Technical Report

UCAM-CL-TR-152
ISSN 1476-2986

Number 152

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Motion development for computer animation

Andrew Mark Pullen

November 1988

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted August 1987 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Churchill College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Contents

1 Introduction 1

2 Traditional Animation—can it be automated? 3
- 2.1 Historical Background . . . 3
- 2.2 Traditional Animation . . . 4
  - 2.2.1 The keyframe animation process . . . 5
- 2.3 Computerising the Keyframe Animation Process . . . 10

3 KAS—a Computer-Assisted Keyframe Animation System 19
- 3.1 KAS—a Keyframe Animation System . . . 19
  - 3.1.1 Keyframe editing . . . 21
  - 3.1.2 In-betweening . . . 25
  - 3.1.3 Superimposition . . . 31
  - 3.1.4 Displaying animated sequences . . . 32
  - 3.1.5 Producing hard-copy output . . . 33
- 3.2 Lessons learnt during the making of the film "Taking a Line for a Walk" . . . 34
- 3.3 "Painting" Animated Sequences . . . 36

4 Computer Animation (review of allied work) 41
- 4.1 The Early Days at Bell Labs . . . 42
- 4.2 Picture-driven Animation . . . 43
- 4.3 Early Attempts at Modelled Animation . . . 45
- 4.4 Keyframe Animation . . . 46
  - 4.4.1 The work of Burtnyk and Wein . . . 46
  - 4.4.2 ANTICS . . . 49
  - 4.4.3 CAAS 2 . . . 49
  - 4.4.4 Keyframe animation using moving point constraints . . . 51
- 4.5 Animation at Ohio State University . . . 53
  - 4.5.1 GRASS—a three-dimensional real time system . . . 53
  - 4.5.2 ANIMA, ANIMA II and ANTTS . . . 54
  - 4.5.3 The Skeleton Animation System . . . 55
- 4.6 Analogue Systems . . . 56

4.7 Three-dimensional Keyframe Animation 56
4.8 Actor-based Animation Languages 57
4.8.1 DIRECTOR 57
4.8.2 ASAS: The Actor/Scriptor Animation System 59
4.8.3 Work with actors & scripts at the Université de Montréal 59
4.9 Commercial Computer Animation Production Houses 62
4.9.1 Work at Lucasfilm 62
4.10 What Next? 63

5 Using Simulation in the Animation Process 65
5.1 Specifying Three-dimensional Motion 65
5.1.1 Three-dimensional keyframing 65
5.1.2 The Actor and Script approach 66
5.2 Using Simulation to Define Motion 66
5.2.1 A street-scene example 67
5.2.2 The production of a new animated sequence 67
5.3 Potential Advantages and Disadvantages of the use of Simulation in the Animation Process 70

6 Snooker—a simple application of simulation to the animation process 73
6.1 An Introduction to the Game of Snooker 74
6.1.1 The rules of the game 74
6.1.2 Modelling the physics of the game of snooker 76
6.2 The Motion Development Program 82
6.2.1 The user interface 84
6.2.2 A typical animation session 87
6.3 Setting up "Camera Angles" 94
6.3.1 The "camera control language" 94
6.3.2 Scene visualisation 99
6.4 Rendering Sequences Ready For Filming 102
6.4.1 The implementation 105
6.5 Evaluation of the Snooker Experiment 108

7 Cellular Automata—a further use of simulation in the animation process 111
7.1 What is a Cellular Automaton? 111
7.2 The Game of Life and its Generalised Descendants 112
7.3 An Overview of the Software Developed for this Experiment . . 114
7.3.1 Getting started 115
7.3.2 The options available from the main menu 119

7.4 The Method Used To Specify The Rules 125
7.4.1 A "maze runner" implemented as a cellular automaton . 127
7.4.2 A Turing machine to recognise prime numbers 130
7.4.3 Illustration of the operation of electronic circuits 131
7.5 Implementation details 132
7.6 Can Cellular Automata be used to Produce Larger Scale Animation? 137
7.7 Conclusion 137

