

1996

# A RULE-BASED APPROACH TO ANIMATING MULTI-AGENT ENVIRONMENTS

YE, VICTOR

<http://hdl.handle.net/10026.1/2812>

---

<http://dx.doi.org/10.24382/4939>

University of Plymouth

---

*All content in PEARL is protected by copyright law. Author manuscripts are made available in accordance with publisher policies. Please cite only the published version using the details provided on the item record or document. In the absence of an open licence (e.g. Creative Commons), permissions for further reuse of content should be sought from the publisher or author.*

# **A RULE-BASED APPROACH TO ANIMATING MULTI-AGENT ENVIRONMENTS**

by

**VICTOR YE**

B.Sc., University of East Anglia, England, 1989

M.A., University of Bournemouth, England, 1990

A thesis submitted to partial fulfilment

of the requirements of

the University of Brighton

for the degree of

Doctor of Philosophy

April 1996

701 YE

90 0660879:9



Ye, Victor Thur Jian

A Rule-Based Approach to Animating Multi-Agent Environments

Thesis directed by Dr. Colin Beardon

## **ABSTRACT**

This dissertation describes ESCAPE (Expert Systems in Computer Animation Production Environments), a multi-agent animation system for building domain-oriented, rule-based visual programming environments.

Much recent work in computer graphics has been concerned with producing behavioural animations of artificial life-forms mainly based on algorithmic approaches. This research indicates how, by adding an inference engine and rules that describe such behaviour, traditional computer animation environments can be enhanced.

The comparison between using algorithmic approaches and using a rule-based approach for representing multi-agent worlds is not based upon their respective claims to completeness, but rather on the ease with which end users may express their knowledge and control their animations with a minimum of technical knowledge.

An environment for the design of computer animations incorporating an expert system approach is described. In addition to direct manipulation of objects on the screen, the environment allows users to describe behavioural rules based upon both the physical and non-physical attributes of objects. These rules can be interpreted to suggest the transition from stage to stage or to automatically produce a longer animation. The output from the system can be integrated into a commercially available 3D modelling and rendering package.

Experience indicates that a hybrid environment, mixing algorithmic and rule-based approaches, would be very promising and offer benefits in application areas such as creating realistic background scenes and modelling human beings or animals either singly or in groups.

A prototype evaluation system and three different domains are described and illustrated with preliminary animated images.

*to my parents*

# ACKNOWLEDGEMENTS

My sincere thanks goes to:

my family, for their unparalleled support and encouragement through all these years.

Colin Beardon, my supervisor, for providing me with so much invaluable advice and discussions.

John Vince, my advisor, for providing me with crucial insights into the world of computer graphics.

Peter Comninos, my advisor, for inspiring me with the initial idea of doing a PhD.

Apple Computer Inc., for introducing the Macintosh to me since 1986.

and mostly, the Rediffusion Simulation Research Centre for financing my research.

© 1996 Victor Ye.



Software used: Claris MacDraw Pro 1.5v1, LPA-MacProlog32 v1.05, PACo Producer 2.0, ResEdit 2.1.3 (Apple Computer Inc.), SwivelPro 2.0.4 (VPL Research), Microsoft Word 6.0.1.

Types set in: Chicago, Garamond, Garamond Book, Garamond BookCondensed, Geneva, Helvetica, LogosPiFont, Monaco, New Baskerville ItalicOsF, New Baskerville, New BaskervilleSC and Zapf Dingbats.

System of referencing used: Fred Sowan & Ellis Horwood (1987) "Publishing with Ellis Horwood. An authors' guide to the publication of works in science and technology". Second Edition. Ellis Horwood Limited.

# CONTENTS

COLOUR PLATES vii

PREFACE 1

OUTLINE 3

## **1 BEHAVIOURAL ANIMATION 5**

1.1 ANIMATION & SIMULATION 7

1.2 BEHAVIOURAL ANIMATION 13

1.3 A RULE-BASED APPROACH TO ANIMATING BEHAVIOUR 17

1.4 A WORKING ENVIRONMENT FOR ANIMATORS 20

1.5 STRUCTURE OF THE THESIS 23

## **2 THE SOFTWARE ENVIRONMENT 25**

2.1 SOFTWARE ARCHITECTURE 27

2.2 DATA STRUCTURES 29

2.3 USER INTERFACES 37

2.4 LANGUAGE PROCESSING OF GRAMMAR RULES 49

2.5 RULEBASE & RULE INTERFACE 55

2.6 INFERENCE ENGINE 63

2.7 COLLISION DETECTION 67

2.8 INPUTS & OUTPUTS 69

2.9 SUMMARY 72

## Contents

### **3 WORLD.JELLY 73**

- 3.1 WORLD.JELLY DATABASE 75
- 3.2 RULES PARSING 81
- 3.3 INFERENCING 86
- 3.4 ORDERING 96
- 3.5 RESULTS 100
- 3.6 SUMMARY 111

### **4 OTHER EXAMPLES 112**

- 4.1 WORLD.TRAFFIC 114
- 4.2 WORLD.BIRD 125
- 4.3 SUMMARY 135

### **5 CONCLUSIONS 136**

- 5.1 RESEARCH CONTEXT AND FINDINGS 138
- 5.2 CONTRIBUTIONS 140
- 5.3 LIMITATIONS & FUTURE WORK 142
- 5.4 INTEGRATION INTO OTHER SOFTWARE 143
- 5.5 CONCLUSIONS 145

### **APPENDICES 147**

- A EXPERT SYSTEMS 148
- B PROLOG 153
- C DCG LISTING 160
- D WORLD.JELLY DOMAIN DATABASE 162
- E WORLD.TRAFFIC DOMAIN DATABASE 167
- F WORLD.BIRD DOMAIN DATABASE 176

### **REFERENCES 180**

### **PUBLISHED PAPERS 186**

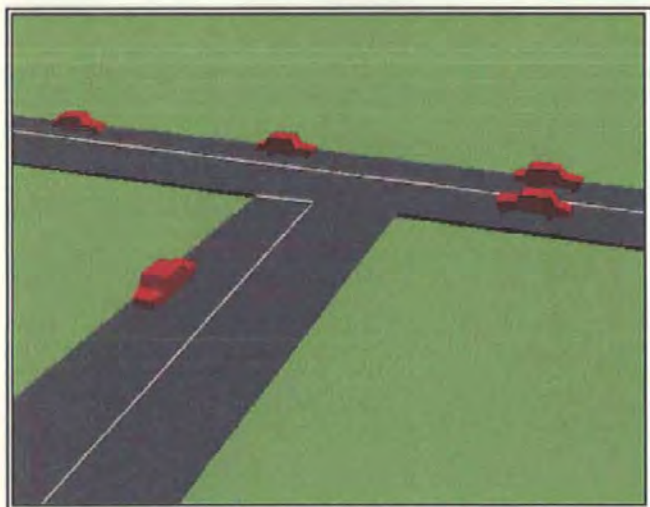


## COLOUR PLATES



**Plate1:**

A rendered scene from the World.Jelly animation: an underwater world of 3 prawns, 2 jellyfish and 1 rock.



**Plate2:**

A rendered scene from the World.Traffic animation: showing the behaviour of 5 drivers approaching a T-junction.



**Plate3:**

A rendered scene from the World.Bird animation: based on a flock of birds in flight that have to navigate around a number of fixed obstacles.

# PREFACE

To date most computer systems that allow for three dimensional modelling and animation are concerned entirely with the physical properties of objects. That is to say, their objects are modelled by being given a physically defined shape, a number of properties (e.g. colour) and a location and orientation in three dimensional space. A 2D representation of an arrangement of such objects (a single 'frame') is produced by specifying the location and orientation of light sources and the point of view. An animated sequence is produced by specifying how these properties change from frame to frame. This can be done explicitly (i.e. by hand), but this is expensive. Normally it is done implicitly by means of an algorithm. Systems can also be built that provide a set of general-purpose procedures that are independent of any particular domain. Craig Reynolds (1982) for example, developed the Actor/Scriptor Animation System (ASAS), a full programming language especially designed for animation and graphics that employs a procedural notation.

There are some very practical reasons why one might wish to supplement these approaches based solely upon the physical attributes of objects by incorporating some ideas from cognitive science. Within complex domains, and particularly where planning and intentional action are involved, capturing realistic behaviour can be very time consuming and expensive. Convincing behavioural animations require both complexity and irregularity and these cannot be easily captured algorithmically. Even if they can be achieved for short sequences, there is a tendency towards repetition if the algorithms are not made more complex as the length of the animation increases.

The purpose of this thesis is to seek a way to model an environment for animating behaviour, and to describe an attempt to enhance physicalist approaches in a principled way by incorporating one of the best known ideas from the field of Artificial Intelligence and cognitive science, namely expert systems (or rule-based systems). A working environment in which a rule-based approach coexists with more traditional modelling and animation methods will be described. The environment is based upon the functions of two different types of user: the end user, who is the person producing a

## P r e f a c e

particular animation; and the domain expert, who provides knowledge about the domain being modelled.

A prototype software system is described and the use that each type of developer will make of the system is shown. In order to validate the model, four sample animations are described (two in Chapter 3 and two in Chapter 4) that use the software environment. Finally some possible application areas and some limitations of the approach will be discussed and an indication given of future work.



# OUTLINE

The four main chapters of this document are centred around the themes: behavioural animation, rule-based system, human-computer interaction, and domain-oriented applications, while a fifth chapter addresses the conclusions that can be drawn from this work. At the end of this thesis are the Appendices, References, followed by two of the papers published by the author which are related to this research.

**Chapter 1: Behavioural Animation.** This chapter includes a brief introduction to the fundamentals of traditional animation and interactive computer graphics, in which past approaches in creating computer animation are discussed. This leads to a description of some of the problems in trying to animate the behaviour of agents in traditional animation environments. Finally the chapter discusses how an animation environment called ESCAPE can be constructed to represent the behavioural properties of the agents, and how this can be used within a research project to discover new knowledge about computer animation.

**Chapter 2: The Software Environment.** The design of the ESCAPE system in the context of the above themes is discussed, some possible limitations depicted, the method of describing the behaviour of agents are illustrated and some scenarios illustrating typical use situations of the system are provided.

**Chapter 3: World.Jelly.** This chapter describes how, by using the ESCAPE system, an underwater scene with three types of object (jellyfish, prawn and rock) can be modelled as a domain, and illustrates the principles of rule-based programming. It also discusses the issues of how optimum results can be derived from the system.

**Chapter 4: Other Examples.** The ESCAPE system is validated by using two more examples: a road junction traffic domain and a flocking bird domain. The chapter also outlines the important issues in creating a new domain.

## Outline

**Chapter 5: Conclusions.** Through the research context and findings, the research's contributions to animating behaviour are summarised, possible application areas are projected, some limitations of the approach discussed and an indication given of future work.

**Appendix A: Expert Systems.** A brief introduction to expert system.

**Appendix B: Prolog.** A brief introduction to Prolog (an expert system based programming language).

**Appendix C: DCG Listings.** The ESCAPE implementation and a listing of the DCG is described.

**Appendix D: World.Jelly domain database.** A complete listing of the World.Jelly database.

**Appendix E: World.Traffic domain database.** A complete listing of the World.Traffic database.

**Appendix F: World.Bird domain database.** A complete listing of the World.Bird database.



**CHAPTER 1**  
**BEHAVIOURAL ANIMATION**



**behaviour** *n.* 1. manner of behaving. 2. the response of an organism to a stimulus. 3. the action or functioning of a machine, etc., under normal or specific circumstances.

The Collins Concise Dictionary (1988)

## Chapter Overview

This chapter begins with an introduction to animation techniques ranging from traditional to 2D and 3D computer animation. Relevant past work is described to give the reader an overview of the techniques involved in computer animation and some of the applications of computer animation, including visualisation and simulation.

The meanings of what is meant in the thesis by “behavioural animation” and “agents” will be explained, and the common problems encountered in a multi-agent environment described. Examples will be given using different traditional animation techniques, and the problems of using these approaches outlined. Some more recent ideas for animating the behaviour and gesture of human beings will also be examined.

The chapter then considers rule-based systems, which are derived from the field of Artificial Intelligence and cognitive science and have been used to perform complex decision making processes in fields such as medicine, robotics, and diagnostics. There will be a discussion of how multi-agent behaviour might be represented in a rule-based system, and the expected benefits of using such a system summarised. The thesis will develop an environment in order to test whether these benefits are achievable and what problems arise in practice.

Finally, there is a discussion of how traditional computer animation techniques, a graphical human computer interface (HCI) and an expert system can be combined in a production environment for animators to represent behavioural knowledge.

### 1.1 Animation & Simulation

This section starts with an introduction to traditional animation, describing how it is produced and some of the difficulties in doing so. It then proceeds to computer animation, outlining its various techniques in visualisation and simulation and some of its uses. It will also look at the relationship between traditional and computer animation.

Since this thesis is centred around 3D computer graphics, '3D computer animation' is referred to as 'computer animation', unless otherwise stated.

#### 1.1.1 Traditional Animation

Any animation is achieved by updating a sequence of images very rapidly at a constant rate. In traditional animation (such as Walt Disney's classic production of *Bambi*, *Snow White*, *Pinocchio* and etc.), the images were individually hand-drawn by the animators, usually onto transparent cels. Traditional animation is a highly skilled profession, and it is still very much practised. Usually, the chief animators would draw a few specified painted 'keyframes', and the in-between frames are interpolated from those and drawn /rendered by helpers. Slight deformation and displacement of the subjects on each cel would then be produced and, when recorded on an output medium such as video or film, frame by frame in sequence, and played back at 25 fps<sup>1</sup>, a moving picture would be created. The smoothness of the animation would generally depend upon the quality of the in-betweening frames between each keyframe.

It is sufficient to say that most animations done this way are 2D, with the exception of 2½D, whereby multiple layers of a drawing overlap each other. For example, the background may be on one cel, static characters on another and the moving character on top. In this way the bottom two cels can be re-used in a number of frames. It might also be that the cels are moved relative to one another in successive frames, without being redrawn, so that, for instance, a background could be scrolled past to suggest the movement of the character in front. This gives 'depth' and pleasing results in the final composition.

---

<sup>1</sup> fps (frames-per-second). The standard rate for the U.K. TV is 25 fps, 24 fps for standard motion movie films, and 30 fps for American TV.



One of the major difficulties in traditional animation is that a huge number of cels have to be hand-drawn just to make a short animation. For a feature film lasting over 160 minutes and containing of 250,000 cels, individual drawings would take 50 years of labour if all were to be drawn by one person (Halas 1974).

### 1.1.2 Computer Animation

Computers have been used for some time as painting tools and for creating geometrically shaped objects. Using computers to explore different techniques for creating interesting visual effects, and to be able to animate them, are high on the agenda for the potential purposes of scientific visualisation, commercial uses (e.g. advertising), and entertainment (e.g. video games). There are some basic principles that can be applied to animation in general and form a close link between traditional animation and computer-based animation.

One of the pioneering works which describes the application of the basic principles of traditional 2D hand-drawn animation to 3D computer animation was written by Lasseter (1987). Lasseter argues that unfamiliarity with the fundamental principles of traditional animation techniques in 3D computer environments will produce *bad* animation, and that an understanding of the principles of traditional animation is essential to producing *good* computer animation. Since 3D computer animation uses 3D models instead of 2D drawings, few traditional animation techniques can be directly applied, but some of the principles mean the same regardless of the medium of animation; for example, motion is achieved by setting keyframes and having the in-between frames generated from them.

Apart from the principles of traditional animation, computer animation also inherits the basic principles of computer graphics (Foley *et al.* 1990, Newman & Sproull 1979, Vince 1992), in that objects are represented using fundamental entities such as coordinates, transformations, modelling, rendering and textures.

The end results produced by computer animation are usually stored digitally either on disks or magnetic tapes. To view an image, the appropriate software is required and, most importantly, a monitor or a VDU (Visual Display Unit). As an example, a 16 bit home PC may be able to display a palette of 512 colours at a resolution of 320 x 200, whilst a 24 bit workstation may display any of a palette of 16.7 million colours at a resolution of 1280 x 1024. The storage size of each digital image usually depends on the resolution and the dimension of the actual image.

To preview a fully-rendered animation (with colours, shadows, lights etc.) directly from the CPU (Centre Processing Unit), a sequence of images would need to be pre-rendered. Depending of the size of the images and the speed of the CPU, the playback is usually so slow that the viewer cannot get the real flow of the animation, with the exception of a few specialist applications such as real flight simulators where super computers are usually being used to render the animation on-the-fly. However, when using a general animation system, the transference of rendered images onto film or video is highly recommended in order to watch the animation playing back in real-time.

Computer animation methodologies can be characterised as belonging to one of three categories: *Keyframe Animation*, *Procedural Animation*, and *Dynamic Simulation*.

### 1.1.3 Keyframe Animation

Keyframe computer animation is derived directly from traditional animation techniques in which the animator constructs what is to appear in each frame and thus explicitly specifies the kinematics or motion of the system. The computer adds efficiency by interpolating *in-between* frames from user supplied *keyframes* (O'Donnell & Olson 1981, Stern 1983). For display generators, early examples used the Evans and Sutherland Picture System. This device contained special-purpose hardware which multiplied 4 x 4 matrices and transformed 3D co-ordinate data.

In more recent years, two such commercially successful keyframe animation packages are the Alias *Wavefront*<sup>2</sup> and Microsoft's *SoftImage*<sup>3</sup> animation systems. These kinds of system usually consist of three main parts: (1) a modeller for modelling objects, (2) a visualisation interface for movement control and (3) a renderer for generating frames for an animation. Modelling the behaviour of objects in such systems would involve the direct manipulation of the objects by the animator via the visualisation tools provided, and setting the keyframes explicitly. An algorithmic behaviour (for example a flight-path defined by an equation) can be generated this way from these keyframes by the application of computerised in-betweening.

---

<sup>2</sup> Alias *Wavefront*, Wavefront Technologies, 530 East Montecito, Santa Barbara, CA 93101.

<sup>3</sup> Microsoft *SoftImage*, Microsoft Corporation, One Microsoft Way, Redmond, WA 98052-6399.

### 1.1.4 Procedural Animation

To achieve the degree of realism found in other areas of computer graphics, the motion of objects can be simulated by the physical principles of kinematics/dynamics that govern their motion, for example objects which are subjected to external forces such as wind and current flow. One way to do this is by scripting - writing procedures so that the animation is determined by the interpretation of the script (i.e. a procedure that is related to an object).

There are animation systems that incorporate this scripting ability. An example of this type of system is CGAL<sup>4</sup>. As the interpretation of scripts is relatively quick, it is possible to generate previews of the animation (using simple line drawings of the objects) very easily and an animation can be developed in a top-down manner (with increasing detail). For this, the end user is required to have some programming knowledge and the capability to handle the script as it becomes larger and more complex.

Procedural graphics (Rogers 1989) are usually created using one of the many algorithmic programming languages available today such as PASCAL, C, or C++. Different programming languages have been designed to handle different tasks. Some have been designed especially to handle animation and graphics, for example Reynolds' ASAS (1982). Some organisations have attempted to set standards for computer graphics libraries to be used in conjunction with the algorithmic programming languages mentioned above, they include the International Standards Organization's (ISO) Graphical Kernel System (GKS), and the American National Standards Institute's (ANSI) Programmers' Hierarchical Interactive Graphics System (PHIGS).

Procedural methods do give animators a great deal of control over the animation, allowing them to work very precisely, specifying where each object has to go and what it should do next. These methods work in a way such that the next state of each object is usually determined and amended with the incrementation of a mathematical function on each pass through a program loop. The mechanism of the function can be quite simple; it could have a linear entity (e.g.  $Y = X + 2$ ), non-linear

---

<sup>4</sup> CGAL (Computer Graphics Animation Language). Developed by Peter Conninos. Information about CGAL can be obtained from Peter Conninos at the National Centre for Computer Animation, Bournemouth University, U.K.

(e.g.  $Y = X^2$ ), or more complex applied mathematics such as kinematic equations. They rely on the computer's ability to determine the kinematics based on implicit instructions rather than explicit positions.

Another class of procedural method is *inverse kinematics*, where the motion of end links in a chain is specified by the animator, but the motion of interior links is determined algorithmically (Badler *et al.* 1991, Girard 1986, Girard *et al.* 1985).

Although procedural methods have produced some of the best animation to date, they often lack dynamic integrity especially when associated with realistic movements under the influence of physical constraints such as gravity and external forces such as the interaction with the ground plane. For them to interact in realistic ways, these dynamic principles would need to be applied.

### 1.1.5 Dynamic Simulation

To capture the realistic appearance of a geometrical model, some degree of dynamic motion control is needed. Dynamic motion control is derived from physical laws, whereby the progress of the motion is controlled by the application of forces such as gravity and friction.

An early example of Waters' (1987) initial facial animation technique, which modelled the effects of muscle tensions over a region of skin, did not respond in a physically realistic fashion to external forces because the model was purely geometrical rather than physical. However, dynamic simulation has been incorporated in his latest associated work (Lee *et al.* 1995), in which the algorithms devised automatically insert contractile muscles at anatomically correct positions within a dynamic skin model and root them in an estimated skull structure with a hinged jaw. They also synthesise functional eyes, eyelids, teeth, and a neck fitted to the final model. This dynamic model offers increased realism compared to the original geometrical model.

Miller (1988) focused on a method for modelling and animating legless figures such as snakes and worms. Each segment of the creature is modelled as a cube of masses with springs along each edge and across the diagonal of each face. For each time interval, muscle contractions are simulated by animating the spring tensions - the spring lengths and spring length velocities are used to compute the forces exerted on the masses at the end of each spring. This takes into account the creature's response to external forces, for example, forces exerted by the floor and gravitational forces.

General systems for dynamic simulation of linked figures, especially the investigations in the field of legged animal movements similar to the human figures, have been described in recent literature relating to robotics and biomechanics as well as computer graphics. To help to lay the basis for controlling individual parts in the limb, problems are addressed that are associated with modelling hierarchical limb kinematics and dynamics (Bruderlin & Calvert 1989, McKenna & Zeltzer 1990, Phillips *et al.* 1990, Raibert & Hodgins 1991), and in optimising motion in the presence of kinematics and physically-based constraints (Cohen 1992, Ngo & Marks 1992, Ngo & Marks 1993, Witkin & Kass 1988), satisfying the *what* constraints and optimising the *how* criteria: *what* it should do: “jump from here to there, clearing a hurdle in between”, and *how* it should be performed: “don’t waste energy”.

Another good example is an animated film called *Animation Aerodynamics* (Wejchert & Haumann 1991), in which a method based on aerodynamics is used to simulate and control the motion of flying leaves blowing in the wind. Since animation has less stringent accuracy requirements, the authors argued that computational expense can be avoided by dividing the system into two parts: a linear flow regime and an object boundary regime. In the linear flow regime, instead of solving the flow numerically, analytic solutions such as using vortices, sinks and uniform flows, are mixed linearly to create a more complex scenario. As for the object boundary regime, each leaf is built out of masses and springs with slight variations in geometry, mass distribution and stiffness. Finally, the total force acting on a mass is made up of contributions from gravity, spring stretching and the external fluid force - linear flow.

Although it serves its purpose in producing physically accurate models, it is computationally expensive, because most dynamic simulations involve the solving of differential equations. When dealing with behavioural animation, dynamic simulation lacks that ability to represent intentional behaviour naturally because the way data or information is represented and stored. By separating out some of the *behavioural* properties from the physical models, and providing a communication tool between the animator and the system, it is hoped that these consequences could be minimised.

### 1.2 Behavioural Animation

In this section, there will be a brief introduction of what “behaviourism” is and its relationship with today’s animation systems. What is meant in this thesis by “behavioural animation” and “agents” will also be described.

#### 1.2.1 Behaviourism

During the 1950s there was a philosophical movement associated with the psychologist B.F. Skinner known as “behaviourism” (Skinner 1974). In the extreme form of behaviourism, behaviourists argued that human behaviour must ultimately be explained in terms of physical, observable properties. They based their approach on observing behaviour and, in particular, they sought to determine the relationships between the observable stimuli received by a subject and the observable response that the subject exhibits as a result. A true science of behaviour, it was maintained, would treat a human being as a black box and merely describe the rules that relate its input to its output. This must be achieved without recourse to internal entities or states. Behaviourists would generally deny the significance of internal states and some extreme behaviourists argue that all mentalistic terms are theoretically unnecessary as they can be redefined in terms of observable, physically describable behaviour.

Modelling and animation software that only allows the representation and manipulation of physically observable features of objects is based upon similar assumptions. Within such software there is no mechanism for representing the inner states of objects and all behaviour has to be expressed and manipulated in terms of physically describable features.

There are many examples of contemporary modelling software that are living monuments to this behaviourist philosophy. Their objects only have physical properties and their transformations only concern changes to the physical attributes of objects. They are also, it has to be admitted, very successful for many purposes, especially where no agents (see § 1.3) are involved.

Skinner’s behaviourism came under attack from Noam Chomsky. Chomsky argued (Chomsky 1959) very convincingly that there are many forms of human activity, language being his central case, which simply cannot be explained in behaviourist terms. He later added to his argument (Chomsky 1965) that there are complex internal and non-observable mechanisms at work and a failure to recognise them will lead to

ineffective explanation. Chomsky himself concentrated upon the structure of language, while others in psychology became interested in cognitive structures.

Around the same period, a program called the *Advice Taker* was proposed by John McCarthy (1959). Its intention was to show common sense and improvable behaviour by using declarative and imperative sentences as the representation, and immediate deduction as the reasoning mechanism. The property of this “programmed-knowledge” was expected to have much in common with what makes us describe certain humans as having *common sense*. McCarthy explains:

‘A program has *common sense* if it automatically deduces for itself a sufficiently wide class of immediate consequences of anything it is told and what it already knows.’

Cognitive psychology, in general, is based upon the model of the human mind as a processor of information (with a memory). Explanations of the behaviour of human beings are not restricted to descriptions of input and output data, but can also refer to the internal representation, storage and processing of that data - different outcomes can arise from different situations for example. Cognitive science and the more applied field of Artificial Intelligence seek to complement cognitive psychology by building computational models of sufficient richness for representing such behaviour. Early cognitive science concerned itself primarily with psychological models of individuals, but later the relationship of the individual to the environment, particularly where that environment contained other individuals, became an important topic of study.

In the early 1980s, a mathematical approach using natural language semantics called *situation theory* (Barwise & Perry 1983) was introduced as an attempt to provide a mathematical foundation for situation semantics, and since then, situation theory has developed into a general framework for the study of information within a multi-agent environment (Devlin 1991, Devlin 1992). Here is Devlin’s definition of situation theory:

‘In the situation-theoretic treatment of cognition, information-processing, and communication, recognition is made of the partiality of information due to the finite, *situated* nature of the agent (human, animal, or machine) with limited cognitive resources. Any real agent must employ necessarily limited information extracted from the environment in order to reason and communicate effectively.’

The conclusion of this philosophical discussion is that: if we are to model worlds in which there are multiple interacting agents, then we will need to enhance our software environments to represent internal states and mechanisms. Furthermore, we will need reasoning mechanism in our software so that “in-betweening” can be done in a way compatible with the richness of the environment. One promising area to look for such mechanisms is cognitive science, or more particularly Artificial Intelligence and rule-based systems. These may provide us with more appropriate tools to represent agents that have imperfect knowledge of the world around them and employ deliberate strategies to decide what action to take next.

### 1.2.2 Animating Behaviour

In the world as described by physics, there are certain constraints upon objects, for example, two solid objects cannot occupy the same point in space, and (on earth) an unsupported object will fall due to the effect of gravity. Modelling and animation systems can be envisaged that check such constraints and make objects bounce off each other if they collide and fall to the ground if they lack support. In such systems, a physical model exists in which it is easy to say that all physical objects display a certain kind of *behaviour*.

Behavioural animation is a means for automatic motion control in which animated objects in a multi-agent environment are capable of sensing their environment and determining their motion (or other change) within it according to certain rules or constraints.

Some of the earliest work on exploring behavioural animation can be seen in the classic computer animation sequence ‘Eurythmy’ (Amkraut *et al.* 1985), in which force fields were employed to influence the behaviour of a flock of flying birds. During their flight, the birds’ wings moved in a life-like fashion, while the birds simultaneously avoided contact with one another. Similar behaviour has also been investigated by Craig Reynolds (1987). In this work, the simulated flock is an elaboration of a particle system in which each particle acts according to an identical algorithm. Each simulated bird is considered as a particle that behaves according to the same behavioural rules as every other. This approach assumes that the behaviour of the flock is simply the result of the interaction between the behaviour of the individual birds. The motion is created by evaluation of the positions and velocities of all birds in the flock. Their next position is



calculated according to three simple rules: avoid collision, match the velocity of neighbouring birds and keep toward the centre of the flock.

Behavioural animation has gained much popularity especially in modelling living organisms. Artificial life (Langton 1989) is an approach to synthesising lifelike behaviours within computers. This approach can be seen in *Artificial Fishes* (Tu & Terzopoulos 1994), which is presented as a virtual marine world inhabited by a variety of fishes acting as autonomous agents. The authors pursue a procedural approach which takes into account the superficial appearance and the basic physics of the creature, including its environment, its means of locomotion, its perception of its world, and its behaviour.

Other means of controlling behavioural animation is proposed by Wilhelms & Skinner (1990) based upon the idea of a connectionist architecture (Feldman 1985). Using the system, the user designs a network of four symbolic components: (1) *sensors* - which detect designated stimuli; (2) *effectors* - which produce a change of state such as motion; (3) *nodes* - which map input signals to output signals; and (4) *connections* - which pass information through the networks from sensors to nodes and on to effectors. Animations of a wide range of similar motions can be generated quite quickly, but the main problem lies in the understanding of the design and the representation of the network.

Behaviour has also been simulated by using the combination of kinematic and dynamic specifications in human articulated figures (Isaacs & Cohen 1987, Phillips & Badler 1991). Using such approaches animators can plan each part of the animation in the way they perceive most suitable for the task. These behaviour constraints model certain behavioural tendencies which capture some of the characteristics of human-like movement, and give greater control over such elements as the balance and stability. Successful experimental results were presented which demonstrated the ability to provide control without disrupting the dynamic integrity of the resulting motion.

Admittedly, procedural approaches have produced convincing animations to date but, for the animators, there is the problem of adaptation. To adapt the required behaviour, the animator is required to program, working precisely with only the physical properties of the subjects. An important issue in animating behaviour is the means by which the user is able to concentrate on controlling the animation and the ease with which the user can alter both the general nature and specific details of the motion. One

possible way to approach this is to construct an animation production environment which consists of two separate components: one for handling the physical attributes, and the other for the behavioural.

The idea is to take a conventional physically-based (as mentioned in § 1.1) animation technique and enhance it with a rule-based system. The physically-based part of the system is used for governing the physical properties of the animation such as shapes, coordinates, and colours; while the rule-based part of the system is used for controlling the behavioural properties such as planning and intentions within the animation. As in any rule-based system, rules are used to give the system instructions of how to behave. The expression of these rules can be formalised so that it is accessible (i.e. easily read and understood) by both the system and, most importantly, the user operating the system - one of the main drawbacks of any procedural approach.

The kinds of problem where this type of system might be of considerable advantage would be in environments where one or more agents are involved, for example in the simulation of flocking birds and artificial life forms.

### 1.3 A Rule-based Approach to Animating Behaviour

The key parameters for creating artificial life in virtual worlds have been identified in Thalmann and Thalmann (1994). The Thalmanns note that there has been very little visual representation of living organisms in such worlds and only behaviourally very simply creatures in them. Already the Thalmanns have successfully incorporated into the animation of photo-realistic human models, some purposeful human behaviour using Schank's scripts (1980) which are cognitive, behavioural descriptions. Schank's scripts are effective for stereotypical scenes but are limited in modelling more open scenarios where there may be a need for planning or, more generally, for a form of situated action (Suchman 1987).

There are some very practical reasons why one might wish to supplement approaches based solely upon the physical attributes of objects by incorporating some ideas from cognitive science. Particularly within domains where planning and intentional action are involved, capturing realistic behaviour requires both complexity and irregularity which cannot easily be captured algorithmically. Even if they can be achieved for short sequences, there is a tendency towards repetition if the algorithms are

not made more complex as the length of the animation increases, and this can be very time consuming and expensive. Another advantage of using ideas from cognitive science is the potential *transparency* of explanations of animated behaviour through accessibility to the chains of reasoning employed by the system.

Rule-based (or expert) systems are a branch of Artificial Intelligence (see **Appendix A**) which has played a major part in today's cognitive science. The following definition from Barr *et al.* (1989) is representative of opinion in the field of AI:

'Artificial Intelligence (AI) is the part of computer science concerned with designing intelligent computer systems, that is, systems that exhibit the characteristics we associate with intelligence in human behaviour - understanding language, learning, reasoning, solving problems, and so on.'

In other words, AI is concerned with programming computers to perform tasks that are presently done better by humans, because they involve such higher mental processes as perceptual learning, memory organisation and judgmental reasoning. Thus, a program that performs complicated statistical calculations would not be seen as performing an Artificial Intelligence activity, while a program that designs experiments to test hypotheses would.

Experiments based on the idea of AI in computer animation which are relevant to multi-agent environments can be seen in systems that use *Actor* (Pugh 1984, Reynolds 1982) and similar object-oriented systems (Agha 1986, Hewitt & Atkinson 1977, Repenning 1993, Rettig *et al.* 1989, Stefik & Bobrow 1984). These systems take the view that a single entity, an *Actor* (or *agent*), be used to represent both data and procedures.

An *agent* is a small processor consisting of a local knowledge base, and defined solely by its behaviours (in a script form). Its behaviour is characterised by its response to receiving a message. Communication between agents is achieved through the single metaphor of message passing. If appropriate, the agent can be provided with operations which allow other agents to interrogate and (possibly) update this knowledge base. More about *agents* can be found in the current volumes (1 & 2) of the proceedings 'Intelligent Agents'<sup>5</sup> (Wooldridge & Jennings 1995, Wooldridge *et al.* 1996).

---

<sup>5</sup> 'Intelligent Agents' are the proceedings published in conjunction with the International Workshop ATAL (Agent Theories, Architectures, and Languages). The aim of the workshop is to bring together the issues surrounding the design and implementation of agents, in

Expert systems are also called 'knowledge-based' or 'rule-based' because their performance depends critically on the use of facts and rules in a similar form to that used by experts. The systems incorporate a body of knowledge from humans who are expert in the subject area concerned, and whose skills in the area are formalised into a database usually in the form of rules. The domain of the subject area is necessarily limited, but within that area the expert system can offer intelligent advice and make intelligent decisions based on the information available in its database.

As in scientific simulation, knowledge-based programming uses digital computers to construct models of a system and to execute these models in order to predict or obtain information. These two methods differ in the way the system is modelled and in how the model is used to make predictions. The behaviour of a system in a simulation model is usually represented by mathematical equations or probability distributions, and in a knowledge-based program by rules, which state which actions will be performed under which conditions. Simulators predict the future state of a system by propagating the values of system variables through time, whereas knowledge-based programs infer facts about the state of the system or show how to achieve a predefined state. Simulators generally function as black boxes that take numerical data as input and produce numerical data as output, whereas knowledge-based programs are often able to explain the reasoning process that led to a given result.

One area of human animation where expert systems have been used is in the modelling of human/robot synthesised hand grasping (Bekey *et al.* 1991, Rijkema & Girard 1991). Observing that people adopt different grasping strategies depending upon the shape of the object and the angle of approach, an expert system approach was used to determine the appropriate strategy (e.g. small objects are picked with the thumb and the index finger; mugs are picked by the handle) within a *hybrid system* that also contained algorithmic approaches for the finer movements such as making contact. The basic underlying steps in the program are: the classic shape of the target object is first identified (e.g. block, sphere, cone, etc.); a grasping strategy associated with this is derived from the knowledge-base of class specific, parameterised techniques; the final grasping motion is then adjusted to adapt to the object's deviation from the classic shape.

---

particularly the link between agent theories and the realisation of using software architectures or languages. Internet URL <http://www.doc.mmu.ac.uk/STAFF/mike/atal95.html>

In the above system, the expert system component was designed for the particular task of grasping and the animator who wishes to modify the rules must become knowledgeable about expert systems - i.e. has a good idea of how expert systems work and how the rules are represented. The opposite approach is adopted by *KidSim* (Canfield *et al.* 1994), a rule-based environment that aims to put the writing of rules into the hands of the end users - in this case children. *KidSim* is a production environment for building animations by declaring a set of agents and graphically describing a set of rules by showing examples (Cypher 1993, Repenning 1993), i.e. by direct manipulation of visual objects (Hutchins *et al.* 1986, Shneiderman 1989, Strauss 1992). The rules are not displayed in symbolic form but rather as transformations of the graphical field. The authors claim that *KidSim* is the model for more adult simulation environments, but while the possibility of defining rules through user-friendly interfaces is to be welcomed, its current limitations on knowledge representation and its inability to describe complex interactions will need to be overcome before more sophisticated domains can be simulated. Nevertheless, *KidSim* does provide an interesting model of user control within a rule-based approach to modelling and animation.

In a rule-based environment, realistic behaviour of the agents can be achieved by strategic planning or multi-level planning. This kind of planning involves reasoning about the effects of actions and the sequencing of available actions to achieve a given cumulative effect. This is an important aspect of behavioural modelling in some domains, and some system such as this could be seen being included in a *hybrid system* (Badler *et al.* 1994) with a more traditional graphical approach such as *Jack*<sup>6</sup>, where animators can use expert knowledge about the behaviour of agents in order to plan foreground animations.

### 1.4 A Working Environment for Animators

In this section, we will look at how a working environment for animator can be built for the representation of behavioural knowledge. The preliminary design of the proposed

---

<sup>6</sup> *Jack* is a software package that provides a 3D interactive environment for controlling articulate figures. It features a detailed human model and includes realistic behavioural controls, anthropometric scaling, task animation and evaluation systems, view analysis, automatic reach and grasp, collision detection and avoidance. Information on *Jack* can be obtained on the Internet <http://www.cis.upenn.edu/~hms/jack.html>

ESCAPE (Expert Systems in Computer Animation Production Environments) system will be described.

There is not one particular method of animation that is preferable in all situations, and in an ideal multi-agent animation environment where behaviour is of major concern, it should be possible to influence the resulting animation in at least three ways:

1. by directly manipulating agents within specific animation frames;
2. by specifying their properties and locations formally in the database; and
3. by representing their behaviour using a rule-based approach.

The proposed ESACPE environment (Beardon & Ye 1995, Ye 1995) has many traditional features and one new one. Individual objects can be defined in exactly the same way as in existing 3D modelling systems. In order to create individual frames of an animation, objects may be manipulated manually through a traditional graphical interface. To these traditional methods an additional facility is added by means of which objects may have non-physical attributes. The state of each object is derived by an inference engine from behavioural rules that are entered separately by the animator. These rules may be qualified to refer to only particular classes of object, or to objects only under certain conditions, and therefore require that objects be classified into various types and have properties or states associated with them. The outcome of applying a rule might be to change an object's properties or states, or to change its location.

A system with the enhancement described above will enable particular aspects of the *behaviour* of the agents in the domain being modelled to be represented as rules. This extension will not just prevent physical impossibilities such as two objects being at the same point at the same time, but also enable us to express *domain knowledge* - the sort of knowledge that pertains to the content of the animation. This rule-based part of the system can also be left to run unattended and will, if properly set up, automatically generate an animation. However, the expert system component is not there to replace skilled animators, but rather to assist them. The animator has control over the rules and can experiment with them, observing their effect on the animation. The animator can also interrupt the generated animation and use one of the other modes of manual manipulation to override what it has produced.

A major design objective here is to find a representational form for rules that can be used by the inference engine effectively, and can lead to multiple user interfaces. One way to construct a rule is to use the "if-then" structure, for example, an ideal representation of a rule would be:

*if a creature is near a rock, then it will avoid colliding into it.*

Prolog is natural choice for the expression of such rules because there are known implementations of expert systems in the language (see **Appendix B**). In addition, what makes Prolog (particularly MacProlog<sup>7</sup>) attractive here is the built-in grammar rule notation DCG (definite clause grammar) which can be used to provide a common structure for the representation of textual and graphical interfaces. Implementing a grammar in a programming language normally means writing a parsing<sup>8</sup> program for the grammar. A grammar written in DCG is already a parsing program for this grammar. This is especially useful in designing rules with a grammar closer to 'natural language' (§ 2.4).

Also in MacProlog, the HCI design can be built quite easily by accessing the standard Macintosh Graphical Library Tools, for the creation of windows, dialog boxes, and graphical representations. There are three main areas for where good HCI design could play an important part:

1. in the main animation window - how agents are represented and for the direct manipulation of the agents;
2. in the presentation of agents' symbolic data; and
3. in the rule interface - so that the user can access the rulebase easily.

It is hoped that with these enhancement, animators will be able to express their behavioural knowledge of their agents in English-like rules, and that the rule-based system is able to handle reasonable amount of complexity and generate explanations related to the reasoning employed by it.

---

<sup>7</sup> LPA MacProlog supports the industrial standard "Edinburgh Syntax". Information on LPA MacProlog can be obtained from Logic Programming Associates Ltd., Studio 4, The Royal Victoria Patriotic Building, Trinity Road, London SW18 3SX, England.

<sup>8</sup> Parsing is a process that, given a sentence, effectively disassembles the sentence into its constituents. The constituents then can be executed accordingly.

For development and from the developers' point of view, the emphasis on rule-based programming implies use of a language such as Prolog; but for performance and if it were to be incorporated into a larger production environment, more efficient languages such as C++ (Hu 1989) and LISP<sup>9</sup> (Allen 1978, Winston & Horn 1984) would be considered as potential languages for creating expert systems.

### 1.5 Structure of the Thesis

Throughout this chapter various computer animation techniques have been investigated with respect to some of the problems encountered in behavioural animation. It is envisaged that a conventional computer animation system could be enhanced with an expert system processor, so that the behavioural part of the animation can be handled separately. This section describes briefly the methodology that is needed to integrate the implementation of such a system into a piece of research, and how the subsequent chapters will elucidate the contribution to knowledge.

From the development point of view, the expected benefits from rule-based systems are:

- transparency - know what the system is doing;
- incrementality - add and remove information independently; and
- presentability - symbolic interpretation of the rules.

A system will be designed and implemented that we fully expect to have the following benefits from the user's point of view:

- the system is able to generate purposeful animations;
- the rules that describe the behaviour of the agents are easy to write;
- user requires minimum technical skills to use the system.

Through the subsequent chapters, we hope to discover whether the expected benefits of rule-based systems could be realised, and what issues might arise in the implementation that might influence an animator's ability to use the system effectively.

---

<sup>9</sup> LISP (LISt Processing) is a symbolic manipulation language, invented in 1956 by John McCarthy.



Chapter 2 will concentrate on the design and the implementation of the system. Chapter 3 & 4 will validate the system by modelling three different worlds with three different types of problem. Here is a description of the significant features of the three worlds:

1. World.Jelly. An underwater scene consists of different types of object interacting with each other. This example will explore two different approaches: goal-oriented and non goal-oriented.
2. World.Traffic. A scene of car traffic at a T-junction. This emphasises the spatial significance of different sections of the road and the inferences of one car driver about other drivers.
3. World.Bird. A simulation of a flock of birds in flight that have to navigate around a number of fixed obstacles. This investigates issues such as the role of a 'leader', flock centering and collision avoidance.



## CHAPTER 2

# THE SOFTWARE ENVIRONMENT



"Design is intuitive. You have to give people something to look at. It doesn't matter what it is, as long as it isn't boring... Although everything has been done before, George (Lucas) somehow finds a way to do it better, differently."

Joe Johnson in (Smith 1985)

### Chapter Overview

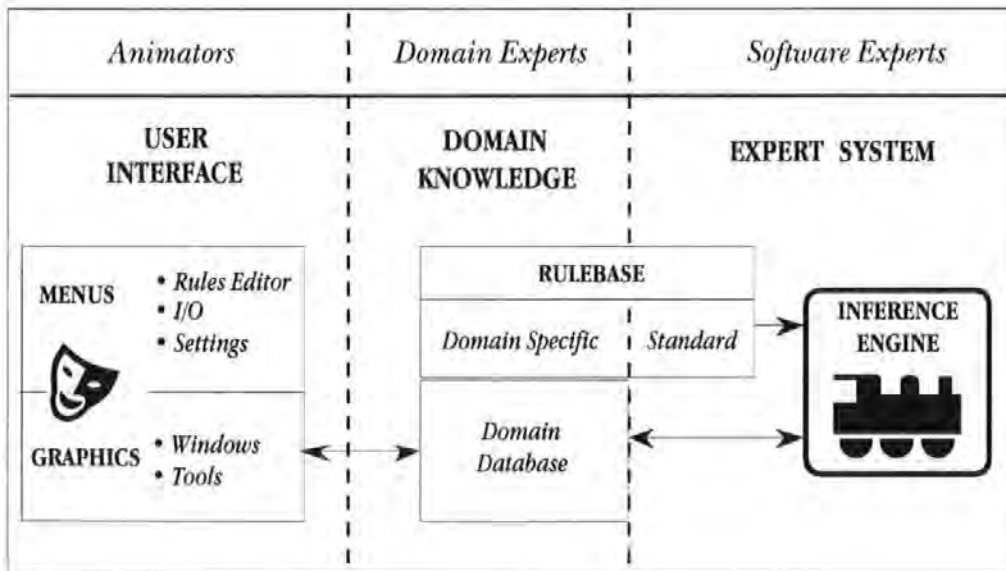
Traditional animation and simulation software usually deals with the physical properties of objects whereas we often have knowledge of how things behave in terms of their intentions or inner states. Traditionally, object behaviour is achieved in computer animation by the application of mathematical algorithms (e.g. procedural equations), or is dependent on the animator explicitly indicating movement from one location to another (e.g. keyframing). These behaviours can be expressed in terms of rules which are used to constrain the location and properties of an object over time. One way in which this might be done is to combine a rule-based system processor with an animation environment.

This thesis describes a rule-based system called ESCAPE (Expert Systems in Computer Animation Production Environments). In this chapter the software architecture of the environment will be discussed. The areas of focus are: the data structures (objects, language and rules), the user interfaces (animation window, tools and rules interface), the database (domains and how information is stored), the inference engine (expert system processor) and input/output formats.

## 2.1 Software Architecture

The ESCAPE system allows for traditional animation through the direct manipulation of objects on the screen and the recording of keyframes by means of an interactive control panel. In addition there is the new facility for defining behavioural rules of objects or classes of objects. The animator can thus use the system to automatically produce an initial animation. This animation can be further refined either by the direct manipulation of objects or by altering the set of rules and re-running the system.

With such an environment it is necessary to distinguish three different functions in the animation production. These functions or levels of users are reflected in the architecture of the software environment, as shown in Figure 2-1:



**Figure 2-1** An Overview of the Software Architecture

The three levels of users shown in Figure 2-1 can be distinguished during the development process in terms of their respective roles:

1. The *software experts* who develop the core expert system component, the part that is common to all application domains/worlds. It consists of an inference engine to interpret the rules against a database and some very low-level rules that provide other users with a useful set of standard predicates and operators.

2. The *domain experts* who set up a particular world by modelling the different types of object, specifying attributes they should have and their default values, and writing the main behavioural rules they follow. The *domain experts* may also specify some lower level domain specific predicates and operators directly by accessing the Domain Database and Rulebase, thus by-passing the User Interface level. In this respect, they need to have some knowledge of Prolog programming.
3. The *animators*, the end users, whose task it is to produce particular animations, are able to update the database by direct manipulation, update the rulebase through a special rule interface called the Rule Editor, and to invoke the inference engine. They use the User Interfaces provided, and knowledge of any kind of programming is not essential.

Once familiarisation with the system is reached, it is possible and relatively easy for an *animator* to become a *domain expert*. However, some advanced knowledge of expert systems would be needed to become a *software expert*.

At the heart of the system lays an *inference engine*, such as one may find in any standard expert system (see **Appendix A**). When invoked it will examine the rulebase and attempt to match the conditions and constraints of each rule against the existing *domain database*, and store any relevant new facts in the working memory. For each iteration, when successful, it will produce a set of possible solutions to be resolved by various methods (discussed in § 3.5) to only one action which will then be carried out. The *graphical windows* will subsequently be updated so the outcome of the actions can be visualised by the animator. All these parts are explained in this chapter.

The rulebase is divided further into two parts: Domain Specific, and Standard. Care has been taken in the design to separate the two and to determine which rules are to be domain specific, and which will remain standard in the expert system as more general-purpose rules capable of handling a range of domains. The rulebase is discussed in § 2.5.

Much of the research described here concentrates on the design of the User Interfaces, the Language Parser and the Rules Interface. The aim being to provide ease of use for the system: there are tools for animating and manipulating *objects* (§ 2.2.1), and the Rule Editor (§ 2.5.3) provides an intuitive interface, allowing the user to create and edit rules easily.

As mentioned in the previous chapter, MacProlog has been chosen for the implementation of the system. While MacProlog's built-in graphical output will suffice for prototyping it is not suitable for a finished product due to its poor graphical output. Since a full frame-by-frame description of each object's precise locations and properties can be obtained from the system, a simple visualisation or preview facility can provide the animator some sense of what the final animation will look like. When considered satisfactory, the results can be output using the appropriate file filters so that they can be read into a commercially available 3D modelling/rendering software package.

A prototype system has been built at the University of Brighton using LPA MacProlog32 running on Apple Macintosh 68030 and 68040 series computers. The system described here models objects in a two-dimensional world as supposed to 3D due to the graphical limitation of MacProlog. The 2D prototype is sufficient to explore the basic architecture of the system without paying particular attention to the quality of the graphical output. To demonstrate its potential it is possible to create a corresponding three-dimensional world in Swivel 3D™ and for the expert system to generate a rendered 3D animation as a QuickPICS file.

## 2.2 Data Structures

Data structures serve to implement and organise complex data sets. These sets are classified into types according to the way they are used. Therefore, depending on the point of view, a data set is characterised by its type (for the user) or by its structure (for the implementer).

The data structures of the ESCAPE system consists of: objects (agents), rule structures, object properties, and user-defined predicates (in standard Prolog code).

In this section, the data structures of objects will first be explained, then the expression of rules and, lastly, how they are collected and bound together to form the Domain database.

### 2.2.1 Objects & Agents

An *object* is looked upon as an *agent* (as mentioned in § 1.5). An agent has behaviour and intentions, which are defined by its properties and modified over time by *rules* (§ 2.4).

There are two layers of object properties: i) *types* that holds information such as class; and ii) *instances*, that holds personal information such as the object's name:

- i) *types*: the part of the object which inherits the properties of objects with the same class/type. The following is an example of the default settings for objects of type **'PRAWN'**:

```

mass:          defaults mass( 'PRAWN', 1).
size:          defaults size( 'PRAWN', 2).
visibility:    defaults visibility( 'PRAWN', 100).
maxSpeed:     defaults maxSpeed( 'PRAWN', 30).
properties:    defaults property( 'PRAWN', [alive]).
picture:      defaults picture( 'PRAWN', resource( 210)).
    
```

Each default property consists of an *attribute* name such as **mass**, **size** and so on, and will have a corresponding *value*. For example, the property **defaults mass( 'PRAWN', 1)** is a formalisation of "the default attribute **mass**, of an object type **'PRAWN'** has a value of 1."

The attribute **picture** holds the information for the picture representation of a particular object type. For example this is for a **'PRAWN'** (Figure 2-2):



**Figure 2-2** Picture Representation of a **'PRAWN'**

This picture representation will be shared by all objects of the type **'PRAWN'**. This particular one is stored as a resource (created by applications such as *ResEdit* - Resource Editor), in PICT format with resource ID=210. During the animation process when a **'PRAWN'** picture is required by the system, it will be referred to by the ID and then copied to the animation window to be updated at the appropriate places.

In an animation environment, it is often desired to animate one or more different types of object at the same time. For example in an underwater

scene, where there are three types of object: jellyfish, prawns and rocks. Introducing and setting up the properties for 10 individual prawns can prove to be a tedious and laborious task. Since objects of the same type usually possess similar properties, such as the ones listed above, a set of default properties for each type of objects can be built, so each object belonging to the same type or class inherits those default properties.

These properties however, will be superseded by the *values* of any *attributes* that are repeated in *instances*.

- ii) *instances*: these are the special properties of an individual object, which are referred to as **facts** here. An object will typically have a specific name (e.g. **prawn1**), be of a specific type (e.g. **'PRAWN'**), and have a specific location. For example:

name:            **fact obj\_type( prawn1, 'PRAWN').**  
location:        **fact location( prawn1, pt( 78, 28)).**

By being a **'PRAWN'**, **prawn1** inherits all the default settings such as those mentioned in i), but the values of the attributes may be overridden by **facts**. Further more, new properties may be applied to **prawn1** by specifying additional information this way, for example:

mass:            **fact mass( prawn1, 2).**  
properties:      **fact property( prawn1, [alive, hungry]).**

Note that for **prawn1**, the value of the attribute **mass** is **2** (heavier than that of the default **'PRAWN'** which is **1**) and an extra property **hungry** (an addition to the default **alive**). These additional properties are the replacement values (i.e. will supersede the default values) while other default properties such as **visibility** and **maxSpeed** etc., will remain unchanged.

For both the animator and the domain expert, it is important that these two layers in the representation of objects are clearly understood, especially when it comes to designing a new domain, which will be discussed later.



### 2.2.2 Expression of Rules

In the ESCAPE system, a formal language has been developed for the expression of rules. The language interpreter and the grammar of the rules is defined using a Definite Clause Grammar (DCG) (Bratko 1990, Clocksin & Mellish 1984, Gal *et al.* 1991, Marcus 1986).

The rules are expressed in the "if-then" format. Since the standard DCG implementation requires Prolog lists for processing, a phrase is presented in the form of a Prolog list. A Prolog list is confined by square brackets ([ ]), and each item (word in this case) is separated by a comma (,). These Prolog lists can be processed by the DCG implementation straightaway without the need for further manipulation. Here are two examples of the rules written using this format:

<b>rule1 :</b>	<b>if</b>	<b>[object1, is_nearby, object2]</b>	% one condition
	<b>then</b>	<b>[object1, meanders].</b>	% one action
<b>rule2:</b>	<b>if</b>	<b>[object1, has_type, PRAWN,</b>	% two conditions
		<b>and, object2, is_nearby, object1]</b>	
	<b>then</b>	<b>[object2, plans_getaway_from, object2,</b>	% two actions
		<b>and, undo, object1, is_hungry].</b>	

The language is composed of:

- keywords (e.g. **:, if, then, and**), % closed set
- variable names (e.g. **object1, object2**), % enumerated sets
- object names (e.g. **jelly1, prawn2**), % closed for particular domain
- properties (e.g. **PRAWN, is\_hungry**), % closed for particular domain
- predicates (e.g. **has\_type, is\_nearby**), % closed for particular domain
- operators (e.g. **plans\_encounter\_to, undo**), % closed for particular domain
- 'generated facts' (e.g. **is\_being\_pursued\_by**). % open for particular domain

The 'generated facts' are not a new type, they are predicates that are not pre-defined but are generated by rules. They are stored in the working memory temporarily and are used to support future testing of the conditions of other rules. For example, **is\_being\_pursued\_by** can be generated as a new fact by a rule such as **rule3**:

```

rule3: if    [object1, has_type, PRAWN,           % conditions
             and, object2, has_type, JELLYFISH,
             and, object1, is_nearby, object2]
then [object1, is_being_pursued_by, object2]. % action
    
```

If the above conditions are met in the database, and **object1** and **object2** are instantiated successfully, then the action would generate a new fact (number is incremental) and insert it into the WM, for example:

```
newfact( 10) : is_being_pursued( prawn1, jelly1).
```

This 'generated fact' will be used in the subsequent testing of the rulebase for other rules. It can be explicitly deleted by another rule using the operator **undo** (§ 2.4.2), or it will be removed automatically at the end of each animation run, as it will have no purposeful meaning in a different run.

Some commonly used system predicates and operators such as **has\_type**, **is\_nearby** and **plans\_encounter\_to** are made available within the inference engine, while other special tasks which are more domain-specific can be defined by the domain expert. These will be covered in the following sub-sections.

The DCG language interpreter also allows for a degree of reformatting to take place so it is ideally suited to build more English-like "front-ends" that allow users to enter rules more naturally, while syntax checking them and converting them to a preferred format for internal operation.

Allowing unrestricted natural language input is currently not feasible. A more formal language has been developed and adopted within which rules are expressed to the system as follows:

```
if [object1, is_nearby, object2, and, object2, is_static] then [object1, meanders].
```

In an ideal version of the system users would be able to express their rules using an interface in natural language as:

```
if object1 is nearby an object2 which is static, then object1 can meander.
```

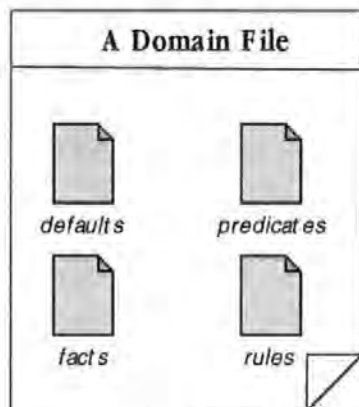
It requires that new keywords such as 'is', 'a', 'an', 'the' and 'can' be implemented, and that the language parser be modified so that the meanings of these new keywords are handled properly (see **Appendix B.3** for what is involved).

Looking to the future, it is possible that a more sophisticated language for rules may be developed and in this case any new grammar written in DCG could be incorporated without affecting other parts of the system (except the Rules Interface dialogue box as it will have to be redesigned).

However, the current implementation of the grammar which involves considerable use of arguments to check context-sensitive features and is able to extract useful data structures, is sufficient for the purpose of this project. Since this sub-section only deals with the expression and the style of rules writing, § 2.4 **Language Processing of Grammar Rules** gives a more comprehensive view on how the language processing is conducted.

### 2.2.3 Domain Database

A Domain file (Figure 2-3) consists of the descriptions of the database and the objects within it. From within the general system environment, different domains can be opened one at a time.



**Figure 2-3** A Domain File

A Domain file is made up of four sub-files: *defaults*, *facts*, *rules*, *predicates* (order is insignificant). The reason for dividing these into different sub-files is because some data are usually more stable and seldom change (*defaults* and *predicates*), whilst others like *Rules* and *Facts* are inclined to be modified and updated constantly by the

system or the operator. The following information or databases are vital for the system to run:

*defaults:* contains resources for the graphics representations and default properties of objects, as described in § 2.2.1:

```
%% Default database graphics representations
defaults picture( 'JELLYFISH', resource( 200)).
defaults picture( 'PRAWN', resource( 210)).

%% Default database for Object properties
defaults types( ['JELLYFISH', 'PRAWN']).

defaults name( 'JELLYFISH', jelly).
defaults mass( 'JELLYFISH', 5).
defaults size( 'JELLYFISH', 16).
defaults vis( 'JELLYFISH', 150).
defaults maxSpeed( 'JELLYFISH', 10).
defaults property( 'JELLYFISH', [alive]).

defaults name( 'PRAWN', prawn).
defaults mass( 'PRAWN', 1).
defaults size( 'PRAWN', 3).
defaults vis( 'PRAWN', 50).
defaults maxSpeed( 'PRAWN', 15).
defaults property( 'PRAWN', [alive]).
```

*facts:* Facts can be added when a problem is set up, or automatically generated by the inference engine, some as described in § 2.2.1:

- instances of a type: `obj_type(<object>, <type>)`  
**fact obj\_type( jelly1, 'JELLYFISH').**
- attributes of a type: `property(<object>, <property list>)`  
**fact property( jelly1, [alive, hungry]).**
- location of an object: `locn(<object>, <time>, <point>)`  
**fact location( jelly1, pt(65, 49)).**

```

fact obj_type( jelly1, 'JELLYFISH').
fact property( jelly1, [alive]).
fact location( jelly1, pt( 65, 49)).

fact obj_type( prawn1, 'PRAWN').
fact property( prawn1, [alive]).
fact location( prawn1, pt( 134, 96)).

fact obj_type( prawn2, 'PRAWN').
fact property( prawn2, [alive]).
fact location( prawn2, pt( 100, 126)).

fact obj_type( rock1, 'ROCK').
fact property( rock1, [static]).
fact location( rock1, pt( 147, 173)).

```

*rules:* contains domain specific Rules for the domain, here are two rule examples:

```

%% When an object is near a static object, then meander
rule1 :
if    [object1, is_nearby, object2]
then [object1, meanders].

%% When an object is_caught_by another object
rule2 :
if    [object1, is_caught_by, object2]
then [object1, dies].

```

*predicates:* Domain specific predicates and operators maybe written by the domain expert here.

```

%% Here's an example for changing different states
%% during driving

change_state( Obj, Old_state, New_state) :-
    get_fact( property( Obj, [Turn, Old_state, Dir])),
    delete( Old_state),
    set_fact( property( Obj, [Turn, New_state, Dir])).

```

The domain predicates are written in Prolog. The necessity for doing so is discussed in § 2.5: Rulebase & Rule Interface.

**Creating New Domains.** When the **New...** command (§ 2.3.3) is selected from the menu, the system creates a new domain file with four windows in it, namely *Facts*, *Rules*, *Defaults* and *Predicates*. There is no logical order of these files, they coexist with each other. These files are initially empty. When writing new rules using the Rule Editor, the rules will be placed in the *Rules* window, and when new objects are created using the New object tool, they will be placed in the appropriate *Facts* and *Defaults* windows.

**Maintenance** of the domain files is simple. After any changes have been made the user can update the files by saving the current settings using the **Save** command, or using the **Save as...** command to save a copy of the same file with a different name. Both are available as menu commands (see § 2.3.3).

### 2.3 User Interfaces

In recent years, human-machine communication has been the focus of a number of computer animation researchers (Blumenthal 1990, Douglas *et al.* 1990, Laurel 1990). For them the main purpose of the user interface is to make the communication easier between the two parties. Hence the implementation of the user interface is a very important part of any system design. Good interface design is the key to communicating between the user and the machine successfully, so it is important that the interface is simple to use and quickly understood.

There are two main forms of interface in the ESCAPE system: a graphical and a menu-driven interface. The graphical interface consists of windows, graphics, icons and texts. Some lesser used tools, commands and system settings, are placed under the menus and will be called or set upon selection, so the screen is not constantly congested with tools and icons.

Windows and icons are amongst the strongest features within the Macintosh OS: this is exploited by taking advantage of these features in the system. The use of dialog boxes, buttons, and all the standard interface techniques thus make the system very 'Mac-like', any experienced Mac-user should find familiarity in the environment as any other Mac programs.

### 2.3.1 Graphical Interfaces

The Graphical Interface is the first main window that the animator sees and is used for the design of animations. It consists of a window showing the current state of the animation. It uses MacProlog Graphics, which provides an easy way of dealing indirectly with the standard Macintosh Graphics Routines. In the prototype a 2D graphical representation of objects is used as shown in Figure 2-4.



Figure 2-4 The Animation Graphical Interface

The above figure shows in close-up the graphical representation of a domain called "World.Jelly 1.4" (domain name shown in the title-bar of the window), which consists of a jellyfish, a prawn and a rock. The window is a view on a larger world (set by `world_size`, see Figure 2-4). If no `world_size` is specified, then it takes the full range of MacProlog's coordinate system (see **Appendix B.4**). On the bottom left corner, the animator can see a dotted rectangle which represents a miniature screen section of what is currently showing on the main window. To reposition the present view, the user can either drag this box over to the desired area, or by using the scroll bars provided situated on the right and bottom (up/down and left/right respectively) on the animation window. The domain definition needed to show the particular scene in Figure 2-4 can be seen in Figure 2-5:



```
% for picture representations
defaults picture( 'PRAWN', resource( 210)).
defaults picture( 'JELLYFISH', resource( 200)).
defaults picture( 'ROCK', resource( 220)).

% world size
fact world_size( (0,0), (400, 300)).

% for defining object types
fact obj_types( ['JELLYFISH', 'PRAWN', 'ROCK']).
fact obj_type( jelly1, 'JELLYFISH').
fact obj_type( prawn1, 'PRAWN').
fact obj_type( rock1, 'ROCK').

% for defining objects' properties and locations
fact property( jelly1, [alive]).
fact location( jelly1, pt( 65, 49)).

fact property( prawn1, [alive]).
fact location( prawn1, pt( 134, 96)).

fact property( rock1, [static]).
fact location( rock1, pt( 147, 173)).
```

Figure 2-5 Domain Definition for Figure 2-4.

There are eight tools shown on the left hand side which can be considered as two sets of four. The top four tools are for changing the status of objects in the database:



**Selection Tool.** This is used for selecting and directly manipulating objects on the screen, so the user can perform click-and-drag operation to move the objects.



**Information Tool,** displays an information window which allows for updating the database by changing symbolic information about an object, see Figure 2-8.



**New object Tool,** is used to create new objects to the domain (by invoking a dialog) or to add an instance of any object currently appearing in the domain, see Figure 2-9.






**Delete Tool,** to delete an object. Having selected this tool, double clicking on the picture representation of the object deletes it from the database.

The lower four tools are for controlling the use of the expert system:



**Reset Tool.** This checks the default domain database, and resets all the objects to their original states. This also clears any previously stored database so the animator can start from fresh.



-  **Run Tool**, to invoke the inference engine and create an animation. Two options are available: Interactive Run and Blind Run (see § 2.3.3 Menus for more details).
-  **Playback Single**, to view the playback of the animation in Single mode (only the current frame is shown). See Figure 2-6.
-  **Playback Multi**, to view the playback of the animation in Multiple mode (the trace of each object is shown). See Figure 2-7.

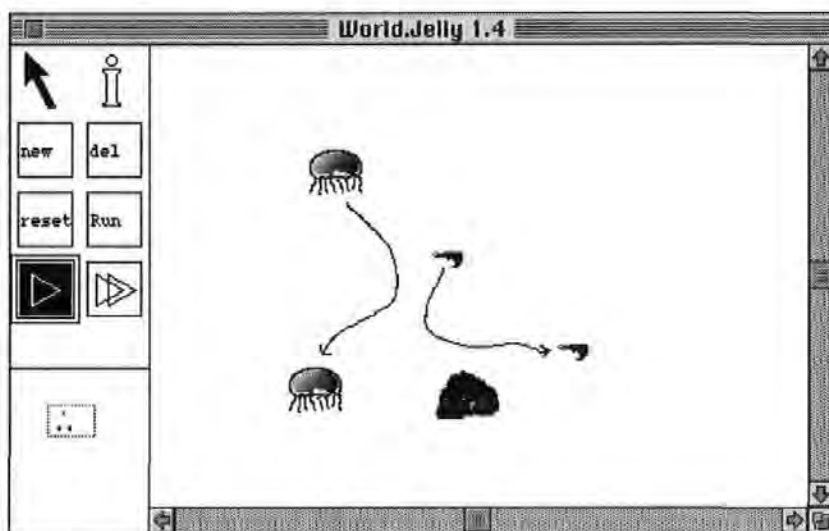


Figure 2-6 Playback: Single with Arrows\*

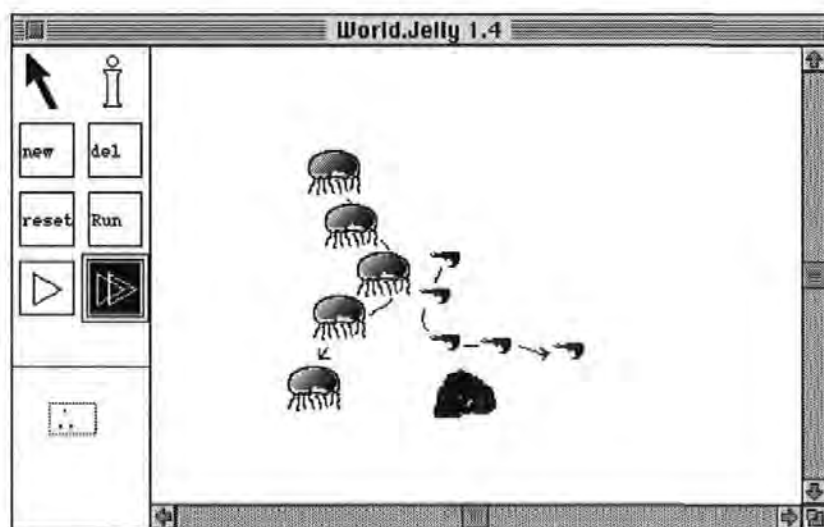



Figure 2-7 Playback: Multi with Arrows\*

---

\* Arrows have been added to the diagram by the author to show the direction the objects are moving.

## 2.3.2 Object Interfaces

By invoking the Object Information tool , the user will be presented with an interface which shows details on the selected object such as its name, and its associated attributes and values, as shown in Figure 2-8:

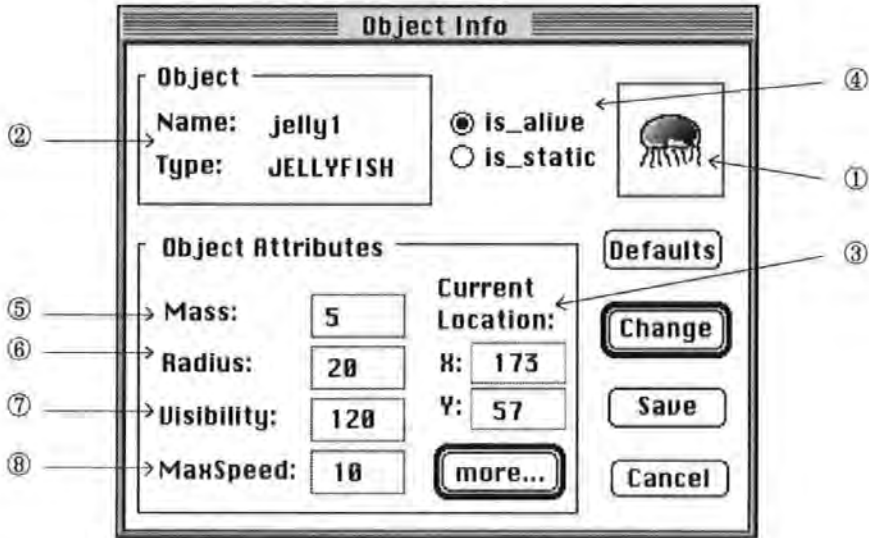


Figure 2-8: The Get Object Information Interface

This information is stored in the database (§ 2.2.2), which has a readable English-like format. Here, the corresponding parts are labelled by numbers, and the database representing the above figure is given as follows:


- ① `picture( 'JELLYFISH', left, resource( 200)).`
- ② `fact obj_type( jelly1, 'JELLYFISH').`
- ③ `fact location( jelly1, pt( 173, 57)).`
- ④ `fact property( jelly1, [alive]).`
- ⑤ `defaults mass( 'JELLYFISH', 5).`
- ⑥ `defaults size( 'JELLYFISH', 20).`
- ⑦ `defaults visibility( 'JELLYFISH', 120).`
- ⑧ `defaults maxSpeed( 'JELLYFISH', 10).`

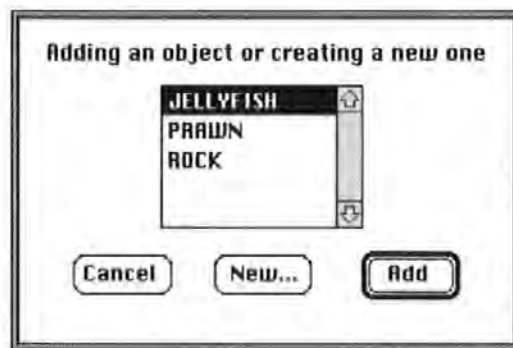
By clicking on the **Defaults** button, the selected object is set back to its original default settings (all properties preceded by the word *defaults*). **Change** will update the database of the object to the currently set values in the fields provided, which the user

can type in via the keyboard; this however, is not a permanent change, because it only updates the database in memory and not the domain file. To make sure that the data is permanently saved to the current domain database, the user invokes the **Save** button. The **Cancel** button closes the interface window and discards any changes made after the last use of the **Save** or **Change** button.

Note that the Get Object Information dialog box (Figure 2-8) displays only the minimum attribute requirements for a 'living' object. The current selection of attributes are adequate for the purposes of this prototype system; however, for more complex object behaviour, additional attributes will have to be accounted for and this can be amended through an additional dialog window by activating the **more...** button.

Since all objects are categorised as either 'alive' or 'static', the properties represented in the Get Object Information dialog can be updated dynamically by the system depending on which category an object belongs to. For example, if an object **is\_static**, the **visibility** and **maxSpeed** attributes will have NIL values, and the edit fields will be disabled so the user cannot make any changes to them.

The addition of a new object to the current domain can be initiated by invoking the  (New Object) tool button and via a dialog box as presented in Figure 2-9:



**Figure 2-9** Adding New Object Dialog box

A desired object type can be selected from a scroll list, activating the **Add** button adds a new object of the selected type to the database, and **Cancel** exits to the previous level. The selection provided in the scroll list is obtained by the system from the database by matching:

```
fact obj_types( X)
```

to

```
fact obj_types( ['JELLYFISH', 'PRAWN', 'ROCK']).
```

and hence instantiating **X** to ['JELLYFISH', 'PRAWN', 'ROCK']).

When a domain is loaded for the first time, the system keeps a record of the number of objects of the same type, so the next object can be incremented accordingly to its type. For example, if initially there were 3 jellyfish (**jellyfish1**, **jellyfish2** and **jellyfish3**), then a new jellyfish will be called **jellyfish4**. Looking back at Figure 2-8, note that the object's name is deliberately made so that it cannot be modified by the user, and this is the reason why. Every effort is made to ensure that the database is kept as simple as possible. By not having the ability to change the object's name by the user, unnecessary code for handling uncommon object names can be omitted. The name of an object can then be used as privileged information by which the system uniquely identifies its objects (a 'primary key' in database terminology).

However, if a new type of object is required which is not already in the types list **X**, clicking on the **New...** button will initiate a further window (see Figure 2-10). For example, if a new type '**FISH**' is to be added to the database, the interface will allow the user to set up its default settings, such as its name (e.g. **fish**), type (e.g. '**FISH**'), size and so on:

Figure 2-10 Creating New Object Type Dialog box

As in the 'Get Object Information dialog box' (Figure 2-8), this dialog displays only the minimum attribute requirements for a 'living' object, additional attributes can be amended through an additional dialog window activated by the **more...** button.

The user is able to import a PICT format picture using the **Get PICT** button for the graphical representation of the newly described object. Once the desired settings/values are set, the user can click on the **Save** button to save it to the database. In the database, the system appends the new type '**FISH**' to the existing **X** list, so the matching of

**fact obj\_types( X)**

will now instantiate **X** to ['**FISH**', '**JELLYFISH**', '**PRAWN**', '**ROCK**']).

The user can then re-select another set of default settings for a new type and save to the database. Several new types can be created in such a way, until the user clicks on the **Done** button to exit the dialog. Clicking the **Cancel** button exits the dialog without saving to the database. Exiting the New Object Type Dialog box brings the user back to the New Object dialog window as shown in Figure 2-9.

In the current implementation of the system, the file which contains the PICT resources have to be in the same directory (location on the disk) as MacProlog. This is the only way that MacProlog can access the resources.

### 2.3.3 Menus

In addition to the tools mentioned earlier, there are menu options for system settings and extra external controls, see Figure 2-11.

The menus can be divided into two categories:

- 1) **System** : main operations for the system, various system settings and preferences.
- 2) **Attributes** : for maintaining, editing, inputs and outputs of the domain attributes.

Under the **System** menus, there are:

- **Solve One Highest Level** : option for 'One Highest Level'. See § 3.5.1.
- **Solve All Highest Level** : option for 'All Highest Level'. See § 3.5.2.

System	Attributes
Solve One Highest Level	New World...
✓ Solve All Highest Level	Open World...
Others...	Close World
✓ Interactive Run	Save
Blind Run	Save as...
Collision Detection is On	Save Swivel Script...
Label On	Save Playback as...
Output Text is off	Open Playback...
Reset Animation	Rule Editor™...
Run Animation	Facts Window
	Rules Window
	Defaults Window
	Predicates Window

Figure 2-11 Menus for External Controls

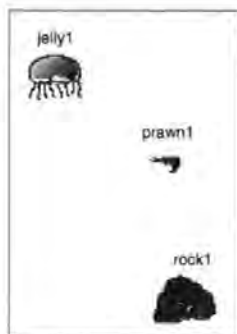
- **Others...** : option for 'Others...'. See § 3.5.3.
- **Interactive Run** : Screen is updated as every frame is resolved, so an instant preview of the animation can be viewed. This slows the running-time of the program.
- **Blind Run** : The screen is dormant while the frames are being resolved by the inference engine. The locations of objects are remembered in buffers. This improves the run-time of the program as the graphics representations of the objects are not updated continually.
- **Collision Detection on/off** : to switch Collision Detection on/off. (see § 2.7).
- **Label on/off** : to switch labels (names) on/off for the objects.
- **Output Text on/off** : to switch the output texts on/off in the log window.
- **Reset Animation** : same function as the 'Reset' Tool.
- **Run Animation** : same function as the 'Run' Tool.

Under the **Attributes** menus, there are:

- **New World...** : to create a new domain file.

- **Open World...** : to load an existing domain file.
- **Close World** : to close the currently loaded domain file.
- **Save** : to save the currently loaded domain file.
- **Save as...** : to save a copy of the currently loaded domain file under a different name.
- **Save Swivel Script...** : to output the current animation in Swivel 3D™ scripts.
- **Save Playback as...** : to save the current animation as a playback file (Prolog format), to be played back in future.
- **Open Playback...** : to open a previously saved playback file.
- **Rules Editor...** : initiates the "Rules Editor" dialog. See § 2.5.3
- **Facts Window** : bring to front the 'Facts' window.
- **Rules Window** : bring to front the 'Rules' window.
- **Defaults Window** : bring to front the 'Defaults' window.
- **Predicates Window** : bring to front the 'Predicates' window.

The **Label on/off** operation is useful if the animator wishes to know or trace the objects on the screen. By turning this option on, the names indicating the objects will be displayed just above them (see Figure 2-12).



**Figure 2-12** Objects with Label On



The **Output Text on/off** operation does not aid the production of the animation itself. Turning it on, the user can have a detailed output log of the running of the animation in a log window, which can be extremely useful for the animator to trace each decision step of the inference engine. With this option on, detailed insight into the operation of the system can be obtained, but at the cost of processing speed for producing the animation. See Figure 2-13 for an example log of an animation run. From the log, the first fact generated by the inference engine was

**New fact: 15 plan\_next 1 to 3**

and note that **New Fact** starts from 15. This is because the first 14 were generated during the initialisation process prior to the animation run (they hold information of all the objects in the domain such as their locations, attributes and so on). **plan\_next 1 to 3** states that the 'keyframe' to be planned next is 1, and the last 'keyframe' to be planned is 3 (a total of 3 to be done). This is only the outcome of the first pass generated by the inference engine.

New fact: 15 plan\_next 1 to 3 by main inference...

**\*\* Pass = 1**

Deleting fact: 15 : plan\_next 1 to 3 by main inference...

New fact: 16 plan\_next 2 to 3 by main inference...

**\*\* Pass = 2**

New fact: 17 poss\_locn(jelly1, 2, 5, pt(73, 60)) by rule4

New fact: 18 poss\_locn(jelly1, 2, 7, pt(70, 28)) by rule5

New fact: 19 poss\_locn(prawn1, 2, 7, pt(126,112)) by rule5

New fact: 20 poss\_locn(rock1, 2, 1, pt(147, 173)) by rule6

Level=5 New fact: 21 locn(jelly1, 2, pt(73, 60)) by resolving...

Level=7 New fact: 22 locn(prawn1, 2, pt(126, 112)) by resolving...

Level=1 New fact: 23 locn(rock1, 2, pt(147, 173)) by resolving...

Deleting fact: 16 : plan\_next 2 to 3 by main inference...

New fact: 24 plan\_next 3 to 3 by main inference...

**\*\* Pass = 3**

New fact: 25 poss\_locn(jelly1, 3, 5, pt(88, 65)) by rule4

New fact: 26 poss\_locn(jelly1, 3, 7, pt(58, 66)) by rule5

New fact: 27 poss\_locn(prawn1, 3, 7, pt(144,110)) by rule5

New fact: 28 poss\_locn(rock1, 3, 1, pt(147, 173)) by rule6

Level=5 New fact: 29 locn(jelly1, 3, pt(88, 65)) by resolving...

Level=7 New fact: 30 locn(prawn1, 3, pt(144,110)) by resolving...

Level=1 New fact: 31 locn(rock1, 3, pt(147, 173)) by resolving...

Deleting fact: 24 : plan\_next 3 to 3 by main inference...

**End of Execution.**

Figure 2-13 Output Log with 'Text On'



During the second pass, **fact 15** is deleted, and

**New fact: 16 plan\_next 2 to 3**

is generated by the inference engine, for planning the next keyframe, 2. The inference engine goes on doing this for each pass until the last keyframe is planned, then it stops.

During inferencing, the rule that was fired to generate a new fact is printed after it. For example,

**New fact: 17 poss\_locn(jelly1, 2, 5, pt(73, 60)) by rule4**

which states that **New fact 17** is generated by **rule4**.

**by resolving...** means that this particular fact is generated by solving several possible solutions/facts suggested by various other rules. This will be covered in more detail in § 3.6: **Results**.

Using the **Output Text Off** option, the user does not have a detailed log. However, the running of the animation will be much faster. A sample of the output when this option is turned off, see Figure 2-14:

```
*
** Pass = 1
**
** Pass = 2
.....
** Pass = 3
.....
End of Execution.
```

**Figure 2-14** Output Log with 'Text Off'

The dots signify that a line of output on the log file would have been generated by the inferencing (if Output Text were turned on).

On the current equipment used, the inference engine does not run in real time and the production of graphical output in conjunction with inferencing slows down its operation further. For this reason it is often advantageous to switch off the graphical display (use **Blind Run** mode and/or **Output Text Off** mode) while generating an animation and, when it has been completed, play it back in **Single** or **Multi** format. See Figure 3-18, Figure 4-7 and Figure 4-12 for time comparison graphs.

### 2.3.4 Help

Online help is available when switching on the standard *Balloon Help*, and the cursor moved over any buttons, control items and user items. Also, by double clicking the *Tool Icons*, a help window will be activated.

## 2.4 Language Processing of Grammar Rules

This section describes the syntax and semantics of the language processing of the grammar rules used in this expert system.

As mentioned in § 2.2.2, the expression of rules are defined using a Definite Clause Grammar (DCG). This makes it very easy to implement formal grammars in Prolog. A grammar stated in DCG is directly executable by Prolog as a syntax analyser.

### 2.4.1 Syntax

**syntax** *n.* 1. the branch of linguistics that deals with the grammatical arrangement of words and morphemes in sentences. 2. the totality of facts about the grammatical arrangement of words in a language. 3. a systematic statement of the rules governing the grammatical arrangement of words and morphemes in a language. 4. a systematic statement of the rules governing the properly formed formulas of a logical system. - (The Collins Concise 1988)

DCG definitions are a standard part of the Prolog language and thus can be expressed within the same programming language as the inference engine. Merely by defining the grammar in DCG form, a parser is automatically available and can be invoked by simply using the *parse* function or one of its variants. This makes it possible for users to directly enter rules as free text if they prefer, though a dialogue box interface to the rulebase is also provided.

As seen earlier, a rule is expressed in the "if-then" format:

**rule: if conditions then actions.**

and the basic syntax of a rule is described in Figure 2-15 (for a more complete listing of the implementation of the DCG see **Appendix C**, and the detailed parsing processes of the DCG see § 3.2). Since **conditions** and **actions** are handled the same way, they can both be represented as a **sentence**:

```

conditions  -> sentence.
actions     -> sentence.

sentence    -> simple_sentence, [and], sentence.
sentence    -> simple_sentence.

simple_sentence      -> noun_phrase, verb_phrase.

noun_phrase -> noun.
noun_phrase -> proper_noun.

verb_phrase -> trans_verb, noun_phrase.
verb_phrase -> intrans_verb.

noun      -> [Var],      { member( Var, Var_list) }.
noun      -> [Type],     { member( Type, Types_list) }.
proper_noun -> [Name],   { member( Name, Name_list) }.
trans_verb -> [Trans],   { member( Trans, Trans_list) }.
intrans_verb -> [Intrans], { member( Intrans, Intrans_list) }.
    
```

**Figure 2-15** Language Parser (DCG for rules)

A **sentence** can be made up of several **simple\_sentences**, or just one **simple\_sentence**. In the former case, **simple\_sentences** are separated by the symbol **[and]** like so:

**sentence -> simple\_sentence [and] simple\_sentence [and] ...**

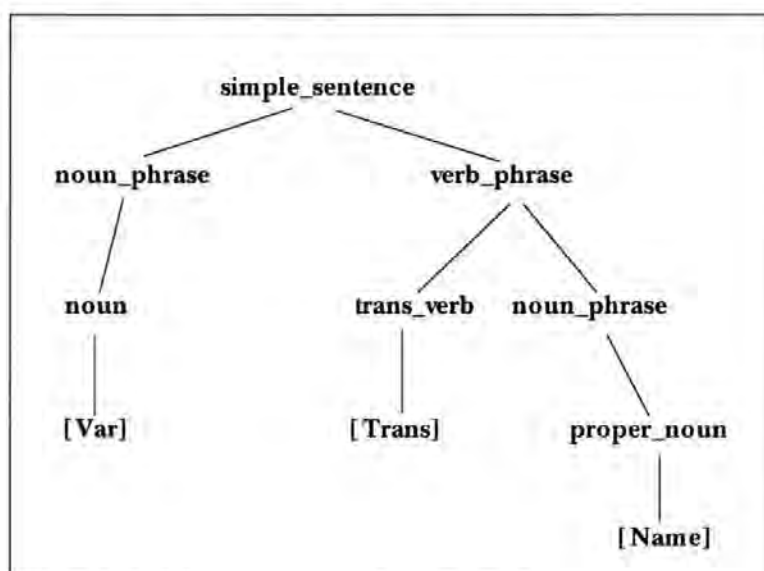
The square brackets ([]) in Prolog denote lists, they are not to be confused here in the DCG implementation as they indicate the terminal symbols. In a parser-tree, terminal symbols are the last nodes which do not branch further.

As an example, one possible **simple\_sentence** constructed from the DCG language parser mentioned above can be represented as a parse-tree in Figure 2-16. This tree allows for the structures of (left to right):

```

simple_sentence    ->  noun_phrase, verb_phrase.
=> simple_sentence ->  noun, trans_verb, noun_phrase.
=> simple_sentence ->  [Var], [Trans], proper_noun.
=> simple_sentence ->  [Var], [Trans], [Name].
    
```

(note: '=>' indicates each step taken for from the parser to reach the terminal symbols)



**Figure 2-16** Parse-tree

Since DCG definitions are a standard part of the Prolog language and thus can be expressed within the same programming language as the inference engine. Prolog clauses can also be used in the DCG implementation, they are included by the curly brackets ({}). (e.g. { **member( Var, Var\_list)** } - a formalisation of "**Var** is a **member** in the list **Var\_list**").

If referring to the underwater jellyfish world for example, the various lists can be obtained from the database:

```

Var_list    = [object1, object2, object3].      % variable list of 3 objects
Types_list  = ['JELLYFISH', 'PRAWN', 'ROCK'].   % type list
Name_list   = [jelly1, prawn1, rock1].          % name of actors in domain
% transitive predicates list, i.e. predicates having two parameters
Trans_list  = [is_nearby, has_type, plans_getaway_from, plans_encounter_to].
% intransitive predicates list, i.e. predicates having only one parameter
Intrans_list = [is_alive, is_static, meanders, dies, stays_there].

```

So, if we were trying to match

```
simple_sentence -> [Var], [Trans], [Name]
```

the syntax of

```
[object1, is_nearby, jelly1]
```

would be feasible, but

```
[object1, is_nearby, object2]
```

would not because **object2** is not a **member** of the **Name\_list**.

Here are a some valid examples written using the DCG rules in Figure 2-15:

```

rule1 : if [object1, is_nearby, object2, and, object2, is_static]
        then [object1, meanders].

rule2:  if [object1, has_type, 'PRAWN', and object2, has_type, 'JELLYFISH']
        then [object1, plans_getaway_from, object2].

rule3:  if [object1, is_static]
        then [object1, stays_there].

```

#### 2.4.2 Semantics

**semantics** *n.* 1. the branch of linguistics that deals with the study of meaning. 2. the study of the relationships between signs and symbols and what they represent.  
- (The Collins Concise 1988)

The semantics of the rulebase is handled by the *Language Interpreter & Translator* module (see Figure 2-19 Inference Engine Flow Diagram). In general, the task of the *Interpreter* is to take a rule and break it up into simple single conditions and

actions (left to right), then to match each condition against the database and generate any facts in the Working Memory as appropriate. A match occurs if an exactly equivalent item is found in the database, or if an equivalent item can be found by binding an unbound variable name to an object name. All conditions within the conditional clause must succeed in matching with the same bindings for the conditional clause as a whole to succeed. For example, if **rule1** is given as “if conditions then actions”:

```
rule1 :    if [object1, is_nearby, object2, and, object2, is_static]
           then [object1, meanders].
```

and assuming it is syntactically correct, then:

```
conditions = [object1, is_nearby, object2, and, object2, is_static].
actions    = [object1, meanders].
```

Let us now look at the **conditions** first. As mentioned earlier, **sentence** takes a complex sentence and breaks it up into simple single sentences called **simple\_sentences** which we will call conditional clauses or *clauses*. They are then passed to the translator, **translate**, so the variable instantiation process of these *clauses* can be carried out.

```
%% if there is more than one clause, then process into two parts
translate( A and B) :-      translate( A), translate( B).

%% if there is only one clause, then tests it.
translate( A)      :-      test( A).
```

```
e.g.,  translate( X is_nearby Y and X is_alive)    % two clauses

=>     translate( X is_nearby Y),                  % clauses 1
       translate(X is_alive)                        % clauses 2

=>     test( X is_nearby Y),                        % test separately
       test (X is_alive).
```

where **test** tests one *clause A* at a time against the current database to see whether:

- ① **A** matches an existing fact in the database (e.g. **X is\_alive**); or
- ② **A** is a calculation (e.g. **X is\_larger\_than Y**); or
- ③ there is no solution (i.e. no match)

If no solution ③ is encountered, the Language Parser will exit the current rule and proceed with the next rule; but if either a successful match ① or a calculated result ② is obtained, some of the variables in the *clause* may be instantiated.

e.g., `test( X is_alive)`                      % if there are two live objects in the domain...

solution No.1 => `X = jelly1`,  
 solution No.2 => `X = prawn1`

For the whole thing to succeed, the instantiations will have to be consistent with clause 1 and clause 2:

e.g., `test( X is_nearby Y), test( X is_alive)`

clause 1:                      `test( X is_nearby Y)`  
                     possible solution No.1 => `X = jelly1, Y = prawn1.`                      % possible match  
                     possible solution No.2 => `X = jelly1, Y = rock1.`                      % possible match  
                     possible solution No.3 => `X = rock1, Y = prawn1.`

clause 2:                      `test( X is_alive)`  
                     possible solution No.1 => `X = jelly1.`                      % possible match  
                     possible solution No.2 => `X = prawn1.`

by matching the possible solutions of the above clauses, there are two solutions:

clause 1 and clause 2: `test( X is_nearby Y), test( X is_alive)`    % matching  
                     solution No.1 => `X = jelly1, Y = prawn1.`  
                     solution No.2 => `X = jelly1, Y = rock1.`

The semantics of **Actions** is similar to that of **Conditions** in the way that a complex **sentence** is broken up into **simple\_sentences** (*clauses*), except that the conditional clauses are for testing purposes, whilst the action clauses are for executions. The connection between the two is that, after a successful test of a conditional clause, the instantiated variables can then be used to match the variables in the action clauses. For example,

**rule1 :**    **if** [`object1, is_alive`] **then** [`object1, meanders`].

**object1** in the conditional clause(`object1 is_alive`) will be matched and 'forced' to be the same as **object1** in the action clause (`object1 meanders`), so that the reference to the same object is consistent throughout the whole rule.

The execution an action (**do\_action**) may take several forms. The algorithm to take an action is based around the following sequence of tests:

- 1 if it contains an **undo** instruction, then the matching item will be removed from the database (no action is taken if the item is not in the database):

e.g., **do\_action( undo( plan( 1 to 16)))**  
 ⇒ **Deleting fact: 13 : plan (1 to 16).**

- 2 to do two actions (A and B), do A and then do B:

e.g., **do\_action( A and B)**  
 ⇒ **do\_action( A and B) :- do\_action( A), do\_action( B).**

- 3 if it contains a valid Prolog expression, then it will be passed to the Prolog system and interpreted:

e.g., **do\_action( TimeN is 4)**  
 ⇒ **TimeN = 4**

- 4 if it contains a predefined predicate or operator, then it will be resolved:

e.g., **do\_action( Distance between jelly1 & prawn1)**  
 ⇒ **Distance = 88.**

- 5 otherwise the item will be added to the database (no action is taken if the item is already present in the database),

e.g., **do\_action( poss\_locn(prawn1, 2, 4, pt(243, 164)))**  
 ⇒ **New fact: 22 poss\_locn(prawn1, 2, 4, pt(243, 164))**

Two detailed examples can be seen in § 3.2.

### 2.5 Rulebase & Rule Interface

The rulebase contains the behavioural rules for producing an animation. The first two parts of this section discuss the two levels of the rulebase: *standard*, which is available within the main system and to any domain that is currently running; and *domain specific*, which contains special behavioural rules for solving particular problems within the currently running domain.

These behavioural rules are represented in two formats that coexist within the system: standard Prolog, and “if-then” format (which will be referred to as *rules*). The



first two parts of this section also discusses the two formats in the designing of the rulebase:

1. is standard Prolog adequate?
2. when to implement in Prolog and when in *rules*?

The last part of this section discusses the design of the rule interface Rule Editor, which is aimed to provide an easy way for the user to write, amend and access the rulebase.

### 2.5.1 Standard Rulebase

The standard rulebase consists of a set of *predicates* which are commonly required yet are not readily available in the 'desirable form' (explained later) in standard Prolog, and some low-level *operators* for performing some commonly used tasks (*predicates* are often found in 'conditions' and *operators* are often used for carrying out 'actions'). These operators and predicates are defined by writing special rules, building them into the system, and providing them to every domain and making them relatively transparent. Users may therefore use such predicates as if they were primitives of the rule-writing language.

An example of a frequently used *predicate* that is not available in a 'desirable form' is `is_smaller_than`. Its apparent equivalent, the standard Prolog predicate '<', does not always behave as expected. The latter deals only with numeric arguments, whilst sometimes the need for comparing objects is desired. Ideally, the two-place predicate `is_smaller_than` should be able to deal with both numeric comparison and object comparison (for example, comparing their sizes). It should perform its functionality accordingly when called and distinguish which possibility to take. Here is the simplified code for `is_smaller_than`:

```
1: A is_smaller_than B :-    number( A),           % check if numbers
                           number( B),
                           A < B.

2: A is_smaller_than B :-    object( A),           % check if objects
                           object( B),
                           size( A) < size( B).    % compare sizes
```

By rewriting such a predicate to include test routines (or other additional features) embedded in Prolog to check for the arguments, different cases can be

handled separately and properly. Although Prolog provides a set of somewhat cumbersome functions and procedures for manipulating numbers, when used in such a way, it can be extended meaningfully to worlds of 2D and 3D objects. The *standard rulebase* deals mainly with the calculation or comparison of locations, distances, sizes, and object properties.