8 Review **139**
8.1 And what next? 142

References **145**

Appendix I
Glossary

List of Figures

2.1 An exposure sheet 9
2.2 A field chart 9
2.3 Problems with interpolation due to the appearance of hidden objects 14
2.4 The inadvisable use of linear interpolation to produce images between keyframes depicting a rectangle at different rotations . . 15

3.1 KAS screen layout 20
3.2 KAS in edit mode 24
3.3 Processing of line-chains with different numbers of points prior to interpolation 26
3.4 Fitting a fourth degree polynomial through five points 28
3.5 Generating an Overhauser curve 28
3.6 Generating an animation path for pairs of corresponding points 30
3.7 The four working areas used during image painting 39
3.8 A simple example of painting an outline image 40

4.1 The use of p-curves to define motion 44
4.2 The skeleton animation technique for computer keyframing . . 48
4.3 Keyframing using moving points 52
4.4 The use of Coons patches in the moving point technique 52
4.5 A DIRECTOR script with comments 58

6.1 The situation ready for the start of a game of snooker 75
6.2 Graph of ball speed against time in both the theoretical and observed cases 77
6.3 The collision of a ball with a cushion 78
6.4 The effect of back spin during collision between balls 81
6.5 Screen layout as snooker program is running 85
6.6 Setting the velocity of a ball 89
6.7 Images produced during camera definition 101
6.8 Demonstration of anti-aliasing 104
6.9 Shaded image of a snooker table 108

7.1 Examples of patterns from the Game of Life 113
7.2 The evolution of a "glider" 114
7.3 Screen layout during editing 117
7.4 "Gosper glider guns" from the Game of Life 122
7.5 Cellular automaton screen images 124
7.6 Simulation rules for the Game of Life 126
7.7 The 'movement' of a maze runner over four generations 128
7.8 Simulation rules for a "maze runner" 129
7.9 A Turing machine to recognise prime numbers 131
7.10 The data structure used to represent the distribution of living cells in a particular generation 133
7.11 The state of the workspace prior to processing a row in the current generation 135
7.12 The state of the workspace after processing a row 136

Chapter 1

Introduction

One of the fastest growing areas of computer-related research in recent years has been in the field of computer graphics, including enormous advances in the field of computer animation. At least part of this research and development activity has been fuelled by the ever increasing demand for computer-generated images for use in advertising, television and feature films. There is always a demand for something new to give a product a hi-tech image, a television station an up to date identity, or a film those extra special special-effects to give it the edge at the box office.

This increased use of computer-generated animation has not however been met with universal acclaim; many artists are horrified at what they see and assume that the results they have seen to date (and which are greeted by computer scientists with such enthusiasm) are the limits of what can be achieved by use of a computer. If this were the case they would have reason to express their horror since the computer-animated sequences produced so far, whilst undoubtedly improving all of the time, are still of a generally poor standard when compared with conventionally produced sequences—both technically and, particularly, artistically. There is a tendency in the artistic community to assume that since only low quality results are being produced, that is all that is possible using a computer, whereas I believe that the quality of the results is more a reflection on the fact that current computer-animated sequences are usually generated by scientists—rather than artists who have been trained in animation techniques.

This dissertation examines some of the problems of producing animated sequences using a computer. It then describes experiments aimed at giving the artist more direct control over the animation process—in an interactive graphical environment. The main reason for wanting to give the animator control over the production process is that it is the artist who has been trained to make aesthetic decisions regarding what is, after all, an artistic process. (I am at this point referring mainly to the use of animation for entertainment purposes as opposed to sequences designed to illustrate some technical or scientific principle.) In the final analysis it is the quality of the work and its effectiveness in portraying what the artist wished to portray that is the best judge of the quality of an animated sequence. Even if it were possible to produce technically superior sequences by the use of a computer programming approach rather than an artist-oriented system, it may be a fruitless exercise since artists are not usu-

ally computer programming experts and computer scientists are not generally capable of producing artistically pleasing animated sequences. It was this conflict between the artistic and scientific styles of working which provided part of the motivation for the development of the experimental systems described later in this dissertation, systems aimed at providing a balance between the two approaches—using an interactive graphical environment over which the artist has control, but to which extra facilities can easily be added by a more technically minded assistant.