Here is a list of example predicates/operators that were written to include as standard in the system:

Predicates/operators with 1 parameter:

<b>X dies:</b>	changes object X's property from alive to static.
<b>X is_alive:</b>	true if object X is alive.
<b>X is_close_to_edge:</b>	true if object X is close to the boundary limits.
<b>X is_static:</b>	true if object X is static.
<b>X lives:</b>	changes object X's property from static to alive.
<b>X moves_within_boundary:</b>	moves object X within the provided fixed boundary.
<b>X stay_there:</b>	X's current location is same as its previous location.
<b>time_now_is X:</b>	returns the current frame X; useful when a specific task is needed to be carried out at a specific time.

Predicates/operators with 2 parameters:

<b>X collides_with Y:</b>	true if objects X and Y are in collision.
<b>X has_type Y:</b>	returns the object X's type as Y.
<b>X is_larger_than Y:</b>	true if X is larger than Y. Both either objects or numeric.
<b>X is_nearby Y:</b>	true if object X is nearby object Y.
<b>X is_not_same_as Y:</b>	true if X is not the same as Y. (* Both arguments must be of the same type - either object or numeric).
<b>X is_same_as Y:</b>	true if X is same as Y. (see *).
<b>X is_smaller_than Y:</b>	true if X is smaller than Y. (see *).

During the initial implementation stages of the system, all *predicates* and *operators* were written in Prolog as part of the building of the system itself. Later, some of them were withdrawn as Prolog code and rewritten using the implemented rule structures, but some remained in Prolog codes for several reasons. There are times

when the need to write rules in Prolog can prove to be advantageous, for example powerful Prolog controls such as backtracking and cuts can be used, when long and complex rules that require flexible controls are required, and also, at some point (e.g. the interpretation of DCG) everything must be expressed in Prolog.

The main advantage of rewriting the *predicates* and *operators* as *rules* is the ability to *explain*. This can be illustrated by comparing the two rule formats. If **rule1** is described as a rule:

```
rule1 : if      [object1, has_type, 'PRAWN',
               and, object2, has_type, 'JELLYFISH']
           then  [object1, is_food_of, object2].
```

equally, **is\_food\_of** can be expressed in Prolog as:

```
is_food_of( Object1, Object2) :-
    Object1 has_type 'PRAWN',
    Object2 has_type 'JELLYFISH'.
```

The resulting behaviour for using the latter is not quite satisfactory from the expert system point of view for explanation cannot be asked for; for example, *how* was the answer derived? In comparison, when the conditions of **rule1** are satisfied, the results/actions are stored dynamically in the memory as 'defined facts', and the *how* question can now be answered, for example "answer derived from rule1, which states that **PRAWN** type **is\_food\_of** **JELLYFISH** type" - currently this is automatically printed out to the log file. But when expressed in Prolog, this explanation is not available since it is either returned as 'succeed' or 'fail'.

However, at present, it is not possible to write all predicates and operators as *rules*, this is largely due to the design of the DCG. In order to do this, a more complex DCG will have to be redesigned. Currently, sufficient complexity has been implemented into the DCG, a sentence can distinguish three different variables (objects, properties and numbers), multiple conditions, multiple actions, unlimited number of user defined operators (memory permitting) and can have references to up to three objects at the same time (**object1**, **object2** and **object3**). If the syntax of the rules is to be made more complex (for example, allowing more primitive operators such as addition '+', output

‘print’ and so on), the design of the language parser will have to be able to cope with the demand. One way to do this is to allow standard Prolog clauses in DCG.

If the redesigning of the DCG is envisaged (for example, to accommodate four or more related objects in one sentence), one difficulty would be keeping track of instantiated objects, and this would require a *software expert* who is fluent in Prolog to modify the DCG to specification.

The above set of the *standard rulebase* is aimed to give the user some foundations for writing behavioural *rules* as described in § 2.2.2: **Expression of Rules**. It, by no means, is a complete set. However, in the likely event that more predicates or operators are needed, the *software expert* can include them in the *Domain Specific Rulebase*, which is described in next sub-section.

### 2.5.2 Domain Specific Rulebase

The standard rulebase is limited to perform generally useful tasks, so it is desirable for some special domain specific rules to be added. This is recommended, however, only for a *domain expert* as some Prolog experience is expected. The *domain expert* can write such special rules in standard Prolog code and place them in the *predicates* file within the domain file. For an example of the *predicates* file see § 2.2.3: **Domain Database**.

Such occasions may arise when for example, to satisfy all the three domains featured in this research (jellyfish, birds, and traffic), three different ways for advancing an object to the next location have been adopted (all have some element of randomness):

1. **meanders**, which calculates a random location in all directions for an object that it is applied to (as used in the jellyfish example);
2. **meanders\_forward**, which calculates the next location mainly in the direction the object is heading, but with some randomness of the object meandering sideways (as used in the flocking bird example);
3. **moves\_forward**, which calculates a random location mainly in the direction the object is heading, without too much randomness (as used in the traffic example);

After these rules have been written by the domain expert, they served their purposes in the particular domain and they have no further use for other domains. Hence they can be extracted and classified as *Domain Specific*.

2.5.3 The Rule Interface

The currently applicable domain rules are kept in a file called 'Rules', a part of the Domain file (see § 2.2.2: Expression of Rules). These *rules* can be written and amended through the use of a specially designed dialog box, see Figure 2-17.

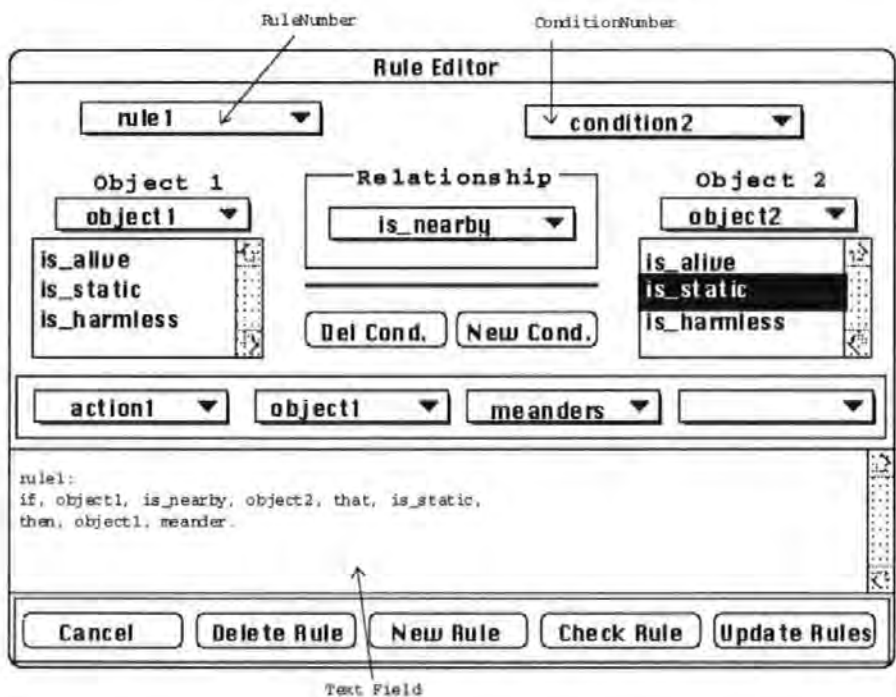


Figure 2-17 The Rule Editor Dialog Box

The dialog box can be used to create new rules and amend existing rules. If an existing rule number is selected then the dialogue box is automatically updated to reflect that particular rule. The following data corresponds to the above Rule Editor interface:

**rule1 :** if [object1, is\_nearby, object2, and, object2, is\_static] then [object1, meander].

RuleNumber = rule1,  
Condition1 = [object1, is\_nearby, object2],  
Condition2 = [object2, is\_static],  
Action1 = object1 meanders.  
Property Obj1 = none selected,  
Property Obj2 = [is\_static]

The Rule Editor dialog box is divided into two main sections. The top section reflects the grammatical structure of a rule. The pop-up menus provide all the terms that are valid at the particular place in the rule so that the animator is restricted to writing only valid rules. The menus also allow the animator to create new variable names if they wish to refer to an object not already known to the system. The lower part of the dialogue box contains a scrolling text field in which a text version of the currently selected rule is displayed. The user may choose to enter texts directly into this field by using input devices such as a mouse and a keyboard. Syntax errors may be introduced this way, though, which will only be detected when the system tries to parse the erroneous rule (which should report the error). Hence it is advised that the typing input is best handled by a domain expert, and rules are syntax checked by using the **Check Rule** button (see later).

For every *rule*, there is at least one *condition* and at least one *action*. When the **RuleNumber** Popup menu is depressed (upon mouse click down), a popup menu shows a selection of available rules from the Rules file within the domain database. When the button is released and a selection made, the *conditions* and the *actions* are then put into the **Text Field** and all parts of the dialog window will be updated. The steps are illustrated as follows:

```
when_click_on( ruleNumberPopup) :-
    get_item( ruleNumberPopup, RuleNumber), % get the selected RuleNumber
    RuleNumber : if Conditions then Actions, % get the Conditions and Actions
    update_text_field( Conditions, Actions), % updates text field
    update_all_menus. % update all other popups/menus
```

**update\_all\_menus** updates all the fields and menus in the dialog box using the corresponding selected rule, always starting with the first condition. There are two types of menu in the dialog box, 'popup' and 'click-select'. The 'popup' menus include **RuleNumber**, **conditionNumber**, **object1**, **object2**, **Actions**, and **Relationship**, the 'click-select' menus include **object1 Properties** and **object2 Properties**.

There has to be at least one object in a rule for it to be valid and, by default, this is set to **object1**. If there is no other object involved, the **Relationship** popup menu will be set to **None**, and **object2** and its properties menus are disabled.

Unlike **RuleNumber**, when selecting a **ConditionNumber** the **Text Field** is not updated because it already has the rule printed in its entirety, so only the popup and the click-select menus need to be updated.

The following describes the functionality of the buttons found in the dialog (see Figure 2-17):

- **Del. Cond. :** deletes the currently selected condition;
- **New Cond. :** creates a new condition, appended to the existing condition list;
- **Cancel :** allows the operator to quit without saving any changes made to the selected rule(s) after the last 'Update' (see **Update Rules** button);
- **Delete Rule :** deletes the currently selected rule;
- **New Rule :** creates a new rule, appended to the existing rules list;
- **Check Rule :** ensures that the screen information is consistent and parses the text version for syntax errors;
- **Update Rules :** updates the currently selected rule to the domain's Rules file.

Creating a new rule and a new condition are somewhat similar:

CREATING A NEW RULE	CREATING A NEW CONDITION
Increment rule number by 1, and	Increment condition number by 1, and
add 1 new rule + create new condition.	add 1 new condition.

Deleting a rule and a condition are somewhat similar too:

DELETING A RULE	DELETING A CONDITION
Decrement rule number by 1,	Decrement condition number by 1,
check: minimum one rule left,	check: minimum one condition left,
Delete Rule (incl. all conditions), purge, and go to first rule	Delete Condition, purge and go to first condition

When the **Check Rule** button is invoked, it carries out the following operations:

```
when_click_on( checkButton) :-  
    get_all_selected_items,           % get all selectedItems on the menus  
    check_syntax,                     % check if syntax is OK  
    /* if OK */ updates_textfield ;   % update Conds & Actions in TextField  
    /* else */ report_errors.         % else if not OK, reports any errors
```

After the syntax of a rule has been checked and if there is no error, it then can be added to the database by invoking the **Update Rules** button:

```
when_click_on( updateRulesButton) :-  
    check_if_rule_db_has_changed,     % check if database has changed...  
    add_to_rulebase.                  % if yes, then add it to the rulebase
```

### 2.6 Inference Engine

The Inference Engine is the heart of the system, the organiser of things, and is responsible for :

- i) *problem solving* - organising steps and domain knowledge (database) to construct a solution to a problem, and
- ii) *reasoning* - organising the computational process whereby needed information is inferred from what is known (such as those stored in the Working Memory).

The inference engine (see **Appendix A: Expert Systems** for references) uses the standard expert system approach of forward-chaining but has some additional features. The system allows for a customised resolution of all the actions that could be fired for any particular step and, when seeking to apply basic operators it will temporarily employ backward chaining.

The reason why forward-chaining is generally used, even in the case of directed animations where some goal is specified, is because it reflects a more open cognitive model. Backward chaining tends to reflect a reasoning process based upon a single argument structure, developed purely to give support to the derived conclusion. Forward chaining is better suited to the exploration of multiple possibilities, some of which may turn out to be ineffective. So by adopting this kind of reasoning process, it



does place an additional responsibility upon the modeller to specify how the system is to resolve multiple competing claims.

Let us look at a rule example which states:

**rule1 : if [object1, is\_nearby, object2] then [object1, meanders].**

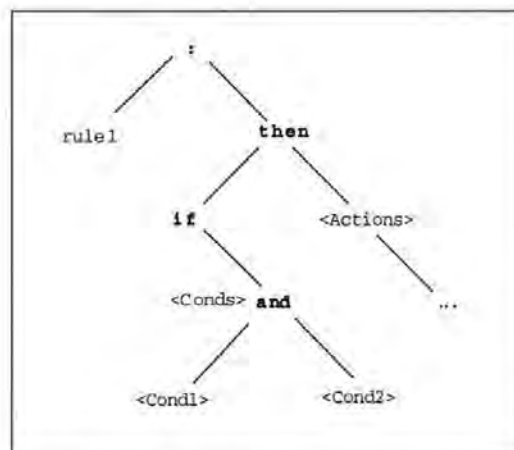
It consists of a conditional part: **[object1, is\_nearby, object2]**

and an action part: **[object1, meanders].**

There are keywords operators (such as 'if', 'then', ':') that enable the inference engine to recognise a rule structure. Some operators have higher precedence order (see **Appendix B.3** for more on precedence order of operators) than others, and they are defined in Prolog as follows:

```
%% special operators defined for the inference engine
:- op( 890, xfx, :).
:- op( 880, xfx, then).
:- op( 870, fx, if).
:- op( 540, xfy, and).
```

Let us look at this example: **rule1 : if <Conds> then <Actions>**. This can be viewed as follows:



**Figure 2-18** Precedence of Operators within a Rule

whereby, <Conds> and <Actions> can be a combination of ANDs, with sub elements <Conds1>, <Conds2> etc. Only if all the sub elements of the <Conds> succeed the testing routines of the Inference Engine, is the <Actions> part carried out.

The Inference Engine forms the main body of the system environment, which links to a number of entities within the system (see § 2.1), and is responsible for the productive output of the system. The *Language Interpreter & Translator* (§ 2.4) form the main testing modules within the Inference Engine. The following Figure 2-19 shows a flow diagram of the inference routines:

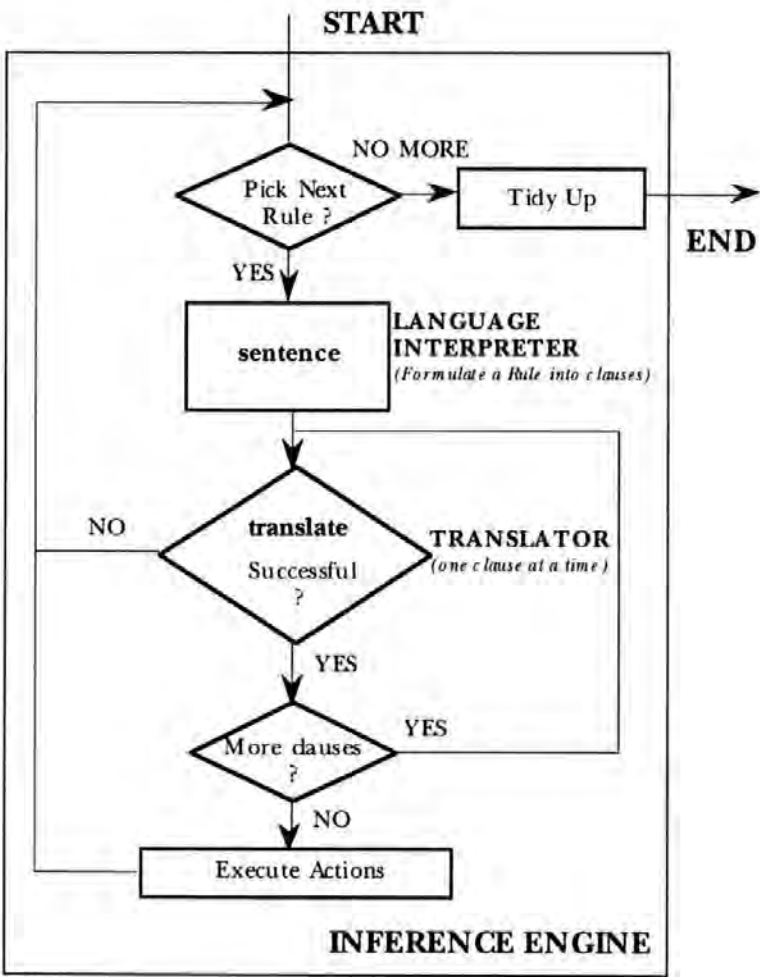


Figure 2-19 Inference Engine Flow Diagram

The inference routine starts by selecting a **RuleNumber** : **if Conds then Actions**, the **Conds** and **Actions** are handled separately from this point. First, the **Conds** are passed to **sentence** (the Language Interpreter) which transforms each condition into

executable Prolog clauses. The Prolog clauses are then passed, one at a time, to **translate** (the Translator) to be tested and instantiated. Instantiation *maps* all the variables of **Conds** against the database to satisfy the Prolog clauses, it is determined at this point whether this particular rule has any overall effect on the objects. If the rule is not satisfied, it fails and will go no further. The parsing steps can be seen in Figure 2-20:

```
infer :-  
  Rule: if Conds then Actions,           % get next Rule: Conds and Actions  
  sentence( X, Y, Z, M1, Conds),        % parse Conds  
  translate( M1),                       % instantiate X, Y, Z  
  sentence( X, Y, Z, M2, Actions),       % parse Actions with obtained X, Y, Z  
  do( M2).                              % execute the instantiated actions M2
```

Figure 2-20 Rules Parsing steps

The main inference routine repeatedly applies all rules to the database until there are no more changes to be made. In each pass, each rule is tested to see if its conditions can be met. If they can then actions are performed and the search continues for any other ways of satisfying the same conditions. Only when all possible ways of meeting the conditions have been explored does the system proceed to look at the next rule. One of the main features of Prolog is that it has the ability to backtrack. Upon failure, Prolog returns to the previous line and searches for the next solution and proceeds until all possible solutions are found. If the tests are successful and instantiations persist, it then continues downwards to executing the **Actions**. Note that the **translation** for **Actions** is not required since the variables are already instantiated by the **translation** of **Conds** (see Figure 2-20). The execution of an action can be seen in § 2.4.2: Semantics.

Stopping criteria are needed to determine when to terminate the problem-solving process. This is done by setting a goal, or several goals, terminating when one is reached. In cases where there is an initial state and a goal state we declare the whole planning process as a goal and introduce rules that try to decompose it into parts, repeating the process until each frame is decided. One example would be, to set up a plan of an animation consisting of X number of keyframes starting from 1 (or any number < X), which would be written as (in the database) **fact plan 1 to X**.

Another kind of goal would be to set up a stopping check, using a conditional expression, for example:

```
rule10:  if    [prawn1, is_captured_by, jelly1]
          then  [stop, animation]
```

In cases where there is no particular goal and the animation just has to run and be realistic, there are no special planning rules and domain rules are applied to each frame to generate the next frame. The process can be repeated indefinitely, though there is no guarantee that the set of rules will generate change from a particular frame or they will not generate a visually recognisable loop.

The inference engine explores every rule, matching it against the database and generating a single list of all possible actions. On completion of the list, it groups the possible actions for each object and passes them to a function which returns the next state of the object. There are many different strategies that could be adopted for this function: one possible action could be selected and the rest discarded (e.g. select the strongest, or the weakest), or a resultant of all the actions could be found, or a threshold could be introduced, resolving those that meet it. Different strategies can be adopted for different domains (see § 3.5: **Results** for how these strategies are used).

Two examples will be used to show how the Inference Engine behaves under different inferencing methods (see § 3.3).

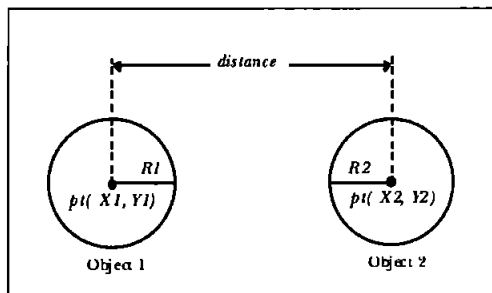
### 2.7 Collision Detection

For any object, detection of possible collision is a consequence of the urge to steer away from imminent impact with another and it plays an important part in creating realistic animation. Much research (Baraff 1990, Moore & Wilhelms 1988, Uchiki *et al.* 1983) has been carried out in the field of CAD/CAM and robotics for collision detection in highly complex scenes.

In ESCAPE, collision detection is a meta constraint on behaviour and has been built into the system using an unsophisticated detection algorithm (a circular bounding box), which is sufficiently visually convincing for the chosen examples (described in the following chapters).

The user can choose to turn collision detection on or off from the System menu as mentioned in § 2.3.3: **Menus**. By default, it is turned on when the system is first loaded. It is recommended to be left on at all time, but if for some reasons that it is not desirable, for testing purposes for example, the option to turn it off is there.

Collisions are detected if the circular bounding boxes of two immediate objects coincide. A static collision avoidance method is used, which is based on the relative position of the objects and ignores their velocity. This can simply be done by adding up the objects' individual sizes (using the radii), if the sum is greater than the distance between the two objects, then a collision is detected:



**Figure 2-21** Collision Detection Calculation

$$distance = \sqrt{(\sqrt{X1 - X2})^2 + (\sqrt{Y1 - Y2})^2}$$

$$Collision = distance < (R1 + R2)$$

or to avoid the expensive square-root calculation, this can be amended to:

$$collision = (\sqrt{X1 - X2})^2 + (\sqrt{Y1 - Y2})^2 < (R1 + R2)^2$$

When a collision is detected between the two objects (**Object1** and **Object2**), only one object is to be recalculated and moved. Which one depends on the following:

```
relocate_after_collision( Object1, Object2) :-
    Object1_is_alive,                % Object1 has to be alive
    ( Object2_is_alive, move( Object2); % move Object2 if it is alive
      move( Object1)),               % else move object1
    collision_detection( Object1, Object2). % redo detection
```

The new location is passed back to the collision detection routine again, to check for further collisions with that particular pair of objects. Unless the new location

is feasible, the next pair of objects are tested, until all possible pairings have been dealt with.

In this system, a new location will have to be proposed recursively until a non-collision is detected, which means that the overheads are going to be relatively high if objects are very close to each other. The recalculations in `move( Object )` are done using the same rule(s) that were applied to the objects initially, so a new location generally relies on the random elements of the operator(s) specified within the rule(s).

Since this system calculates keyframe output for Swivel 3D™, and the smoothing of the animation is calculated by the interpolation process within the Swivel 3D™ package, it is possible that objects will collide into each other between keyframes. An immediate solution to this is to implement a more sophisticated collision detection algorithm that checks the paths of objects between each keyframe, or to perform the calculations of all the in-between frames in Prolog. This would be a task for future implementation.

Other future work would be to include the implementation of another constraint, gravity. Environments using the laws of physics could be set up with different gravity values, such as zero for a free floating experience. An alien or surreal environment could be explored in such a way, and could enhance other disciplines of computer animation such as VR (Virtual Reality).

### 2.8 Inputs & Outputs

ESCAPE is capable of handling certain input and output files which can be seen in Figure 2-22.

The Domain file is solely used by the ESCAPE system itself, and is written and read in MacProlog format. The Playback file which is used to store animation information (e.g. the location of objects) for a particular run, can also be saved or opened from within the system in MacProlog format. Picture representation is read in PICT format, and the interface for accessing this can be seen in Figure 2-10 Creating New Object Type Dialog box.

DESCRIPTION:	FORMAT	INPUT	OUTPUT
Domain file: (Defaults, Facts, Rules, and Predicates).	MacProlog	✓	✓
Playback file	MacProlog	✓	✓
Picture representation	PICT	✓	✗
Swivel Script	Swivel 3D™ Script	✗	✓

Figure 2-22 Input and Output File Formats in ESCAPE

Swivel Scripts can be constructed and output by the system for use in conjunction with the Swivel 3D™ software for creating 3D colour animations. A Swivel Script can be seen in Figure 2-23, which shows an example of a three keyframes animation run of our WorldJelly. The output is written in Swivel 3D™ script format, comments are in brackets '()'.

Prior to running this script, a World (a Swivel 3D™ terminology for its workspace) will have to be set up. In the example shown in Figure 2-23, there are three objects (**jelly1**, **prawn1** and **rock1**) so, in the Swivel World, 3D models of the three object will have to be built and each appropriately named (**jelly1**, **prawn1** and **rock1**), for the script to work.

The script works by selecting one object at a time using **FindObject** (by its name), moving it to the **RelPos( X, Y, Z)** which usually follows on the next line of the script. After all objects have been moved, the whole scene is set by using **AddKeyFrame** (which adds the current scene as a keyframe). When all the keyframes are set, the in-betweening operation can be started automatically by including the script **Tween**. The number of in-between frames can be pre-set from within Swivel 3D™. In this example, only the objects' locations are amended, but other properties can also be changed in this way, for example size and colour.

```

( This is swivel script output from World: World.Jelly 1.2)
( Planning keyframes from 1 to 3 )

( This is KeyFrame = 1 )
FindObject jelly1          ( get the object jelly1...   )
RelPos: 133 88 0 ,        ( ...set its relative position )
FindObject prawn1         ( get the object prawn1...   )
RelPos: 123 188 0 ,      ( ...set its relative position )
FindObject rock1
RelPos: 150 220 0 ,
AddKeyFrame              ( add this as a keyframe...   )
DrawWorld                ( redraw and update World... )

( This is KeyFrame = 2 )
FindObject jelly1
RelPos: 131 113 0 ,
FindObject prawn1
RelPos: 127 167 0 ,
FindObject rock1
RelPos: 150 220 0 ,
AddKeyFrame              ( add this as a keyframe...   )
DrawWorld                ( redraw and update World... )

( This is KeyFrame = 3 )
FindObject jelly1
RelPos: 130 154 0 ,
FindObject prawn1
RelPos: 112 173 0 ,
FindObject rock1
RelPos: 150 220 0 ,
AddKeyFrame              ( add this as a keyframe...   )
DrawWorld                ( redraw and update World... )

Tween                    ( start in-betweening now... )

```

Figure 2-23 A Swivel 3D Script File\*

This produces a QuickPICS file which contains the complete series of frames for the animation. Since Swivel 3D™ does not have the facility to preview QuickPICS files, it is at this point that another piece of software is required to display the animation from QuickPICS. This is called PACo Producer. It can convert QuickPICS files into animations to be displayed on the screen. Various animation options can also be applied such as interlace (reduce tearing), scaling, changing background colour, looping the animation, adding sounds from external sources and so on.

---

\* Comments on the right had side of the script are added in for the purpose of explanation.



## 2.9 Summary

The software architecture for the incorporation of a rule-based system (cognitive approach) with a more traditional computer animation (physical approach) is described, establishing the needs and the roles of different level of users: animators, domain experts and software experts. Areas covered in this chapter include the data structures, the user interfaces, the language processing and grammar rules, the inference engine, and input/output formats.

The physical part of the production environment is adopted by a keyframing method, which is centred around the direct manipulation of objects on the screen and the recording of keyframes by means of an interactive control panel. The cognitive approach provides the facility to define behavioural rules of objects or classes of objects by means of using the user interfaces. The behavioural rules are expressed in natural language-like grammar and a parser is needed for handling the meanings of the rules.

Using these behavioural rules, the production system (an inference engine) can then be driven to suggest transition from one stage to another, the animator can thus use the system to automatically produce an initial animation. This animation can be further refined either by the direct manipulation of objects or by altering the set of rules and re-running the system. This prototype system cannot produce high quality output, but it does however allow for the output to be incorporated into a 3D modelling or rendering software, such as Swivel 3D™.

There is no straight forward method for implementing a general purpose system for the production of behavioural animation, but to learn from experiments. One important fact learned from the system is that it needs to contain a reasonable set of *standard* operators, predicates and vocabulary (for describing the grammar rules) so that different types of behaviour can be described based on the existing set. Some predicates may be of no expressive value to a certain world, for example the 'flock centring' behaviour of living animals has no real meaning for modelling a scene of car traffic (although the outcome maybe an interesting one!), so they can be separated out to be included in the *domain-specific* set.

CHAPTER 3

WORLD JELLY



"There is no particular mystery in animation... it's really very simple, and like anything that is simple, it is about the hardest thing in the world to do."

Bill Tytla.

## Chapter Overview

From the problems and difficulties in animating behaviour outlined in Chapter One, to the design of an animation environment which combines traditional animation techniques with an expert system processor in Chapter Two, we have seen how the components of a system such as ESCAPE can be implemented.

In this chapter, the viability of ESCAPE is demonstrated by showing its application to the world of underwater jellyfish and prawns (which is referred to as World.Jelly) to give the reader an insight of how a world is set up, and how animation is produced from it. This chapter is divided into two parts:

1. *Descriptions* (§ 3.1 - § 3.2): These sections describe and show how such a world is set up, how desired behaviours are defined in terms of rules that are comprehensible both to the user and the Inference Engine, and how the rules are translated and interpreted.
2. *Discussion and Programming Style* (§ 3.3 - § 3.5): These sections discuss the implications of using different inferencing control techniques within the Expert System, how output is generated, how transition from stage to stage is generated, and how objects should be ordered. Finally, in Results, the question of what to do with the solutions suggested by the Expert System are discussed.

### 3.1 World.Jelly Database

A fish-tank scene called World.Jelly is set up, in which there are three main types of object: 'JELLYFISH', 'PRAWN', and 'ROCK'. To simplify the animation, one agent of each type of object is used; they are named **jelly1**, **prawn1** and **rock1** respectively. The task is to construct a set of rules to enable the simulated agents to behave in such a way that they roam freely within the tank in any direction, seek food, avoid collisions, avoid danger and so on: static objects must stay where they are. The graphical interface of the World.Jelly scene can be seen in Figure 2-4 (p 34).

To set up the database for this world, four sub-files (*defaults*, *facts*, *rules* and *predicates*) for the domain have to be constructed as mentioned in § 2.2.3: **Domain Database**. Here, extracts from the database will be used to explain their significance in the following sub-sections. A full listing of the database for this domain can be seen in **Appendix D**.

#### 3.1.1 Defaults

Using the example above, there are three types of object and they can be defined in the database as:

```
defaults types( ['JELLYFISH', 'PRAWN', 'ROCK']).
```

and for each type of object, a generic **name** is given as:

```
defaults name( 'JELLYFISH', jelly).
defaults name( 'PRAWN',  prawn).
defaults name( 'ROCK',   rock).
```

**name** is not only beneficial for immediate recognition of an object and its type, but is also useful for creating a new objects by incrementing the number of the objects of the same type, for example, **jelly1**, **jelly2**, **jelly3** and so forth.

Other default properties/attributes needed to assign to the objects:

```
defaults property( 'JELLYFISH', [alive]).
defaults mass( 'JELLYFISH', 5).
defaults size( 'JELLYFISH', 21).
defaults vis( 'JELLYFISH', 150).
defaults maxSpeed( 'JELLYFISH', 40).

defaults property( 'PRAWN', [alive]).
defaults mass( 'PRAWN', 1).
defaults size( 'PRAWN', 4).
defaults vis( 'PRAWN', 80).
defaults maxSpeed( 'PRAWN', 40).

defaults property( 'ROCK', [static]).
defaults mass( 'ROCK', 10).
defaults size( 'ROCK', 25).
```

Note: the numeric values are just units, which do not represent imperial or metric entities in any way. World.Jelly is set in 2D.

The properties/attributes may be taken to have the following meanings:

- **property** sets the list of properties of an object, for example whether they are initially alive or static.
- **mass** is the mass of an object, it is used for dynamic calculations when the mass of an object is needed to estimate its speed, forces and so on.
- **size** is the size of an object, the value is its radius.
- **vis** is the visibility threshold beyond which distance the object *cannot see*.
- **maxSpeed** is the maximum speed an object can travel.

Note also that since objects with the type '**ROCK**' are static in nature, **vis** and **maxSpeed** are not required here. This part of the program depends on individual intuitions - common sense.

In order for the viewer to see these objects, picture representations are required to display them on the screen. These pictures are imported as PICT images using the object tool as mentioned in § 2.3: **User Interfaces**. The PICTs are stored as resources in a file called **Objs.PICT.rsrc** (one for all domains) which is opened by the system when it

is first run. These PICTs have their own unique ID, and the system needs to know which PICT is representing which object, so the first thing that needs to be done is to tell the system which ID represents which object:







<code>set_prop( 'JELLYFISH', left, resource( 200)).</code>	
<code>set_prop( 'JELLYFISH', right, resource( 201)).</code>	
<code>set_prop( 'PRAWN', left, resource( 210)).</code>	
<code>set_prop( 'PRAWN', right, resource( 211)).</code>	
<code>set_prop( 'ROCK', left, resource( 220)).</code>	
<code>set_prop( 'ROCK', right, resource( 221)).</code>	

Figure 3-1 Object Pictures for World.Jelly

`set_prop` (set property) is a standard MacProlog command for inserting a property into the working memory so it can be retrieved at any time (in § 2.2.1, `picture` is used instead which has the same feature). To make the preview a little more interesting, left/right facing pictures for each object type have been incorporated. Although 'ROCK' type does not move, a pleasing variation is to have two different looking rocks when setting up a scene.

### 3.1.2 Facts

After the defaults are set, *instances* (§ 2.2.1: **Objects & Agents**) can be introduced. The information about the instances are kept as **facts**, and they can be entered via the Get Object Information Interface as shown in Figure 2-8 (p. 41), or directly input into the Domain: Facts file by a domain expert.

In this example, the objects `jelly1`, `prawn1`, and `rock1` are declared as instances of some type:

```
fact obj_type( jelly1, 'JELLYFISH').
fact obj_type( prawn1, 'PRAWN').
fact obj_type( rock1, 'ROCK').
```

then addition properties and locations are given as follows:

```
fact property( jelly1, [alive]).
fact location( jelly1, pt( 65, 49)).

fact property( prawn1, [alive]).
fact location( prawn1, pt( 134, 96)).

fact property( rock1, [static]).
fact location( rock1, pt( 147, 173)).
```

When a new object is created, it inherits all the **defaults** settings of its chosen type (see page 76). For example, a further new object of type '**ROCK**' will be called **rock2**, and will have **mass** = 10, **size** = 25, and **property** = [static]. Note that **defaults** properties are not the same as **fact** properties, here is the difference:

```
defaults property( 'ROCK', [static]). % properties for type 'ROCK'

fact property( rock2, [static]).      % properties for object 'rock2'
```

The definition of the world and the objects is now complete. The domain is now ready to run once some rules are defined. A start and an end goal can be set to tell the system when to start and when to stop, this is done by including the following line:

```
fact plan 1 to 16.
```

which tells the inference engine to plan a sequence from 1 to 16 (16 time intervals), these time intervals will form the keyframe intervals after each successful iteration. If no special instructions are given to the system, it will run continuously (memory permitting) until interrupted by the user by holding down the 'command - . (period)' key combination.

### 3.1.3 Rules

The Rules are the driving force for the inference engine. The following are some the rules written to produce our World.Jelly animation (full listing available in **Appendix D**).

In this World.Jelly example, the behaviour of an object of type **PRAWN** will be followed closely. The relationships between objects of the type **PRAWN** and other

objects of the types **JELLYFISH** and **ROCK** in the domain are represented as rules (shown in Figure 3-2). The active objects are **prawn1**, **jelly1** and **rock1** respectively.

```
% objects of PRAWN type are 'food_of' JELLYFISH type
rule1 : if      [object1, has_type, 'PRAWN',
                and, object2, has_type, 'JELLYFISH']
            then [object1, is_food_of, object2].

% objects of JELLYFISH type are 'predator_of' PRAWN type
rule2 : if      [object1, has_type, 'PRAWN',
                and, object2, has_type, 'JELLYFISH']
            then [object2, is_predator_of, object1].

% ROCK type are 'shelters_for' PRAWN type if they are near
rule3 : if      [object1, is_nearby, object2,
                and, object1, has_type, 'PRAWN',
                and, object2, has_type, 'ROCK']
            then [object2, is_shelter_of, object1].

% if an object is_nearby 'a_predator' (rule2),
% then object has property 'is_escaping_from' predator
rule6 : if      [object1, is_nearby, object2,
                and, object2, is_predator_of, object1]
            then [object1, is_escaping_from, object2].

% if an object 'is_escaping' (rule6) and a_shelter (rule3) is nearby,
% then moves towards it
rule7: if      [object1, is_escaping_from, object2,
                and, object3, is_shelter_of, object1,
                and, object1, is_nearby, object3]
            then [object1, plans_encounter_to, object3].

% if object 'is_escaping' (rule6) from 'a_predator',
% then it tries to get away from predator
rule8 : if      [object1, is_escaping_from, object2]
            then [object1, plans_getaway_from, object2].

% if an object is alive, then meanders
rule9 : if      [object1, is_alive]
            then [object1, meanders].
```

**Figure 3-2** Some Rules from World.Jelly

In Figure 3-2, the user generated facts (such as 'is\_food\_of', 'is\_predator\_of' and etc.) are shown in plain-style (non-bold) so they can be distinguished from the standard



keywords, predicates and operators (in **bold**). These user generated facts can be used in subsequent rules, combining with other generated facts or any of the standard predicates/operators to create new facts. There is no limit to how many facts can be generated in such a way.

### 3.1.4 Predicates

*Predicates* includes clauses written in Prolog for carrying out domain-specific low level tasks such as movements. These tasks include predicates and operators, and they need to be "labelled" at the top level so that they exist for the inference engine prior to executing the codes (as this is essential for the language interpreter to handle the syntax properly). The "labelling" process is accomplished by giving a precedence order (§ 2.6) to each operator and predicate and then dividing them into two lists: **domainActionList** and **domainPredicateList**. If there is no defined domain predicate or operator, an empty list ([]) is given.

In this domain, there are three domain specific actions being introduced: **meanders** - for an object to move or meander in all directions; **plans\_encounter\_to** - for an object to plan an encounter with a chosen object; **plans\_getaway\_from** - for an object to plan an escape from another object. The "labelling" process is demonstrated as follows:

```
%% operator precedence definitions
:- op( 220, xf, meanders).
:- op( 220, xfx, plans_encounter_to).
:- op( 220, xfx, plans_getaway_from).

%% domain defined lists
domainPredicateList( []).      % no defined domain predicate
domainActionList( [ meanders, plans_encounter_to, plans_getaway_from]).
```

Since this domain is a model of a underwater scene, living objects can freely move in any direction (360° freedom in 2D world), **meanders** is the means which an object advances to the next location. The transition of **meanders** is carried out by obtaining two random numbers (**X** and **Y**) based on the **maxSpeed** of the object concerned, with the upper limit (**Max**) being the **maxSpeed** value, and the lower limit (**Min**), the negative value of **maxSpeed**. This is demonstrated as follows:

```

/*                      meanders                      */
/*****/

Object meanders :-
  get_prop( level, meanders, Level),      % get Level for meander
  get_prop( rule, number, RuleNo),        % get the current rule number
  get_prop( frame, current, Now),         % get the current frame number
  get_prev_locn( Object, Now, pt( X, Y)), % get Object's previous locn
  maxSpeed( Object, Max),                 % get Object's max speed
  make_neg( Max, Min),                    % Min = -Max
  random( XRand, Min, Max),               % Xrand = random( Min, Max)
  random( YRand, Min, Max),
  X1 is X + Xrand,                        % add the difference
  Y1 is Y + Yrand,
  Action = poss_locn( Object, Now, Level, pt( X1, Y1)),
  do( Action, RuleNo), !.                 % carry out Action

```

This is a simple form of meandering that does not take momentum into consideration, so sometimes objects may appear to 'jump' unpredictably from left to right, up and down. However, if the same object is under the influence of other rules (hence other actions), the resultant outcome can be quite convincing. (This is covered in § 3.5: Results to show how different methods can be used to achieve this visual subtlety).

(See Appendix D.4 for a full listing of `plans_encounter_to` and `plans_getaway_from`.)

### 3.2 Rules Parsing

The Language Parser uses standard Prolog DCG (Definite Clause Grammar). The main purposes of the Language Parser (**sentence**) are, for any given rule, to check the syntax and to decompose a complex rule into simpler and manageable phrases while still preserving meaning. The output is then passed to the **translator** for the instantiation of variables to be carried out. See § 2.4: Language Processing of Grammar Rules for a full description of how the Language Parser and Translator work.

In the following sub-sections, two rule examples taken from World.Jelly are used to show how rules are parsed. The steps are illustrated in Figure 2-20 (p. 66).

### 3.2.1 Rule Example 1

Rule example 1 consists of two conditions and one action that states:

```

if      [object1, is_nearby, object2,
        and, object2, is_static]
then    [object1, meanders].

```

this can be extracted in the form:

```

Conds =  [object1, is_nearby, object2, and, object2, is_static].
Actions = [object1, meanders].

```

When **Conds** is passed to **sentence**:

```

sentence( X, Y, Z, M1, Conds)

⇒      X = _158,      Y = _159,      Z = _160,
        M1 = X is_nearby Y and Y is_static

```

where, a number preceded by an under score represents a variable in Prolog. **X**, **Y**, and **Z** are variables for **object1**, **object2** and **object3** respectively. **M1** is instantiated to the original clause but as a Prolog structured object (not a Prolog list).

**M1** is then passed to **translate** for the variables **X** and **Y** to be instantiated by matching **M1** against the current database. **Z** is insignificant in this example as there are only two objects involved, and will be discarded automatically without affecting the result.

When **translate( M1)** is initiated, the clauses **X is\_nearby Y** and **Y is\_static** are tested separately, but the variables **X** and **Y** are kept consistent. Since **is\_nearby** and **is\_static** can be found in the standard rulebase (see § 2.5.1, p. 56), this operation is effectively interpreted as standard Prolog: **is\_nearby( X, Y), Y is\_static**. This will search for all possible solutions of any two objects that are nearby each other at this time, where one is a static object. After matching the database, in this case, there are two solutions:

- N<sup>o</sup>1    **M1 = jelly1 is\_nearby rock1 and rock1 is\_static.**  
           (where object1 = jelly1 and object2 = rock1)
- N<sup>o</sup>2    **M1 = prawn1 is\_nearby rock1 and rock1 is\_static.**  
           (where object1 = prawn1 and object2 = rock1)

The syntax of **Actions** is also tested in the same way as mentioned above:

**Actions = [object1, meanders].**

**sentence( X, Y, Z, M2, Actions)**

⇒     **X = \_158,        Y = \_159,        Z = \_160,**  
          **M2 = \_158 meanders**

Since only **object1** is required in **M2** and it has already been found in **M1** that there are two possible solutions to **object1** (N<sup>o</sup>1 = **jelly1**, N<sup>o</sup>2 = **prawn1**), they can be put directly into **M2** (further **translation** for **Actions** is not required since the variables are already instantiated by the **translation** of **Conds**, see Figure 2-20). So there are two actions to be carried out:

- N<sup>o</sup>1    **M2 = jelly1 meanders.**  
           (when object1 = jelly1)
- N<sup>o</sup>2    **M2 = prawn1 meanders.**  
           (when object1 = prawn1)

A graphical approach showing how the Language Parser handles this particular rule can be seen in Figure 3-3. There are three objects in the domain: **jelly1**, **prawn1** and **rock1**.

Chapter 3: World.Jelly



```
fact jelly1 has_type 'JELLYFISH'.
fact location( jelly1, pt(X1, Y1)).
fact jelly1 is_alive.
```

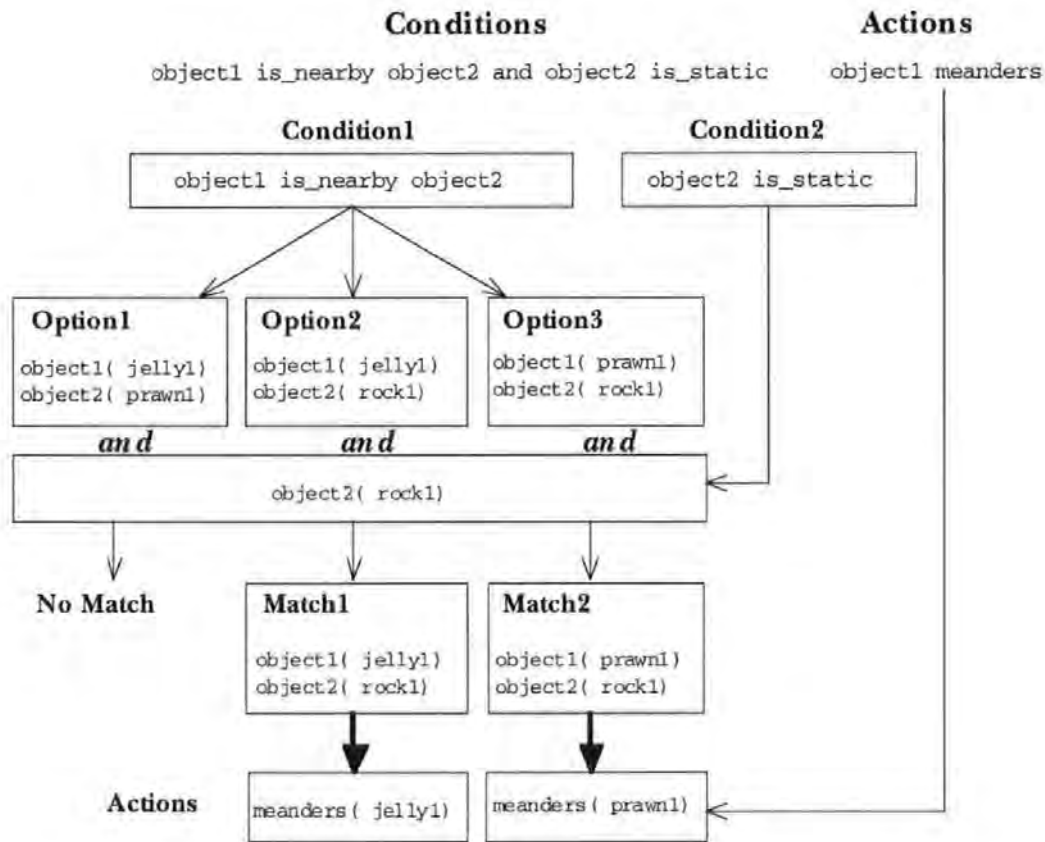


```
fact prawn1 has_type 'PRAWN'.
fact location( prawn1, pt(X2, Y2)).
fact prawn1 is_alive.
```



```
fact rock1 has_type 'ROCK'.
fact location( rock1, pt(X3, Y3)).
fact rock1 is_static
```

(a) Initial State



(b) Interpretation of a Rule

Figure 3-3 Rule Parsing to Executing Actions: (a) Initial State (b) Interpretation of a Rule

By matching the database, **Condition1** (**object1 is\_nearby object2**) gives 3 options and **Condition2** (**object2 is\_static**) gives only 1 option. When the combinations are paired, only two match, **Match1** and **Match2**, and as a result, two **Actions** (**meanders( jelly1)**, and **meander( prawn1)**) are carried out.