This thesis is divided into a number of logical sections:-

- Chapter 2 introduces keyframe animation (the process most often used for traditional hand-crafted animation) and examines to what extent it is well suited to computer assistance.

- Chapter 3 describes my own implementation of such a system on a computer and assesses its effectiveness as an animation tool—based on experience gained during its use by a professional animator for the making of a short film for Channel 4 television.

- Chapter 4 presents some of the landmark systems in computer animation history and explores how these systems have attempted to solve the various problems to which they have addressed themselves.

- Chapter 5 introduces the idea of using computer simulation to guide the process of motion development for animated sequences—the aim being to provide the animator with powerful artist-oriented tools giving high-level control over an animated object's motion and interactions, thus avoiding the need for artists to be computer-experts.

- Chapter 6 describes an initial experiment in the realm of simulation/animation, namely the animation of snooker balls on a billiard table. Tools are provided to produce a realistic depiction of a game of snooker—or for that matter a totally un-realistic one. The software is aware of much of the physics of the game of snooker (together with the rules of play) in order to facilitate the speedy production of a true-to-life sequence. There are also many ways of 'bending' the built-in rules to achieve any required outcome.

- Chapter 7 develops the simulation/animation theme further in considering the use of cellular automata in an animation environment. Using the *Game of Life* as a starting point, an extensible system is described—in which the user can specify rules to guide the action of an automaton for interesting mathematical effects or simple illustrative animation.

- Finally, chapter 8 attempts to draw the threads together and reach some conclusions on the effectiveness of the simulation/animation approach and its relationship to other methods of motion development.

Chapter 2

Traditional Animation—can it be automated?

Before considering computer animation, it is necessary to understand a little of what is involved in the traditional animation process, in order to begin to appreciate the possible advantages the use of computers could bring. Another reason for knowing something about conventional animation is that a possible first step towards developing a computer animation system may be to adapt traditional techniques for computer use.

This chapter will present a few conventional animation techniques and consider the possibility of automating them using a computer. The main emphasis will be on the *keyframe cel animation* technique, since this is the most commonly used method for hand animation; it is also the most obvious candidate for computer assistance.

2.1 Historical Background

In 1831 the Frenchman Joseph Antoine Plateau invented the phenakistoscope, a device consisting of a spinning disc which held a series of drawings, and windows which caused these drawings to be presented to the viewer in quick succession. In this way the illusion of movement was created. Horner, an Englishman, extended the idea in 1834 when he invented the zoetrope. Here the drawings were stuck to the inner wall of a rotating drum and viewed through slits in the side of the drum. A further refinement, by the Frenchman Emile Reynaud, was to replace the slits by mirrors which rotated in the centre of the drum. He called this device a praxinoscope.

It was not until the opening of the first cinema, in 1892, that it became possible for large numbers of people to view longer sequences of moving pictures. For the first fourteen years all of the films presented consisted only of live action; it was not until 1906 that the American J. Steward Blackton created the first animated film, called *Humorous Phases of a Funny Face.*

Many people would consider the first *cartoon* to be *Gertie the Trained Dinosaur*, by another American, Winsor McCay. Although the sequence was very short it needed 10,000 pencil sketches to be produced.

For animated films up until 1915 all drawings were normally produced on paper; but, in that year, Earl Hurd introduced the idea of working on transparent celluloid sheets which could be photographed in front of painted backgrounds. By using several layers of celluloid, different parts of the foreground action could be animated independently. This technique quickly became known as *cel animation.*

It was Walt Disney who became the first serious commercial user of cel animation. He began in 1928 with Mickey Mouse, then went on to Donald Duck and the Silly Symphonies. In 1938 he produced the first full length animated film, *Snow White and the Seven Dwarfs.* This was a very expensive and risky venture due to the enormous detail and complexity of the project. It almost caused financial ruin for the Disney company during its production but on completion was very well received by the public and was a commercial success.

Over the years many people in different countries have pioneered new animation techniques and styles; notably Ivanov in Russia, Trnka in Czechoslovakia, Halas in Britain and McLaren in Canada. The majority of animators used cel animation techniques, but other techniques were also developed. Clay animation in particular has become quite popular in certain circles. With this technique the artist fractionally moves or deforms objects made of clay, between successive shots taken with a stop-frame camera—a good example is *Morph* on the BBC television programme *Take Hart.* Indeed a wide range of different effects can be achieved by using a stop-frame camera to photograph a whole range of different two and three dimensional objects. Films produced by building moving patterns of sand or pasta pieces, for example, are becoming quite common in children's television programmes such as *Sesame Street.*

An enormous variety of more unusual techniques have been developed too. These include such things as the *pin screen* technique in which an array of shiny metal pins is arranged in a grid; pushing up some of the pins from the back of the grid can cause them to catch the light differently to their neighbours thus creating patterns which can be subtly altered between successive frames.

2.2 Traditional Animation

Conventional hand animation is a frame-by-frame technique. That is to say the illusion of movement, or change, is created by photographing a series of individual static images onto successive frames of film or video tape. The illusion is produced by showing these frames in rapid succession (typically 24 frames per second (f.p.s.) for film and 25 f.p.s. for video) so that the viewer is unable to distinguish the individual images but perceives, instead, a smoothly changing scene.

As mentioned above, there are many different methods of producing hand animation. For two-dimensional work these methods include keyframing (the technique used for cel animation), pin-screen and model animation. For three-dimensional work they include clay animation and other forms of model animation. Some of these techniques may be suitable for computer assistance but it is likely that the techniques which are most successful for hand animation will

not be the same ones that are ideally suited to use with a computer. There may be completely new techniques—either not possible or not used for hand animation—which are the most useful contribution for computers to make to the animation process.

2.2.1 The keyframe animation process

Perhaps the most commonly seen examples of animation are the animated *cartoons* produced by such studios as Walt Disney and Hanna Barbera. These range from simple five minute programme-fillers to very sophisticated full-length feature films.

Walt Disney studios are the best known producers of full-length animated films. Most people would agree that such films as *Snow White, Bambi* and *Fantasia* represent the finest and most elaborate animation the world has ever seen. These films were produced using the most common of all conventional animation techniques, the *keyframe* animation method which takes its name from the fact that the most significant events in the animated sequence (the *key frames*) are drawn first (by the senior animators), leaving lesser animators the less demanding job of producing the in-between drawings.

The keyframe method has also been adopted by large numbers of small animation houses, the main difference being that in a major animation company a large number of artists may be involved at each separate stage in the process, whereas in a small company a small number of artists will undertake the entire project, performing several tasks each.

The precise details of the stages involved in producing a cartoon-style film will vary slightly from company to company, but typically the steps will be as given below. The animation of the foreground characters is roughly a sequential pipeline, with painting of the backgrounds going on in parallel. It is necessary to break the process down into these standard steps to facilitate the over-all organisation of the project—without a standard method, the organisation of the tens of thousands of drawings necessary for even a short animated film would quickly become unmanageable. The steps are:-

1. **The story**

   As in an ordinary film, the animated film generally tells a story. The formal description of this story involves three steps, with each step a more detailed refinement of the previous one.

   (a) **The synopsis** is produced first. It is a very short summary of the story (one page maximum).