The log output would be:

```
...
jelly1 is_nearby prawn1.
jelly1 is_nearby rock1.
prawn1 is_nearby rock1.
New fact : 18 poss_locn( jelly1, 1, pt( 73, 60))
New fact : 19 poss_locn( prawn1, 1, pt( 26, 12))
...
```

### 3.2.2 Rule Example 2

Rule example 2 consists of three conditions and one action that states:

```
if      [object1, is_escaping_from, object2,
        and, object1, is_nearby, object3,
        and, object3, is_shelter_of, object1]
then    [object1, plans_encounter_to, object3].
```

this can be extracted in the form of:

```
Conds =  [object1, is_escaping_from, object2,
          and, object1, is_nearby, object3,
          and, object3, is_shelter_of, object1].

Actions = [object1, plans_encounter_to, object3].
```

When **Conds** is passed to **sentence**:

```
sentence( X, Y, Z, M1, Conds)

=>   X = _192,      Y = _193,      Z = _194,
      M1 = X is_escaping_from Y and X is_nearby Z and Z is_shelter_of X
```

Similarly, **M1** is then passed to **translate** for the variables **X**, **Y** and **Z** to be instantiated by matching **M1** against the current database. After matching the database, in this case, there is only one solution for the conditions:

Nº1    **M1 = prawn1 is\_escaping\_from jelly1,  
               and prawn1 is\_nearby rock1,  
               and rock1 is\_shelter\_of prawn1.**  
       (where object1 = prawn1, object2 = jelly1 and object3 = rock1)

**object1** and **object3** from the solution can now be applied to the action associated with it (**object2** is not required in **Actions**):

**Actions = [object1, plans\_encounter\_to, object3]**  
**sentence( X, Y, Z, M2, Actions)**  
 ⇒    **X = \_192,            Y = \_193,            Z = \_194,**  
       **M2 = \_192 plans\_encounter\_to \_194**

Since only **object1** and **object3** are needed now, and they have to be consistent in both cases of **Conds** and **Actions**, there is only one solution:

Nº1    **M2 = prawn1 plans\_encounter\_to rock1.**  
       (where object1 = prawn1 and object3 = rock1)

These illustrations show that the system generates all possible future states from an initial state and a set of rules. At present, all actions eventually lead to the generation of suggested transitions from one location to another. However, in the future, more sophisticated behaviour such as the changing of colour and size can also be incorporated.

### 3.3 Inferencing

In the second part of this chapter, the various different inferencing/control methods and programming styles in implementing the system are discussed.

At the global level of control within an expert system, the rules can be driven backwards or forwards. A conclusion that a user wishes to establish can be chained backward, by establishing the conditions necessary for its veracity, and thus seeing if it can be supported by the initial facts. In this case, special goal statements in the working memory are matched against the right-hand side of the rules. Modifications to working memory then manipulate these goal statements (e.g. replacing them with subgoals), as

well as modifying patterns of data. In the case of medical diagnosis, where backward chaining methods are often used, (for example MYCIN **Appendix B2**), the existence of evidences about the symptoms of an illness can be traced backwards to find their medical cause. This type of reasoning gives a tight logical argument in a narrow structure to work directly towards a goal.

However, those conditions that are known to be true (established facts) can be chained forward towards conclusions by matching the data in working memory against the left-hand side of the rules. An example of type of system is R1, the program that configures VAX computers (**Appendix B2**).

When trying to model varied and more life-like behaviour of living objects, one possible way is to influence the future of individual object by taking different intentional actions under certain situations. As in real life, the future is unpredictable. One could argue that one cannot predict future events, but one can try to steer towards a goal. Intentional action is not aimed at bringing about a particular state but rather at steering a course between various constraints hopefully in the general direction of one or more goals - what Suchman (1987) calls "situated action". A goal for a simple animal as an example would be to stay alive; but this also means that it has to eat and avoid being eaten by another animal.

Forward chaining seems to suit this situation well because it has a tendency to branch forward, exploring all possibilities. Forward chaining gives a wider variation of choices (am I hungry? eat now or later? is there a danger nearby?), however, it has to be carefully controlled and the branching has to be confined otherwise it will produce more information than that needed. This will be discussed further in the next two sections (§ 3.4 & § 3.5).

Although the system uses a forward chaining algorithm, problems can be presented to it in two different forms: goal-oriented and non goal-oriented. Each of these will be demonstrated with working examples.

### 3.3.1 Example 1 : Goal-Oriented

The system is given a set of facts representing the final goal and a set of rules containing all constraints and it applies all its rules to these facts, generating new facts until the goal state is reached or no solution can be found (e.g. all frames are planned or it can go no further).



The important part of the problem solving strategy adopted in this example is how the initial time slot is divided and subsequently subdivided into smaller ones, so different actions can be carried out according to the locations of the objects and their speed. For example, if object X is going from location A to location B, depending on the distance between A and B and also the amount of time available to get there, X can choose to **propel** (swim faster) or **meander** (swim slower). These will be explained later.

In this example there are three objects (**jellyfish1**, **prawn1** and **rock1**). Their initial and final locations as well as the time and location of two encounters are appraised. Figure 3-4 shows the facts that are needed and these form the initial state of the database. A set of rules are also needed which tell the system such things as how to plan a sequence (by subdividing it), how to plan a particular state and how to make objects move. Figure 3-5 shows some such typical rules.

```
% basic facts about each object
obj_type( jelly1, [alive]).
obj_type( prawn1, [alive]).
obj_type( rock1, [static]).

% facts about this run: planning from time 1 to 16
plan 1 to 16.

% initial positions for objects
% location( Object, Time, Loc)
location( rock1, 1, pt( 83, -108)).
location( jelly1, 1, pt( -200, -130)).
location( prawn1, 1, pt( 172, -170)).

% final positions for objects
% location( Object, Time, Loc)
location( jelly1, 16, pt( -200, -130)).
location( prawn1, 16, pt( 172, -170)).

% the encounters
jelly1 encounters rock1 at time( 3) at pt( 83, -108).
jelly1 encounters prawn1 at time( 10) at pt( -183, 27).
```

**Figure 3-4** Facts Required for Example 1: Goal Oriented

```

...

% if there is an encounter within a sequence, then plan the encounter,
% then plan the sequences before and after it
rule2: if    plan Time1 to Time2
            and jelly encounters Something at time(TimeN) at P
            and TimeN between Time1 & Time2
    then plan_state TimeN at P
            and plan Time1 to TimeN
            and plan TimeN to Time2
            and undo( plan Time1 to Time2).

...

% if the jellyfish has to move slowly (average speed =<45),
% then make it meander
rule6: if    move( Time1, Time2)
            and is_average_speed( Time1, Time2, A)
            and less_than_or_equal_to( A, 45)
    then meander( Time1, Time2).

...

% if the jellyfish encounters a static object,
% then make the jellyfish move upwards 80 units one frame later
rule12:    if encounter( jelly, Object, Time1, SomePlace)
            and property( Object, static)
            and move( Time1, Time2)
            and after( Time1, 1, TimeN)
            and not( location( jelly, TimeN, _))
    then above( SomePlace, 80, NewPlace)
            and plan_state TimeN at NewPlace
            and planmove TimeN to Time2
            and undo( move( Time1, Time2)).

...

```

**Figure 3-5** Some Rules Required for Example 1: Goal Oriented

The main inference routine repeatedly applies all rules to the database until there are no changes. In each pass, each rule is tested to see if its conditions can be met. If they can, the actions are performed and the search continues for any more ways of satisfying the same conditions. Only when all possible ways of meeting the conditions have been explored does the system proceed to look at the next rule.

The language used for defining rules in Figure 3-5 has certain operations which, as we have seen, will be performed when the rule is fired. These may vary from application to application but are mainly logical, spatial and temporal operators, these are some example of operators for example 1:

above,	after,
below,	between,
greater_than,	greater_than_or_equal_to,
has_type,	is_average_speed,
is_between_places,	is_between_times,
is_bigger_than,	is_nearby,
is_not_same_as,	is_same_as,
is_same_size_as,	is_same_type_as,
is_smaller_than,	less_than_or_equal_to,
less_than,	position_of

The system is initiated and an extract from its log output is shown in Figure 3-6:

```

** Pass 1
New fact: 13 plan( 1 to 16) by rule1
New fact: 14 plan_state( 3 at pt( 83, -108)) by rule2           % encounter at time 3
New fact: 15 plan( 1 to 3) by rule2                             % move plan 1
New fact: 16 plan( 3 to 16) by rule2                             % move plan 2
Deleting fact: 13 plan( 1 to 16) by rule2
%% an encounter at time 3 has been used to divide the plan into 2 parts
...
New fact: 22 move( 1, 3) by rule7                                % move identified
New fact: 23 move( 10, 16) by rule7                             % move identified
New fact: 24 location( jelly1, 3, pt( -83, -108)) by rule9       % encounter pt 1 fixed
New fact: 25 location( jelly1, 10, pt( -183, 27)) by rule9       % encounter pt 2 fixed
New fact: 26 location( prawn1, 10, pt( -183, 27)) by rule10      % encounter pt 2 fixed
%% some positions are fixed and two moves are identified
...

** Pass 2
New fact: 27 plan_state( 10 at pt( -183, 27)) by rule2           % encounter at time 10
New fact: 28 plan( 3 to 10) by rule2                             % move plan 3
New fact: 29 plan( 10 to 16) by rule2                             % move plan 4
New fact: 30 propel( 1, 3) by rule4
New fact: 31 meander( 10, 16) by rule6
%% the nature of the jellyfish's move is decided
...
New fact: 37 location( jelly1, 6, pt( -107, -18)) by rule6
%% an intermediate position of the jellyfish move is calculated
...

** Pass 3
New fact: 39 meander( 3, 10) by rule6
%% jelly1 meanders from time 3 to 10
...

```

Figure 3-6 Runtime Extracts for Example 1: Goal Oriented

It can be seen that how the initial time slot is being divided and subsequently subdivided into smaller ones. A simple graphical view of how the time slots are planned can be seen in Figure 3-7:

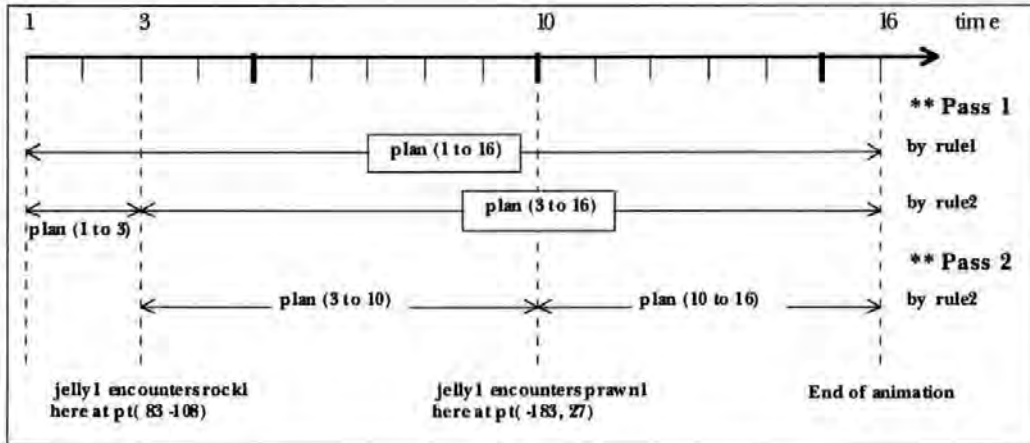


Figure 3-7 Time-planning

As shown in Figure 3-6, processing starts from **\*\* Pass 1**. **rule1** initiates the planning process by planning the whole period as one, **plan( 1 to 16)**. When **rule2** detects the first encounter at time = 3 between **jelly1** and **rock1** (told, see Figure 3-4), a new plan is obtained by division:

**plan( 1 to 16) = plan( 1 to 3) and plan( 3 to 16).**

For administrative purposes, the old plan **plan( 1 to 16)** will have to be deleted from the database (shown in Figure 3-7 in boxes) by using the **undo** operator. If left, it would continue to generate sub-plans for each time-slot.

During **\*\* Pass 2**, again **rule2** detects an encounter at time = 10 (between **jelly1** and **prawn1**), a further plan is needed from **plan( 3 to 16)** (as 10 falls between these times):

**plan( 3 to 16) = plan( 3 to 10) and plan( 10 to 16).**

Similarly, **plan( 3 to 16)** is deleted from the database. Since there are no other known encounters, the final number of plans to be completed are three:

**plan( 1 to 3), plan( 3 to 10) and plan( 10 to 16).**

Within each of these time-slots, the movement is conducted by using either the **meander** or **propel** operators. Since the encounter locations and the time (e.g. **plan( 1 to 3)** to get there are known, an average speed can be calculated, a threshold value can then be set (in this case 45 units, see **rule6** in Figure 3-5) to determine which operator will be used. If the average speed is larger than the threshold value, it means that the object is in a hurry to get to its destination, so it will have to **propel**, otherwise it will **meander**. The need to have these two different operators lies in the randomness in each case: **propel** tends to have a 'focus' on the location it is heading for, thus the randomness in the degree of direction would be less than that of **meander**. When an object has more time than it needs to get to its destination, it can 'waste-time' by meandering about.

After 13 passes, the system can make no more changes to the database and creates, as output, a list of all the locations it has determined, see Figure 3-8:

```
% locations for jelly1
location(jelly1, 1, pt(-200, -130)).
location(jelly1, 3, pt( 83, -108)). % encounter with rock1
location(jelly1, 5, pt( -65, -71)).
location(jelly1, 6, pt(-107, -18)).
...
location(jelly1, 10, pt(-183, 27)) % encounter with prawn1
...
location(jelly1, 16, pt(-200, -130)).

% locations for prawn1
location(prawn1, 1, pt( 172, -170)).
location(prawn1, 3, pt( 72, -55)).
location(prawn1, 5, pt( -33, 9)).
location(prawn1, 6, pt(-103, 31)).
...
location(prawn1, 10, pt(-183, 27)). % encounter with jelly1
...
location(prawn1, 16, pt(172, -170)).

% locations for rock1
location( rock1, 1, pt( 83, -108)). % since rock1 is static..
... % ...all the locations are..
location( rock1, 16, pt( 83, -108)). % ...the same.
```

**Figure 3-8** Location output for Example 1: Goal Oriented

Where there are no details for a time slot (for example Time = 2, 4 etc.) then simple averaging is used to calculate these positions from within the system. The final output can then be converted and fed into the Swivel 3D™ model to drive the animation.

The goal-oriented approach requires that the goals (as facts) and rules be entered into the system. Rules written this way do not *suggest* but rather, determine the future states by sub-dividing time-slots. The main drawbacks with this version are the complexity of writing the rules, the inability to see the detailed effect of rules on the animation and the absence of any means to control the animation rather than by formally specifying facts or rules.

### 3.3.2 Example 2 : Non Goal-Oriented

In the second example, the system is given only an initial state and a set of rules and successive frames are planned until the required time period is completely planned. There is no final goal state and no additional constraints.

The main inference routine used in this example differs slightly from the previous example. All the rules in this example refer to changes between one frame and its successor so the result from one complete pass of the rules is a set of possible moves that each object could sensibly make. The inference engine then decides on the resulting location for each object. The progress of the animation can be monitored in a graphics window.

The same three objects, **jelly1**, **prawn1** and a **rock1** participate. Figure 3-9 shows the facts that are needed and these form the initial state of the database.

**plan 1 to 7.**

**object( rock1, rock, [static]).**  
**location( rock1, pt( 150, 220)).**

**object( prawn1, prawn, [alive]).**  
**location( prawn1, pt( 168, 154)).**

**object( jelly1, jellyfish, [alive]).**  
**location( jelly1, pt( 123, 103)).**

**Figure 3-9 Facts Required for Example 2: Non-Goal Oriented**

Unlike the goal-oriented example as seen earlier, non-goal oriented planning does not subdivide time-slots, but works from a set of rules which change the state of objects or directly calculate their next location. Figure 3-10 shows some typical rules:

```

...

% When objects are near, the prey plans to escape
rule 3 : if      object1 is_nearby object2
              and object2 is_predator_of object1
              then object1 plans_getaway_from object2.

% When objects are near, the predator plans to attack
rule 4 : if      object1 is_nearby object2
              and object2 is_food_of object1
              then object1 plans_encounter_to object2.

%      When no objects are near, then meanders
rule 5 : if      object1 is_alive
              then object1 meanders.

%      When no objects are near, then stay there
rule 6 : if      object1 is_static
              then object stay_there.

...

```

**Figure 3-10** Some Rules Required for Example 2: Non-Goal Oriented

When the animation is run, an extract from the log is shown in Figure 3-11. Here, operators (such as **plans\_encounter\_to**, **meanders** etc.) triggered by the rules are assigned priority levels (lower number denotes higher priority). Looking at the log output in Figure 3-11, at **Pass = 7**, there are two **poss\_locn** (possible locations) for **jelly1** as generated in **Newfact 62** and **64**:

```

New fact : 62 poss_locn(jelly1, 7, 3, pt(166, 94)) by rule 4    % level 3
New fact : 64 poss_locn(jelly1, 7, 4, pt(156, 55)) by rule 5    % level 4

```

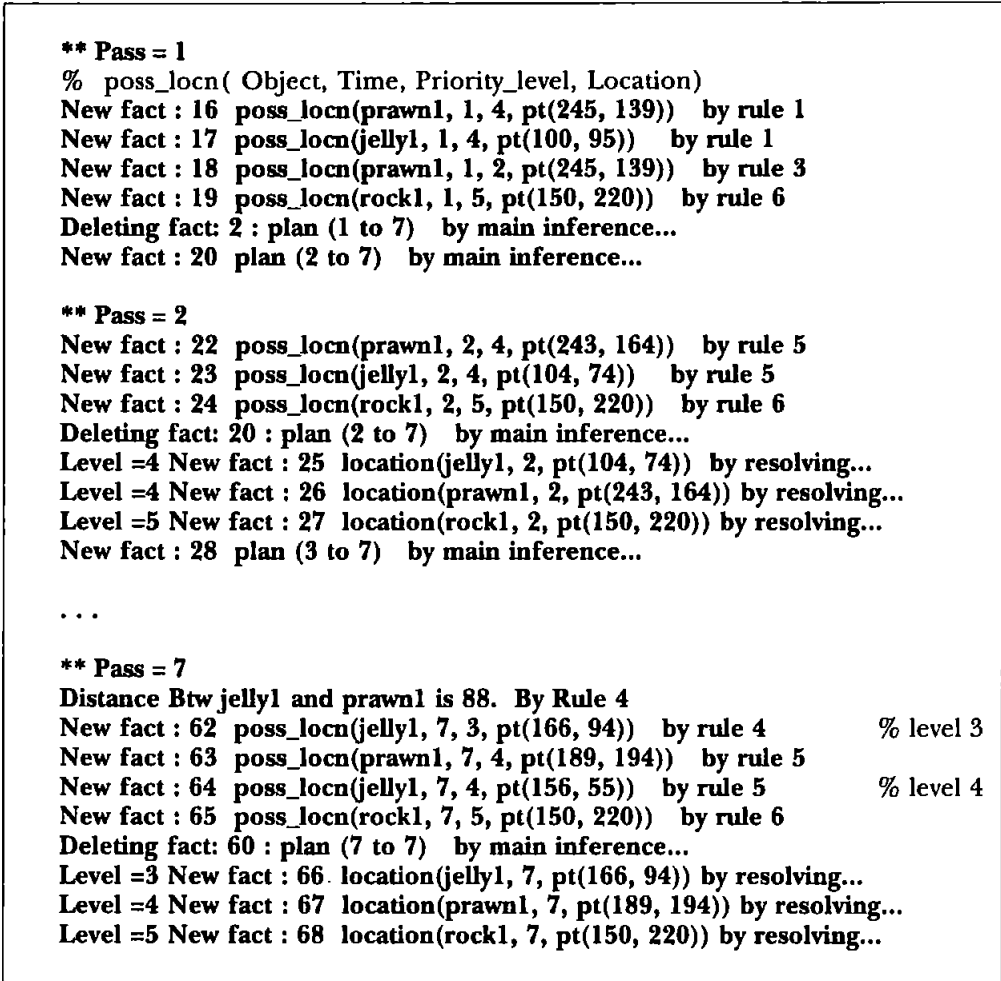


Figure 3-11 Runtime Extracts for Example 2: Non-Goal Oriented

This situation arises when **jelly1** satisfies the two rules, **rule4** : “if nearby food then plans encounter to food” and **rule5** : “if alive, meanders”. Since **plans\_encounter\_to** (level 3) has a higher priority than **meanders** (level 4) it determines the final location of **jelly1** (see § 3.5: Results for how levels are defined and how different solutions are solved):

```

New fact : 66 location(jelly1, 7, pt(166, 94)) by resolving...

```

Unlike the goal-oriented example, having completed the run, all seven keyframes will have a location allocated to each object involved so no extra averaging or in-betweening process is needed. A list of these locations for all objects can be printed out



in Swivel Script format to generate the keyframes and animations from within Swivel 3D™, see Figure 3-12:

```
( This is KeyFrame = 1 )
jelly1: 100, 95, 0
prawn1: 245, 139, 0
rock1: 150, 220, 0

( This is KeyFrame = 2 )
jelly1: 104, 74, 0
prawn1: 243, 164, 0
rock1: 150, 220, 0

. . .

( This is KeyFrame = 7 )
jelly1: 166, 94, 0
prawn1: 189, 194, 0
rock1: 150, 220, 0
```

**Figure 3-12** Output for Swivel Script of Example 2: Non-Goal Oriented

The non goal-oriented approach, once again produced a number of acceptable animations. This approach was adopted for further development as the number of rules and the complexity of writing them are reduced considerably when compared with the goal-oriented approach. Moreover, this approach also produced more interesting results because it allowed for further branching and exploring all possibilities of different situations. However, these 'branches' would need to be filtered in order to make the system more productive at solving problems rather than suggesting more than what was needed. These filtering processes will be discussed in § 3.5: **Results**.

### 3.4 Ordering

Why is ordering a problem? Prolog objects are handled in the memory by their names (e.g. `jelly1`), which are picked sequentially one at a time when called from the existing objects list (e.g. `[jelly1, prawn1, rock1]`). By (Prolog's) default, these names are sorted into ascending alphabetical order within the described ESCAPE system.

Take an example of pairing any two of these four objects, A, B, C, and D. Assuming objects cannot pair with themselves, (e.g. AA, BB, CC and DD) and repeated

pairings are discarded ( i.e.  $AB = BA$ ,  $AC = CA$  etc.), it would give six possibilities as shown in Figure 3-13:

AB, AC, AD,	( <del>AA</del> )
BC, BD,	( <del>BA, BB</del> )
CD.	( <del>CA, CB, CC</del> )

**Figure 3-13** Pairing Sequence of Four objects

Also, assuming that the testing routines within the inference engine involve only the interaction between two objects, the above pairings would form the basis of all the testing sequences to be examined by all the rules.

In this section, the issues involved in the ordering of objects are discussed. The problem lies in the order in which they are presented. This is one of those problems that was not foreseen until the later stages of the designing of the system.

## 3.4.1 Object Ordering

The order of the pairing sequence as shown in Figure 3-13 is satisfactory if these objects are in a world where there is not much happening apart from roaming around, avoid colliding into each other, and there is no fear of getting eaten or killed. In the case of resolving a collision, it doesn't matter which object remains static and which has its position adjusted.

However, such an approach would not be feasible when representing examples such as flocking birds, a school of fish, or a herd of animals, because here, ordering does play an important role. In these cases, there is usually a leader which other flockmates tend to follow. Each flockmate has the same chance of moving ahead and effectively becoming the leader, so it is important that the new leader and their location is resolved first.

Let us assume that there are four birds in a flock, **bird1**, **bird2**, **bird3**, and **bird4**. If finding a leader is not a priority, then the pairing sequence as shown in Figure 3-13 is adequate; but if a leader is required then some ordering process is needed. Given their locations and the directions the birds are travelling, an order can be constructed in which the front most bird (relative to the flock) is placed first in the list, followed by the second and so on.

Let us say, after sorting, the order of a flock of birds is: [**bird3**, **bird1**, **bird4**, **bird2**], where the leader is **bird3**, then followed by **bird1**, **bird4**, and **bird2**. The inference engine works in a way that **bird3** is examined first and after certain routines of movement and collision detection, a desirable location is given to **bird3**. When the second in the list, **bird1** is being examined, it has only one bird to check against for collision(**bird3** with its new location). If all goes well for **bird1**, **bird4** is in place for examination, against **bird3** and **bird1**. If say, **bird4** is found to be in collision with **bird1**, immediately a new location can be recalculated for **bird4** and rechecked so that the new location is compatible with **bird3** and **bird1**. This continues until a suitable location is obtained, then it goes on to the next bird, **bird2**. If no such ordering is applied, the whole behaviour of the flock could be different.

### 3.4.2 Situated Action Ordering

Situated action ordering derives from *Situated Action* (Suchman 1987), which suggests that in a multi-agent environment, under certain situations, agents naturally possess different intentional actions which are aimed at steering towards the general direction of one or more goals.

This can be illustrated in an imaginary chasing scene where a hungry jellyfish is pursuing a nearby prawn. It can be looked at from two different points of view: the pursuing agent and the agent being pursued. Realistically, the motion of the jellyfish depends on the prawn's location, so it only makes sense that a location for the prawn is determined before the jellyfish's. Otherwise, if the location of the pursuing jellyfish is obtained before the prawn's, the prawn could be moving in the opposite direction and the jellyfish would not be pursuing it.

What will happen if during the chasing scene, the prawn sees a small fish (prawn's favourite food) meandering nearby? Will it pursue the fish, or will it continue to avoid the jellyfish?

If objects are given 'memory', and can 'remember' their situated actions at the previous time interval, an order for objects as mentioned earlier would not have any significant impact. In the jellyfish example above, the next location for the jellyfish will still be advanced one step closer to the prawn because it remembers it was pursuing the prawn. These 'memories' can be suggested to include in the property list of an object as:

```
property( fish1, [is_alive, meandering at_time 13]).
property( jelly1, [is_alive, is_hungry, pursuing( prawn1) at_time 14]).
property( prawn1, [is_alive, is_hungry, escaping( jelly1) at_time 13,
                  sees_food( fish1) at_time 13]).
```

This should give the same continuity as ordered object lists with respect to the pursuing ability of the jellyfish, but also combine a hint of unpredictability into the objects' actions. Imagine, if the prawn suddenly makes a sharp turn to the left, the jellyfish would not have the empirical information (the next location of the prawn) until one time interval later, so the action can only be rectified after the subsequent time interval. Similarly for the jellyfish, using its 'memory' of where the prawn was last seen, it can predict the direction of the prawn's next move.

But what should the prawn (or in fact any other object) do when faced with the choice of eat or run? If some kind of hierarchical ordering is applied whereby different situated actions are given priorities (situated action ordering), a suitable action can be resolved. An example can be constructed as:

(higher priority) *escaping* > *pursuing* > *eating* > *meandering* (lower priority)

For the prawn, there are two situations to be considered: *eating* and *escaping*. The domain can be set up and the inference engine adopted so that both these situations are taken into consideration, and depending on the chosen resolving method (for example, it could consider both situations, either one, or the one with the higher priority and so on), a suitable solution can be obtained.

Both the *object ordering* and *situated action ordering* methods contribute to achieving realistic animation, but it is the *situated action ordering* method that has been chosen to be implemented into the ESCAPE system, due to the fact that it produces significant improvement and provides better control than the other. This, and different resolution methods, with some of their uses, will be explained in § 3.5: Results.

In future, it is envisaged that both methods will be implemented into the system leaving the user to choose which method is most suitable in a given domain.

### 3.5 Results

The inference engine explores every rule, matching it against the database and generating a single list of all possible actions. When this has been completed, it groups the possible actions for each object and passes them to a function which returns the next state of the object. There are many different strategies that could be adopted for this function: one possible action could be selected and the rest discarded (e.g. select the strongest, or the weakest), or a product of all the actions could be found, or a threshold could be introduced, resolving those that meet it. Different strategies can be adopted for different domains.

The World.Jelly domain is used as an example here. Several methods will be discussed for resolving these possible locations. Assuming, the animation of World.Jelly is governed by these rules,

```
rule2: if [object1, is_nearby, object2, and, object2, is_predator_of, object1]
      then [object1, plans_getaway_from, object2].

rule3: if [object1, catches, object2, and, object2, is_food_of, object1]
      then [object2, dies].

rule4: if [object1, is_nearby, object2, and, object2, is_food_of, object1]
      then [object1, plans_encounter_to, object2].

rule5: if [object1, is_alive]
      then [object1, meanders].

rule6: if [object1, is_static]
      then [object1, stay_there].
```

that the animation has already been started and it is currently at frame number 2. This is the current database:

```
frame number = 2,
% there are 3 objects and these are their properties
jelly1 is_alive,
prawn1 is_alive,
rock1 is_static

jelly1 is_nearby prawn1,          % these are the situations
jelly1 is_predator_of prawn1.
prawn1 is_food_of jelly1.
```

Upon running, the system produces the following possible locations (**poss\_locn**) for all the objects for this frame, the rules that fired the facts are printed after the facts:

```

** Pass = 2
% poss_locn( Object, Frame, Priority_level, Location)      by rule number
New fact : 16 poss_locn(prawn1, 2, level_3, pt(126, 112)) by rule2
New fact : 17 poss_locn(prawn1, 2, level_2, pt(130, 120)) by rule3
New fact : 18 poss_locn(jelly1, 2, level_4, pt( 73, 60))  by rule4
New fact : 19 poss_locn(jelly1, 2, level_6, pt( 70, 28))  by rule5
New fact : 20 poss_locn(prawn1, 2, level_6, pt(100, 110)) by rule5
New fact : 21 poss_locn(rock1, 2, level_1, pt(147, 173))  by rule6

```

Figure 3-14 Possible locations of objects

Looking at the output in Figure 3-14, the immediate problem here is that some of the objects have more than one possible location effected by different rules, for example, **prawn1** appeared in three newfacts(16, 17, and 20), and **jelly1** appeared in two (18 and 19). This is to be expected, because the conditions of different rules can be satisfied in more than one way (this will be explained later). But, there can only be one final move assigned to each object, so a method will have to be found to solve this problem. Since every **poss\_locn** is triggered by a rule, for example,

```

New fact : 16 poss_locn(prawn1, 2, level_3, pt(126, 112))      by rule2

```

is triggered by **rule2**:

```

rule2:  if object1 is_nearby object2 and object2 is_predator_of object1
        then object1 plans_getaway_from object2.

```

and it can be seen that the situated action for this rule is **plans\_getaway\_from**.

With careful observation, different priorities are given to different actions according to their importance and urgency. Figure 3-15 shows how a priority table can be constructed (which may vary for different objects).

PRIORITY LEVELS	ACTIONS
level_1(Highest)	stay_there
level_2	dies
level_3	plans_getaway_from
level_4	plans_encounter_to
level_5	meanders_forward
level_6(Lowest)	meanders

Figure 3-15 Default Priority Levels for Various Actions

It is arranged in a way such that, **meanders** is considered to be least important (because all living objects move anyway), so it is given the lowest priority 6. Escaping (**plans\_getaway\_from**) is considered to be more important than **plans\_encounter\_to** (food for example), so **plans\_getaway\_from** has a higher priority level. When an object is trying to escape from a predator (**plans\_getaway\_from**), and it is caught and killed (**dies**), the information about it being killed is more important than that of escaping, so **dies** has a higher priority. Static objects usually don't move, so **stay\_there** is considered to have the highest priority. These are the default settings shared by all objects in World.Jelly.

A different order can be constructed in a similar fashion according to the actions adopted and the type of domain being modelled, and entered into the database (at present, only) by the domain expert. One future implementations would be, for all objects to have a 'priority level table' which can be accessed by the user through a graphical interface, so that different settings can be applied to different objects.

Having applied the priorities to different actions, the one final solution of an object can be deduced by using the priority table. Several methods will be discussed following this. For the next three methods, Priority Level methods will be used, and for the fourth a Vector method. Then other factors such as randomisation will be discussed, and finally, the results will be presented.

3.5.1 One Highest Priority Level

This is the simplest method. In this case, only the highest priority level is considered. If there is more than one possible solution for the same priority, then only one (the first one) is chosen and the rest discarded.

From Figure 3-14, for **jelly1**, the possible levels were 4 and 6:

New fact : 18 poss\_locn(jelly1, 2, level\_4, pt( 73, 60)) by rule4  
New fact : 19 poss\_locn(jelly1, 2, level\_6, pt( 70, 28)) by rule5

Since **level\_4** has a higher priority than **level\_7**, so **Newfact: 18** is chosen to become the final location for **jelly1**:

Level = 4 New fact : 22 locn(jelly1, 2, pt( 73, 60))

For **prawn1**, **level\_2** has the highest priority over **level\_3** and **level\_6**:

New fact : 16 poss\_locn(prawn1, 2, level\_3, pt(126, 112)) by rule2  
New fact : 17 poss\_locn(prawn1, 2, level\_2, pt(130, 120)) by rule3  
New fact : 20 poss\_locn(prawn1, 2, level\_6, pt(100, 110)) by rule5

so **Newfact: 17** is chosen:

Level = 2 . New fact : 23 locn(prawn1, 2, pt(126, 112))

This option is implemented in the system, under the **System** menu : **Solve One Highest Level** option.

### 3.5.2 All Highest Priority Levels

Here, the principle is the same as One Highest Priority, with the exception that if there is more than one possible solution for the highest priority, then all will be taken into consideration. This situation arises when different solutions are suggested by different rules but with the same action. For example,

rule3: if [object1, is\_alive]  
then [object1, meanders].  
  
rule7: if [object1, is\_nearby, object2, and object2, is\_static]  
then [object1, meanders].



If both the rules succeed (say for **prawn1**), there will be two different (randomness permits) possible solutions (say in **level\_6**):

New fact : 17	poss_locn(prawn1, 2, level_6, pt(126, 112))	by rule3
New fact : 18	poss_locn(prawn1, 2, level_6, pt(130, 120))	by rule7

then the average of the two points **pt( 126, 112)** and **pt( 130, 120)** is the new location:

Level = 2	New fact : 22	locn(prawn1, 2, pt(128, 116))
-----------	---------------	-------------------------------

This method can represent actions in the face of multiple competing goals and give that extra unpredictability characteristic of a real-world situation. For example, when a jellyfish senses a group of prawns nearby, naturally, it tends to go for the centre of the group, thinking that the chances of catching more than one is high, but that in fact is not a good strategy.

This option is implemented in the system, under the **System** menu : **Solve All Highest Level** option.

3.5.3 Combination of Highest Priority Levels

This method involves selecting a number of possible priority levels and finding the resulting vectors from these selected possible locations.

Let us assume that an object at any time, has the following three possible locations from the following actions:

	PRIORITY LEVELS	ACTIONS
✓	level_3	plans_getaway_from
✓	level_4	plans_encounter_to
	level_6	meanders

The user can set a “depth of priority level” value **K**, whereby only calculations to the **K** number of vectors of the highest priorities are performed. Say if **K = 2** in the above example, the two highest priority levels are 3 & 4 as indicated with ‘✓’.

This gives the user that extra option to choose how much information (out of all those suggested by the system) is to be used. This option is not implemented in the system, but is planned for future implementation under the **System** menu : **Others...** option.

## 3.5.4 Resultant Vectors

The current approach of considering each object in isolation clearly has some limitations and it would be preferable, particularly in complex and crowded domains, for the states of all objects to be resolved together. It is envisaged that a more sophisticated resolution method could be introduced here. If agents were treated as sets of vectors, a more sensitive method could be used to resolve them into a single movement.

The possible actions can be represented as vectors (Andersen 1992) and more complex vector resolution algorithms can be introduced. This form of representation could also have benefits for future debugging software.

Let us look at the possible locations for **jelly1**. In this resultant vector method, the priority level is made redundant. The following is extracted from Figure 3-14:

New fact : 18	poss_locn(jelly1, 2, level_4, pt( 73, 60))	by rule4
New fact : 19	poss_locn(jelly1, 2, level_6, pt( 70, 28))	by rule5

Since the previous location of **jelly1** is predetermined from the database:

locn( jelly1, 1, pt( 65, 49))
-------------------------------

the vector values for pt( 73, 60) and pt( 70, 28) can be calculated from pt( 65, 49) .

A vector has value and direction, so by resolving all the vectors acting on an object, a resultant vector can be obtained, and therefore determining where the object is going next. See Figure 3-16.

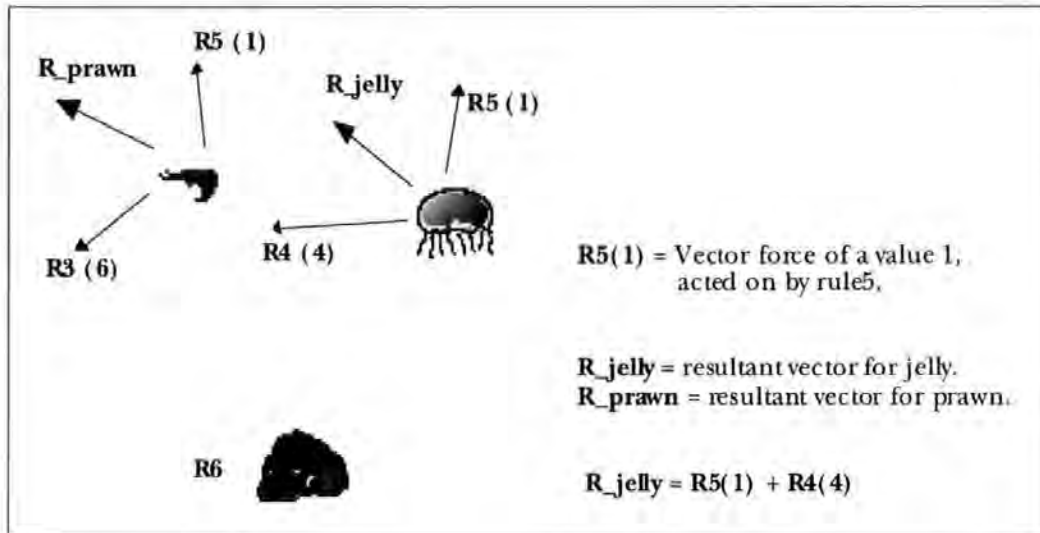


Figure 3-16 Resultant Vectors

It is also envisaged that these vectors could be presented by arrows (as shown above) in the graphical windows to give the animator an indication of which rules have the greatest influence on the whole. This would also be helpful for debugging purposes, as the rules that are not behaving as intended can be located and perhaps amended. This is one of the areas to be implemented in future.

### 3.5.5 Randomisation

Randomisation is important because it helps the user to avoid unrealistically smooth motion in any results obtained. However, extreme caution has to be taken to ensure its effect is kept within limits, otherwise the results appear frenetic. It is important to know how to cut down randomisation when needed.

Randomisation is used in all the implemented operators involved in obtaining a possible location. For example, the operator **meanders** is used to determine a random location of an object in all direction. The code is as follow:

```

/*                                meanders                                */
/*****/

Object meanders :-
  get_prop( level, meanders, Level),          % get Level for meander
  get_prop( rule, number, RuleNo),            % get the current rule number
  get_prop( frame, current, Now),             % get the current frame number
  get_prev_locn( Object, Now, pt( X, Y)),      % get Object's previous locn
  maxSpeed( Object, Max),                     % get Object's max speed
  make_neg( Max, Min),                        % Min = -Max
  random( XRand, Min, Max),                   % Xrand = random( Min, Max)
  random( YRand, Min, Max),
  X1 is X + XRand,                           % add the difference
  Y1 is Y + YRand,
  Action = poss_locn( Object, Now, Level, pt( X1, Y1)),
  do( Action, RuleNo), !.                     % carry out Action

```

A random number is obtained from a range limited by the min. and max. values derived from **maxSpeed** (see § 2.2.1)- min. is negative value and max. is the positive value. These random numbers, **Xrand** and **Yrand**, are then added to the previous location of the object, **X1** and **Y1**, to get **X** and **Y**. **X** and **Y** are then set as a newfact with **level\_6** (priority level for **meanders**).

### 3.5.6 Results and Output

These are the results produced by the ESCAPE system for World.Jelly with various options selected. Times are measured in seconds by averaging several runs of the same animation using MacProlog's internal time resolver, based on the following statistics:

World:	<b>World.Jelly 2.3</b>
Nº of Frames:	<b>10</b>
Nº of Rules:	<b>12</b>
Nº of Objects:	<b>6 (2 jellyfish, 3 prawns and 1 rock)</b>

	Blind Run		Interactive Run	
	Collision On	Collision Off	Collision On	Collision Off
Text On	51.93	50.42	60.30	59.87
Text Off	38.83	38.62	49.83	49.10
Interactive Run vs. Blind Run				18%
Text On vs. Text Off				21%
Collision On vs. Collision Off				1%
"Full-option" vs. "Minimum-option" *				36%

Figure 3-17 Various Time (in Seconds) Taken to Run World.Jelly 2.3

And it can be seen in the chart in Figure 3-18:

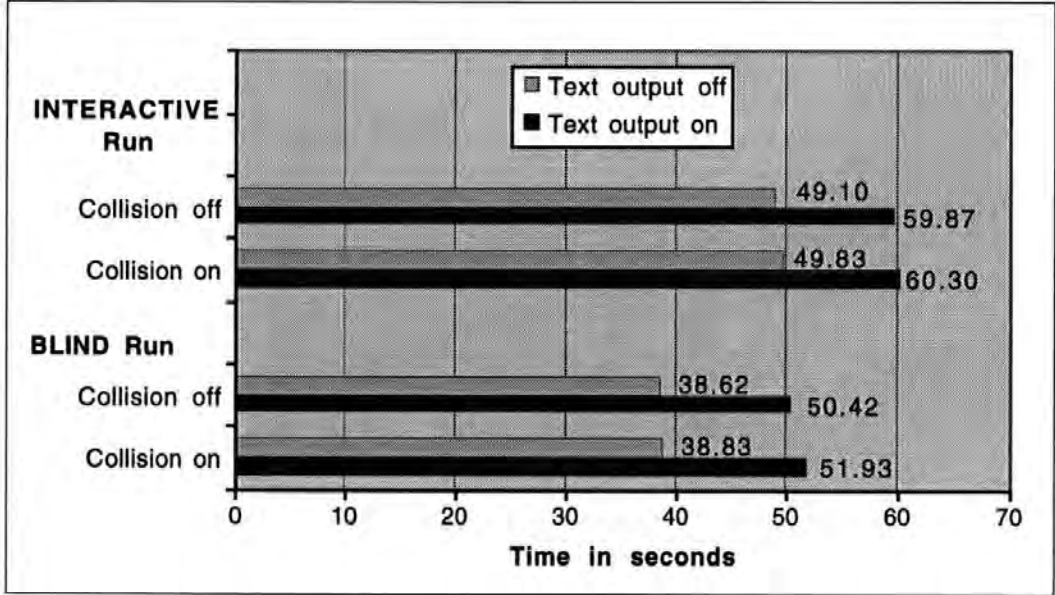


Figure 3-18 Time Chart for World.Jelly 2.3

\* "Full-option" = Interactive Run + Text Output On + Collision On and "Minimum-option" = Blind Run + Text Output Off + Collision On.

From Figure 3-17, it is clear that by using the **Blind Run** option, on average, it runs 18% (all percentages here are approximated) faster than the **Interactive Run** option. The biggest time saver is the **Text Output Off** option, which can save time up to 21% on some runs. Collision detection on the other hand, has not contributed any significant impact due to small number of objects involved (only 1% longer here with 6 objects as supposed to 33% in World.Bird with 16 objects, see Figure 4-12, p. 130).

If collision detection is to be switched on at all times, comparing the "Full-option" (Interactive Run + Text Output On) to the "Minimum-option" (Blind Run + Text Output Off), the "Full-option" takes nearly 36% longer than the "Minimum-option".

Although time is not a factor in this thesis, this comparison is important because it gives the viewer an idea of how to produce fruitful animations given the resources and options available. Compromises can then be made based on these consumption in different situations to save time, for example, if a traced log is not a prime priority, it is best switched off. It also gives the viewer a concept of processing time (machine dependent) to run an animation given different number of frames, rules and objects (see § 4.2.2).

Figure 3-19 shows 8 keyframes (out of 10) generated from the above run. Some behaviour can be observed from these keyframes especially for the prawns. In  $f_1$ , the prawns detect the presence of the two jellyfish, and take immediate actions to move towards the rock. This behaviour is defined in a collection of rules and facts which tell the prawns to plan an encounter to a shelter (rock) if a predator (jellyfish) is nearby. Similarly for the jellyfish, rules and facts contributed to the detection of food (prawns) nearby, so they move towards them.

A rendered image taken from the World.Jelly animation, produced by Swivel 3D™ can be seen in **Colour Plate 1** (p. vii)

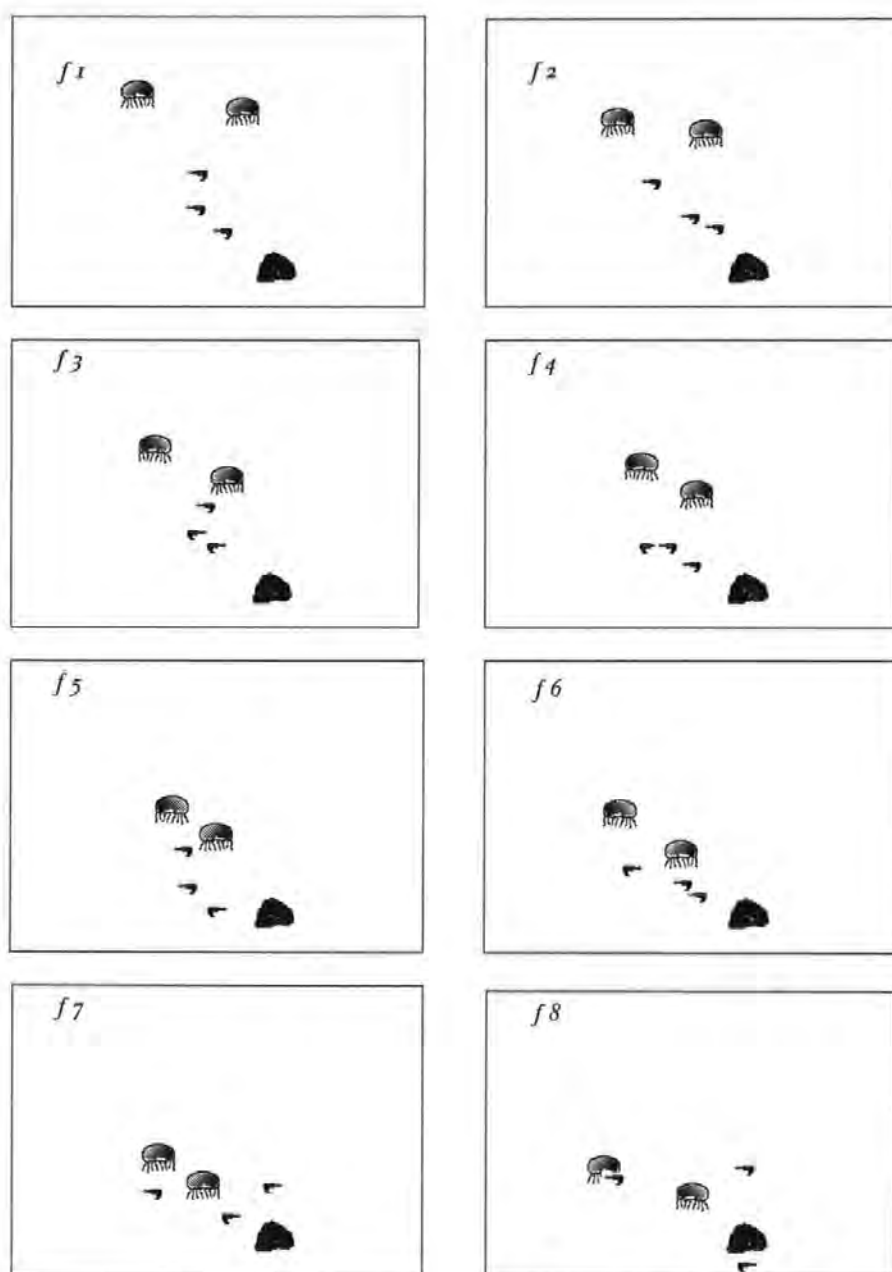


Figure 3-19 Keyframes for World.Jelly 2.3

The pursuing/escaping scenes goes on until  $f_7$ , when one of the prawns is entangled between the two jellyfish and cannot get nearer to the rock, and is consequently caught by one of the jellyfish ( $f_8$ ), while one shoots off to the clear ground above the rock (partly to avoid colliding into the rock), and the other moves in front of the rock to hide.

### 3.6 Summary

In this chapter, a domain of a fish-tank scene of jellyfish, prawns and rocks is demonstrated, and is used as an example of how such a domain database is constructed. Rules for describing the behaviour of the objects in this domain are also illustrated, and it is shown how they are parsed using some underlying rules.

Within the ESCAPE system, a forward chaining algorithm has been adopted, and two different strategies were also presented: goal-oriented and non goal-oriented. It has been shown that by adopting these two different approaches, problems can be presented and solved in different ways. Generally, distinct sets of rules had to be written for each approach.

The goal-oriented approach presented in this example achieved the objective of producing purposeful animation on the basis of facts and rules entered into the system. The main drawbacks with this version are the complexity of writing the rules, the inability to see the detailed effect of rules on the animation and the absence of any means to control the animation by direct manipulation of the objects.

Unlike the goal oriented approach, in the non goal-oriented approach, new types of states are *suggested* by the rules which affect future behaviour of an object but do not determine it. These states can then be used to control the animation accordingly using various ordering methods (§ 3.4: Ordering) and resolution methods (§ 3.5: Results). Also, the number of rules and the complexity of writing them are reduced considerably using this method. The non-goal approach has been adopted for further development.





CHAPTER 4

OTHER EXAMPLES



"Occasionally, it [computer graphics] will be used to create banal graphics, and it will get the blame. Equally, it will produce stunning, exciting images and be denied recognition. But I believe eventually it will find its natural place alongside other systems and enjoy a healthy life, and eventually we will all wonder what the fuss was about."

John Vince (1985)

## Chapter Overview

It has been shown from the WorldJelly domain that the ESCAPE system is capable of producing convincing animation from the problems presented. It should be equally important that it has the ability to handle a diversity of problems in order to be a useful tool.

To validate this, the integrity of the ESCAPE system is tested by applying it to two further domains which have different kinds of problem to solve, and are governed by different rule sets to that of the WorldJelly domain. They are domains of a traffic junction and a flock of birds.

By extending the application to a total of three domains we are hoping, with a degree of confidence, to further clarify the boundaries between those tasks that belong with the system, those in the domain and those with the end user.

## 4.1 World.Traffic

In this domain there is a T-junction and a number of cars. Each car arrives on the scene at a specified location and at a specified time with its destination through the junction predetermined. A set of general rules are written to determine how each driver behaves. In general cars aim to go as fast as they can up to their maximum speed, but have to slow down if their way ahead is blocked or they wish to turn. If a vehicle should stop, then other vehicles behind it will slow down and stop; if a vehicle wishes to turn it will stop and wait until it estimates that the way ahead is clear.

A UK left-hand-drive road system is adopted, and for simplicity, the motorists have the following special conditions:

- they have perfect vision and knowledge of the road;
- there is no overtaking on the road;
- they either move vertically (north <-> south) or horizontally (west <-> east) within the road;
- priority is given to vehicles on the major road;
- they follow the flow of the traffic (see Figure 4-1).

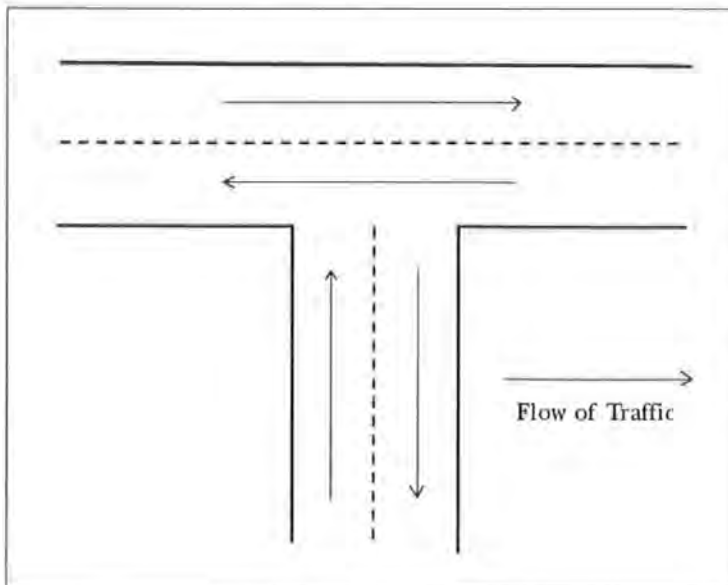


Figure 4-1 Flow of Traffic

#### 4.1.1 Traffic Database

In order for the motorists to have knowledge of the road, different areas at the junction have to be defined. These areas are the key to describing the behaviour of the traffic because they give indications of the direction of the flow of traffic as shown in Figure 4-1. Eight areas can be divided and illustrated as below:

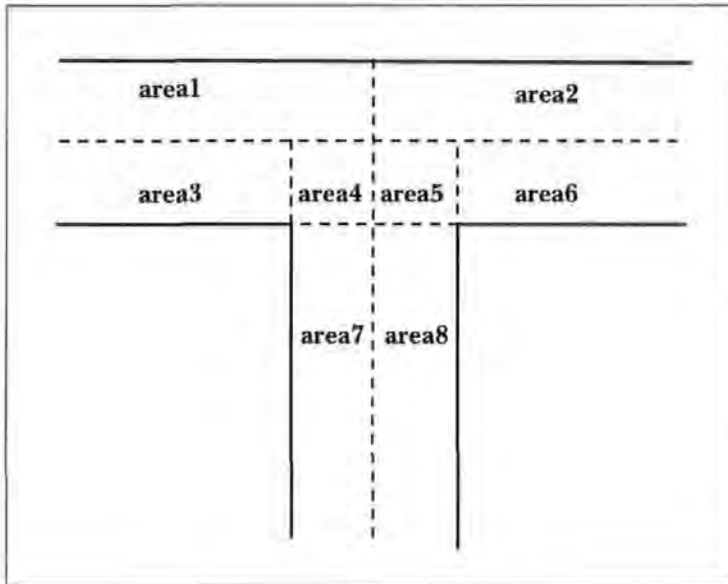


Figure 4-2 Direction Areas

Areas 1-3 and 6-8 are straightforward as they allow only a single flow of traffic. **area4** and **area5** are slightly different because traffic in these areas could be moving in a different direction depending on whether it is turning or moving forward. For instance, traffic in **area1** can only travel in the easterly direction, while in **area4**, either northerly or westerly. The directions and areas are defined as follows:

```
set_junction_areas :-
    % set property into memory
    set_prop( poss_dir, area1, [east]),           % unidirectional
    set_prop( poss_dir, area2, [east]),
    set_prop( poss_dir, area3, [west]),
    set_prop( poss_dir, area4, [north, west]),     % bi-directional
    set_prop( poss_dir, area5, [west, south]),
    set_prop( poss_dir, area6, [west]),
    set_prop( poss_dir, area7, [north]),
    set_prop( poss_dir, area8, [south]).
```

To represent the motorists, there is the introduction of one object type 'CAR' in this domain, and it is defined as:

```
defaults types( ['CAR']).
```

whereby objects of the type 'CAR' have the following default settings:

```
%% Default Objects database
defaults mass( 'CAR', 1).
defaults size( 'CAR', 2).
defaults vis( 'CAR', 100).           % visibility distance
defaults maxSpeed( 'CAR', 30).       % maximum speed
defaults stopTime( 'CAR', 6).        % time taken to stop
defaults stopDistance( 'CAR', 40).   % distance taken to stop
```

Five cars are introduced and, initially, each given a behaviour plan (e.g. 'take first left turn', 'park there'). If no additional behaviour plan is given, then it just follows the road ahead. Cars which are on the move will have the property 'alive' and those which are parked or stopped have the property 'static'. Five instances are created as follow:

```
fact obj_type( car1, 'CAR').
fact property( car1, [alive]).       % moving (no turning)
fact location( car1, pt( 39, 28)).

fact obj_type( car2, 'CAR').
fact property( car2, [alive]).       % moving (no turning)
fact location( car2, pt( 152, 26)).

fact obj_type( car3, 'CAR').
fact property( car3, [static]).      % park there
fact location( car3, pt( 279, 26)).

fact obj_type( car4, 'CAR').
fact property( car4, [alive, left]). % turning left at the first junction
fact location( car4, pt( 279, 60)).

fact obj_type( car5, 'CAR').
fact property( car5, [alive, left]). % turning left at the first junction
fact location( car5, pt( 170, 174)).
```

Once these are constructed, the cars can then be displayed in the graphics window (Figure 4-3), and the system awaits input from the user.

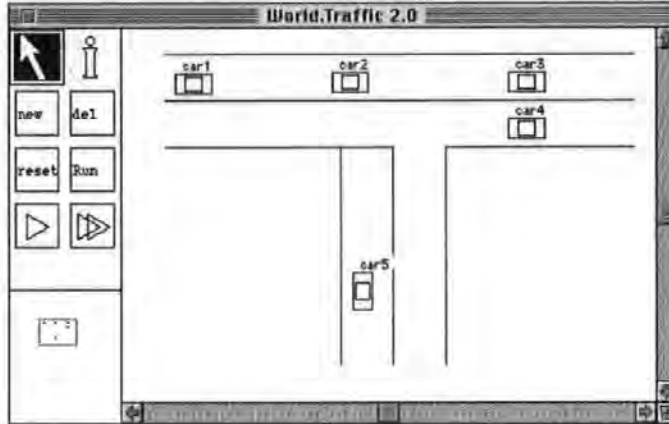


Figure 4-3 World.Traffic Graphical Representation

The drivers know the initial direction to go (by being in the allocated areas), and they also obey rules which describe the behaviour for dealing with events such as turning, slowing down when approaching a junction or coming close to the car in front:

```
% if a car is moving and ahead is clear, then moves forward
rule1 : if [object1, is_alive, and, object1, ahead_is_clear]
        then [object1, moves_forward].

% if a car is moving and ahead is not clear, then slows down
rule2 : if [object1, is_alive, and, object1, ahead_is_not_clear]
        then [object1, slows_down].

% if a car is stop, then stay there
rule3 : if [object1, is_stop]
        then [object1, stay_there].

% if a car is turning, then slows down
rule4 : if [object1, is_turning]
        then [object1, slows_down].

% if a car is stop waiting to turn but ahead is not clear, then stay there
rule5 : if [object1, is_stop, and, object1, is_turning,
            and, object1, ahead_is_not_clear]
        then [object1, stay_there].

% if a car is stop waiting to turn and ahead is clear, then turns
rule6 : if [object1, is_stop, and, object1, is_turning,
            and, object1, ahead_is_clear]
        then [object1, turns].

% if a car is parked there, then stay there
rule7 : if [object1, is_static]
        then [object1, stay_there].
```

Figure 4-4 Rules for World.Traffic

There domain predicates and operators needed in this domain, and they are:

<b>ahead_is_clear</b> <b>ahead_is_not_clear</b> <b>is_stop</b> <b>is_turning</b>	<b>moves_forward,</b> <b>slows_down,</b> <b>turns,</b>
---	--

**ahead\_is\_clear** and **ahead\_is\_not\_clear** are domain predicates for determining if the road ahead is clear or not. They are carried out by checking if there are any vehicles immediately in front. This depends on the visibility distance (**vis**) of each vehicle, and is confined to the ongoing traffic that is moving in front. At this stage of modelling, overtaking is not allowed (as stated in one of the special conditions in p. 114), so only the forward moving traffic is checked. It is estimated that an extra set of rules, predicates and operators would have to be defined to implement this feature. For instance, when overtaking is to be incorporated, all areas have the same potential of handling vehicles moving in any direction, and a more complex algorithm for determining whether ahead is clear or not is needed. Also when overtaking, a car does not only have to deal with the immediate traffic in front, but also any oncoming traffic.

This is the structure (though not the actual code) of the **ahead\_is\_clear** predicate:

```

Object ahead_is_clear :-
  get_objects_ahead( Object, List),      % get all the objects ahead of Object
  ( List = [] -> EXIT                     % if none then EXIT(ahead is clear)
  ; nearest_object( List, Nearest_obj),  % else get the nearest obj from List
    visible( Object, Nearest_obj, Ans),  % is the nearest object visible?
    Ans = NO -> EXIT                     % if No, then EXIT (ahead is clear)
    % else FAIL (ahead_is_not_clear)
  ).

```

**moves\_forward** and **slows\_down** are operators for advancing an individual vehicle one time slot, the distance travelled depending on the 'speed' of the vehicle. 'speed' is a property which has a value between 0 and 5, where 0 means stopping (for the predicate **is\_stop**) and 5 travelling at full speed. **moves\_forward** implies that the 'speed' of the vehicle is maintained (i.e. the current value is the same as previous), or accelerating (i.e. the current value is higher than previous). **slows\_down** means that the current 'speed' value is set lower than that previously defined.

**is\_turning** is a predicate that checks if a vehicle has a plan for turning and if it is approaching a junction. Since the junction areas are set and are known to the motorists, the predicate **approaching\_junction** can be used to check if a junction is approached:

```
Object is_turning :-
  property( Object, List),           % get the properties of Object
  ( member( [left], List)           % if List has either the property 'left',
    ; member( [right], List)),      % or the property 'right'
  approaching_junction( Object).    % and if approaching a junction
```

A vehicle possesses an additional property called 'direction' which records its direction at different time intervals. This additional property is useful for manoeuvring operators (such as **turns**) which use it to resolve the subsequent direction for any vehicle.

**turns** is another operator for advancing movements, and it is used in a rule that describes what happens when a car moves into the relevant area. For example in Figure 4-3, **car5** is travelling north, and has a property to take the first left turn at the next junction. When the driver arrives at the junction, steps into **area4** (provided the road is clear), then compares its direction and the possible directions of **area4** and turns accordingly. In this case, **car5** turns and drives in a westerly direction, see below:

car5 direction	turns direction	area4 [directions]	result
north	north + left = west	[west, north]	west [match]

And as mentioned earlier, one of the special conditions for the motorists is that:

- they either move vertically (north <-> south) or horizontally (west <-> east) within the road;

this is arranged for simplicity so that only four basic directions are considered (north, east, south, west). A table for the turning directions can be constructed based on the four directions as below:



car directions	+ turns	= turning directions
north	left	west
north	right	east
east	left	north
east	right	south
south	left	east
south	right	west
west	left	south
west	right	north

Figure 4-5 Turning Directions

The seven rules in Figure 4-4 produce convincing and predictable road traffic animation. Of course, more complex behaviours such as overtaking, crashing into each other by introducing human fatigue, random elements and so on can be achieved by constructing additional rules, or more importantly, a different and more complex road area layout. Some complex road layout may restrict drivers' vision, for example when vision is obscured by a building or a hedge. Other external incentives such as traffic lights could also be applied, and it would be interesting to see what might occur if some motorists were given a predisposition to jump red lights.

For a full listing of current database see **Appendix E**.

#### 4.1.2 Traffic Results

These are the results produced by the ESCAPE system for World.Traffic with various options selected. Times are measured in seconds by averaging several runs of the same animation using MacProlog's internal time resolver, based on the following statistics:

World:           **World.Traffic 2.0**  
 N° of Frames:   **12**  
 N° of Rules:     **7**  
 N° of Objects:   **5 (cars)**

	Blind Run		Interactive Run	
	Collision On	Collision Off	Collision On	Collision Off
Text On	27.60	26.33	36.67	35.47
Text Off	21.05	19.40	29.82	28.72
Interactive Run vs. Blind Run				28 %
Text On vs. Text Off				21 %
Collision On vs. Collision Off				5 %
"Full-option" vs. "Minimum-option"				43 %

Figure 4-6 Various Time (in Seconds) Taken to Run World.Traffic 2.0

And it can be seen in the following chart:

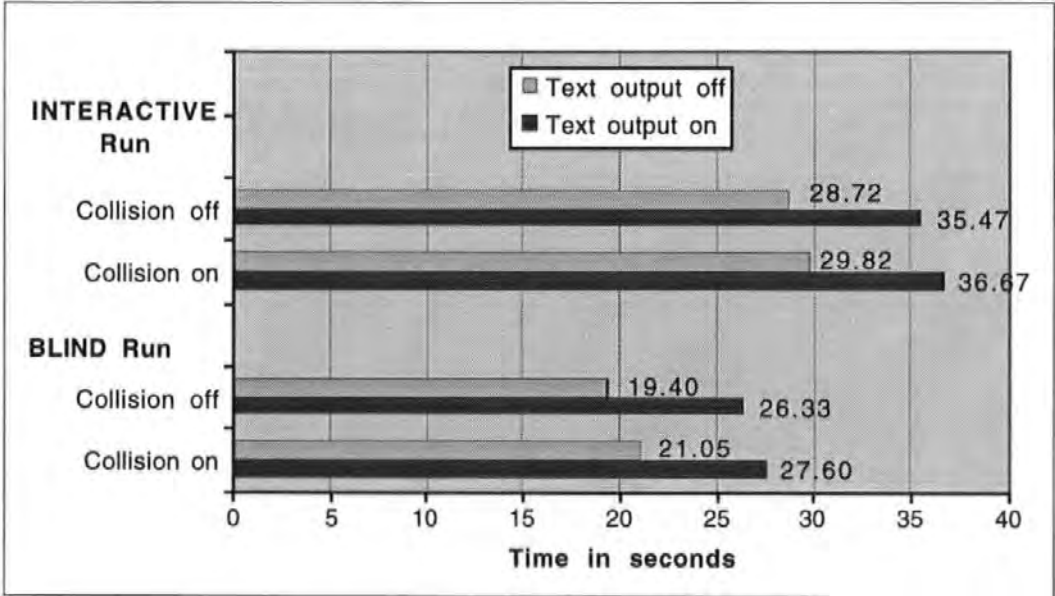


Figure 4-7 Time Chart for World.Traffic 2.0

\*"Full-option" = Interactive Run + Text Output On + Collision On and  
"Minimum-option" = Blind Run + Text Output Off + Collision On.

From the two figures above, by using the **Blind Run** option, on average, it runs 28% faster than the **Interactive Run** option. The **Text Output Off** option saves time up to 21% on some runs. Collision detection has not contributed any significant impact due to the smallness of the number of objects involved, 5% longer here with 5 objects compared to 33% in World.Bird with 16 objects (see Figure 4-12 p.130).

If collision detection is to be switched on at all times, the "Full-option" takes nearly 43% longer than the "Minimum-option". The results are similar to those found for World.Jelly because the pre-condition settings are rather alike. See § 4.2.2 for a comparison for all three domains.

Here are some comments on the keyframes as illustrated in Figure 4-8, showing the first 10 keyframes (out of 12) generated from the above run. In  $f_1$  (each car is labelled here so they can be followed with the explanations), **car3** on the top right hand corner is parked there, and **car2** is moving eastwards behind **car3**. **car2** soon detects that **car3** is not moving, and gradually reduces its speed, eventually coming to a halt (around  $f_7$ ) just behind **car3**, but within a safe distance. This safe distance is the result of the calculations of the **stopDistance** (valued at 40 units) and the **stopTime** (which is set to be 6) between the car and the car in front. Similarly, **car1**, which is behind **car2**, slows down as **car2** slows down, keeps on moving at a safe distance, and eventually stops at around  $f_9$  after **car2** has come to a complete stop.

**car4** is travelling westwards and is told to take the first left turn at the junction, and since there is no traffic in front of it, it just makes the turn as it arrives at the junction and continues its journey southwards. **car5** is travelling northwards, and is also told to take the first left turn at the junction. It slows down as it approaches the junction and eventually stops at  $f_5$  when **car4** is detected coming from the right, and **car1** is detected immediately in front. **car5** starts to turn as soon as **car4** has made its turn, and can see that **car1** is travelling on a different route, then continues in a westerly direction.

This produces a predictable animation as the objective is to demonstrate the effects of the rules (as a recognisable road scene) by positioning the vehicles in a predictable way.

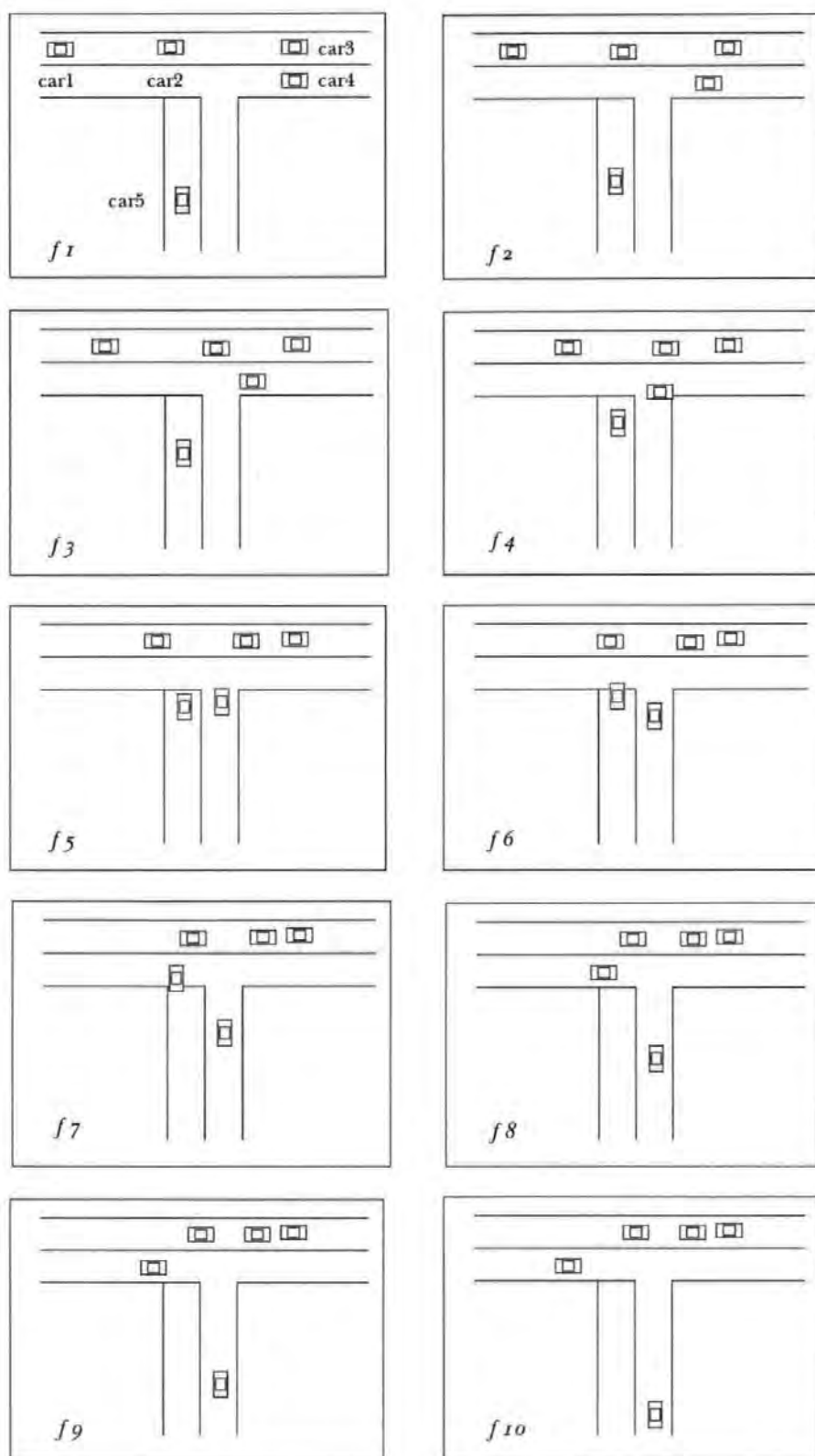


Figure 4-8 Keyframes for World.Traffic 2.0

Modelling of this domain has highlighted two important issues. In the WorldJelly example, points or regions in the spatial field were not significant, but in this one different regions are important. The background represents the road junction and it is relatively complex. It can be divided into road and non-road regions and within the road regions there are important distinctions to be made (for example, each side of the road, the region before the stop-line for traffic approaching the major road, etc.). The modelling and representation of the domain background becomes an important issue.

Also featured in this model is the ability for drivers to plan on the basis of their models of other drivers. When considering a move, each driver must calculate the current velocity and acceleration of every other vehicle to decide whether or not it is likely to interfere with their progress. In normal cases, if there is any possibility of collision then no move is attempted. However, if an attribute representing the degree of patience of the driver is added and it is reduced as their waiting time increases, then the result will be that the driver is prepared to make a move even if this depends upon vehicles slowing down. This interaction of agents is a novel feature for the model but it seems to introduce no particular difficulties. In a more complex situation, the strategy of moving one object at a time might not suffice and the type of solution proposed in the next example may be preferable here too.

A high-level strategy has also been experimented, which identified a number of 'drive-states' (e.g. 'free', 'turning', 'slowing'). All cars are in one of these states at any time. Rules have been written for the identification of particular traffic situations and which lead to the transition from state to state. Further rules have been written to describe the behaviour of cars within each driving-state. As a general approach this seems to be a good way to handle potential complexity.

A rendered image taken from this domain can be seen in **Colour Plate 2** (p. vii).

## 4.2 World.Bird

In this domain, a model is investigated where a flock of birds in flight that have to navigate around a number of fixed obstacles. Reynolds (1987) has developed a sophisticated animation of birds in flight using an algorithmic approach. It simulates the flock using an elaboration of a particle system in which each particle acts according to an identical algorithm so that each bird is considered as behaving according to the same constraints as every other bird. This approach assumes that the behaviour of the flock is simply the result of the interaction between the behaviour of the individual birds. The next position of each bird is calculated according to three simple rules:

- 1) *Collision Avoidance: avoid collision with nearby flockmates,*
- 2) *Velocity Matching: attempt to match velocity with nearby flockmates,*
- 3) *Flock Centring: attempt to stay close to nearby flockmates.*

The following two paragraphs are quoted from Reynolds' 1987 paper because it is felt that Reynolds' explanations cannot be made any clearer;

"Static collision avoidance and dynamic velocity matching are complementary. Together they ensure that the members of a simulated flock are free to fly within the crowded skies of the flock's interior without running into one another. Collision avoidance is the urge to steer away from an imminent impact. Static collision avoidance is based on the relative position of the flockmates and ignores their velocity. Conversely, velocity matching is based only on velocity and ignores position. It is a predictive version of collision avoidance: if the bird does a good job of matching velocity with its neighbours, it is unlikely that it will collide with any of them any time soon. With velocity matching, separations between birds remains approximately invariant with respect to ongoing geometric flight. Static collision avoidance serves to establish the minimum requirement separation distance; velocity matching tends to maintain it."

"Flock centring makes a bird want to be near the centre of the flock. Because each bird has a localised perception of the world, 'centre of the flock' actually means the centre of the nearby flockmates. Flock centring causes the bird to fly in a direction that moves it closer to the centroid of the nearby birds. If a bird is deep inside a flock, the population density in its neighbourhood is roughly homogeneous; the bird density is approximately the same in all directions. In this case, the centroid of the neighbourhood birds is approximately at the centre of the neighbourhood, so the flock

centring urge is small. But if a bird is on the boundary of the flock, its neighbouring birds are on one side. The centroid of the neighbourhood toward the body of the flock. Here the flock centring urge is stronger and the flight path will be deflected somewhat toward the local flock centre."

Reynolds' formulation of the above rules has been adopted and used to construct the World.Bird domain.

#### 4.2.1 Bird Database

For this domain, there are two types of object: **'BIRD'** and **'OBST'** (obstacles). The picture representations and default database settings can be organised in the **Domain: defaults** file as follows.

```
%% Default Objects pictures
set_pict_descs :-
    set_prop( 'OBST', _, circle( 0, 0, 10)), % is a circle with radius 10 units
    set_prop( 'BIRD', left, resource( 300)), % PICT resource ID is 300
    set_prop( 'BIRD', right, resource( 400)).

%% Default Objects database
defaults types( ['BIRD', 'OBST']). % 2 types of object in this domain

defaults name( 'BIRD', bird).
defaults mass( 'BIRD', 1).
defaults size( 'BIRD', 6). % size with radius of 6 units
defaults vis( 'BIRD', 100).
defaults maxSpeed( 'BIRD', 15).
defaults property( 'BIRD', [alive]). % with default property alive

defaults name( 'OBST', obstacle).
defaults size( 'OBST', 10). % size with radius of 10 units
defaults property( 'OBST', [static]). % with default property static
```

And for the instances, there are three birds and three obstacles. The properties and their initial locations can be included in **Domain: facts** file and defined as follows:



```

%% facts about the birds
fact obj_type( bird1, 'BIRD').
fact property( bird1, [alive]).
fact location( bird1, pt( 258, 178)).

fact obj_type( bird2, 'BIRD').
fact property( bird2, [alive]).
fact location( bird2, pt( 279, 162)).

fact obj_type( bird3, 'BIRD').
fact property( bird3, [alive]).
fact location( bird3, pt( 321, 190)).

%% facts about the obstacles
fact obj_type( obstacle1, 'OBST').
fact property( obstacle1, [static]).
fact location( obstacle1, pt( 232, 125)).

fact obj_type( obstacle2, 'OBST').
fact property( obstacle2, [static]).
fact location( obstacle2, pt( 175, 204)).

fact obj_type( obstacle3, 'OBST').
fact property( obstacle3, [static]).
fact location( obstacle3, pt( 91, 162)).

```

When the above **defaults** and **facts** files are set, the user can immediately see the objects appearing in the animation window as illustrated in Figure 4-9.

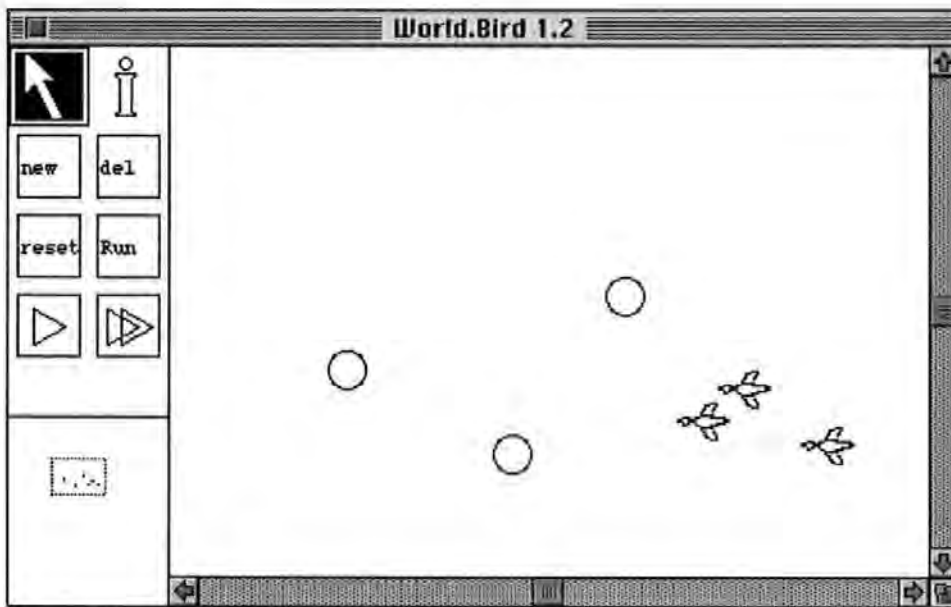


Figure 4-9 Animation Window for World.Bird



In this domain, the birds are set to fly only in one direction (e.g. toward the west), and while doing so, they must obey the three following rules:

```
% if an object is alive, then just meanders forward
rule1 : if [object1, is_alive]
        then [object1, meanders_forward].

% if an object is alive, then try to flock towards the centre of the flock
rule2 : if [object1, is_alive]
        then [object1, flock_centering].

% if an object is static, then stay there (mainly for the obstacles)
rule3 : if [object1, is_static]
        then [object1, stay_there].
```

When compared with the three rules suggested by Reynolds (in p. 125):

	REYNOLDS' MODEL	ESCAPE
Collision Avoidance	YES	built-in
Velocity matching	YES	<b>rule1: meanders_forward</b>
Flock centring	YES	<b>rule2: flock_centering</b>
others	-	<b>rule3: Static rule</b>

**Figure 4-10** Rules Implementation Comparison Between Reynolds' Model and ESCAPE

All Reynolds' rules are implemented in ESCAPE (in some way). However, there is an extra rule in ESCAPE, that is the Static rule (**rule3**, which states: if an object is static, then stay there).

The origin of this Static rule is from the first two examples, World.Jelly and World.Traffic, where it was realised that objects can change their property from one state to another (for example, from alive to static) through the action clause of rule. In World.Jelly, a world where there are predators and potential preys, one living object might become static when captured/eaten; and in the latter, a world where a moving car might stop at a junction and become static.

There are two options for the implementation of the Static rule. One is to build it into the system as basic (similar to collision detection), and the other to write it explicitly as a rule. Since movements of objects (having the property **is\_alive**) are put into rules (e.g. **meanders**, **meanders\_forward** and so on), why not have a rule for static objects? It just happens that all the examples featured here have permanently non-moving static objects. There are occasions when static objects could move under some conditions, for example, hit by a moving object. The advantage of describing such behaviour as a rule is that it can be changed easily to accommodate the requirements without having to modify the inference engine.

For a full listing of current database including how **meanders\_forward** and **flock\_centering** operate, see **Appendix F**.

### 4.2.2 Bird Results

The database for the objects are slightly different for this example. Firstly, the '**BIRD**' picture representation is changed from the picture of a 'bird' to merely a 'black dot' in the animation window to gain processing time, and the number of objects has been increased from 6 to 16 to show the effects better. The three rules still remain the same.

These are the results produced by the ESCAPE system for World.Bird with various options selected. The times are measured in seconds, by averaging several runs of the same animation using MacProlog's internal time resolver, based on the following statistics:

World:	<b>World.Bird 2.0</b>
N <sup>o</sup> of Frames:	<b>16</b>
N <sup>o</sup> of Rules:	<b>3</b>
N <sup>o</sup> of Objects:	<b>16 (9 birds and 7 obstacles)</b>

	Blind Run		Interactive Run	
	Collision On	Collision Off	Collision On	Collision Off
Text On	206.27	134.28	238.25	165.97
Text Off	162.95	104.23	199.55	138.48
Interactive Run vs. Blind Run				18%
Text On vs. Text Off				19%
Collision On vs. Collision Off				33%
"Full-option" vs. "Minimum-option"				32%

Figure 4-11 Various Time (in Seconds)Taken to Run World.Bird 2.0

And it can be represented in the following chart:

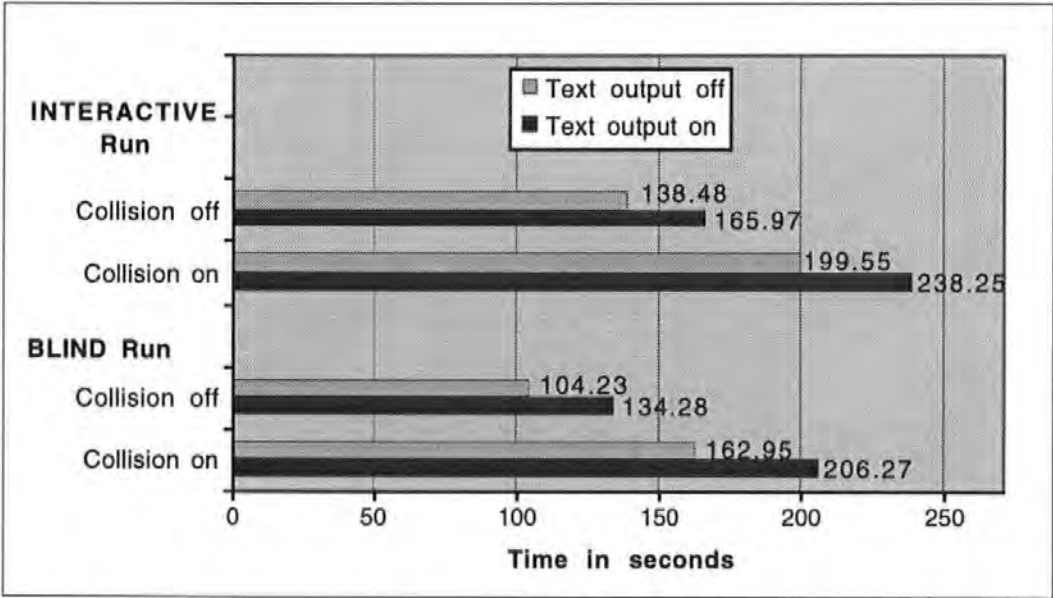


Figure 4-12 Time Chart for World.Bird 2.0

\*"Full-option" = Interactive Run + Text Output On + Collision On and  
"Minimum-option" = Blind Run + Text Output Off + Collision On.

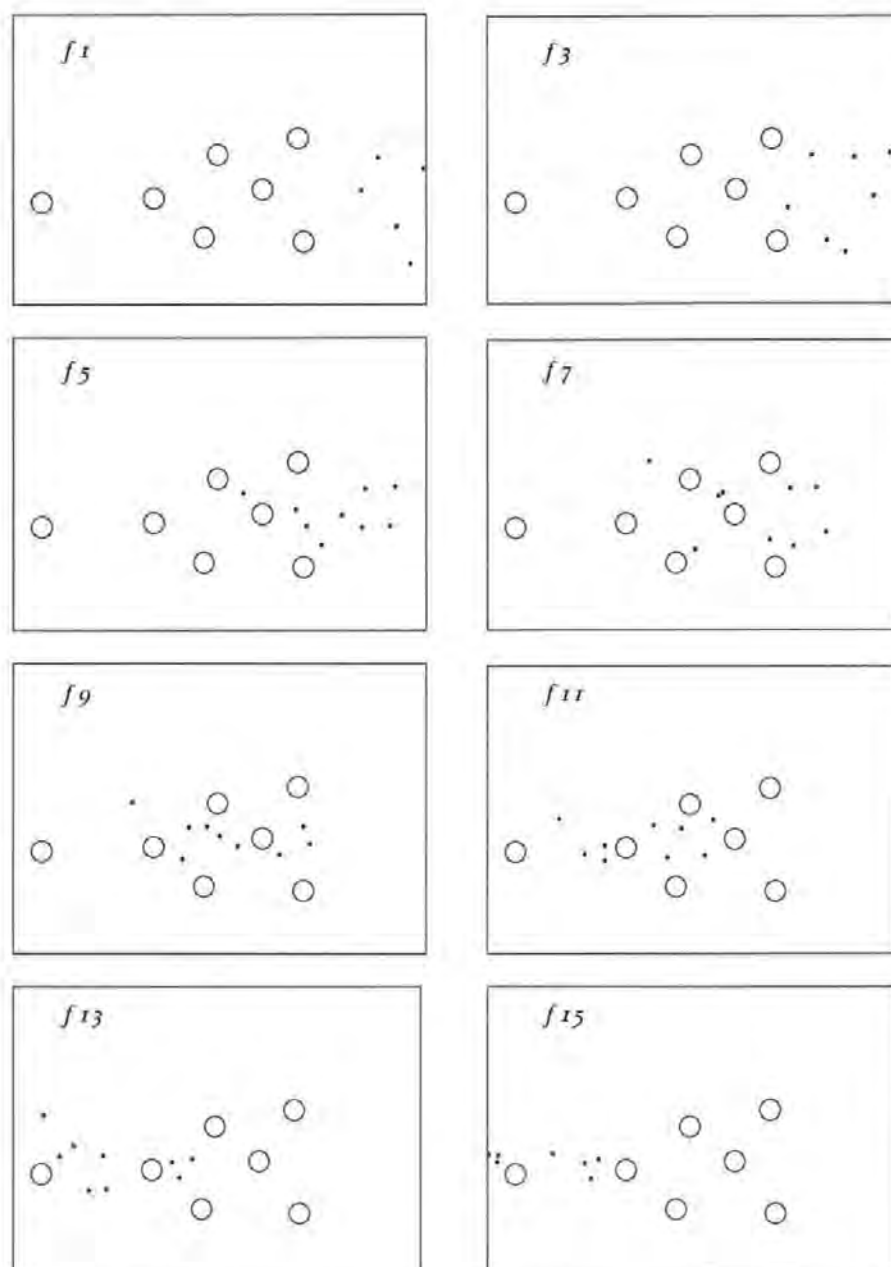
From both the figures above, it can be seen that by using the **Blind Run** option, on average, it runs approximate 18% faster. The **Text Output Off** option, which can save time up to 19% on some run. Collision detection however has contributed a significant impact due to the numerous number of objects involved (33% here with 16 objects comparing to, 1% in World.Jelly with 6 objects - see Figure 3-18 in p. 108). It can be concluded that, as objects increase, collision detection algorithm becomes apparent.

Here is a comparison between the all three domains featured in the examples:

	World.Jelly	World.Traffic	World.Bird
number of frames	10	12	16
number of rules	12	7	3
number of objects	6	5	16
Time for "Full-option"	60.3	36.67	230.25
Time for "Min-option"	38.62	19.4	104.23

The relevant of this exercise is to link the performance of the system influenced under different options, so that the viewer becomes aware of what contributes to the production of an animation using this kind of system. From the table, a pattern can be seen whereby as the number of objects increases, the time it takes to complete an animation increases almost uniformly. By contrast, the number of rules and the number of frames do not seem to contribute much to the computational time. So writing fewer rules does not improve the execution time, but reducing the number of objects does.

Figure 4-13 shows 8 keyframes (out of 16) generated from the above run:



**Figure 4-13** Keyframes for World.Bird 2.0 (with obstacles)

Initially, the birds (presented by black dots) are deliberately placed at distance from each other, so the effect of flock centring can be seen as the animation progresses: they stay together as a group, occasionally split up when encountering the obstacles (circles), but rejoin again whenever possible, while maintaining a safe distance between

them (effect of collision avoidance). They are also flying roughly at the same speed (effect of velocity matching) as each other, as none fly too far ahead, nor too far behind.

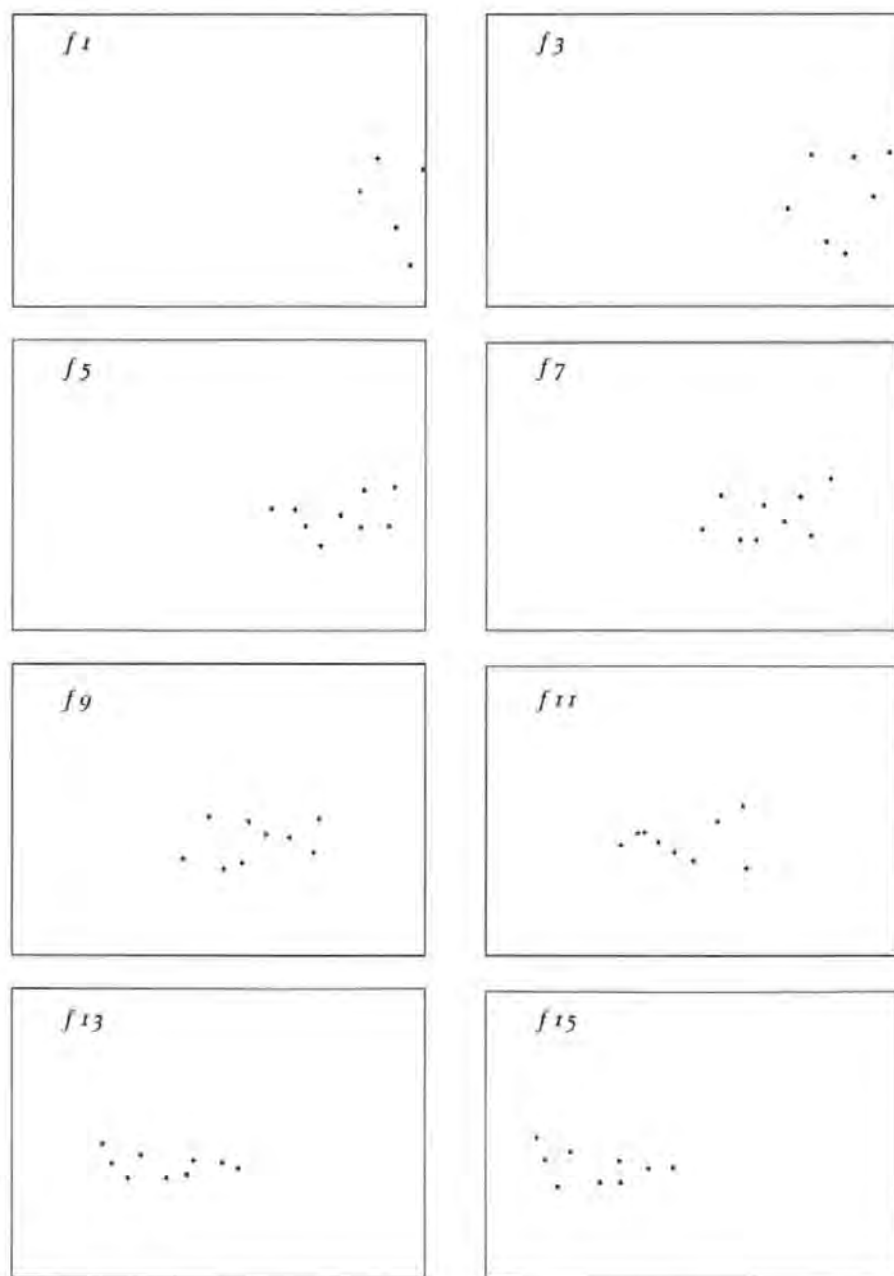


Figure 4-14 Keyframes for World.Bird 2.1 (without obstacle)

In Figure 4-14, the keyframes were produced using the same set of rules and the same number of birds, but without the obstacles.

When comparing the last keyframe (*fr5*) between Figure 4-13 and Figure 4-14, the birds facing obstacles tend to fly 'faster', this is caused by a number of factors. As an individual bird tries to match its speed with the rest of the flock, when combined with the quick change of direction in order to swirl around and to avoid colliding into an obstacle, this combination of actions makes the bird fly forward marginally faster.

It can be seen that in Figure 4-13 the flock sometimes splits apart to go around an obstacle. This shows one of the advantages for adopting the situated action ordering approach over 'follow the leader' ordering (§ 3.4). Flock centring allows simulated flocks to bifurcate. Whilst an individual bird stays close to its nearby neighbours, it does not matter if the rest of the flock turns away. More simplistic models proposed for flock organisation (such as a central force model or a follow the designated leader model) do not allow splits.

These principles are followed in this approach, but instead of the procedural approach to specify each bird's behaviour, they are expressed in rules. By writing a few rules concerning how birds position themselves with respect to the other birds around them, it is possible to generate possible moves for each bird. In this case the sophistication really lies in the method of resolution for it seems infeasible to allow each bird to determine its own position. The major reason for this is that the overriding principle of flocking motion is that there are never any collisions. If an approach is adopted whereby the position of each bird is determined in turn and an unavoidable collision is identified when placing one of the later birds, the system would have to backtrack to place a previous bird in another place. At present, finding a new place depends upon a new random number being generated. A random procedure is not really wanted here for it may take a very long time before it comes up with an acceptable solution. If a crowded situation is likely to occur it is preferable to calculate an optimal solution rather than hoping to hit upon a set of acceptable placements by chance. Therefore, it is our intention to introduce a resolution algorithm for this domain that will consider all birds together and find a solution with least penalties.

A rendered image can be seen in **Colour Plate 3** (p. vii).

### 4.3 Summary

The examples included so far establish that convincing animations can be generated using expert system techniques, and that they can be embedded in an environment that emulates those proficiencies provided by more sophisticated software. Because the environment package is domain selective, 'plug-ins' of different domains can be chosen and this gives us the ability to test the more general environment on several different domains and types of animation problem.