   (b) **The scenario** is a detailed text which describes the complete story (without any cinematographic references).

   (c) **The storyboard** provides the details of what the film should look like. It consists of a number of illustrations, laid out in comic-strip fashion with appropriate captions to indicate the action that is occurring and the length of time this action should take in the final film. The number of illustrations used will vary depending on

the amount of action in the film, but typically there will be at least one drawing per *shot* (a *shot* may be considered to be a continuous piece of filming from one "camera position"). Several shots may be considered together to constitute a *scene* (in which the location and main characters remains constant). Several scenes may constitute a *sequence*, in which a particular idea is conveyed.

There must always be enough illustrations to describe the film's key moments.

2. **Animatic**

   If the idea for the animation has to be demonstrated to a third party (such as the company which has commissioned an animated television commercial), it may be necessary to make a very crude animated film (an *animatic*) to demonstrate the format of the final film. This may well be produced by photographing the storyboard onto movie film, in such a way that each storyboard illustration is shown for the same length of time as that shot will take up in the final film. The addition of a suitable sound-track, or voice-over, can help to give an impression of the style and timing of the finished article.

3. **Layout**

   This step involves deciding on the final designs for the characters and "sets" (the environment within which the action takes place).

4. **Sound-track recording and reading**

   The sound-track must be produced before the animated images, since the motion (particularly mouth movements during speech) must match the sound-track. Once the sound has been recorded it is possible to measure the time between significant events on the sound-track in order to determine how many frames of animation are needed to match a particular sound. This can be a very tedious and time consuming process, often performed by recording the sound-track onto special magnetic tape with holes at equal intervals arranged in such a way that one hole passes the tape recorder's playback head for each frame of film so that the time between significant sound events can be determined by counting holes.

5. **Animate extremes**

   Important (*key*) frames for each shot are drawn by the main animator. The number of frames which the main animator will draw for each second of film time will depend on several factors, including the smoothness and amount of movement in the scene—more complex motion will necessitate more keyframes in order to define the motion accurately.

6. **Draw some in-betweens**

   The keyframes produced in the previous stage are passed to an assistant animator who draws some of the frames between these extreme positions.

7. **In-betweening**

   A conventional film is projected at 24 frames per second but relatively few of these frames will be drawn by the senior animators. It is the *in-betweener*'s job to draw all of the frames which the other animators have not produced. This obviously requires a good drawing ability, but is much more automatic than the assistant animator's contribution and is considered to require less artistry.

   Depending on the amount of motion in a scene, and the quality required of the final result, it is sometimes possible to produce only 12 different pieces of artwork per second and project each frame twice—a technique known as half framing; this obviously saves time (and hence money) which makes it popular with some animators for certain applications.

8. **Line test**

   All of the sketches produced so far have probably been drawn on paper using a pencil. For the final photography the images must be transferred to acetate "*cels*" and painted. However, before transferring to cels it is possible to photograph the pencil sketches onto movie film, or video tape, so that the dynamics of the motion can be checked. If there are any problems with the smoothness or speed of movement, it is obviously better to find out at this stage so that the offending frames can be re-drawn and re-tested. This test is referred to as a *Line Test*, or sometimes a *Pencil Test*.

9. **Xeroxing and inking**

   The original pencil sketches of each frame must be traced or photocopied onto acetate "cels". Lines must be inked in by hand.

10. **Painting**

    Most objects to be animated contain areas of solid colour, or should at least be opaque to the background. Each individual cel must be painted by hand. The painting is performed on the back of the cel (the opposite side to the outlines) in order to avoid obliterating the inked outlines. This work requires a great deal of patience and accuracy.

11. **Checking**

    Before shooting, the animators in charge of each scene must check that everything is complete and correct.