The following are "World" dependent (i.e. information derived varies from "World" to "World"); the definition of the environment, the types of object, the graphics representation of object, the instances of object, and the user defined predicates and operators.

As may have been expected, the performance of the system depends on the quality of graphical output, the quality of log output and, especially, the collision detection calculations - the significance is particularly clear as the number of object increases.





## CHAPTER 5

# CONCLUSIONS



"Note that wanting to make computers *be* intelligent is not the same as wanting to make computers *simulate* intelligence."

Patrick Henry Winston on AI (1977)

### Chapter Overview

The demonstrations of the ESCAPE system have helped clarify the type of architecture (Figure 2-1) needed, the functionality that might be expected and the users who may be able to successfully exploit such a system. It provides the opportunity to use a new kind of knowledge, designed for the animators to simulate the behaviour of objects within a modelled environment.

The three examples (World.Jelly, World.Traffic and World.Bird) illustrated some potential for rule-based enhancements to animation software. They have shown that the architecture can be used in different domains, and have enabled us to begin to separate domain specific problems from more general ones.

Some important issues were raised during the implementation of the system that had not been foreseen, for example, the significance of the respective textual and graphical interfaces, the use of different chaining methods for the inference routines, and the different methods for resolving possible solutions suggested by the inference engine.

This final chapter also provides some useful insights into the limitations of the approach, the types of animation problem where it might be useful, the possible future work that needs to be done, and some suggestions of utilising the output generated by the system with more purposeful generalised graphical languages available in the current graphics industry.

### 5.1 Research Context and Findings

In behavioural animation, an animator using the computer usually does not have any programming skills, but has some good ideas of how his/hers agents to behave. One of the main difficulties here is the representation of the information which allows these agents to act in a purposeful manner - as the animator intended. ESCAPE is a prototype system with an 'enhanced environment', designed especially for the animator in the pursuit of capturing behavioural animation within a multi-agent environment. This enhancement is developed by incorporating a rule-based system with conventional computer animation techniques so that the behaviour of the agents can be driven by the inference engine, and the physical side of the animation by the conventional counterparts. The rule-based system uses English-like rules which allow the user to read and write rules easily. It is claimed that through the kind of experimentation described in this thesis, our understanding of some of the problems in behavioural animation has been increased.

There is not one particular method of animation that is preferable in all situations, and in an ideal multi-agent animation environment where modelling intentional behaviour is of main concern it should be possible to influence the resulting animation in at least three ways: by directly manipulating agents within specific animation frames; by specifying their properties and locations formally in the database, and by providing rule-based constraints on their behaviour using an expert system approach. All three have been allowed for in the ESCAPE system, however, it is the implication of using the rule-based approach that is highlighted as the major contribution of this research.

Within the ESCAPE system, a user can describe the behaviour of the agents in English-like rules using the textual interfaces and the proposed rulebase. The utilisation of natural language technology in the rulebase is aimed at the user, so that their knowledge about agents' behaviour is not only easier to describe and understood (by both the user and the system), but is also capable of producing realistic behavioural animations.

Realistic animations need planning, the planning process of the animation can be done either explicitly, or implicitly by the animator. In traditional explicit techniques such as keyframing, the outcome of the animation depends entirely on the animator moving each object to the desired locations usually using a graphical interface. This

works well in animation where the number of objects remains small, but in a multi-agent environment, problems arise when the number of objects increases because every single object has to be dealt with by the animator. One way to address this problem is to write routines to deal with the objects implicitly.

Implicit methods are generally implemented using algorithmic approaches. Algorithmic approaches work well in producing physically accurate models, giving the animator convenience control by applying scripts to obtain certain desired behaviour, but they lack the ability to represent intentional behaviour where the modelling of complex behaviour in a multi-agent environment is of concern. One way to do this is to consider some ideas from AI and rule-based systems.

In complex scenarios such as flocking, algorithmic techniques are difficult to organise because the individual behaviour as well as the group behaviour has to be captured. Reynolds (1987) for example has developed AI-based realistic flocking behaviour using an object-oriented approach: he represents his 'rules' in the form of an abstract programming language, but not in the way that an animator or a domain expert can understand easily.

One of the features of rule-based systems is that they are suitable for handling symbolic meanings and domain-oriented problems. A dedicated rulebase can therefore be constructed incorporating a natural language syntax to handle simple grammatical meanings, so that the symbolic form of the rules can be readily comprehended by the user. Incrementality is another attractive feature of the rule-based approach, so that new knowledge can be added to the system at any time to build up the knowledge database. This knowledge can be presented in the form of rules, facts and properties that describe an agent.

Once properly set up, the rule-based part of the system can be left to run unattended and will automatically generate an animation. However, the expert system component is not there to replace skilled animators, but rather to assist them. The animators can interrupt the generated animation at any time by manual manipulation to override it. This coexistence between the user interactions and the expert system can be seen as a context of working practice for the animators, as the animators have control over the rules and can experiment with them, observing their effect on the animation.

The kind of architecture (§ 2.1) described in this thesis allows us to adopt a methodology that will begin to separate domain specific problems from more general

ones. The more general ones are built into the ESCAPE system which include the main inference engine, the natural language interpreter and other shared system routines, and the domain specific ones are built into separated modules called domains or worlds. To make a readily usable system it is necessary to provide a good set of general predefined predicates and operators in terms of which domain-specific behavioural predicates may be defined. A useful set has been identified along the way, but it is by no means complete.

### 5.2 Contributions

The ESCAPE system is a fully working prototype with its limitations (§ 5.3), and it is capable of generating convincing animation sequences as illustrated in the examples throughout the thesis. Following is a summary of the successful features of the current implementation of ESCAPE system. This implementation:

- \* confirms the software architecture for the incorporation of a rule-based system with a more conventional keyframe animation;
- \* establishes the needs and the roles of different level of users: animators, domain experts and software experts;
- \* provides a working environment incorporating a rule-based element based on the architecture mentioned above;
- \* enables us to separate domain-independent materials from domain-dependent ones;
- \* contains a formal language for the expression of rules;
- \* incorporates interfaces for user interaction;
- \* integrates GUI, symbolic and rule-based approaches; and
- \* has been successfully demonstrated on three model worlds.

Some important issues were raised during the implementation of the system that had not been foreseen, these include the significance of the respective textual and graphical interfaces, ordering, hybrid systems, and the quality of graphics output.

The textual interfaces are closely linked with the rulebase, they provide a high level of readable natural language interfaces aiming to make the tasks of rule-writing (edit, amend and delete) easier for the animator and the domain expert. Initially, these interfaces were designed to show only the available options in some special popup menus, so that they could be manipulated with the use of a mouse. It was hoped that by

## Chapter 5: Conclusions

minimising the use of other input devices such as a keyboard, any possible erroneous syntax input by the user could be minimised. Later, it was discovered that sometimes the use of a keyboard was inevitable, when for example a new word was to be introduced into the rulebase, so a syntax checker was incorporated into the system to account for that. The final textual interfaces allowed for both menus and keyboard input methods. For an experienced user of the system, the latter method would seem to be more attractive because a particular rule can be viewed in its entirety and be accessed directly.

The reason to use natural language processing techniques is that they provide a formal mechanism for expressing our verbal knowledge, for instance in describing abstract information of how something should behave. Since the formalisation of the complete syntax and meaning of the English language would not be possible within the scope of this research, the adaptation of relative simple subsets of natural language has been adequately formalised and implemented in the ESCAPE system. Even with the limited syntax capability, this natural language extension has allowed us to represent (some of) our verbal knowledge as simple rules and these have been successfully applied to the example animations.

The graphical interfaces are mainly designed for the animators for indirect access to the domain database which was originally set up by either the domain expert or the software expert. The functionality of the graphical interfaces includes the manipulation of agents, controlling the animation by means of using the animation tools, and choosing the current environment settings. The main animation window allows the animators to manipulate the agents directly and they can immediately observe the actions being taken, as each agent is represented by its physical location rather than its precise coordinates in numerical notation.

The order in which rules are applied to object is especially significant. For example, given two rules in that order (rule1, rule2):

rule1: if conds1 **then** actions1.

rule2: if conds2 **then** actions2.

The inference engine reads in a rule from 'top' to 'bottom', in this case if action1 would result in conds2 being true, then action2 will be performed. However, if they are in the reverse order, then action2 will not be performed. It is important that the user has an understanding of how expert systems work.

Rule-based systems seem more appropriate for some types of problem than others. As some researchers discovered (Koga *et al.* 1994, Rijpkema & Girard 1991), an expert system was useful for dealing with complicated tasks where strategic planning was involved, for example determining a grasping strategy, but a procedural approach was best suited to the finer movements involved, for example in making contact. A rule-based approach is not likely to serve well in areas that require much mathematical calculation, but an algorithmic approach is also unlikely to serve well where strategic planning is required. A possible direction is therefore the flexibility of *hybrid systems* (see § 1.3), incorporating both algorithmic and rule-based approaches.

Since ESCAPE system was developed using MacProlog, the poor quality of graphics output from MacProlog was not fully realised until the expert system elements have been implemented. This can be seen as a limitation rather than an inadequacy, and this will be discussed in the next section.

### 5.3 Limitations & Future Work

The small rulebases in the examples raise the question of whether the approach will scale up. The question here is not primarily a technical one, for there is little problem (except execution time) in presenting the inference engine with 250 rules instead of 15. The question is more whether domain experts and animators can comprehend what the system is doing if it contains a large number of rules. In this respect, it is believed, the interesting research question is how animators might use this new facility creatively, and this will depend crucially on their ability to understand the effect of individual actions. The current text-based log file may give some insight into the workings of the inference engine but it is not always possible to quickly extract from it a clear understanding of why an object is behaving in a particular manner.

The major problem is not the ability of the inference engine to cope with things on a larger scale but the ability of users to visualise the decisions the system is making. To this end there is a need to develop specific techniques for visualising the actions of the inference engine itself. For debugging purposes, in the future, it is envisaged that the user would be able to switch on an option that makes visible the vectors acting on an object at each stage in the inferencing which relates them to the rules from which they originated. There is even a suggestion of a visual representation for rules that might be called up and assigned to objects, possibly with strengths that may be set by the

animator. Several researchers have contributed this visual way of representing expert system for analysis, debugging and understanding (Selig 1990, TPM<sup>10</sup>, Yao *et al.* 1990).

At present, the planners developed in ESCAPE system only use very basic search strategies (§ 3.4: **Ordering** and § 3.5: **Results**) and are integrated fixed into the system. An animator activates any one mode at a time by selecting from a selection of pre-defined options in the menu. These strategies are completely uninformed in the sense that they do not use any domain-specific knowledge in choosing among alternatives. In future, a second level of rulebase can be incorporated into the system, so an extra set of rules can be used to describe strategic planning. Instead of selecting an option from the menu, the animator can construct a custom-made planner by writing rules to decide:

- what order of the goals are attempted;
- which alternative action will be tried to achieve the given goal; and
- which of the alternative regressed goal sets to consider next.

This aims to give the animator the provision for higher level planning and greater control over the animation, and an opportunity to try out different strategies for the agents.

LPA Prolog for the Apple Macintosh was chosen for the development of the ESCAPE system, it provides a full set of operators for graphics windows and interactivity, as well as DCG as an extra bonus. The greatest limitation of this choice is the software's approach to graphical output. The agents in the animation are represented by PICT resources which are contained within their own rectangle. Whenever an agent is moved on the screen the entire rectangle becomes white and is then re-drawn resulting an extremely jerky animation. This is adequate for quick previewing purposes, but the only solution for a smooth preview is to use rendering software, such as Swivel 3D<sup>TM</sup>.

---

<sup>10</sup> TPM (Transparent Prolog Machine), a debugging tool for MacProlog. By Fred Kwakkel. Information about TPM for Macintosh can be obtained from Human Cognition Research Laboratory, The Open University, Milton Keynes MK7 6AA, U.K.



### 5.4 Integration Into Other Software

The existing ESCAPE system is a prototype and does not produce quality output. To do this the approach needs to be integrated into a more sophisticated animation production system. The architecture that would be needed for this to be done has been described, the only issue to be resolved would be the interface language between the systems.

Given this vision of a fully working system and the limited resources available, it became necessary to define a project that addressed only the most central issues and utilised the most effective software for the task. Though the eventual output will need to be fully-rendered three dimensional colour graphics, it is not necessary to go to these lengths to demonstrate the principles of rule-based constraints. What is required is that the system produce a full description of each frame in terms of the objects involved, their precise location and a list of their properties, that some visualisation or preview facility is available, with the possibility of linking to a more sophisticated 3D modelling and rendering software package. One possible future extension would be to produce descriptions in more generalised languages such as VRML<sup>11</sup> (Fiévet 1995) or Java<sup>TM12</sup>.

VRML is an open, platform-independent file format for 3D graphics on the Internet. Similar in concept to the Web standard for text, HTML<sup>13</sup>, VRML encodes computer-generated graphics into a compact form for transportation over a network. As with HTML, a user can view the contents of a file - in this case an interactive 3D graphics file - as well as navigate to other VRML "worlds" or HTML pages. However, VRML is an emerging standard for 3D-model definition, but it does not yet support transformation, so any transformation has to be done from within the VRML browsers (e.g., WebSpace<sup>TM</sup> Navigator<sup>14</sup>, and WhurlWind<sup>TM15</sup>); or to have the ESCAPE system to produce a full description of the animation in Java.

---

<sup>11</sup> VRML (Virtual Reality Modelling Language). Further information and relevant bibliography on VRML can be obtained at the Internet URL <http://www.ncsa.uiuc.edu/General/VRML/>

<sup>12</sup> Java<sup>TM</sup>, see SUN MicroSystem's Java home page at the Internet URL <http://java.sun.com/>

<sup>13</sup> HTML (HyperText Markup Language). Comprehensive online information on HTML can be accessed at the Internet URL <http://lcweb.loc.gov/global/html.html>

<sup>14</sup> WebSpace<sup>TM</sup> Navigator, a VRML browser available for most platform, see SGI's (in Europe) home page on the Internet at <http://www-europe.sgi.com/>

Java is an object-oriented programming language intended to be used in networked/distributed environments, its neutral architecture means that the same version of the application encoded in Java will run on all platforms (SUN, SGI, IBM, Macintosh, PowerPC etc.), hence making it ideal for the net. Java applications can be incorporated into any HTML files and can open and access data across the net via URLs with the same ease that programmers are used to when accessing a local file system. To take full advantage of these features, the use of HotJava™ Browser (see footnote 12, p. 144) is highly recommended.

### 5.5 Conclusions

Would rule-based systems do a *better* job than traditional methods such as a simulator in producing behavioural animations? Simulators written in high-level programming languages or specialised simulation languages can, in theory, handle systems of any complexity. These simulators are still black boxes, however; they take numbers as input and generate numbers as output. There is a lot of information about the system that cannot be obtained in this way, for example, they can predict the occurrence of events such as traffic flow, but they do not tell us how to handle such events. Perhaps most importantly, a numerical simulator provides no interpretation or explanation of its output; it just generates numbers and leaves their interpretation to the user.

By integrating a rule-based system into the ESCAPE system, it has enabled the possibility of handling the interpretations and explanations to a certain limit. The generated text-based log files keep track of every step of the inference routine, showing which rule is being fired and what the outcomes are, and the interpretation of these can be followed in a (limited) natural language format. Provided the rulebase is of a 'manageable size' (comprehensible by the animator), the animator should not find difficulties in following the explanations. This approach has also shown the potential of giving the animator valuable information especially for debugging proposes.

The comparison between using algorithmic approaches and using a rule-based approach for representing multi-agent worlds is not based upon their respective claims

---

<sup>15</sup> WhirlWind™, a VRML browser available for the Apple Power Macintosh, a free copy can be obtained from the ftp site: <ftp://ftp.sdsc.edu> in the `/pub/vrml/software/browsers` directory.

to algorithmic completeness or efficiency, but rather on the ease with which end users may express their knowledge and control their animations with a minimum of technical knowledge (i.e. very little or no programming knowledge). For example, an (English) expert in the behaviour of bees who does not have any programming knowledge, can operate the system by expressing his/hers expertise about the behaviour of the bees in English-like rules, without having to spend time trying to learn the programming language.

It is not proposed that a rule-based approach is suitable for all modelling or animation needs. There are many aspects of modelling that are best expressed algorithmically, but with the limitations mentioned above. However, it is believed that there are at least two types of problem where a rule-based approach could have distinct advantages.

One is the creation of a realistic animated background scenery, where it can be used in stand-alone mode to produce convincing, non-repetitive animations where detail is not particularly significant, e.g. World.Traffic, situations like this can be advantageous to real-time flight simulator especially when a street scene is present, the system can automatically produce a realistic simulation of traffic flow without having to prepare the animation earlier. The other is strategic planning. Strategic planning is an important aspect of modelling in some domains and some system such as this could be seen being included in a *hybrid system*.

To conclude, it is argued that, animating sophisticated beings with minds of their own requires a modelling environment that can represent internal mental states and internal mental processes. It is believed that in order to model and produce behavioural animations in a complex multi-agent environments, there is a need to enhance current applications software to include the ability to model cognitive mechanisms and processes. One way to do this is to take some ideas from Artificial Intelligence and rule-based systems and to provide an additional tool whereby users can define properties, predicates, operators and rules to determine the behaviour of objects. It is not possible, at this point in time, to determine the best representations and strategies for all domains, or even for each particular domain, but the precise nature of the approach will develop through experimentation of the kind that has been described in this work.

## APPENDICES

## APPENDIX A

# EXPERT SYSTEMS

### A.1 An Overview of Expert Systems

An expert system (Bratko 1990, Clocksin & Mellish 1981, Jackson 1986, Kononenko & Lavrac 1988, Lloyd 1984, Marcus 1986) is a computing system capable of representing and reasoning about some knowledge-rich domain. With a view to solving problems and giving advice, some more famous examples have been in the areas of internal medicine or geology (Shapiro 1987). An expert system can be distinguished from other kinds of AI program by the following characteristics:

#### A.1.1 Knowledge Acquisition

Knowledge acquisition is defined as 'the transfer and transformation of potential problem-solving expertise from some knowledge source to a program', which sometimes is also known as 'machine learning'. Learning programs associated with expert systems differ considerably in the extent to which the program learns by being told, by modifying or manipulating what it already knows, by induction from some set of examples, or by discovering new concepts.

#### A.1.2 Knowledge Representation

Knowledge representation is a substantial sub field in its own right on the borderline between AI and cognitive science. It is concerned with the way in which information might be stored in the human brain, and the (possibly analogous) ways in which large bodies of knowledge can conveniently be stored in data structures for the purposes of symbolic computation (non-numeric computations in which the symbols can be constructed as standing for various concepts and relationships between them).

### A.1.3 Knowledge Application

This related to the issues of planning and control in the field of problem solving. Expert systems design involves paying close attention to the details of how knowledge is accessed and applied during the search for a solution. Knowing what one knows, and knowing when and how to use it, seem to be an important part of expertise; this is usually termed 'meta-knowledge', i.e. knowledge about knowledge.

Different strategies for bringing domain-specific knowledge to bear will generally have marked effects upon the performance characteristics of programs. Most knowledge representation formalisms can be employed under a variety of control regimes, and expert systems researchers are continuing to experiment in this area.

### A.1.4 Generating Explanations

The whole issue of how to help a user understand the structure and function of some complex piece of software relates to the comparatively new field of human/computer interaction, which is emerging from an intersection of AI, engineering, psychology and ergonomics. The contribution of expert systems researchers to date has been to place a high priority upon the accountability of consultation programs, and to show how explanations of program behaviour can be systematically related to the chains of reasoning employed by rule-based systems. This issue sometimes goes under the name of 'transparency', i.e. the ease with which one can understand what the program is doing and why.

## A.2 Inside An Expert System

The driving mechanism of an expert system consists of three major parts, namely, the inference engine(A.2.1), the rules (A.2.2) and the controls (A.2.3). The resultant programs consist of a number of relatively independent modules (e.g. rules, structures or clauses) which are matched against incoming data and which manipulate data structures.

Although expert systems research has grown out of more general concerns in Artificial Intelligence, it still maintains strong links with related topics in its parent discipline. Some of these links are outlined below:

## Appendix A: Expert Systems

Expert systems are known for the following features:

- *problem-solving*: capable of using domain-specific knowledge.
- *user-interaction*: which includes explanation of the system's intentions and decisions during and after the problem-solving process.
- *rules*: which allows for the if-then scenario, giving us the choice of using two basic ways of control: backward chaining, and forward chaining.

### A.2.1 Inference Engine

There are three essential ingredients to any such engine:

1. A collection of modules which are capable of being activated by incoming data which matches their 'trigger' patterns.
2. One or more dynamic data structures that can be examined and modified by an active module.
3. An interpreter that controls the selection and activation of modules on a cyclic basis.

For the inference engine to recognise these modules, a *precedence order* would have to be constructed. See **Appendix B.3** for how this is assembled.

### A.2.2 Rules

The rules are a formalism which saw some use in automata theory, formal grammars and the design of programming languages, before being pressed into the service of expert systems.

A production system consists of a rule set, a rule interpreter that decides how and when to apply which rules, and a working memory (WM) that can hold data, goals or intermediate results.

Rules consist of condition-action pairs, for example:

if  $C1 \ \& \dots \ \& \ Cn$ ,  
then  $A1 \ \& \dots \ \& \ An$

They can be interpreted in two ways:

1. production rules (forward chaining): with the reading 'if conditions  $C1 \ \& \dots \ Cn$  are true, then perform actions  $A1 \ \& \dots \ An$ '.
2. logical implication (backward chaining): with the reading 'if you want to prove  $A1 \ \& \dots \ An$  are true, then one way to do this is to prove  $C1 \ \& \dots \ Cn$  are true'.

(see **Appendix A.2.3: Controls** for forward and backward chaining.)

These are the advantageous features within the *production rules*:

- *Expressibility*: each rule defines a small, relatively independent piece of knowledge.
- *Incrementability*: new rules can be added to the knowledge base relatively independently of other rules.
- *Degradability*: existing rules can be deleted from the knowledge base relatively independently of other rules.
- *Modifiability* (as a consequence of modularity): old rules can be changed relatively independently of other rules.
- Support system's *transparency*: the system's ability to explain its decisions and solutions.

### A.2.3 Controls

Controlling the behaviour of rule-based systems poses non-trivial problems. There are two general approaches to this: global and local control. A global control regime tends to be domain-free, in that the strategy employed does not use domain knowledge to any significant extent. Local control regimes tend to be domain-dependent, in that special rules are required which use domain knowledge to reason about control. Global techniques are usually 'hard-coded' into the interpreter, and therefore difficult for the programmer to change, while local techniques are often 'soft-coded' in the sense that the programmer can write explicit rules to create particular effects.

At the global level of control, production rules can be driven *forward* or *backward*. We can chain forward from those conditions that we know to be true, towards conclusions which the facts allow us to establish, by matching data in working memory (WM) against the left-hand side of the rules. However, we can also chain



backward from a conclusion that we wish to establish, towards the conditions necessary for its truth, to see if they are supported by the facts. In this case, we match special goal statements in WM against the right-hand side of rules, modifications to working memory then manipulate these goal statements (e.g. replacing them with subgoals), as well as modifying patterns of data.

*Forward chaining* does not start with a hypothesis, but with some confirmed findings. An example of this type of expert system is R1 - now called XCON (McDermott 1980), which incorporated about 1000 if-then rules needed to configure orders for Digital Equipment's VAX computers and eliminated the need for DEC to hire and train many new people to perform a task that had proved difficult and that had resisted solution by conventional computer techniques.

*Backward chaining* starts with a hypothesis, then reasons backwards following a chain of rules in the inference network, to the pieces of evidence. An example of this type of expert system is MYCIN (Shortliffe 1976) - which incorporated about 400 heuristic rules written in an English-like if-then formalism to diagnose and treat infectious blood diseases, but its major impact on the field arose from its ability to explain lucidly any conclusion or question it generated.

❧

## APPENDIX B

# PROLOG

### B.1 An Overview of Prolog

Prolog stands for *programming in logic* (Clocksin & Mellish 1981) - an idea that emerged in the early 1970s to use logic as a programming language. The early developers of this idea included Robert Kowalski at Edinburgh (on the theoretical side), Maarten van Emden at Edinburgh (experimental demonstration), and Alain Colmerauer at the University of Marseilles (implementation). The present popularity of Prolog is largely due to David Warren's efficient implementation at Edinburgh in the mid 1970s.

Prolog is a programming language for symbolic, non-numeric computation. It is specially well suited for solving problems that involve objects and relations between objects. Prolog as a programming language is centred around a small set of basic mechanisms, including pattern matching, tree-based data structuring, and automatic backtracking. This small set constitutes a surprisingly powerful and flexible programming framework. Prolog is especially well suited for problems that involve objects - in particular, structured objects - and relations between them. Prolog can reason about the spatial relations and their consistency with respect to the general rule. Features like this make Prolog a powerful language for Artificial Intelligence and non-numerical programming in general.

A question to Prolog is always a sequence of one or more goals. To answer a question, Prolog tries to satisfy all the goals. To satisfy a goal means to demonstrate that the goal is true, assuming that the relations in the program are true. In other words to satisfy a goal means to demonstrate that the goal *logically follows* from the facts and rules in the program. If the question contains variables, Prolog also has to find what are the particular objects (in place of variables) for which the goals are satisfied. The particular instantiation of variables to these objects is displayed to the user. If Prolog

cannot demonstrate for some instantiation of variables that the goals logically follow from the program, then Prolog's answer to the question will be 'No'.

An appropriate view of the interpretation of a Prolog program in mathematical terms is then as follows: Prolog accepts facts and rules as a set of axioms, and the user's question as a *conjectured theorem*; then it tries to prove this theorem - that is, to demonstrate that it can be logically derived from the axioms.

Real numbers are not handled well in Prolog, as it is primarily a language for symbolic, non-numeric computation, as compared to scientific mathematical languages such as Fortran. In symbolic computation, integers are often used, for example, to count the number of items in a list, but there is little need for real numbers.

### B.2 Language Processing in Prolog

All Prolog implementations provide a notational extension called DCG (definite clause grammars) (Bratko 1990, Clocksin & Mellish 1981, Gal *et al.* 1991, Marcus 1986). This makes it very easy to implement formal grammars in Prolog. A grammar stated in DCG is directly executable by Prolog as a syntax analyser. DCG also facilitates the handling of the semantics of a language so that the meaning of a sentence can be interleaved with the syntax. This section shows how a grammar parser can be constructed in DCG.

#### B.2.1 Categories and Structures

Constituent structures (or simply, constituents) are built up from the basic syntactic categories. The most usual basic lexicon categories are as follows:

- nouns (N): *e.g., table, computer, John*
- determiners (DET): *e.g., the, some, most*
- verbs (V): *e.g., eats, eating, sleep, slept, have*
- adjectives (ADJ): *e.g., big, fast, his*
- adverbs (ADV): *e.g., yesterday, rarely, rather, very*
- auxiliaries (AUX): *e.g., has, will, is, are, must, should*
- conjunctions (CONJ): *e.g., and, or*
- prepositions (PREP): *e.g., to, on, with, in*
- pronouns (PRON): *e.g., he, who, which*

## Appendix B: Prolog

The constituents found in standard theories are defined in terms of these basic categories. Here are some of the more usual ones:

- sentence (S): a whole sentence,  
*e.g., It is raining.*  
*The man who John met yesterday is a painter.*  
*What is the capital of France?*
- noun phrase (NP): a noun or pronoun together with elements which modify it, i.e. determiner, adjectives, relative clause, etc.  
*e.g., the computer*  
*the man who John met yesterday*  
*some of the richest people in the world*
- verb phrase (VP): a verb or verb group and its immediate complements (direct, indirect and prepositional objects).  
*e.g., is sleeping*  
*gave Mary a book*  
*must be looking for something*
- prepositional phrase (PP): a preposition followed by a noun phrase.  
*e.g., with a telescope*  
*at the bottom of the garden*
- adjective phrase (ADJP): an adjective together with its modifiers, i.e. adverbial, qualifying phrases.  
*e.g., very big*  
*bigger than anyone expected*  
*small for an elephant*
- adverbial phrase (ADVP): an adverb together with its modifiers.  
*e.g., yesterday evening*  
*less often*  
*as quickly as possible*

This list is intended to be indicative rather than definitive: for different applications it may be more advantageous to recognise different intermediate

constituents between sentence and word. Also, not all types of text will have all the possible varieties of each type of construction.

### B.2.2 Description of the Structures

The basic syntactic categories mentioned above can be adjoined to form the description of a kind of “natural language” sentence. Note that “natural language” is not to be confused with “human language”. To be able to understand human language, a computer would need to possess the kind of knowledge about the language that humans possess referred to as “context” (Marcus 1986).

The DCG syntax of the natural language with categories can be seen as follows:

```
S    -> NP, VP.  
VP   -> V, NP, ADVP.  
VP   -> AUX, V, PP  
PP   -> PREP, NP.  
NP   -> DET, N.  
NP   -> DET, ADJP, N.  
N    -> [dog].  
N    -> [cat].  
DET  -> [the].  
PREP -> [in, front, of].  
ADJP -> [big, small].
```

Since allowing unrestricted natural language input is currently not feasible, a more formal language has been developed and adopted. The current implementation of the grammar is sufficient for the purpose of this project, which involves considerable use of arguments to check context-sensitive features and able to extract useful data structure. This will be mentioned in **Appendix C**.

## B.3 Operator Precedence in Prolog

As mentioned in **Appendix A.2.1**, an inference engine needs a collection of modules to drive. For the inference engine to recognise these modules, a *precedence order* would have to be constructed so that they can be distinguished and extracted easily by the language processing routines. Here is a rule example:

```
rule1 : if [object1, is_nearby, object2] then [object1, meanders].
```

there are keywords (such as 'if', 'then', ':' and 'and') and modules (such as 'is\_nearby' and 'meanders') that enable the inference engine to recognise a rule structure. Some have higher precedence order than others, and they are defined in Prolog as follows:

```
%% special operators defined for the inference engine

:- op( 890, xfx, :).
:- op( 880, xfx, then).
:- op( 870, fx, if).

:- op( 540, xfy, and).
:- op( 220, xfx, is_nearby).
:- op( 220, xf, meanders).
```

Take 'and' for instance. This tells Prolog that we want to use 'and' as an operator, whose precedence is 540 and its type is 'xfy', which is a kind of infix operator. The form of the specifier 'xfy' suggests that the operator, denoted by 'f', is between the two arguments denoted by 'x' and 'y'.

Notice that operator definitions do not specify any operation or action. Operators are normally used, as functors, only to combine objects into structures and not to invoke actions on data, although the word 'operator' appears to suggest an action. Operator names are atoms, and their precedence must be in some range which depends on the implementation. Typical range is between 1 and 1200, but the upper limit is often implementation dependent.

There are three groups of operator types:

1) infix operators of three types:

```
xfx    xfy    yfx
```

2) prefix operators of two types:

```
fx      fy
```

3) postfix operators of two types:

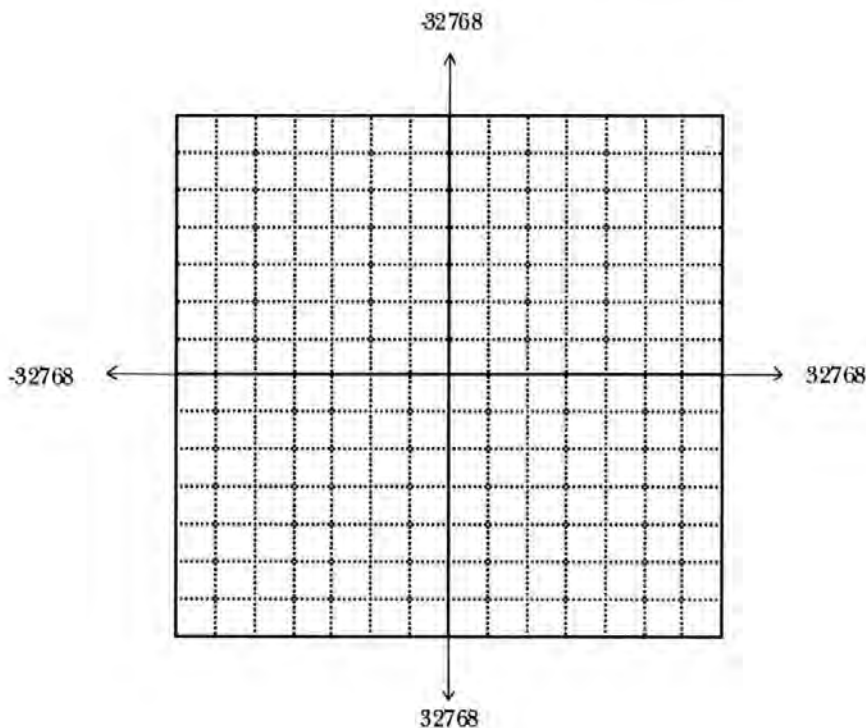
```
xf      yf
```

There is a difference between 'x' and 'y'. If an argument is enclosed in parentheses or it is an unstructured object then its precedence is 0; if an argument is a structure then its precedence is equal to the precedence of its principal functor. 'x' represents an argument whose precedence must be strictly lower than that of the operator. 'y' represents an argument whose precedence is lower or equal to that of the operator. Note also that, the higher the number, the higher the precedence order.

#### B.4 LPA MacProlog's Coordinate system

Within the MacProlog's GDL (Graphics Description Language), the mathematical coordinate system used corresponds to the coordinate system used by the internal graphic system *QuickDraw* on the Macintosh. Note that this differs in some important ways from traditional mathematical coordinate systems.

The coordinate plane is a two-dimensional grid, as shown in Figure B-1.



**Figure B-1** The Coordinate System of MacProlog

## Appendix B: Prolog

Coordinates must be integers in the range -32768 to +32768. The coordinate origin (0,0) is in the centre of the grid. As in traditional coordinate systems, the horizontal coordinates increase as you move from the left to right, but, unusually for mathematical systems, the vertical coordinates increase as you move from top to bottom.



# APPENDIX C

## DCG LISTING

This is the implemented DCG version of the *Language Interpreter & Translator* in the ESCAPE system. The language parser (`sentence`) can refer to three variables in a phrase. Each variable can be an object (`object1`, `object2` or `object3`), an object type (domain-dependent), or a calculated value (numeric).

```
%% - DCG LANGUAGE COMPILER
%% - Contains codes for checking Rules languages,
%%
%% - Written by Victor Ye, © 1995, RSRC, University of Brighton

% X = object1, Y = object2, Z = object3
sentence( X, Y, Z, P and Q)      -> simple_sentence( X, Y, Z, P), [and], sentence( X, Y, Z, Q).
sentence( X, Y, Z, P)           -> simple_sentence( X, Y, Z, P).

% simple_sentence can be an 'undo', a 'calculation' or a 'standard' simple_sentence
simple_sentence( X, Y, Z, undo( Assn))  -> [undo], simple_sentence( X, Y, Z, Assn), !.
simple_sentence( X, Y, Z, Assn)         -> calculation( X, Y, Z, Assn), !.
simple_sentence( X, Y, Z, Assn)         -> noun_phrase( _, Obj), verb_phrase( X, Y, Z, Obj, Assn).

% a noun_phrase can be a proper_noun (e.g. jelly1) or a type (e.g. 'JELLYFISH')
noun_phrase( X, Y) -> proper_noun( A, T), { T = obj, X = A ; T = type, Y = A }, !.
noun_phrase( X, Y) -> noun( X, Y).

% if a trans_verb is between two objects; e.g.: object1 trans_verb object2
verb_phrase( X, Y, Z, Obj1, Assn) -> trans_verb( A, B, Assn), noun_phrase( B, Obj2),
{ Obj1 = object1, Obj2 = object2, A = X, B = Y; Obj1 = object1, Obj2 = object3, A = X, B = Z
; Obj1 = object2, Obj2 = object1, A = Y, B = X; Obj1 = object2, Obj2 = object3, A = Y, B = Z
; Obj1 = object3, Obj2 = object1, A = Z, B = X; Obj1 = object3, Obj2 = object2, A = Z, B = Y }, !.

% if a trans_verb is between an object and a type; e.g.: object1 trans_verb type
verb_phrase( X, Y, Z, Obj, Assn) -> trans_verb( A, Type, Assn), noun_phrase( _, Type),
{ Obj = object1, A = X; Obj = object2, A = Y; Obj = object3, A = Z }, !.

% if it is an intransitive verb
verb_phrase( X, Y, Z, Obj, Assn) -> intrans_verb( A, Assn),
{ Obj=object1, A = X ; Obj=object2, A = Y; Obj=object3, A = Z }, !.

% new-verb = user defined
% if a new_verb is between two objects; e.g.: object1 new_verb object2
verb_phrase( X, Y, Z, Obj1, Assn) -> new_verb( A, B, Assn), noun_phrase( B, Obj2),
{ Obj1 = object1, Obj2 = object2, A = X, B = Y; Obj1 = object1, Obj2 = object3, A = X, B = Z
; Obj1 = object2, Obj2 = object1, A = Y, B = X; Obj1 = object2, Obj2 = object3, A = Y, B = Z
; Obj1 = object3, Obj2 = object1, A = Z, B = X; Obj1 = object3, Obj2 = object2, A = Z, B = Y }, !.

% if a new verb is between an objects and a type; e.g.: object1 new_verb type
verb_phrase( X, Y, Z, Obj, Assn) -> new_verb( A, Type, Assn), noun_phrase( Y, Type),
{ Obj = object1, A = X; Obj = object2, A = Y; Obj = object3, A = Z }, !.

% if it is an new intransitive verb
verb_phrase( X, Y, Z, Obj, Assn) -> new_verb( A, Assn),
{ Obj=object1, A = X ; Obj=object2, A = Y; Obj=object3, A = Z }, !.
```

## Appendix C: DCG Listing

```

% if it is a calculation of one parameter, then calculate the value (e.g. time_now_is X)
calculation( X, Y, Z, Calc( A)) -> [Calc], value( _, A), { calculableList( List), member( Calc, List)}, !.

% if it is a calculation of two parameters, then perform the calculates
calculation( X, Y, Z, Calc( A, B)) -> calculation2( X, Y, Z, Calc( R, B)),
{ R = object1, A = X; R = object2, A = Y; R = object3, A = Z }, !.
calculation( X, Y, Z, Assn )      -> calculation2( X, Y, Z, Assn).

calculation2( X, Y, Z, Calc( A, B)) -> [Calc( A, B)], { calculableList( List), member( Calc, List)}, !.

% a noun can be one of the variables object1, object2 or object3
noun( X, Object) -> [Object], { member( Object, [object1, object2, object3]) }, !.
% or it can be a value (performed by a calculation)
noun( X, Var)    -> value( X, Var), { atomic( Var); var( Var)}.

% if it is a value, then copy it
value( X, Var)   -> [Var].

% a proper noun can be one of the object types (domain-dependent, e.g. 'JELLYFISH')
proper_noun( Type, type)      -> [Type], { defaults types( TList), member( Type, TList) }.
% or it can be one of the existing actors in the domain (domain-dependent, e.g. jelly1)
proper_noun( Object, obj)     -> [Object], { get_actors_list( AList), member( Object, AList) }.

% an intrans_verb can be a 'property' or a '1 parameter operator' (domain-dependent e.g. is_alive)
intrans_verb( X, Intrans( X)) -> [Intrans],
{ propertyList( PList), action1List( AList), append( PList, AList, TList), member( Intrans, TList) }, !.

% an trans_verb can be a 'relation' or a '2 parameters operator' (domain-dependent e.g. is_nearby)
trans_verb( X, Y, Trans( X, Y)) -> [Trans],
{ relationList( RList), action2List( AList), append( RList, AList, TList), member( Trans, TList) }, !.

% new verb (user defined) can be transitive (2 parameters) or intransitive (1 parameter)
new_verb( X, Y, Verb( X, Y))    -> [Verb].
new_verb( X, Verb( X))          -> [Verb].

```

❧

## APPENDIX D

# WORLD.JELLY DOMAIN DATABASE

### Overview

This section lists the database of the domain World.Jelly 2.2.

Number of objects: 6 ( 2 JELLYFISH, 3 PRAWNS, 1 ROCK).

Number of rules: 12

### D.1 Facts

```
%% facts required for World.Bird 2.2

%% planning script from X to Y
fact plan 1 to 10.

%% defining the boundary of the world
fact world_size( ( 0, 0), ( 400, 300)).

%% facts and properties of objects
fact obj_type( jelly1, 'JELLYFISH').
fact property( jelly1, [alive]).
fact location( jelly1, pt( 197, 72)).

fact obj_type( jelly2, 'JELLYFISH').
fact property( jelly2, [alive]).
fact location( jelly2, pt( 99, 56)).

fact obj_type( prawn1, 'PRAWN').
fact property( prawn1, [alive]).
fact location( prawn1, pt( 203, 206)).

fact obj_type( prawn2, 'PRAWN').
fact property( prawn2, [alive]).
fact location( prawn2, pt( 180, 153)).

fact obj_type( prawn3, 'PRAWN').
fact property( prawn3, [alive]).
fact location( prawn3, pt( 178, 185)).

fact obj_type( rock1, 'ROCK').
fact property( rock1, [static]).
fact location( rock1, pt( 226, 212)).
```

## D.2 Defaults

%% Default Objects pictures for World.Jelly 2.2 for use with the Rules Maker

```
set_pict_descs :- set_prop( 'JELLYFISH', left, resource( 200)),
                  set_prop( 'JELLYFISH', right, resource( 201)),
                  set_prop( 'PRAWN', left, resource( 210)),
                  set_prop( 'PRAWN', right, resource( 211)),
                  set_prop( 'ROCK', left, resource( 220)),
                  set_prop( 'ROCK', right, resource( 221)),
                  set_area.
```

```
set_area :-      fact world_size( ( _, _), ( Y, X)),
                  set_prop( region, areal, box( -Y, -X, Y, X)),
                  set_prop( poss_dir, areal, [left, right]).
```

%% Default Objects database

```
defaults types( ['JELLYFISH', 'PRAWN', 'ROCK']).
```

```
defaults mass( 'JELLYFISH', 5).
defaults size( 'JELLYFISH', 21, 21).
defaults vis( 'JELLYFISH', 150).
defaults maxSpeed( 'JELLYFISH', 40).
defaults property( 'JELLYFISH', [alive]).
```

```
defaults mass( 'PRAWN', 1).
defaults size( 'PRAWN', 4, 4).
defaults vis( 'PRAWN', 80).
defaults maxSpeed( 'PRAWN', 40).
defaults property( 'PRAWN', [alive]).
```

```
defaults mass( 'ROCK', 10).
defaults size( 'ROCK', 25, 25).
defaults property( 'ROCK', [static]).
```

```
defaults name( 'JELLYFISH', jelly).
defaults name( 'PRAWN', prawn).
defaults name( 'ROCK', rock).
```

## D.3 Rules

```

%% rulebase required for World.Jelly 2.2

% objects of 'PRAWN' type are food_of 'JELLYFISH' type
rule1 :   if      [object1, has_type, 'PRAWN',
                  and, object2, has_type, 'JELLYFISH']
          then    [object1, is_food_of, object2].

% objects of 'JELLYFISH' type are predator_of 'PRAWN' type
rule2 :   if      [object1, has_type, 'PRAWN',
                  and, object2, has_type, 'JELLYFISH']
          then    [object2, is_predator_of, object1].

% if an object is nearby a predator, then set escaping property
rule3 :   if      [object1, is_nearby, object2,
                  and, object2, is_predator_of, object1]
          then    [object1, is_escaping_from, object2].

% if an object is nearby food, then plans encounter and set pursuing property
rule4 :   if      [object1, is_nearby, object2,
                  and, object2, is_food_of, object1,
                  and, object2, is_alive]
          then    [object1, plans_encounter_to, object2,
                  and, object1, is_pursuing, object2].

% if an object collides into food, kills it and stay for 1 keyframe
rule5 :   if      [object1, is_pursuing, object2,
                  and, object1, collides_with, object2,
                  and, object2, is_food_of, object1]
          then    [object2, dies,
                  and, object1, stay_there,
                  and, undo, object1, is_pursuing, object2].

% if a 'PRAWN' is nearby a 'ROCK', 'ROCK' becomes a shelter for 'PRAWN'
rule6 :   if      [object1, is_nearby, object2,
                  and, object1, has_type, 'PRAWN',
                  and, object2, has_type, 'ROCK']
          then    [object2, is_shelter_of, object1].

% if an object is escaping and is nearby a shelter, then move towards it
rule7 :   if      [object1, is_escaping_from, object2,
                  and, object3, is_shelter_of, object1,
                  and, object1, is_nearby, object3]
          then    [object1, plans_encounter_to, object3].

% if an object has an escaping property, then plans get away
rule8 :   if      [object1, is_escaping_from, object2]
          then    [object1, plans_getaway_from, object2].

% if an object is alive, then meanders
rule9 :   if      [object1, is_alive]
          then    [object1, meanders].

% if an object is static, then stay put
rule10 :  if      [object1, is_static]
          then    [object1, stay_there].

% if an object is alive and is a 'PRAWN', then flock centering
rule11 :  if      [object1, is_alive, and, object1, has_type, 'PRAWN']
          then    [object1, flock_centering].

% move objects within the boundary
rule12 :  if      [object1, is_alive, and, object1, is_close_to_edge]
          then    [object1, moves_within_boundary].

```

## D.4 Predicates

```

%% domain specific operators
:- op( 220, xf, meanders).
:- op( 220, xf, flock_centering).
:- op( 220, xfx, plans_encounter_to).
:- op( 220, xfx, plans_getaway_from).

%% domain defined lists

% domainAction1List = 1 parameter, domainAction2List = 2 parameters
domainAction1List( [ flock_centering, meanders]).
domainAction2List( [ plans_encounter_to, plans_getaway_from]).

% domainPredicate1List = 1 parameter, domainPredicate2List = 2 parameters
domainPredicate1List( []).
domainPredicate2List( []).

%% see F.4 World.Bird: Predicate for code for flock_centering

/*                      meanders                      */
/******/

Object meanders :-
    get_prop( level, meanders, Level),           % Level (6) for random move
    get_prop( rule, number, RuleNo),
    get_prop( frame, current, Now),
    get_prev_locn( Object, Now, pt( X, Y)),
    maxSpeed( Object, Max),                      % get maxSpeed of object
    make_neg( Max, Min),
    rand_ize( XRand, Min, Max),                  % random number generator
    rand_ize( YRand, Min, Max),
    X1 is X + XRand,
    Y1 is Y + YRand,
    Action = poss_locn( Object, Now, Level, pt( X1, Y1)),
    do( Action, RuleNo), !.

/*                      plans_encounter_to            */
/******/

Object1 plans_encounter_to Object2 :-
    Object1 is_alive,                            % make sure that only live object moves
    get_actors_list( AList),
    member( Object2, AList),                    % make sure that object2 is a valid obj
    get_prop( rule, number, RuleNo),
    do_plans_encounter( Object1, Object2, Action),
    do( Action, RuleNo).

do_plans_encounter( Object1, Object2, Action) :-
    get_prop( frame, current, Now),
    get_prev_locn( Object1, Now, pt( X1, Y1)),
    get_prev_locn( Object2, Now, pt( X2, Y2)),
    distance( X1, X2, Y1, Y2, ObjsDist),
    maxSpeed( Object1, ObjMaxSpeed),

    Ratio is ObjMaxSpeed/ObjsDist,              % then move only within the speed range
    distance( X1, X2, X),
    distance( Y1, Y2, Y),
    DX is Ratio*X, DX_int is int( DX),          % DX = X distance to travel, maximum

```

```

DY is Ratio*Y, DY_int is int( DY),      % DY = Y distance to travel, maximum

% get the X and Y signs, whether they're +ve of -ve
X_diff is X2 - X1, X_sign is sign( X_diff),
Y_diff is Y2 - Y1, Y_sign is sign( Y_diff),

(   ObjMaxSpeed < ObjDist               % if object2 is outside the speed range
-> X_temp is X1 + X_sign*DX_int,         % move towards it
    Y_temp is Y1 + Y_sign*DY_int
    ; X_temp is X1 + X_sign*X,           % else move within full speed
    Y_temp is Y1 + Y_sign*Y ),
NewLoc is_between_places pt( X1, Y1) & pt( X_temp, Y_temp) ,
get_prop( level, plans_encounter_to, Level),
Action = poss_locn( Object1, Now, Level, NewLoc), !.

/*                      plans_getaway_from                      */
/*****

%% At the moment, the one escapes in random mode
Object1 plans_getaway_from Object2 :-
    Object1 is_alive,                % make sure that only live object moves
    get_actors_list( AList),
    member( Object2, AList),         % make sure that object2 is a valid obj
    get_prop( frame, current, Now),
    get_prop( level, plans_getaway_from, Level),
    get_prop( rule, number, RuleNo),
    do_plans_getaway( Object1, Object2, Now, Level, Action),
    do( Action, RuleNo).

do_plans_getaway( Object1, Object2, Now, Level, Action) :-
    find_direction( Object2),        % find object2's direction
    Prev_time is Now - 1,
    direction( Object2, Prev_time, Direction),
    Upper_limit is Direction + 45,
    Lower_limit is Direction - 45,
    rand_ize( Movin_direction, Upper_limit, Lower_limit),
    get_direction_signs( Movin_direction, X_sign, Y_sign), % standard in system

    get_prev_locn( Object1, Now, pt( X1, Y1)),
    get_prev_locn( Object2, Now, pt( X2, Y2)),

    distance( X1, X2, Y1, Y2, BigDist),
    maxSpeed( Object1, Speed),
    Max_dist is Speed / sqrt(2),
    Max_dist_int is int( Max_dist),
    (   Speed < BigDist                  % if Obj2 is outside the speed range
    -> Percentage is Max_dist_int * 8/10, % 80 percent speed
        Percentage_int is int( Percentage), % force to become an integer
        rand_ize( X_dist, Percentage_int, Speed),
        rand_ize( Y_dist, Percentage_int, Speed),
        X_temp is X1 + X_sign*X_dist,
        Y_temp is Y1 + Y_sign*Y_dist,
        NewLoc is_between_places pt( X1, Y1) & pt( X_temp, Y_temp)
    ; X is X1 + X_sign*Speed,            % else just move at maxSpeed
      Y is Y1 + Y_sign*Speed,
      NewLoc = pt( X, Y)),
    Action = poss_locn( Object1, Now, Level, NewLoc), !.