12. **Final Photography**

    For filming, the cels are placed in front of painted backgrounds (which have been produced by a background artist) and photographed, one frame at a time, onto movie film. If only part of an object in a scene is moving, it may not have been necessary to redraw the whole image for each

frame—the part which is moving could be painted on a separate layer of acetate from the static part and only that need be altered for each frame. This technique may obviously be extended for several independent moving parts, but there is a practical limit to the number of layers of acetate which may be used before they begin to affect the quality of the background image. If multiple layers are used the colours used for painting the image must be subtly altered to compensate for the position of the cel in the stack.

13. **Editing and post-production**

    Once the film has been developed, it can be spliced together and any necessary optical effects (such as fades and wipes) can be added.

As already mentioned, the organisational aspects of even a simple animated film can be enormous. Four main kinds of information sheets are used to control the flow of artwork from the initial storyboard through to finished film:-

- **Bar sheets** carry a visual synopsis of the animation sequence. They indicate the number of frames allotted per action, and other timing information.
- **Route sheets** are mainly of use to a director. For each scene, they list the artist in charge, the length of the scene, and the current status of the work.
- **Model sheets** contain sketches of the characters to be animated in a number of representative poses—thus allowing several people to work on the animation of a single character, with consistent results.
- **Exposure sheets** are very detailed documents (see figure 2.1). They contain a line for every frame in the film. They are built up by the animators and ultimately used by the camera operator to ensure that the correct cels are overlaid for each frame (in the correct order) and to specify any necessary camera movements. This sheet will also deal with sound synchronisation if necessary.

Another useful tool, used throughout the animation process, is a transparent sheet of plastic, 12" x 8.75", known as a *field chart* (see figure 2.2). This contains a grid which may be laid over original artwork and used to define all sizes and positions—for example, the camera operator may be asked to zoom in on the four field centred on position (-4,7).

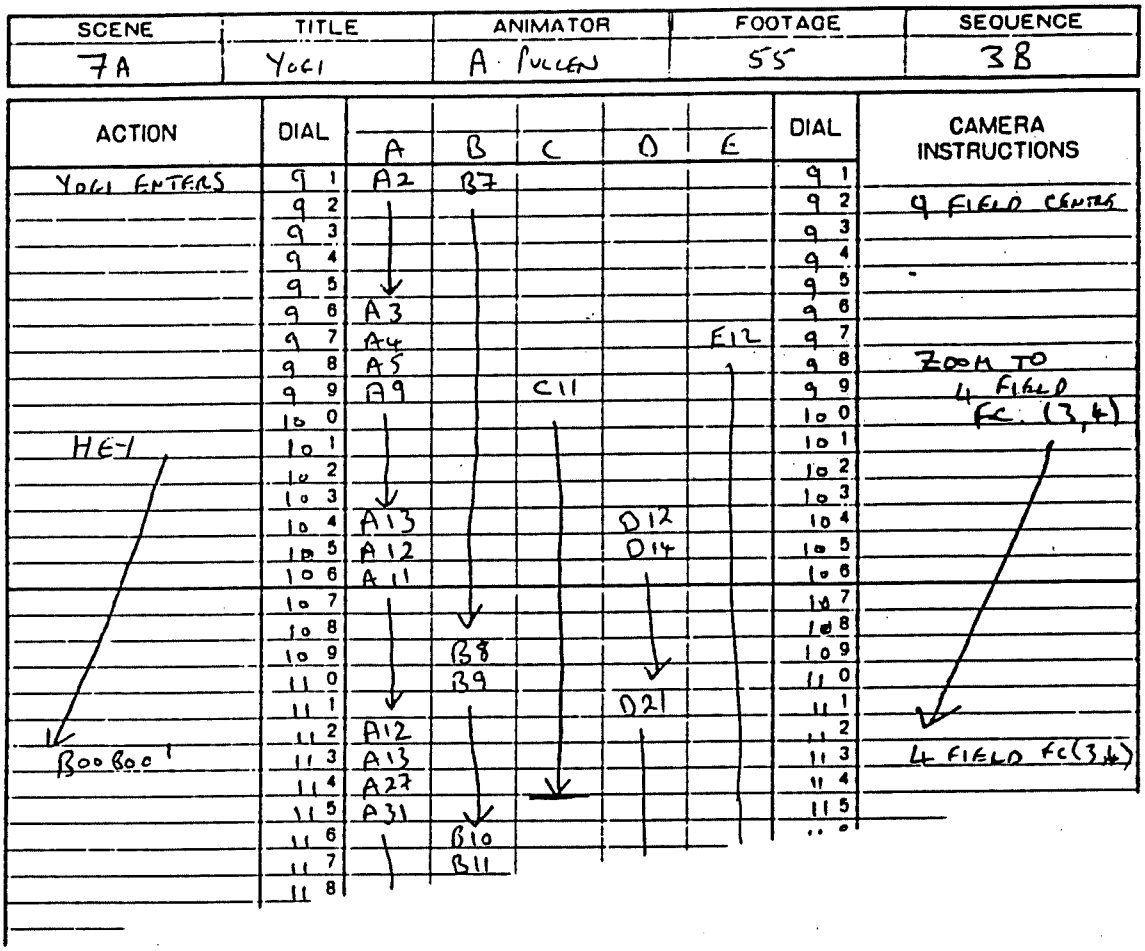


Figure 2.1: An exposure sheet.

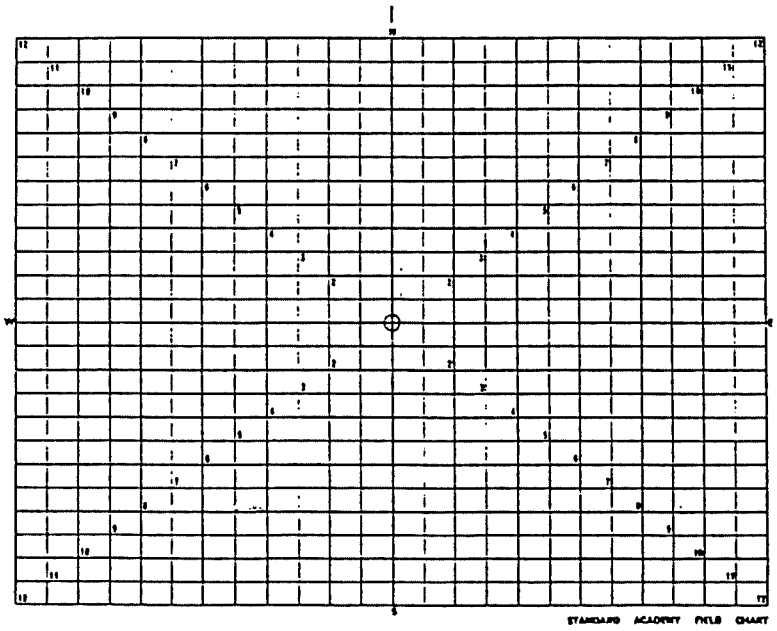


Figure 2.2: A field chart (not real size).

2.3 Computerising the Keyframe Animation Process

As can be seen from the above description, cartoon animation is a time consuming and tedious process. It necessitates a certain amount of artistic involvement (such as the design of a storyboard, and production of original drawings), but also a large amount of relatively automatic and straightforward work (such as painting the acetate cels) which is, none-the-less, very time consuming. It would appear that gains could be made by using computers to speed up particularly these more automatic processes. The question to be answered is "for which processes should the computer be used?". Let us consider each of the steps in the traditional process in turn and assess how a computer could be of benefit to them.

The story

Automatic story writing or storyboard layout is not usually considered relevant to the production of animation—the animator will normally want to have total control over the story to be told and how it should be presented. In any case, automatic story writing and understanding is a complex artificial intelligence problem which is not yet sufficiently well understood to be a useful tool.