```

# APPENDIX E

## WORLD.TRAFFIC DOMAIN DATABASE

### Overview

This section lists the database of the domain World.Traffic 2.0.

Number of objects:     5 ( 5 CARS).

Number of rules:       7

### E.1 Facts

```
%% facts required for World.Traffic 2.0
```

```
%% planning script from X to Y
```

```
fact plan 1 to 12.
```

```
%% facts and properties of objects
```

```
fact obj_type( car1, 'CAR').  
fact property( car1, [alive, none]).  
fact location( car1, pt( 39, 28)).
```

```
fact obj_type( car2, 'CAR').  
fact property( car2, [alive, none]).  
fact location( car2, pt( 152, 26)).
```

```
fact obj_type( car3, 'CAR').  
fact property( car3, [none, static]).  
fact location( car3, pt( 279, 26)).
```

```
fact obj_type( car4, 'CAR').  
fact property( car4, [alive, left]).  
fact location( car4, pt( 279, 60)).
```

```
fact obj_type( car5, 'CAR').  
fact property( car5, [alive, left]).  
fact location( car5, pt( 170, 174)).
```



## E.2 Defaults

```

%% Default Objects pictures for World.Traffic 2.0

set_pict_descs :-      set_prop( 'CAR',  east,  resource( 500)),
                      set_prop( 'CAR',  west,  resource( 500)),
                      set_prop( 'CAR',  north, resource( 501)),
                      set_prop( 'CAR',  south, resource( 501)),
                      set_junction_areas.

set_junction_areas:-  set_prop( region, area1, box( 18,  0,  34, 197)),
                      set_prop( region, area2, box( 18, 196,  34, 240)),
                      set_prop( region, area3, box( 51,  0,  34, 159)),
                      set_prop( region, area4, box( 51, 158,  34,  39)),
                      set_prop( region, area5, box( 51, 196,  34,  39)),
                      set_prop( region, area6, box( 51, 234,  34, 200)),
                      set_prop( region, area7, box( 84, 158, 200,  39)),
                      set_prop( region, area8, box( 84, 196, 200,  39)),

                      set_prop( poss_dir, area1, [east]),
                      set_prop( poss_dir, area2, [east]),
                      set_prop( poss_dir, area3, [west]),
                      set_prop( poss_dir, area4, [north, west]),
                      set_prop( poss_dir, area5, [west, south]),
                      set_prop( poss_dir, area6, [west]),
                      set_prop( poss_dir, area7, [north]),
                      set_prop( poss_dir, area8, [south]).

%% Default Objects database

defaults types( ['CAR']).
defaults mass( 'CAR', 1).
defaults size( 'CAR', 2).
defaults vis( 'CAR', 100).
defaults maxSpeed( 'CAR', 30).
defaults stopTime( 'CAR', 6).
defaults stopDistance( 'CAR', 40).
defaults property( 'CAR', [alive, left]).

defaults name( 'CAR', car).

```

### E.3 Rules

```
%% rulebase required for World.Traffic 2.0

% if an object is alive and ahead is clear, then move forward
rule1 :   if      [object1, is_alive,
                  and, ahead_clear( object1, yes)]
          then    [object1, moves_forward].

% if an object is alive and ahead is not clear, then slow down
rule2 :   if      [object1, is_alive,
                  and, ahead_clear( object1, no)]
          then    [object1, slows_down].

% if an object is stop, then stay there
rule3 :   if      [object1, is_stop]
          then    [object1, stay_there].

% if an object is turning, then slow down
rule4 :   if      [object1, is_turning]
          then    [object1, slows_down].

% if an object is stop, ready to turn, and ahead is not clear, then stay put
rule5 :   if      [object1, is_stop,
                  and, object1, is_turning]
          and, ahead_clear( object1, no),
          then    [object1, stay_there].

% if an object is stop, ready to turn, and ahead is clear, then go ahead and turn
rule6 :   if      [object1, is_stop,
                  and, object1, is_turning]
          and, ahead_clear( object1, yes),
          then    [object1, turns].

% if an object is static, copy previous position
rule7 :   if      [object1, is_static]
          then    [object1, stay_there].
```

## E.4 Predicates

```

%% domain specific operators
:- op( 220, xf, ahead_is_clear).
:- op( 220, xf, ahead_is_not_clear).
:- op( 220, xf, is_stop).
:- op( 220, xf, is_turning).
:- op( 220, xf, moves_forward).
:- op( 220, xf, slows_down).
:- op( 220, xf, turns).

%% domain defined lists

% domainPredicate1List = 1 parameter, domainPredicate2List = 2 parameters
domainPredicate1List( [ahead_is_clear, ahead_is_not_clear, is_stop,
                       is_turning]).
domainPredicate2List( []).

% domainAction1List = 1 parameter, domainAction2List = 2 parameters
domainAction1List( [ moves_forward, slows_down, turns]).
domainAction2List( []).

/*          check if object1 is stopped          */
/******/

Obj is_stop :- get_actors_list( List),
               member( Obj, List),
               get_prop( Obj, speed, 0). % Obj is_stop if speed = 0

/*          check if object1 is turning          */
/******/

Object1 is_turning :-
    get_prop( frame, current, Time),
    Object1 is_alive, % make sure that only live object moves
    get_prev_locn( Object1, Time, pt( X, Y)),
    which_area( pt( X, Y), Area),
    get_prop( poss_dir, Area, List_of_poss_dirs),
    get_prev_direction( Object1, Time, Dir),

    stopDistance( Object1, StopDist),
    Dist is StopDist/2, % make it halved the travel StopDist ahead
    correct_location( Dir, X, Y, X1, Y1, Dist),
    which_area( pt( X1, Y1), Area2), % find which area it is in

    change_direction( Object1, Time, Area2, Dir2),
    Dir2 \== Dir, !. % object is turning if directions are not the same

/*          ahead_is_clear or not          */
/******/

Object ahead_is_clear :- ahead_clear( Object, yes).
Object ahead_is_not_clear :- ahead_clear( Object, no).

ahead_clear( Object1, Answer) :-
    Object1 is_alive,

```

```

    get_prev_time( Prev_time),
    find_objects_ahead( Object1, Objects_ahead_List), % get list of objects in front
    ( Objects_ahead_List = []
    -> ( set_prop( Object1, is_behind, none), % if list is empty, then it's clear
        Answer = yes)
    ; % else check if the nearest object in front is within the viewing distance
        find_nearest_object_in_list( Object1, Prev_time, Objects_ahead_List,
            Nearest_object),
        outside_viewing_distance( Object1, Nearest_object, Prev_time, Answer)).

%% find objects ahead

find_objects_ahead( Object1, List_of_objects_ahead) :-
    get_prop( frame, current, Time),
    get_actors_list( AList),
    if_in_front( Object1, AList, Time, [], List_of_objects_ahead).

if_in_front( _, [], _, List, List) :- !.

if_in_front( Object1, [Object2|Rest], Time, List1, List2) :-
    get_prev_locn( Object1, Time, pt( X1, Y1)),
    which_area( pt( X1, Y1), Area1), % find which area is object1 in
    which_direction( Object1, Time, Area1, Dir), % direction it's travelling
    get_prev_locn( Object2, Time, pt( X2, Y2)),
    which_area( pt( X2, Y2), Area2), % find which area is object2 in
    ( is_infront_of( Dir, X2, Y2, X1, Y1), % if in front and same direction
        which_direction( Object2, Time, Area1, Dir),
        append( [Object2], List1, List3), !
    ; List3 = List1),
    if_in_front( Object1, Rest, Time, List3, List2).

% if pt( X2,Y2) is in front of pt( X1,Y1) towards the east
is_infront_of( east, X2, Y2, X1, Y1) :- X2 > X1, Y1 =< Y2, Y2 < Y1.

% if pt( X2,Y2) is in front of pt( X1,Y1) towards the west
is_infront_of( west, X2, Y2, X1, Y1) :- X2 < X1, Y1 =< Y2, Y2 < Y1.

% if pt( X2,Y2) is in front of pt( X1,Y1) towards the north
is_infront_of( north, X2, Y2, X1, Y1) :- Y2 < Y1, X1 =< X2, X2 < X1.

% if pt( X2,Y2) is in front of pt( X1,Y1) towards the south
is_infront_of( south, X2, Y2, X1, Y1) :- Y2 > Y1, X1 =< X2, X2 < X1.

is_infront_of( right, X2, Y2, X1, Y1) :- is_infront_of( east, X2, Y2, X1, Y1).
is_infront_of( left, X2, Y2, X1, Y1) :- is_infront_of( west, X2, Y2, X1, Y1).

%% find nearest object in list

% if only one, then it is the nearest
find_nearest_object_in_list( _, _, [Nearest_object], Nearest_object).

find_nearest_object_in_list( Obj1, Time, [First, Second|Rest], Nearest_object) :-
    find_nearest_object_in_list( Obj1, Time, [Second|Rest], NearRest),
    nearer_object( Obj1, Time, First, NearRest, Nearest_object).

nearer_object( Obj1, Time, Obj2, Obj3, Nearer) :-
    get_objects_dist( Obj1, Obj2, Time, Distance1),
    get_objects_dist( Obj1, Obj3, Time, Distance2),
    min( Distance1, Distance2, MinDist),
    ( MinDist = Distance1

```

```

-> Nearer = Obj2
; Nearer = Obj3).

%% check if objects are within the viewing distance
outside_viewing_distance( Object1, Object2, Time, Answer) :-
    vis( Object1, VisDist),           % get the Visible Distance
    get_objects_dist( Object1, Object2, Time, Distance),
    ( Distance less_than_or_equal_to VisDist % front not clear)
    -> ( ( Object2 is_static           % if the object in front is stopped
        ; ( direction( Object1, Time, Dir), % or if both going in same direction
            direction( Object2, Time, Dir ) ) )
        -> Answer = no, set_prop( Object1, is_behind, Object2)
        ; Answer = yes, set_prop( Object1, is_behind, none) ) , !
    ; Answer = yes, set_prop( Object1, is_behind, none) ).

/*                      moves_forward                      */
/*****

Object moves_forward :-
    get_prop( frame, current, Now),
    get_prop( level, moves_forward, Level),
    get_prop( rule, number, RuleNo),
    do_moves_forward( Object, Now, Level, Action),
    do( Action, RuleNo), !.

do_moves_forward( Object, Now, Level, Action) :-
    get_prev_locn( Object, Now, pt( X, Y)),
    which_area( pt( X, Y), Area),           % find which area is object in
    which_direction( Object, Now, Area, Dir), % see below
    set_object_direction( Object, Now, Dir), !,

    maxSpeed( Object, MaxSpeed),
    stopDistance( Object, StopDist),
    stopTime( Object, StopTime),

    set_speed_values( Object, acc),           % see later
    get_prop( Object, speed, Value),
    Step is Value/StopTime,
    Acc is ( 2*StopDist/(StopTime^2)),
    XX is (MaxSpeed*Step + (0.5*Acc*(Step^2))), % distance to go
    Distance_travelled is int( XX),           % make XX an integer

    correct_location( Dir, X, Y, X1, Y1, Distance_travelled),
    check_within_boundary( Area, Dir, X1, Y1, X2, Y2),
    Action = poss_locn( Object, Now, Level, pt( X2, Y2)).

/* set the correct location for Directions, and distances travelled */
correct_location( north, X_in, Y_in, X_out, Y_out, Distance_travelled) :-
    Y_out is Y_in - Distance_travelled, !.

correct_location( east, X_in, Y_in, X_out, Y_in, Distance_travelled) :-
    X_out is X_in + Distance_travelled, !.

correct_location( south, X_in, Y_in, X_in, Y_out, Distance_travelled) :-
    Y_out is Y_in + Distance_travelled, !.

correct_location( west, X_in, Y_in, X_out, Y_in, Distance_travelled) :-
    X_out is X_in - Distance_travelled, !.

```

```

/* keep location within the road boundary */
check_within_boundary( Area, north, X_in, Y_in, X_out, Y_out) :-
    check_within_boundary( Area, south, X_in, Y_in, X_out, Y_out).

check_within_boundary( Area, south, X_in, Y_in, X_out, Y_in) :-
    get_prop( region, Area, box( _, Left, _, Width)),
    X1 is ( ( Width/3) + Left),
    X2 is int( X1),                % make X1 an integer
    rand_ize( XRand, -1, 1),      % obtain a little randomisation
    X_out is X2 + XRand, !.

check_within_boundary( Area, east, X_in, Y_in, X_out, Y_out) :-
check_within_boundary( Area, west, X_in, Y_in, X_out, Y_out).

check_within_boundary( Area, west, X_in, Y_in, X_in, Y_out) :-
    get_prop( region, Area, box( Top, _, Depth, _)),
    Y1 is ( ( Depth/3) + Top),
    Y2 is int( Y1),                % make Y1 an integer
    rand_ize( YRand, -1, 1),      % obtain a little randomisation
    Y_out is Y2 + YRand, !.

check_within_boundary( _, _, X_in, Y_in, X_in, Y_in).    % else if nothing fits

/*                                slows_down                                */
/*****

Object slows_down :-
    Object is_alive,                % make sure that only live object moves
    get_prop( frame, current, Now),
    get_prop( level, slows_down, Level), % Level (5) for meanders_forward
    get_prop( rule, number, RuleNo),
    do_slows_down( Object, Now, Level, Action), !,
    do( Action, RuleNo).

do_slows_down( Object, Now, Level, Action) :-
    get_prev_locn( Object, Now, pt( X, Y)),
    which_area( pt( X, Y), Area),    % find which area is object in
    which_direction( Object, Now, Area, Dir),
    set_object_direction( Object, Now, Dir), !,

    get_prev_time( Time),
    maxSpeed( Object, MaxSpeed),
    stopDistance( Object, StopDist),
    stopTime( Object, StopTime),

    get_prop( Object, is_behind, Object2),
    get_actors_list( List),
    (   member( Object2, List)        % if Object2 is an object then...
    -> get_objects_dist( Object1, Object2, Time, Distance)
    ;   Distance = StopDist ),        % in case of turning

    (   Distance <= StopDist
    -> set_prop( Object, speed, 0),    % emergency stop !
        StoppingDistance = 0
    ;   StoppingDistance is Distance - StopDist ),    % StopDist = 40 (2 cars length)

    set_speed_values( Object, deacc),
    get_prop( Object, speed, Value),

    Step is Value/StopTime,
    DeAcc is ( 2*StoppingDistance/(StopTime^2)),
    XX is (MaxSpeed*Step - (0.5*DeAcc*(Step^2))), % distance to go

```

```

Distance_travelled is int( XX),                % make XX an integer

correct_location( Dir, X, Y, X1, Y1, Distance_travelled),
check_within_boundary( Area, Dir, X1, Y1, X2, Y2),
Action = poss_locn( Object, Now, Level, pt( X2, Y2)), !.

get_junction_stopping_distance( Area, pt( X, Y), Distance) :-
get_prop( region, Area, box( Top, Left, Bottom, Right)),
(   Area = area1 -> D is Left + Right - X, Distance is abs( D), !
; Area = area6 -> D is X - Left,      Distance is abs( D), !
; Area = area7 -> D is Y - Top,      Distance is abs( D), !).

set_speed_values( Object, Type) :-
get_prop( Object, speed, Value),
set_speed_value( Object, Type, Value)
; set_prop( Object, speed, 4), !.                % if it has not a value, then give it 4

set_speed_value( Object, Type, stop).           % if it is stop, then do nothing
set_speed_value( Object, Type, Value) :-        % else check the value
(   Type = acc -> Temp = 1
; Type = deacc -> Temp = -1 ),
NewValue is Value + Temp,
(   NewValue >= 6 -> set_prop( Object, speed, 6)
; NewValue < 0 -> set_prop( Object, speed, 0)
; set_prop( Object, speed, NewValue) ), !.

/*              making an object turn              */
/******/

turns Object :-
Object is_alive,                                % make sure that object is alive
get_prop( frame, current, Now),
get_prop( level, turns, Level),
get_prop( rule, number, RuleNo),
write( Object), writenl( ' is supposed to turn here...'),
do_turns( Object, Now, Level, Action),
write( ' ...turned here with '), write( RuleNo), write( '...'),
do( Action, RuleNo), !.

do_turns( Object, Now, Level, Action) :-
get_prev_locn( Object, Now, pt( X, Y)),
which_area( pt( X, Y), Area),                  % find which area is object in
change_direction( Object, Now, Area, Dir),     % see below
set_object_direction( Object, Now, Dir), !,

maxSpeed( Object, MaxSpeed),
stopDistance( Object, StopDist),
stopTime( Object, StopTime),

set_speed_values( Object, acc),
get_prop( Object, speed, Value),
Step is Value/StopTime,
Acc is ( 2*StopDist/(StopTime^2)),
XX is 5+ (MaxSpeed*Step + (0.5*Acc*(Step^2))), % distance to go
Distance_travelled is int( XX),                % make XX an integer

correct_location( Dir, X, Y, X1, Y1, Distance_travelled),
check_within_boundary( Area, Dir, X1, Y1, X2, Y2),
Action = poss_locn( Object, Now, Level, pt( X2, Y2)), !.

```

```

%% Check object when turning, turning + possible directions
change_direction( Obj, Now, Area, Direction) :-
    property( Obj, PropertyList),
    find_next_turing( PropertyList, Next_turn), % find the next turning
    get_prop( poss_dir, Area, List_of_poss_dirs), % get all the possible directions
    length( List_of_poss_dirs, Length), % find the number of directions
    ( Length = 1
      -> [Direction] = List_of_poss_dirs
      ; get_prev_direction( Obj, Now, Heading),
        resolve_dir( Heading, Next_turn, TheDirection),
        ( member( TheDirection, List_of_poss_dirs) % if it is possible
          -> Direction = TheDirection
          ; Direction = Heading ) ). % else stay with the original direction

find_next_turing( Prop_List, Next_turn) :-
    member( none, Prop_List), Next_turn = none, !.
find_next_turing( Prop_List, Next_turn) :-
    member( left, Prop_List), Next_turn = left, !.
find_next_turing( Prop_List, Next_turn) :-
    member( right, Prop_List), Next_turn = right, !.

resolve_dir( north, left, west).
resolve_dir( north, right, east).
resolve_dir( east, left, north).
resolve_dir( east, right, south).
resolve_dir( south, left, east).
resolve_dir( south, right, west).
resolve_dir( west, left, south).
resolve_dir( west, right, north).
resolve_dir( Heading, none, Heading).

```



# APPENDIX F

## WORLD.BIRD DOMAIN DATABASE

### Overview

This section lists the database of the domain World.Bird 2.0.

Number of objects: 16 ( 9 BIRDS, 7 OBSTACLES).

Number of rules: 3

### F.1 Defaults

```
%% Default Objects pictures for World.Bird 2.0

set_pict_descs :- set_prop( 'OBST', left, circle( 0, 0, 10)),
                  set_prop( 'OBST', right, circle( 0, 0, 10)),
                  set_prop( 'BIRD', left, fillcircle( 0, 0, 2)),
                  set_prop( 'BIRD', right, fillcircle( 0, 0, 2)),
                  set_area.

set_area :-      set_prop( region, areal, box( -500, -500, 500, 500)),
                  set_prop( poss_dir, areal, [left, right]).

%% Default Objects database
defaults types( ['BIRD', 'OBST']).
defaults mass( 'BIRD', 1).
defaults size( 'BIRD', 6, 6).
defaults vis( 'BIRD', 100).
defaults maxSpeed( 'BIRD', 15).
defaults property( 'BIRD', [alive]).

defaults size( 'OBST', 13, 13).
defaults property( 'OBST', [static]).

defaults name( 'BIRD', bird).
defaults name( 'OBST', obstacle).
```

## F.2 Facts

fact plan 1 to 16. %% planning script from X to Y

fact obj\_type( bird1, 'BIRD').  
fact property( bird1, [alive]).  
fact location( bird1, pt( 325, 161)).

fact obj\_type( bird2, 'BIRD').  
fact property( bird2, [alive]).  
fact location( bird2, pt( 341, 131)).

fact obj\_type( bird3, 'BIRD').  
fact property( bird3, [alive]).  
fact location( bird3, pt( 358, 194)).

fact obj\_type( bird4, 'BIRD').  
fact property( bird4, [alive]).  
fact location( bird4, pt( 383, 141)).

fact obj\_type( bird5, 'BIRD').  
fact property( bird5, [alive]).  
fact location( bird5, pt( 371, 228)).

fact obj\_type( bird6, 'BIRD').  
fact property( bird6, [alive]).  
fact location( bird6, pt( 405, 191)).

fact obj\_type( bird7, 'BIRD').  
fact property( bird7, [alive]).  
fact location( bird7, pt( 409, 96)).

fact obj\_type( bird8, 'BIRD').  
fact property( bird8, [alive]).  
fact location( bird8, pt( 432, 194)).

fact obj\_type( bird9, 'BIRD').  
fact property( bird9, [alive]).  
fact location( bird9, pt( 436, 75)).

fact obj\_type( obstacle1, 'OBST').  
fact property( obstacle1, [static]).  
fact location( obstacle1, pt( 271, 208)).

fact obj\_type( obstacle2, 'OBST').  
fact property( obstacle2, [static]).  
fact location( obstacle2, pt( 266, 113)).

fact obj\_type( obstacle3, 'OBST').  
fact property( obstacle3, [static]).  
fact location( obstacle3, pt( 233, 160)).

fact obj\_type( obstacle4, 'OBST').  
fact property( obstacle4, [static]).  
fact location( obstacle4, pt( 178, 204)).

fact obj\_type( obstacle5, 'OBST').  
fact property( obstacle5, [static]).  
fact location( obstacle5, pt( 191, 128)).

fact obj\_type( obstacle6, 'OBST').  
fact property( obstacle6, [static]).  
fact location( obstacle6, pt( 131, 168)).

```
fact obj_type( obstacle7, 'OBST').
fact property( obstacle7, [static]).
fact location( obstacle7, pt( 27, 172)).
```

### F.3 Rules

```
%% Rulebase required for World.Bird 2.0

% if an object is alive, then meanders forward
rule1 :   if      [object1, is_alive]
         then    [object1, meanders_forward].

% if an object is static, then copy previous position
rule2 :   if      [object1, is_static]
         then    [object1, stay_there].

% if an object is alive, the flock centering
rule3 :   if      [object1, is_alive]
         then    [object1, flock_centering].
```

### F.4 Predicates

```
%% domain specific operators
:- op( 220, xf, flock_centering).
:- op( 220, xf, meanders_forward).

%% domain defined lists

% domainAction1List = 1 parameter, domainAction2List = 2 parameters
domainAction1List( [ flock_centering, meanders_forward]).
domainAction2List( []).

% domainPredicate1List = 1 parameter, domainPredicate2List = 2 parameters
domainPredicate1List( []).
domainPredicate2List( []).

/*                      flock_centering                      */
/******/

Object flock_centering :-
  get_prop( frame, current, Now),
  Time is Now - 1,
  Object has_type Type, 1,
  find_flock_centre( Type, Time, pt( CentreX, CentreY)),
  not( CentreX = -9999),    % proceed if there's a centre
  findall( Level, poss_locn( Object, Now, Level, _), LevelList),
  sort( LevelList, [Highest|_]),    % find the current heightst priority level
  poss_locn( Object, Now, Highest, pt( PossX, PossY)),
```

```

centering_calc( PossY, CentreY, Y),
centering_calc( PossX, CentreX, X),
get_prop( rule, number, RuleNo),
do( poss_locn( Object, Now, Highest, pt( X, Y)), RuleNo), !.

% get it if the centre coordinates are already available
find_flock_centre( Type, Time, Point) :-
    get_prop( flock, Time, [Type, Point]), !.

% otherwise, find it
find_flock_centre( Type, Time, pt( X, Y)) :-
    findall( Loc, (Obj has_type Type, Obj is_alive, Obj is_nearby _,
        locn( Obj, Time, Loc)), List),
    ( % if this is the only object, then do not apply flock centering
        length( List, 1) -> X = -9999, Y = -9999, !
    ; List = [pt( A, B)|Rest], !, % else
        length( List, Number_in_list),
        find_sum_of_locs( A, B, Rest, pt(X_sum, Y_sum)),
        X is int( X_sum/Number_in_list),
        Y is int( Y_sum/Number_in_list),
        set_prop( flock, Time, [Type, pt( X, Y)]), !.

centering_calc( Poss, Centre, X) :-
    Diff is Poss - Centre,
    N is int( sqrt( abs( Diff))),
    Mid_point is int(( Poss + Centre)/2),
    rand_ize( The_point, Poss, Mid_point), % put some randomisation in it
    X is The_point - sign( Diff)*N.

/*                      meanders_forward                      */
/******/

Object meanders_forward :-
    Object is_alive, % make sure that object is alive
    get_prop( frame, current, Now),
    get_prop( level, meanders_forward, Level),
    get_prop( rule, number, RuleNo),
    do_meanders_forward( Object, Now, Level, Action),
    do( Action, RuleNo). % see infer, test...

do_meanders_forward( Object, Now, Level, Action) :-
    get_prev_locn( Object, Now, pt( X, Y)),
    which_area( pt( X, Y), Area), % find which area is object in
    which_direction( Object, Now, Area, Dir), % see below
    set_object_direction( Object, Now, Dir),
    get_prev_locn( Object, Now, pt( X, Y)),
    maxSpeed( Object, Speed),
    UpperLimit is Speed + 10,
    rand_ize( XRand, Speed, UpperLimit), % fixed, so it goes in one direction
    rand_ize( YRand, -20, 20), % fixed
    X1 is X - XRand,
    Y1 is Y + YRand,
    Action = poss_locn( Object, Now, Level, pt( X1, Y1)) .

```

## REFERENCES

## References

"With regard to the practice of quoting, in the margin, such books and authors as have furnished you with sentences and sayings for the embellishment of your history..."

Tobias Smollett's translation of Don Quixote (1986),  
André Deutsch

- Allen, J. (1978) *Anatomy of LISP*. McGraw-Hill, New York
- Andersen, P. B. (1992) Vector Spaces as the Basic Components of Interactive Systems: Towards a Computer Semiotics. *Hypermedia* 4(1), 53-76
- Agha, G. (1986) *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, Massachusetts
- Amkraut, S., Girard, M. & Karl, G. (1985) Motion Studies for a Work in Progress entitled 'Eurythmy'. In *SIGGRAPH Video Review*, 21, 2nd item, 3:58-7:35
- Badler, N. I., Barsky, B. & Zeltzer, D.; Eds. (1991) *Making Them Move - Mechanics, control and animation of Articulated Figures*. Morgan Kaufmann
- Badler, N. I., Phillips, C. B. & Webber, B. (1994) *Simulating Humans*. Oxford University Press, Oxford
- Baraff, D. (1990) Curved surfaces and coherence for non-penetrating rigid body simulation. *Computer Graphics (Proceedings SIGGRAPH)* 24, pp.19-28
- Barr, A., Cohen, P. R. & Feigenbaum, E. A. (1989) *The Handbook of Artificial Intelligence*. Volume IV. Addison-Wesley
- Barwise, J. & Perry, J. (1983) *Situations and Attitudes*. MIT Press, London
- Beardon, C. & Ye, V. (1995) Using Behavioural Rules in Animation. *Computer Graphics: Developments in Virtual Environments*. Academic Press, London, pp. 217-234
- Bekey, G. A., Iberall, T., Tomovic, R. & Liu H. (1991) Knowledge Based Models of Human & Robot Grasping. *Identification & System Parameter Estimation 1991*. 1&2, pp. 699-704
- Blumenthal, B. (1990) Strategies for Automatically Incorporating Metaphoric Attributes in Interface Design. *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, Snowbird, Utah, pp. 66-75
- Bratko, I. (1990) *PROLOG Programming for Artificial Intelligence*. Second Edition, Addison-Wesley, Wokingham
- Bruderlin, A. & Calvert, T. W. (1989) Goal-Directed, Dynamic Animation of Human Walking. *Computer Graphics* 23(3), 233-242

## References

- Canfield, D. S., Cypher, A. & Spohrer, J. (1994) KidSim: Programming Agents Without a Programming Language. *Communications of The ACM* 37(7), 55-67
- Chomsky, N. (1959) Review of Skinner's *Verbal Behaviour* *Language* 35, pp 26-58
- Chomsky, N. (1965) *Aspects of the Theory of Syntax*. MIT Press, Boston, Massachusetts
- Clocksin, W. F. & Mellish, C. S. (1981) *Programming in Prolog*. Springer-Verlag, New York
- Cohen, M. F. (1992) Interactive Spacetime Control for Animation. *Computer Graphics (Proceedings SIGGRAPH)* 26(2), 293-302
- Cypher, A. (1993) *Watch What I Do: Programming by Demonstration*. MIT Press, Boston, Massachusetts
- Delvin, K. (1991) *Logic and Information*, Cambridge University Press, Cambridge
- Delvin, K. (1992) Situation Theory and Design. *Department of Mathematics and Computer Science, Colby College, Waterville, Maine, USA*. Rewrite
- Douglas, S., Doerry, E. & Novik, D. (1990) QUICK: A User-Interface Design Kit for Non Programmers. *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, Snowbird, Utah, pp. 47-56
- Feldman, J. A.; Ed. (1985) *Cognitive Science*. 9(1). Special Issue on Connectionist Models and Their Applications.
- Fiévet, C. (1995) The 3rd dimension of Internet. *Cybersphere*, August 24, 1995. Available on the Internet <http://www.quelm.fr/CSphere/N3/Ana3DU.html>
- Foley, J. D., van Dam, A., Feiner, S. & Hughes, J. (1990) *Computer Graphics - Principles & Practice*. Second Edition, Addison-Wesley, Massachusetts
- Gal, A., Lapalme, G., Saint-Dizier, P. & Somers, H. (1991) *PROLOG for Natural Language Processing*. Wiley, Chichester
- Girard, M. (1986) Interactive Design of 3D Computer Animated Legged Animal Motion. *1986 Workshop on Interactive 3D Graphics*. Chapel Hill, North Carolina, October 1986
- Girard, M. & Maciejewski, A. A. (1985) Computational Modelling for the Computer Animation of Legged Figures. *Computer Graphics (Proceedings SIGGRAPH)* 19(3), 263-270
- Halas, J.; Ed. (1974) *Computer Animation*. Focal Press, London
- Hewitt, C. & Aitkinson, R. (1977) Parallelism and Synchronization in Actor Systems. *ACM Symposium on Principles of Programming Languages 4*, Los Angeles, California, January 1977
- Hu, D. (1989) *C/C++ for Expert Systems*. MIT Press, Boston, Massachusetts
- Hutchins, E. L., Hollan, J.D. & Norman, D. A. (1986) Direct Manipulation Interfaces, in *User Centred System Design*, Norman, D. A & Draper, S. W, Eds., Lawrence Erlbaum Associates, Hillsdale, NJ, pp. 87-127

## References

- Isaacs, P. M. & Cohen, M. F. (1987) Controlling Dynamic Simulation With Kinematic Constraints, Behaviour Functions and Inverse Dynamics. *Computer Graphics (Proceedings SIGGRAPH)* 21(4), 215-224
- Jackson, P. (1986) *Introduction to Expert Systems*. Addison-Wesley, Wokingham
- Koga, Y., Kondo, K., Kuffner, J. & Latombe, J. (1994) Planning Motions with Intentions. *Computer Graphics (Proceedings SIGGRAPH)* 28, pp. 395-408
- Kononenko, I. & Lavrac, N. (1988) *Prolog - Through Examples: A Practical Programming Guide*. SIGMA Press, Wilmslow
- Langton, C. (1989) *Artificial Life*. Addison-Wesley, Reading, Massachusetts
- Lasseter, J. (1987) Principles of Traditional Animation Applied to 3D Computer Animation. *Computer Graphics (Proceedings SIGGRAPH)* 21(4), 35-44
- Laurel, B. (1990) *The Art of Human-Computer Interface Design*. Addison-Wesley, Wokingham
- Lee, Y., Terzopoulos, D. & Waters, K. (1995) Realistic Modeling for Facial Animation. *Computer Graphics (Proceedings SIGGRAPH)* 29, pp. 55-62
- Lloyd, J. W. (1984) *Foundations of Logic Programming*. Springer-Verlag, Berlin
- Marcus, C. (1986) *Prolog Programming: Application for Database Systems, Expert Systems, and Natural Language Systems*. Addison-Wesley, Reading, Massachusetts
- McCarthy, J. (1959) Programs with Common Sense. *Mechanization of Thought Process*. Her Majesty's Stationary Office, London. pp. 77-84
- McDermott, J. (1980) R1: An Expert in the Computer Systems Domain, in *Proceedings of the First Annual National Conference on Artificial Intelligence*, Stanford, CA, pp. 269-271
- McKenna, M. & Zeltzer, D. (1990) Dynamic Simulation of Autonomous Legged Locomotion. *Computer Graphics (Proceedings SIGGRAPH)* 24, pp. 29-38
- Miller, G. (1988) The Motion Dynamic of Snakes & Worms. *Computer Graphics (Proceedings SIGGRAPH)* 22(4), 169-178
- Moore, M. & Wilhelms, J. (1988) Collision detection and response for computer animation. *Computer Graphics (Proceedings SIGGRAPH)* 22(4), 289-298
- Newman, W. & Sproull, R. (1979) *Principles of Interactive Computer Graphics*. 2nd Edition, McGraw-Hill, New York
- Ngo, J. T. & Marks, J. (1992) Physically Realistic Trajectory Planning in Animation: A Stimulus-Response Approach. *Technical Report TR-21-92. Centre for Research in Computing Technology, Harvard University*, October 1992
- Ngo, J. T. & Marks, J. (1993) Spacetime Constraints Revisited. *Computer Graphics (Proceedings SIGGRAPH)* 27, pp. 343-350
- O'Donnell, T. J. & Olson, A. J. (1981) GRAMPS - A Graphics Language Interpreter for Real-Time, Interactive, Three-Dimensional Picture Editing and Animation. *Computer Graphics (Proceedings SIGGRAPH)* 15(3), 133-142



## References

- Phillips, C. B. & Badler, N. I. (1991) Interactive Behaviours for Bipedal Articulated Figures. *Computer Graphics (Proceedings SIGGRAPH)* 25(4), 359-362
- Phillips, C. B., Zhao, J. & Badler, N. I. (1990) Interactive Real-time Articulated Figure Manipulation Using Multiple Kinematic Constraints. In *ACM Proceedings of Symposium on Interactive 3D Graphics (Snowbird, Utah, March, 1990)*. 24, pp. 245-250
- Pugh, J. R. (1984) Actors - The Stage is Set. *SIGPLAN Notices* 19(3), 61-65
- Raibert, M. H. & Hodgins, J. K. (1991) Animation of Dynamic Legged Locomotion. *Computer Graphics (Proceedings SIGGRAPH)* 25(4), 349-358
- Repenning, A. (1993) Agentsheets: A Tool for Building Domain Oriented Dynamic, Visual Environments. *PhD. Thesis, University of Colorado*
- Rettig, M., Morgan, T., Jacobs, J & Wimberly, D. (1989) Object Oriented Programming in AI. *AI EXPERT*, January 1989, pp.53-69
- Reynolds, C. W. (1982) Computer Animation with Scripts and Actors. *Computer Graphics (Proceedings SIGGRAPH)* 16(3), 289-296
- Reynolds, C.W. (1987) Flocks, Herds, and Schools: A Distributed Behavioural Model. *Computer Graphics (Proceedings SIGGRAPH)* 21(4), 25-34
- Rogers, D. F. (1989) *Procedural Elements for Computer Graphics*. McGraw-Hill, London
- Rijkema, H. & Girard, M. (1991) Computer Animation of Knowledge-Based Human Grasping. *Computer Graphics (Proceedings SIGGRAPH)* 25(4), 339-348
- Schank, R. C. (1980) Language and Memory, *Cognitive Science* 4(3), 243-284
- Selig, W. J. & Johannes, J. D. (1990) Reasoning Visualization in Expert Systems: The Applicability of Algorithm Animation Techniques, in *3rd International Conference on Industrial & Engineering Applications of AI & Expert Systems 1 & 2*, pp. 457-466
- Shapiro, S.C. Ed. (1987) *Encyclopedia of Artificial Intelligence*. Volume 1 & 2. John Wiley, Chichester
- Shneiderman, B. (1989) Direct Manipulation: A Step Beyond Programming Languages, in *Human-Computer Interaction: A Multidisciplinary Approach*, Beaecker & Buxton Eds., Morgan Kaufmann Publisher, Inc. pp. 461-467
- Shortliffe, E.H. (1976) *Computer-Based Medical consultation: MYCIN*. Americal Elsevier, New York
- Skinner, B. F. (1974) *About Behaviourism*. Alfred Knopf, New York
- Smith, T. G.; Ed. (1985) *Industrial, Light & Magic: The Art of Special Effects*. Columbus, London
- Stefik, M. & Bobrow, D. G. (1984) Object-Oriented Programming: Themes and Variations. *The AI Magazine*, pp. 40-61

## References

- Stern, G. (1983) Bbop - A Program for 3-Dimensional Animation. *Nicograph 1983 Proceedings*, pp. 403-404
- Strauss, P. S. & Carey, R. (1992) An Object-Oriented 3D Graphics Toolkit. *Computer Graphics (Proceedings SIGGRAPH) 26(2)*, 341-347
- Suchman, L. A. (1987) *Plans and Situated Actions: The Problem of Human-Machine Communication*. Cambridge University Press, Cambridge
- Thalmann, D. (1994) Animating Autonomous Virtual Humans in Virtual Reality. *13th World Computer Graphics Congress 94*, 3, pp. 177-183
- Thalmann, N. M. & Thalmann, D.; Eds. (1994) Introduction: Creating Artificial Life in Virtual Reality. In *Artificial Life and Virtual Reality*, pp. 1-10, Wiley, London
- Tu, X. & Terzopoulos, D. (1994) Artificial Fishes: Physics, Locomotion, Perception, Behaviour. *Computer Graphics (Proceedings SIGGRAPH) 28*, pp. 43-50
- Uchiki, T., Ohashi, T. & Tokoro, M. (1983) Collision Detection in Motion Simulation. *Computers and Graphics* 7 No. 3-4
- Vince, J. (1985) *Computer Graphics for Graphic Designers*. Frances Pinter, London
- Vince, J. (1992) *3-D Computer Animation*. Addison-Wesley, Wokingham
- Waters, K. (1987) A Muscle Model for Animating Three-Dimensional facial Expression. *Computer Graphics (Proceedings SIGGRAPH) 21(4)*, 17-24
- Wejchert, J. & Haumann, D. (1991) Animation Aerodynamics. *Computer Graphics (Proceedings SIGGRAPH) 25(4)*, 19-22
- Wilhelms, J. & Skinner, R. (1990) A "Notion" for Interactive Behavioural Animation Control. *IEEE Computer Graphics and Applications* 10, pp. 14-22
- Winston, P. H. (1977) *Artificial Intelligence*. Addison-Wesley, Wokingham
- Winston, P. H. & Horn, K. P. H. (1984) *Lisp*. 2nd Edition, Addison-Wesley, Wokingham
- Witkin, A. & Kass, M. (1988) Spacetime Constraints. *Computer Graphics (Proceedings SIGGRAPH) 22(4)*, 159-168
- Wooldridge, M. & Jennings, N. R.; Eds. (1995) *Intelligent Agents*. Springer-Verlag, London
- Wooldridge, M., Müller, J. P. & Tambe, M.; Eds. (1996) *Intelligent Agents II*. Springer-Verlag, London
- Yao, Y., Jawahir, I. S., Jamieson, D. & Fang, X. D. (1990) Computer Animation of Chip Flow and Chip Curl in an Expert Process Planning System for Metal Machining. *Transactions of the North American Manufacturing Research Institution of SME*. pp. 161-166
- Ye, V. (1995) Expert Systems in Computer Animation. *Digital Creativity: (Proceedings CADE'95)*, University of Brighton, pp. 222-227

## PUBLISHED PAPERS

(1)

**Expert Systems in Computer Animation  
Production Environments (ESCAPE)**

Victor Ye

Paper presented at CADE Conference,

University of Brighton,

18-21 April 1995.

As Printed in Digital Creativity: Proceedings CADE '95, pp 222-227.

## **Expert Systems in Computer Animation Production Environments (ESCAPE)**

**Victor Ye**

Rediffusion Simulation Research Centre,  
University of Brighton,  
Grand Parade,  
Brighton  
BN2 2JY, UK

Email: V.T.J.Ye@bton.ac.uk

### **Abstract:**

Traditional computer animation environments can be enhanced by adding rules that describe the behaviour of objects and an inference engine. An environment for the design of computer animations incorporating an expert system approach is described. In addition to direct manipulation of objects the environment allows users to describe behavioural rules based upon both their physical and non-physical attributes. These can be interpreted to suggest the transition from frame to frame or to automatically produce a longer animation. The output from the system can be integrated into a commercially available 3D modelling and rendering package. Experience indicates that a hybrid environment, mixing algorithmic and rule-based approaches, would be very promising and offer benefits in application areas such as creating realistic background scenes and modelling human beings or animals either singly or in groups.

### **Keywords:**

Artificial life, behavioural models, computer animation, computer animation environments, expert systems, human modelling, intentional action, rule-based systems.

## 1 Introduction

To date most computer systems that allow for three dimensional modelling and animation are concerned entirely with the physical properties of objects. That is to say, their objects are modelled by being given a physically defined shape, a number of properties (e.g. colour) and a location and orientation in three dimensional space. A 2D representation of an arrangement of such objects (a single 'frame') is produced by specifying the location and orientation of light sources and the point of view. An animated sequence is produced by specifying how these properties change from frame to frame. This can be done explicitly (i.e. by hand) but this is expensive; or implicitly by means of an algorithm (e.g. Miller, 1988; Waters, 1987; Wejchert & Haumann, 1991). Systems can also be built that provide a set of general purpose procedures that are independent of any particular domain. Craig Reynolds (1982), for example, developed the Actor/Scriptor Animation System (ASAS), a full programming language especially designed for animation and graphics that employs a procedural notation.

There are some very practical reasons why one might wish to supplement these approaches based solely upon the physical attributes of objects by incorporating some ideas from cognitive science. Within complex domains, and particularly where intentional action is involved, capturing realistic behaviour can be very time consuming and expensive. Convincing animations require both complexity and irregularity and these cannot be easily captured algorithmically, also, there is a tendency towards repetition if the algorithms are not made more complex as the length of the animation increases.

In this paper we describe an attempt to enhance physicalist approaches in a principled way by incorporating one of the best known ideas from the field of artificial intelligence and cognitive science, namely expert systems (or rule-based systems). A working rule-based approach environment is described, which is based upon the functions of two different types of user: the end user, who is the person producing a particular animation; and the domain expert, who provides knowledge about the domain being modelled. In order to validate the model, three different domains (jellyfish world, traffic junction with cars, and flocking birds) are tested that use the software environment. Finally some possible application areas and some limitations of the approach will be discussed and an indication given of future work.

## 2 An Artificial Intelligence approach to animation

There are strong similarities between the physicalist approach embodied in current animation software and the psychological theory of behaviourism put forward by B. F. Skinner (1974). Behaviourists argue that a true science of behaviour must be modelled on the physical sciences.

Within such domain, there is no mechanism for representing the inner states of objects and all behaviour has to be expressed in terms of physically describable features.

Cognitive psychology on the other hand, is based upon the model of the human mind as a processor of information, which exhibits the characteristics we human take for granted - understanding language, learning, reasoning, solving problems and so on.

Rijpkema and Girard (1991) introduced an expert system when modelling a human hand grasping an object. In their system, the classic shape of the target object is first identified (e.g. block, sphere, cone, etc.) and a grasping strategy associated with this is derived from the knowledge-base of class specific, parameterised techniques. The final grasping motion is then adjusted to adapt to the object's deviation from the classic shape.

The above system is designed for the particular task of grasping and the animator who wishes to modify the rules must become knowledgeable about expert systems. The opposite approach is adopted by KidSim (Canfield Smith, Cypher & Spohrer, 1994), a rule-based environment that aims to put the writing of rules into the hands of the end users - in this case children. KidSim is an environment for building simulations by declaring a set of agents and graphically describing a set of rules by showing examples. The rules are not displayed in symbolic form but rather as transformations of the graphical field. The authors claim that KidSim is the model for more adult simulation environments, but while the possibility of defining rules through user-friendly interfaces is to be welcomed, its current limitations on knowledge representation and its inability to describe complex interactions will need to be overcome before more sophisticated domains can be simulated. Nevertheless, KidSim does provide an interesting model of user control within a rule-based approach to modelling and animation.

In Artificial Fishes (1994), Tu & Terzopoulos designed an underwater world of fishes. These fishes are hard-coded to have sensors and intention, and bringing computer animation into the Artificial Life era.

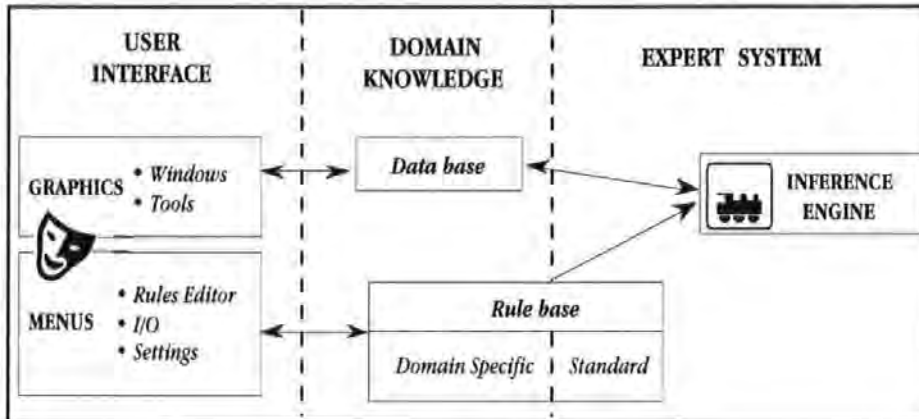
### **3 The software environment**

#### **3.1 Software Architecture**

The three functions in the development process are reflected in the architecture of the software environment, as shown in Figure 1.

At the heart of the system lay an inference engine such as one may find in any standard expert system. When invoked it will examine the rule base and attempt to match the conditional part of

each rule against the data base. Where successful this will generate a set of possible actions. When all the rules have been fully explored the system will examine all the possible actions for each object and will decide how it should be transformed for the next frame. This will update the database which, in turn, will update the graphical window being used by the animator.



**Figure 1** Software architecture

### 3.2 User Interfaces

The Graphical Interface used by animators contains a window showing the current state of the animation. In the prototype a 2D graphical representations of objects is used as shown in Figure 2.

A set of tools is provided. The Select tool is used for the direct manipulation of objects on the screen, while the Information tool allows for updating of the database by changing symbolic information about an object. See Figure 3.

In addition there are menu options that allow the user to: load or unload a description of a particular animation; load or unload the definition of a particular domain; allow labels to be displayed next to objects on the screen; invoke the rules interface; direct output to a Swivel-3D Script file; switch the graphical display on or off while the inference engine is running.

Rules can be written and amended through the use of a specially designed Rules Editor. See Figure 4.

## 4 Conclusions

The preceding examples illustrate some potential for rule-based enhancements to animation software. They provide the opportunity to use a new kind of knowledge to simulate the behaviour






Figure 2 The animator's graphical interface

Object Info

Object Name: jelly1

☒ is\_alive



Object Type: JELLYFISH

☐ is\_static

Object Attributes

Mass: 5

Current Location:

Size H: 20

H = 173

Y: 20

Y = 57

Visibility: 120

Stopping Time: NOT\_A

MaxSpeed: 10

Stopping Dist: NOT\_A

Defaults

Change

Save

Cancel

Figure 3 Information about an object

Rule Editor™

rule1

condition1

Object 1

object1

is\_alive

is\_static

is\_harmless

Relationship

is\_nearby

Del Cond.

New Cond.

Object 2

object2

is\_alive

is\_static

is\_harmless

action1

object1

meander

rule1:

if, object1, is\_nearby, object2, that, is\_static,

then, object1, meander.

Cancel

Delete Rule

New Rule

Check Rule

Compile Rule

Figure 4 The Rules Editor

of objects within a modelled environment and these initial demonstrations have helped clarify the type of architecture needed, the functionality that might be expected and the types of user who may be able to successfully exploit such a system. They also provide us with some useful insights into the limitations of the approach, the types of animation problem where it might be useful and the significant work that needs to be done.

The existing system is a prototype and cannot produce quality output. To do this the approach needs to be integrated into a more sophisticated animation production system. We have described the architecture that would be needed for this to be done. To make a readily usable system it is necessary to provide a good set of predefined predicates and operators in terms of which domain-specific behavioural predicates may be defined. We have gone some way in identifying a useful set but it is by no means complete.

The smallness of the rule bases in the examples raises the question of whether the approach will scale up. The question here is not primarily a technical one, for there is little problem (except execution time) in presenting the inference engine with 250 rules instead of 15. The question is more whether domain experts and animators can comprehend what the system is doing if it contains so many rules. In this respect the interesting research question, we believe, is how animators might use this new facility creatively and this will depend crucially on their ability to understand the effect of their actions. The text-based log file may give some insight into the workings of the inference engine but it is not always possible to quickly extract from it a clear understanding of why an object is behaving in a peculiar manner.

The major problem is not the ability of the inference engine to cope with things on a larger scale but the ability of users to visualise the decisions the system is making. To this end there is a need to develop specific techniques for visualising the actions of the inference engine itself. In the future we would like to be able to switch on an option that makes visible the vectors acting on an object at each stage in the inferencing and relates them to the rules from which they originated. We even go so far as to suggest a visual representation for rules that might be called up and assigned to objects, possibly with strengths that may be set by the animator.

Rule-based systems seem more appropriate for some types of problem than others. As Rijpkema and Girard (1991) discovered, an expert system was useful for determining a grasping strategy, but a procedural approach was best suited to the finer movements involved in making contact. A rule-based approach is not likely to serve well in areas that require much mathematical calculation, for example, but an algorithmic approach is also unlikely to serve well where strategic planning is required. The solution is therefore the flexibility of hybrid systems, incorporating both algorithmic and rule-based approaches.

There seem to be several situations where such an approach could give real benefit. One is the creation of a realistic animated background scenery, the detail of which is not particularly

significant. For example, an animation may need a background of a realistic street scene that is not predictable, but neither is it particularly remarkable. A suitably set up domain could generate endless such scenes which would be very tedious or difficult to produce by any other method.

The other area of beneficial use is where animators can use expert knowledge about the behaviour of humans (or other animals) in order to plan foreground animations. Thalmann is already incorporating some modelling of purposeful human behaviour into the animation of photo-realistic human models (Thalmann, 1994). The system currently uses pre-written scenes derived from the work of Schank (1980). Schank's scripts are effective for stereotypical scenes but are limited in modelling more open scenarios where there may be a need for planning or, more generally, for a form of situated action.

It is not possible, at this point in time, to determine the best representations and strategies for all domains, or even for each particular domain. The precise nature of the approach will develop through experimentation of the kind we have described in this paper. What we do argue is that animating sophisticated beings with minds of their own requires a modelling environment that can represent internal mental states and internal mental processes.

## References

- Canfield Smith, D., Cypher A. & Spohrer J. (1994) KidSim: Programming Agents Without a Programming Language. *Communications of The ACM* 37(7), 55-67.
- Miller G. (1988) The Motion Dynamic of Snakes & Worms. *Computer Graphics* 22(4), 169-178.
- Reynolds C.W. (1987) Flocks, Herds, and Schools: A Distributed Behavioural Model. *Computer Graphics* 21(4), 25-34.
- Reynolds C.W. (1982) Computer Animation with Scripts and Actors. *ACM SIGGRAPH '82 Proceedings* 16 (3), 289-296.
- Rijkema H. & Girard M. (1991) Computer Animation of Knowledge-Based Human Grasping. *Computer Graphics* 25(4), 339-348.
- Schank, R.C. (1980) Language and Memory. *Cognitive Science*, 4 (3), 243-284.
- Skinner, B.F. (1974) About Behaviorism. Alfred Knopf, New York.
- Thalmann, D. (1994) Animating Autonomous Virtual Humans in Virtual Reality. In: Duncan, K. & Kreuger, K. (Editors) *Proceedings 13th World Computer Congress 94, Volume 3*. Elsevier Science B.V. (North-Holland), Amsterdam, 177-184.
- Tu X. & Terzopoulos D. (1994) Artificial Fishes: Physics, Locomotion, Perception, Behaviour. *Computer Graphics* 28, 43-50.
- Waters K. (1987) A Muscle Model for Animating Three-Dimensional facial Expression. *Computer Graphics* 21(4), 17-24.
- Wejchert J. & Haumann D. (1991) Animation Aerodynamics. *Computer Graphics* 25(4), 19-22.

(2)

## **Using Behavioural Rules in Animation**

Colin Beardon & Victor Ye

Paper presented at Computer Graphics International 95,

University of Leeds,

26-30 June 1995

In Computer Graphics: Developments in Virtual Environments, pp 217-234.

Academic Press, London.

# Using Behavioural Rules in Animation

Colin Beardon & Victor Ye

## 1 Introduction

To date most computer systems that allow for three dimensional modelling and animation are concerned entirely with the physical properties of objects. That is to say, their objects are modelled by being given a physically defined shape, a number of properties (e.g. colour) and a location and orientation in three dimensional space. A 2D representation of an arrangement of such objects (a single 'frame') is produced by specifying the location and orientation of light sources and the point of view. An animated sequence is produced by specifying how these properties change from frame to frame. This can be done explicitly (i.e. by hand) but this is expensive. Normally it is done implicitly by means of an algorithm (e.g. Waters, 1987; Wejchert & Haumann, 1991). Systems can also be built that provide a set of general purpose procedures that are independent of any particular domain. Craig Reynolds (1982), for example, developed the Actor/Scriptor Animation System (ASAS), a full programming language especially designed for animation and graphics that employs a procedural notation.

There are some very practical reasons why one might wish to supplement these approaches based solely upon the physical attributes of objects by incorporating some ideas from cognitive science. Within complex domains, and particularly where intentional action is involved, capturing realistic behaviour can be very time consuming and expensive. Convincing animations require both complexity and irregularity and these cannot be easily captured algorithmically. Even if they can be achieved for short sequences, there is a tendency towards repetition if the algorithms are not made more complex as the length of the animation increases.

In this paper we describe an attempt to enhance physicalist approaches in a principled way by incorporating one of the best known ideas from the field of artificial intelligence and cognitive science, namely expert systems (or rule-based systems). A working environment in which a rule-based approach coexists with more traditional modelling and animation methods will be described. The environment is based upon the functions of two different types of user: the end user, who is the person producing a particular animation; and the domain expert, who provides knowledge about the domain being modelled. A particular prototype software system is described and the use that each type of developer will make of the system is shown. In order to validate the model, four sample animations are described that use the software environment. Finally some possible application areas and some limitations of the approach will be discussed and an indication given of future work.

## 2 Background

### 2.1 An Artificial Intelligence approach to animation

There are strong similarities between the physicalist approach embodied in current animation software and the psychological theory of behaviourism put forward by B. F. Skinner (1974). Behaviourists argue that a true science of behaviour must be modelled on the physical sciences. They base their approach on observing behaviour and, in particular, they seek to determine the relationships between the observable stimuli received by a subject and the observable response that the subject exhibits as a result. Behaviourists generally deny the significance of internal states and some extreme behaviourists argue that all mentalistic terms are theoretically unnecessary as they can be redefined in terms of observable, physically describable behaviour. Modelling and animation software that only allows the representation and manipulation of physically observable features of objects is based upon similar assumptions. Within such software there is no mechanism for representing the inner states of objects and all behaviour has to be expressed in terms of physically describable features.

Cognitive psychology is an alternative approach which, in general, is based upon the model of the human mind as a processor of information. Explanations of the behaviour of human beings is not restricted to descriptions of input and output data, but can also refer to the internal representation, storage and processing of that data. Cognitive science and the more technical field of artificial intelligence seek to complement cognitive psychology by building computational models of sufficient richness. In order to explore the potential of this approach, environments must be developed in which the manipulation of data that refers to physical attributes can be enhanced through the use of a model of internal mental processes and expert systems seem particularly well-suited to represent such processes.

Rijkema and Girard (1991) introduce an expert system when modelling a human hand grasping an object. While previous studies of grasping have sought to record and analyse the physical motions, there argue the need to focus on the problem of synthesising grasping motion. Observing that people adopt different grasping strategies depending upon the shape of the object and the angle of approach, they use an expert system approach to determine the appropriate strategy. In their system, the classic shape of the target object is first identified (e.g. block, sphere, cone, etc.) and a grasping strategy associated with this is derived from the knowledge-base of class specific, parameterised techniques. The final grasping motion is then adjusted to adapt to the object's deviation from the classic shape.

In the system designed by Rijkema and Girard the expert system is designed for the particular task of grasping and the animator who wishes to modify the rules must become knowledgeable about expert systems. The opposite approach is adopted by KidSim, a rule-based environment that

aims to put the writing of rules into the hands of the end users - in this case children (Canfield *et al.*, 1994). KidSim is an environment for building simulations by declaring a set of agents and graphically describing a set of rules by showing examples. The rules are not displayed in symbolic form but rather as transformations of the graphical field. The authors claim that KidSim is the model for more adult simulation environments, but while the possibility of defining rules through user-friendly interfaces is to be welcomed, its current limitations on knowledge representation and its inability to describe complex interactions will need to be overcome before more sophisticated domains can be simulated. Nevertheless, KidSim does provide an interesting model of user control within a rule-based approach to modelling and animation.

## 2.2 A working environment for animators

In the world as described by physics there are certain constraints upon objects. For example, two solid objects cannot occupy the same point in space, and (on earth) an unsupported object will fall due to the effect of gravity. Modelling and animation systems can be envisaged that check such constraints and make objects bounce off each other if they collide and fall to the ground if they lack support. In such systems a physical model exists in which it is easy to say that all physical objects display a certain kind of "behaviour".

The rule-based enhancement that we propose enables an extension of the scope of such rules to include knowledge about the domain being modelled. By 'domain knowledge' we mean the sort of knowledge that refers to the subject matter or content of the animation. For example, in an animation of an underwater scene knowledge may be available of how various creatures behave under different conditions (e.g. if a jellyfish is hungry and it detects the presence of a prawn it will pursue it; or, if a prawn is being pursued by a predator it will try to hide). The proposed system provides a way of using such knowledge so that the objects can be constrained to act in a purposeful manner. It is hoped that this will result in animations that are not only more realistic, but whose development cost is fixed regardless of their duration.

The environment proposed has many traditional features and one new one. Individual objects can be defined in exactly the same way as in existing 3D modelling systems. In order to create individual frames of an animation, objects may be manipulated manually through a traditional graphical interface. A scripting system can also be employed to algorithmically describe changes in the world from frame to frame. To these traditional methods an additional facility is added by means of which objects may have non-physical attributes and the state of each object is derived by an inference engine from behavioural rules that are entered separately.

This rule-based part of the system can be left to run unattended and will, if properly set up, automatically generate an animation. However, the expert system component is not there to replace skilled animators, but rather to assist them. The animator has control over the rules and can experiment with them, observing their effect on the animation. The animator can also interrupt the

generated animation and use one of the other modes (manual manipulation or algorithmic) to override it.

With such an environment it is necessary to identify different functions in the development process. The software experts develop the core expert system component, the part that is common to all application domains. It consists of an inference engine to interpret the rules against a data base and some very low-level rules that provide other users with a useful set of standard predicates and operators. The domain expert sets up a particular world by modelling the different types of object, specifying attributes they should have and their default values, and writing the main behavioural rules they follow. The domain expert may also specify some lower level domain specific predicates and operators. The animator, whose task it is to produce particular animations, is able to update the data base by direct manipulation, update the rule base through a special rules interface and invoke the inference engine.

### 3 The software environment

#### 3.1 Software Architecture

The three functions in the development process are reflected in the architecture of the software environment, as shown in Figure 1.

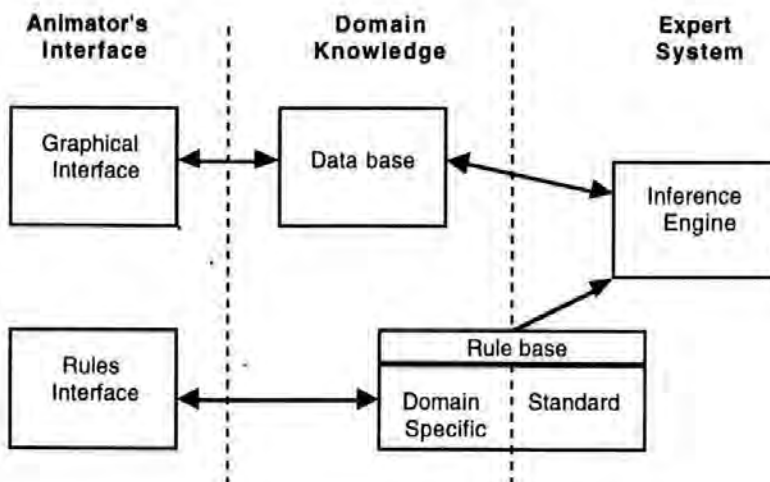


Figure 1 Software architecture

At the heart of the system lay an inference engine such as one may find in any standard expert system. When invoked it will examine the rule base and attempt to match the conditional part of each rule against the data base. Where successful this will generate a set of possible actions. When all the rules have been fully explored the system will examine all the possible actions for each



object and will decide how it should be transformed for the next frame. This will update the database which, in turn, will update the graphical window being used by the animator.

Great care has been taken in this design to separate domain-specific information from the other parts of the system which remain as a general-purpose animation environment capable of handling a range of domains.

The rules are of the standard "if ... then ..." format. In an ideal version of the system users would be able to express their rules in natural language:

*if a prawn knows it is being pursued and it is aware of the presence of a medium or large rock then it will tend to try to get behind it.*

Allowing unrestricted natural language input is currently not feasible and so a more formal language has been developed within which rules are expressed to the system. For example,

```
if    object1 is_a prawn and is_nearby predator
then  object1 plans_getaway.
```

The language is composed of:

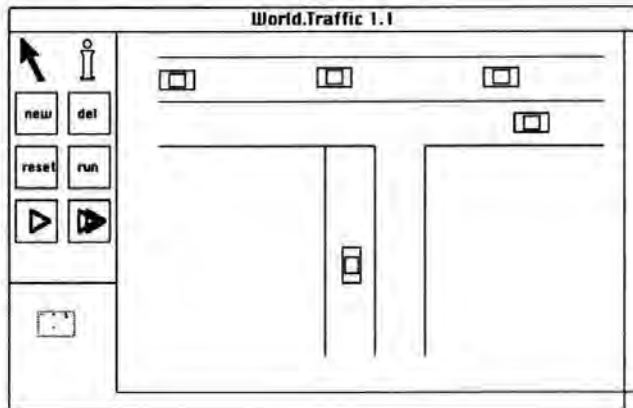
- keywords (e.g. 'if', 'then' & 'and'),
- variable names (e.g. 'object1'),
- properties (e.g. 'prawn', 'predator'),
- predicates (e.g. 'is\_a', 'is\_nearby'), and
- operators (e.g. 'plans\_getaway').

Some commonly-used predicates and operators are made available with the inference engine while other, more domain-specific, operators are defined by the domain expert or the animator. The language is described in more detail in Section 3.3.

A prototype system has been built at the University of Brighton using LPA MacProlog32 running on an Apple Macintosh series II machine. The system described here models objects in a two-dimensional world which enables us to concentrate upon significant design issues. A version that contains a three-dimensional modelling environment has been built and while this introduces more complexity it raises no fundamentally new issues. The 2D prototype is sufficient to explore the basic architecture of the system without paying particular attention to the quality of the graphical output. To demonstrate its potential it is possible to create a corresponding three-dimensional world in Swivel 3D™ and for the expert system to generate a rendered 3D animation as a QuickPICS file.

### 3.2 Graphical Interface

The Graphical Interface used by animators contains a window showing the current state of the animation. In the prototype a 2D graphical representations of objects is used as shown in Figure 2.



**Figure 2** The animator's graphical interface

The eight tools shown on the left can be considered as two sets of four. The tools at the top are for changing the status of objects in the data base (and hence on the screen).

- Select                to move an object;
- Information        to view or update symbolic information about an object;
- New                 to create a new object (by invoking a dialog);
- Delete              to delete an object.

The Select tool is used for the direct manipulation of objects on the screen, while the Information tool allows for updating of the database by changing symbolic information about an object (Figure 3).

**Figure 3** Information about an object

The lower tools are for controlling the use of the expert system.

- **Reset** to reset the objects to an original state;
- **Run** to invoke the inference engine and create an animation;
- **Playback Single** to view the playback of the animation in Single format (only the current frame is shown);
- **Playback Multi** to view the playback of the animation in Multi format (the trace of each object is shown).

In addition there are menu options that allow the user to:

- load or unload a description of a particular animation;
- load or unload the definition of a particular domain;
- allow labels to be displayed next to objects on the screen;
- invoke the rules interface;
- direct output to a Swivel-3D Script file.
- switch the graphical display on or off while the inference engine is running.

On current equipment the inference engine does not run in real time and the production of graphical output slows down its operation further. For this reason it is often advantageous to switch off the graphical display while generating an animation and, when it has been completed, play it back in Single or Multi format.

### 3.3 Rule Base & Rule Interface

As indicated in Section 3.1, a formal language has been developed for the expression of rules. The grammar of rules is defined using a Definite Clause Grammar (DCG) (Clocksin & Mellish, 1984) which has a number of advantages. DCG definitions are a standard part of the Prolog language and thus can be expressed within the same programming language as the inference engine. Merely by defining the grammar in DCG form, a parser is automatically available and can be invoked by simply using the 'parse' function or one of its variants. This makes it possible for users to directly enter rules as free text if they prefer, though a dialogue box interface to the rule base is also provided. Looking to the future, it is possible that a more sophisticated language for rules may be developed and in this case any new grammar written in DCG could be incorporated without affecting other parts of the system (except the Rules Interface dialogue box). The grammar used involves considerable use of arguments to check context-sensitive features and extract useful data structures but the basic syntax of rules is as described in Figure 4.

rule	--> [if], conditions, [then], actions.
conditions	--> condition, [and], conditions.
conditions	--> condition.
actions	--> action, [and], actions.
actions	--> action.
condition	--> noun_phrase, verb_phrase.
action	--> verb_phrase.
action	--> condition.
noun_phrase	--> noun, rel_clause.
noun_phrase	--> proper_noun.
verb_phrase	--> vp, [and], vp.
verb_phrase	--> vp.
vp	--> trans_verb, noun_phrase.
vp	--> intrans_verb.
rel_clause	--> [that], verb_phrase.
rel_clause	--> [].
noun	--> [Variable] {member_of(Variable, [object1, object2, ... ])}.
noun	--> [Type], {member_of( Type, types_list)}.
proper_noun	--> [Name], {member_of( Name, actors_list)}.
trans_verb	--> [Trans], {member_of( Trans, trans_list)}.
intrans_verb	--> [Intrans], {member_of( Intrans, intrans_list)}.

Figure 4 DCG for rules

Rules can be written and amended through the use of a specially designed dialog box (see Figure 5). The Rules Interface dialog box is divided into two main sections. The top section reflects the grammatical structure of a rule. The pop-up menus provide all the terms that are valid at the particular place in the rule so that the animator is restricted to writing only valid rules. The menus also allow the animator create new variable names if they wish to refer to an object not already known to the system. The lower section of the dialogue box contains a scrolling field in which a text version of the rule that is being created appears. The user may enter text directly into this field, though syntax errors may then be introduced which will only be detected when the system comes to parse the rule.

The dialog box can be used to create new rules and amend existing rules. If an existing rule number is selected then the dialogue box is automatically updated to reflect that particular rule. The "Check Rule" button will ensure that the screen information is consistent and parse the text version for syntax errors. The "Compile Rule" button will permanently update the Rule base for the selected rule.

The semantics of rules is based upon the way that the inference engine interprets them. In general, the conditional clause is broken up into simple conditions (left to right) and each condition is matched against the database to see if there is an item that matches it. A match occurs if an

exactly equivalent item is found in the database, or if an equivalent item can be found by binding an unbound variable name to an object name. All conditions within the conditional clause must succeed in matching with the same bindings for the conditional clause as a whole to succeed.

**Figure 5** The Rules Interface dialog box

If all the conditions are satisfied, then the action clause is interpreted using the bindings generated by the conditional clause. Each simple action has one of four possible interpretations:

- if it contains a valid Prolog expression, then it will be passed to the Prolog system and interpreted,  
TimeN is 4 by Prolog.
- if it contains an 'Undo' instruction, then the matching item will be removed from the database (no action is taken if the item that is not in the database),  
Deleting fact: 13 : plan (1 to 16) by rule 2.
- if it contains a predefined predicate or operator, then it will be resolved,  
Distance Between jelly1 and prawn1 is 88 by predicate 4.
- otherwise the item will be added to the database (no action is taken if the item is already present in the database),  
New fact : 22 poss\_locn(prawn1, 2, 4, pl(243, 164)) by rule 5.

Predefined predicates and operators are those which are commonly required yet are not readily available in the desirable form in standard Prolog. (Predicates appear in conditions and operators appear in actions.) For example, we may frequently wish to use a predicate 'less\_than' (as in 'A

less\_than B') but find that the standard Prolog predicate '<' does not always behave as we would expect. We may need to embed the call in code to check whether the arguments are instantiated to numbers, for example. We handle such cases by writing special rules, providing them to every domain and making them relatively transparent. Users may therefore use such predicates as if they were primitives of the rule-writing language. Some examples are: 'between', 'after', 'above', 'greater than', 'greater than or equal to', 'is bigger than', 'is same size as' and 'is same as'.

### 3.4 Data Base

The database contains descriptions of the domain and the objects within it. A domain is specified by declaring predicates and operators and defining each type of object. They are declared as having either one or two arguments (and will therefore be constrained to appear as intransitive or transitive verbs):

```
predicate_1_List( [is_alive, is_static, is_stopped, is_turning]).
predicate_2_List( [is_near]).
action_1_List( [move_forward, slow_down, stay_there]).
action_2_List( [avoid_collision_with, turn]).
```

A type is declared by describing its graphic(s) and default facts that will apply if no other facts are declared:

```
set_prop( 'CAR', east, resource( 500)),
defaults mass( 'CAR', 1).
defaults size( 'CAR', 2, 2).
defaults vis( 'CAR', 100).
defaults maxSpeed( 'CAR', 30).
defaults stopTime( 'CAR', 6).
defaults stopDistance( 'CAR', 40).
defaults property( 'CAR', [alive, left]).
defaults name( 'CAR', car).
```

Particular objects are declared by entering facts. There are certain standard formats for representing common data which are referred to by predefined predicates and operators. Other facts can be added when the problem is set up or by the inference engine.

- instances of a type: obj\_type(<object>, <type>)
 

```
fact obj_type( car1, 'CAR').
```
- attributes of a type: property(<object>, <property list>)
 

```
fact property( car1, [alive, none]).
```
- location of an object: locn(<object>, <time>, <point>)
 

```
fact locn( car1, 1, pt( 33, 28)).
```

- planning task: plan <time1> to <time2>  
fact plan 1 to 12.

From within the general environment different domains can be opened one at a time. Domains can be kept as files.

### 3.5 Inference Engine

The inference engine primarily uses the standard expert system approach of forward-chaining but has some variations. The system allows for a customised resolution of all the actions that could be fired for any particular step and, when seeking to apply basic operators it will temporarily employ backward chaining. The reason why forward-chaining is generally used, even in the case of directed animations where some goal is specified, is because it reflects a more open cognitive model. Backward chaining tends to reflect a reasoning process based upon a single argument structure, developed purely to give support to the derived conclusion. Forward chaining is better suited to the exploration of multiple possibilities, some of which may turn out to be ineffective. This seems to match better the reasoning processes that needs to be modelled, though it does place an additional responsibility upon the modeller to specify how the system is to resolve multiple competing claims.

In cases where there is an initial state and a goal state we declare the whole planning process as a goal and introduce rules that try to decompose it into parts, repeating the process until each frame is decided. In cases where there is no particular goal and the animation just has to run and be realistic, there are no special planning rules and domain rules are applied to each frame to generate the next frame. The process can be repeated indefinitely, though there is no guarantee that the set of rules generate no change from a particular frame or they generate a visually recognisable loop.

The inference engine explores every rule, matching it against the data base and generating a single list of all possible actions. When this has completed, it groups the possible actions for each object and passes them to a function which returns the next state of the object. There are many different strategies that could be adopted for this function: we could select one possible action and discard the rest (e.g. select the strongest, or the weakest), or we could find the resultant of all the actions, or we could introduce a threshold and then resolve those that meet it. Different strategies can be adopted for different domains.

The current approach of considering each object in isolation clearly has some limitations and it would be preferable, particularly in complex and crowded domains, for the states of all objects to be resolved together. To make this easier, possible actions can be represented as vectors (Andersen, 1992) and more complex vector resolution algorithms can be introduced. This form of representation could also have benefits for future debugging software (see Section 5).

## 4 Sample animations

A number of animations have been generated using the environment and these have served to clarify the distinction between the functionality of the core software system and the kinds of knowledge that a domain expert might be expected to express.

### 4.1 Example 1: Goal planning in the Jellyfish domain

The first domain we consider contains three types of object: jellyfish, prawns and rocks. Realistic motion in such domains can be achieved by the algorithmic approach (e.g. Miller, 1988). An attempt to model behavioural characteristics of the motion of fish is described by Tu and Terzopoulos (1994). They declare a set of intentional attributes of their fish and adopt an algorithmic approach to testing them and applying the outcome to the fish's motion. This is therefore not an expert system approach, though inner states have been modelled.

In our first example there is one object of each type and we are told the initial and final locations of the objects as well as details of two encounters that are to take place.

```
%      basic facts about each object
fact obj_type(rocky1, rock).
fact obj_type(jelly1, jellyfish).
fact obj_type(prawn1, prawn).
fact property(rocky1, [static]).
fact property(jelly1, [alive]).
fact property(prawn1, [alive]).
%      initial position for objects
fact locn(rocky1, 1, pt(168, 128, 0)).
fact locn(jelly1, 1, pt(-200, -130, -50)).
fact locn(prawn1, 1, pt(172, -170, -150)).
```

We then add facts that state the problem to be solved

```
%      facts about this run
fact plan 1 to 16.
%      final position for objects
fact locn(jelly1, 16, pt(-200, -130, -50)).
fact locn(prawn1, 16, pt(172, -170, -150)).
%      the encounters
jelly1 encounters rocky1 at time(3) at pt(83, -108, 0).
jelly1 encounters prawn1 at time(10) at pt(-183, 27, 0).
```



The set of rules is defined next and contain two types of rule. The first is a rule that is concerned with how to plan a sequence of frames (i.e. by subdividing it).

```
%      if the jellyfish encounters something within a sequence then plan the
%      encounter, and plan the sequences before and after it
rule 2 : if      plan Time1 to Time2
              and  jelly encounters Something at time(TimeN) at P
              and  TimeN between Time1 & Time2
              then plan_state TimeN at P
              and  plan Time1 to TimeN
              and  plan TimeN to Time2
              and  undo(plan Time1 to Time2).
```

The second is a rule that is concerned with how and when to change the location of an object.

```
%      if the jellyfish encounters a static object, then it will move
%      upwards 80 units one frame later
rule 12 : if    jelly encounters Something at time(T1) at SomePlace
              and  property(Something, static)
              and  move(T1, T2)
              then above(SomePlace, 80, NewPlace)
              and  TN is T + 1
              and  plan_state TN at NewPlace
              and  planmove TN to T2
              and  undo(move(T1, T2)).
```

For this example fifteen rules are put in the rule base and the inference engine is invoked. The output is an animation (which can be played back) and a log file. The facts loaded at initialisation were all numbered and new facts are given the next number in sequence. The log file entry contains details of all new facts and deleted facts, indicating the rule that led to the particular action.

Deleting fact: 13 : plan (1 to 16) by rule 2

New fact : 23 move(10 to 16) by rule 7

In the following extract an encounter at time 3 has been used to divide the plan into two parts:

```
** Pass 1
New fact : 14 plan_state (3 at pt(83, -108, 0)) by rule 2
New fact : 15 plan (1 to 3) by rule 2
New fact : 16 plan (3 to 16) by rule 2
Deleting fact: 13 : plan (1 to 16) by rule 2
. . .
```

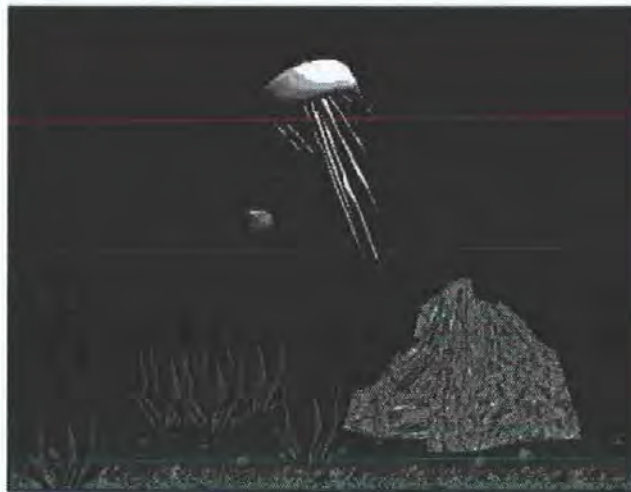
Later, some of the strategies for moving are decided and an intermediate position calculated:

```

** Pass 2
. . .
New fact : 31 propel(1 to 3) by rule 4
New fact : 32 meander(10 to 16) by rule 6
    %% the jellyfish's strategy for moving is decided
. . .
New fact : 37 location(jelly1, 12, pt(-190, -37, -23)) by rule 9
    %% an intermediate position of the jellyfish move is calculated
. . .

```

After thirteen passes the inference engine can make no more changes to the database and creates, as output, a list of all the locations it has determined. Where there are gaps in the location of an object for a particular time slot then an inbetweening algorithm is used to calculate the location. If the appropriate option has been selected by the user, commands are then passed to Swivel-3D to create a rendered animation (see Figure 6).



**Figure 6** Rendered scene from jellyfish animation

The approach adopted in this example achieves the objective of producing credible purposeful animations on the basis of facts and rules entered into the system. Because of a random factor introduced in some rules, each animation is different. In producing the animation some modification of the original set of rules was required, particularly the rules that determine how a jellyfish should 'meander'. This was readily identified as the problem and solving it was not particularly difficult.

#### 4.2 Example 2: non-goal-oriented movement in the Jellyfish domain

In the second example, the system is given an initial state and a set of rules and successive frames are to be produced until the system is told to stop. The same three objects (a jellyfish, a prawn and a rock) participate but a more complex set of attributes is employed. The following facts form the

initial state of the database.

```

fact obj_type( rocky1, rock).
fact object( rocky1, [static]).
fact locn( rocky1, 1, pt( 150, 220)).
fact obj_type( prawn1, prawn).
fact object( prawn1, [alive]).
fact locn( prawn1, 1, pt( 168, 154)).
fact obj_type( jelly1, jellyfish).
fact object( jelly1, [alive]).
fact locn( jelly1, 1, pt( 123, 103)).
fact mass(jellyfish, 5).
fact mass(prawn, 1).
fact size(jellyfish, 20, 20).
fact size(prawn, 3, 3).
fact visibility(jellyfish, 100).
fact visibility(prawn, 50).
fact maxSpeed(jellyfish, 10).
fact maxSpeed(prawn, 8).
fact food_type(prawn, jellyfish).           % prawn is food to a jellyfish
fact predator_type(jellyfish, prawn).       % jellyfish is predator of prawn

```

By defining function names as infix or prefix operators, a more English-like presentation is achieved, as can be seen in the following typical rules.

```

% When a predator is near, the prey plans to escape
rule 3 : if      object1 is nearby predator
        then    plans_getaway.

% When the prey is near, the predator plans to attack
rule 4 : if      object1 is nearby food
        then    plans_encounter.

% Move randomly (if no other move is made)
rule 5 : if      object1 is alive
        then    random_move.

```

An extract from the output log is shown in Figure 7. Having explored all the rules and generated all possible actions, the inference engine then decides on the resulting action to take for each object. In this example, rules are assigned a priority level and lower numbered rules take precedence over higher numbered ones. We see the effect in Frame 7 of Figure 13 where two possible locations for jelly1 are considered (New facts 62 & 64), but resolution results in only the former becoming realised (New fact 67).

```

** Frame = 1
New fact : 16 poss_locn(prawn1, 1, 4, pt(245, 139)) by rule 1
New fact : 17 poss_locn(jelly1, 1, 4, pt(100, 95)) by rule 1
New fact : 18 poss_locn(prawn1, 1, 2, pt(245, 139)) by rule 3
New fact : 19 poss_locn(rocky1, 1, 5, pt(150, 220)) by rule static. . .
Deleting fact: 2 : plan (1 to 7) by rule main inference. . .
New fact : 20 plan (2 to 7) by rule main inference. . .

** Frame = 2
New fact : 22 poss_locn(prawn1, 2, 4, pt(243, 164)) by rule 5
New fact : 23 poss_locn(jelly1, 2, 4, pt(104, 74)) by rule 5
New fact : 24 poss_locn(rocky1, 2, 5, pt(150, 220)) by rule static. . .
Deleting fact: 20 : plan (2 to 7) by rule main inference. . .
Level= 4 New fact : 25 location(jelly1, 2, pt(104, 74)) by resolving . . . .
Level= 4 New fact : 26 location(prawn1, 2, pt(243, 164)) by resolving . . . .
Level= 5 New fact : 27 location(rocky1, 2, pt(150, 220)) by resolving . . . .
New fact : 28 plan (3 to 7) by rule main inference. . .

. . .

** Frame = 7
Distance Between jelly1 and prawn1 is 88. By predicate 4
New fact : 62 poss_locn(jelly1, 7, 3, pt(166, 94)) by rule 4
New fact : 63 poss_locn(prawn1, 7, 4, pt(189, 194)) by rule 5
New fact : 64 poss_locn(jelly1, 7, 4, pt(156, 55)) by rule 5
New fact : 65 poss_locn(rocky1, 7, 5, pt(150, 220)) by rule static. . .
Deleting fact: 60 : plan (7 to 7) by rule main inference. . .
Level= 3 New fact : 66 location(jelly1, 7, pt(166, 94)) by resolving . . . .
Level= 4 New fact : 67 location(prawn1, 7, pt(189, 194)) by resolving . . . .
Level= 5 New fact : 68 location(rocky1, 7, pt(150, 220)) by resolving . . . .

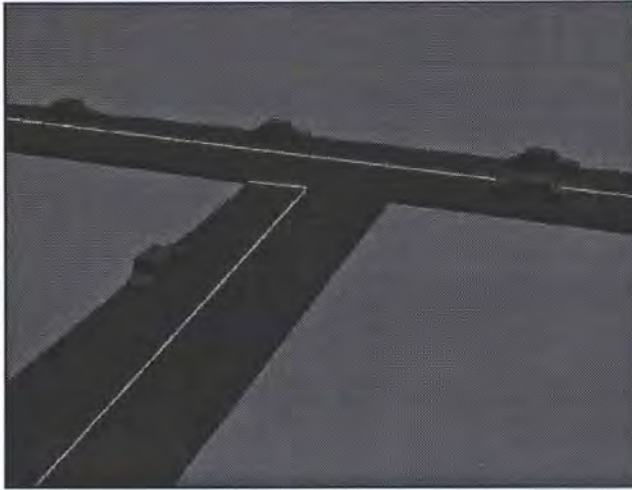
```

**Figure 7** Example 2 - log file

When the required number of frames are planned the system has output a list of all the locations it determined. Though the animations are currently rather short, some purposeful behaviour consistent with the rules can be observed.

#### 4.3 Example 3 - traffic at a road junction

In this domain there is a T-junction and a number of cars. Each car arrives on the scene at a specified location and at a specified time with its destination through the junction predetermined. A view of the animators interface is shown in Figure 2 and a view of the resulting animation shown in Figure 8.



**Figure 8** Rendered frame from the Traffic World

Two approaches have been used in modelling parts of this domain. As a high-level strategy, a number of 'driving-states' have been identified (e.g. 'free', 'turning', 'slowing') and all cars are in one of these states at any time. Rules have been written for the identification of particular traffic situations and which lead to the transition from state to state. Further rules have been written to describe the behaviour of cars within each driving-state. As a general approach this seems to be a good way to handle potential complexity.

At a lower level, a set of general rules have been written to determine how each driver behaves. In general cars aim to go as fast as they can up to their maximum speed, but have to slow down if their way ahead is blocked or they wish to turn. If a vehicle should stop, then other vehicles behind it will slow down and stop; if a vehicle wishes to turn it will stop and wait until it estimates that the way ahead is clear.

Modelling of this domain has highlighted two important issues. In previous examples points or regions in the spatial field were not significant, but in this one different regions are important. The background represents the road junction and it is relatively complex. It can be divided into road and non-road regions and within the road regions there are important distinctions to be made (for example, each side of the road, the region before the stop-line for traffic approaching the major road, etc.). The modelling and representation of the domain background becomes an important issue.

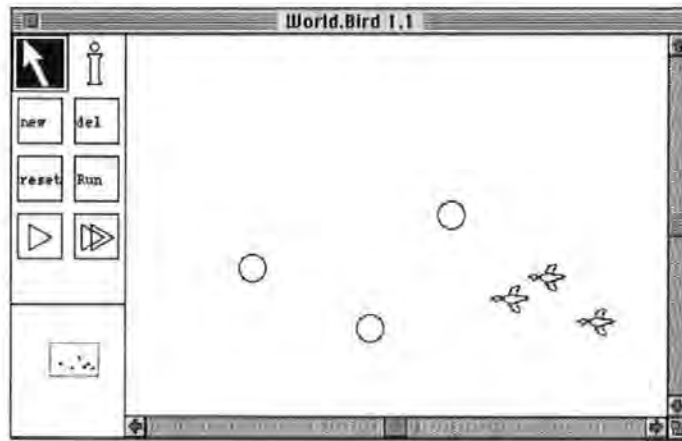
We also include in our model the ability for drivers to plan on the basis of their models of other drivers. When considering a move, each driver must calculate the current velocity and acceleration of every another vehicle to decide whether or not it is likely to interfere with their progress. In normal cases, if there is any possibility of collision then no move is attempted. However, if we add attributes representing the degree of patience of the driver and reduce this as their waiting time increases then the result will be that the driver is now prepared to make a move even if collision avoidance depends upon other vehicles slowing down. This interaction of agents is a novel feature



for the model but it seems to introduce no particular difficulties. In a more complex situation, the strategy of moving one object at a time might not suffice and the type of solution proposed in the next example may be preferable here too.

#### 4.4 Example 4 - a flock of birds

In this domain we model a flock of birds in flight that have to navigate around a number of fixed obstacles (Figure 9). Reynolds has developed a sophisticated animation of birds in flight (Reynolds, 1987). The simulated flock is an elaboration of a particle system in which each particle acts according to an identical algorithm so that each bird is considered as behaving according to the same rules as every other bird. This approach assumes that the behaviour of the flock is simply the result of the interaction between the behaviour of the individual birds.



**Figure 9** Graphical Interface showing three birds and a number of obstacles

We follow this principle in our approach but instead of the procedural approach to specify each bird's behaviour we express the bird's behaviour in rules. By writing a few rules concerning how birds position themselves with respect to the other birds around them, we are able to generate possible moves for each bird. In this case the sophistication really lies in the method of resolution for it seems infeasible to allow each bird to determine its own position. The major reason for this is that the overriding principle of flocking motion is that there are never any collisions. If we adopt an approach whereby the position of each bird is determined in turn and we identify an unavoidable collision when placing one of the later birds, we would have to backtrack to place a previous bird in another place. At present, finding a new place depends upon a new random number being generated. We do not really want a random procedure here for it may take a very long time before it comes up with an acceptable solution. If a crowded situation is likely to occur it is preferable to calculate an optimal solution rather than hoping to hit upon a set of acceptable placements by chance. We therefore intend to adopt a resolution algorithm for this domain that will consider all birds together and find a solution with least penalties.

## 5 Conclusions

The preceding examples illustrate some potential for rule-based enhancements to animation software. They provide the opportunity to use a new kind of knowledge to simulate the behaviour of objects within a modelled environment and these initial demonstrations have helped clarify the type of architecture needed, the functionality that might be expected and the types of user who may be able to successfully exploit such a system. They also provide us with some useful insights into the limitations of the approach, the types of animation problem where it might be useful and the significant work that needs to be done.

The existing system is a prototype and cannot produce quality output. To do this the approach needs to be integrated into a more sophisticated animation production system. We have described the architecture that would be needed for this to be done. To make a readily usable system it is necessary to provide a good set of predefined predicates and operators in terms of which domain-specific behavioural predicates may be defined. We have gone some way in identifying a useful set but it is by no means complete.

The smallness of the rule bases in the examples raises the question of whether the approach will scale up. The question here is not primarily a technical one, for there is little problem (except execution time) in presenting the inference engine with 250 rules instead of 15. The question is more whether domain experts and animators can comprehend what the system is doing if it contains so many rules. In this respect the interesting research question, we believe, is how animators might use this new facility creatively and this will depend crucially on their ability to understand the effect of their actions. The text-based log file may give some insight into the workings of the inference engine but it is not always possible to quickly extract from it a clear understanding of why an object is behaving in a peculiar manner.

The major problem is not the ability of the inference engine to cope with things on a larger scale but the ability of users to visualise the decisions the system is making. To this end there is a need to develop specific techniques for visualising the actions of the inference engine itself. In the future we would like to be able to switch on an option that makes visible the vectors acting on an object at each stage in the inferencing and relates them to the rules from which they originated. We even go so far as to suggest a visual representation for rules that might be called up and assigned to objects, possibly with strengths that may be set by the animator.

Rule-based systems seem more appropriate for some types of problem than others. As Rijpkema and Girard (1991) discovered, an expert system was useful for determining a grasping strategy, but a procedural approach was best suited to the finer movements involved in making contact. A rule-based approach is not likely to serve well in areas that require much mathematical calculation, for example, but an algorithmic approach is also unlikely to serve well where strategic

planning is required. The solution is therefore the flexibility of hybrid systems, incorporating both algorithmic and rule-based approaches.

There seem to be several situations where such an approach could give real benefit. One is the creation of a realistic animated background scenery, the detail of which is not particularly significant. For example, an animation may need a background of a realistic street scene that is not predictable, but neither is it particularly remarkable. A suitably set up domain could generate endless such scenes which would be very tedious or difficult to produce by any other method.

The other area of beneficial use is where animators can use expert knowledge about the behaviour of humans (or other animals) in order to plan foreground animations. Thalmann is already incorporating some modelling of purposeful human behaviour into the animation of photo-realistic human models (Thalmann, 1994). The system currently uses pre-written scenes derived from the work of Schank (1980). Schank's scripts are effective for stereotypical scenes but are limited in modelling more open scenarios where there may be a need for planning or, more generally, for a form of situated action.

It is not possible, at this point in time, to determine the best representations and strategies for all domains, or even for each particular domain. The precise nature of the approach will develop through experimentation of the kind we have described in this paper. What we do argue is that animating sophisticated beings with minds of their own requires a modelling environment that can represent internal mental states and internal mental processes.



## References

- Andersen P.B. (1992) Vector spaces as the basic components of interactive systems: towards a computer semiotics. *Hypermedia*, 4(1), 53-76.
- Canfield Smith, D., Cypher A. & Spohrer J. (1994) KidSim: Programming Agents Without a Programming Language. *Communications of The ACM* 37(7), 55-67.
- Clocksin, W.F. & Mellish, C.S. (1984) *Programming in Prolog*, 2nd Edn. Springer-Verlag, Berlin.
- Miller G. (1988) The Motion Dynamic of Snakes & Worms. *Computer Graphics* 22(4), 169-178.
- Reynolds C.W. (1987) Flocks, Herds, and Schools: A Distributed Behavioural Model. *Computer Graphics* 21(4), 25-34.
- Reynolds C.W. (1982) Computer Animation with Scripts and Actors. *ACM SIGGRAPH '82 Proceedings* 16 (3), 289-296.
- Rijkema H. & Girard M. (1991) Computer Animation of Knowledge-Based Human Grasping. *Computer Graphics* 25(4), 339-348.
- Schank, R.C. (1980) Language and Memory. *Cognitive Science*, 4 (3), 243-284.
- Skinner, B.F. (1974) About Behaviorism. Alfred Knopf, New York.
- Thalmann, D. (1994) Animating Autonomous Virtual Humans in Virtual Reality. In: Duncan, K. & Kreuger, K. (Editors) *Proceedings 13th World Computer Congress 94, Volume 3*. Elsevier Science B.V. (North-Holland), Amsterdam, 177-184.
- Tu X. & Terzopoulos D. (1994) Artificial Fishes: Physics, Locomotion, Perception, Behaviour. *Computer Graphics* 28, 43-50.
- Waters K. (1987) A Muscle Model for Animating Three-Dimensional facial Expression. *Computer Graphics* 21(4), 17-24.
- Wejchert J. & Haumann D. (1991) Animation Aerodynamics. *Computer Graphics* 25(4), 19-22.