Animatic

It may be possible to produce these very simple animations using a computer. An advantage is that the computer monitor could be used as the display medium rather than having to use film or videotape. This should dramatically reduce the time needed to produce and view the animatic.

One possible production method would be to input simple keyframes directly at a graphics tablet (see below) and use simple interpolation techniques to produce a moving image on a computer display. Hardware limitations may mean that only very crude animation is possible if the necessary display rate is to be achieved.

Layout

This stage involves producing static drawings to be referred to by the artists producing the film. As such it is probably best done using pencil and paper.

Sound-track

The nature of the sound-track, and its method of production, will vary from film to film. Few generalisations can be made about its production. The computer may, however, be able to make a useful contribution in analysing the sound-track to determine the timing of significant sound events. These events could be noted on an exposure sheet (either automatically or manually), so as to allow the animators to arrange the necessary synchronisation between picture and sound. Alternatively, some method could be devised to automate part of the picture-generation phase—based on information obtained from the sound-track. A major use for this would be for lip-synchronised speech. The

computer could analyse the dialogue to determine which phoneme was current for each frame; the correct mouth position could then either be inserted into the frame automatically or, more likely, the computer could indicate the correct mouth position for each frame and allow the animator to edit it in. As far as I am aware this technique is not currently used in any existing animation system but I believe it has the potential to be a useful tool.

Animate extremes

This is a creative process which involves producing drawings. It is therefore desirable that the artist should be able to use the materials of his choice since this will make his job easier. Traditionally, animators have used pencil and paper; they tend to be reluctant to change to a new medium (such as sketching with a stylus on a graphics tablet). However, it is possible that as time progresses it will be less necessary to cater for the traditional methods of working quite so much, since younger artists may learn to use the new techniques at the beginning of their career and not feel that they are being forced to adapt to an alien medium.

If the computer is going to be used for in-betweening, it is necessary to store the key frames in the computer's memory before any interpolation can be performed. This can be achieved by any one of several methods:-

(a) Perhaps the most obvious method is to produce the original drawings by sketching on a graphics tablet using a stylus.

It is a relatively simple matter to write a program which senses the position of the stylus on the tablet and draws lines in corresponding positions on a computer graphics monitor. Artists often initially find this style of working awkward since they are used to looking at the paper when drawing, whereas, with a tablet, they must look at the screen. However, this is not a significant problem since the artist is not in fact looking at his hand as he draws on paper, he is looking at the image left behind by the pencil—the only difference is that now this image appears on a computer monitor. As long as there is an intuitive and immediate correspondence between hand movements and the image on the screen, and the artist can tell where his logical pencil is in relation to the rest of the image (probably by use of a cursor on the screen) he will soon accept the new method of working as being perfectly natural. Indeed there are positive advantages with the computer method such as the fact that the artist can see the entire image rather than having part of it obscured by his hand.

Even after the artist becomes used to looking at the screen instead of his hand there are still problems with using a tablet as a high-tech sketch pad. One problem is that most automatic in-betweening methods insist on being told how successive keyframes correspond. For practical purposes this means that each of the keyframes should be composed of the same number of lines, and these lines should be drawn in the same order. When sketching two or more keyframes freehand on a tablet it is almost impossible to remember exactly how the objects you are drawing should be decomposed into lines; and, especially, the order in which these lines should be drawn.

This problem may be partially overcome by editing an existing keyframe to produce the next keyframe in the sequence. The editing process could take several forms: one method would be to move each individual vertex on a line-chain to its new position for the next keyframe, this would ensure that corresponding lines contained the same number of points but would be a very tedious and unnatural way of working. A slightly better alternative would be to also allow collections of points to be operated on simultaneously (say, translate or rotate an entire line-chain or object), however this reverts to being a point-at-a-time operation if corresponding line-chains are not related in a simple mathematical way—in that case it would be useful to be able to delete the current line and redraw it freehand. Even when given all of these facilities the artist may well feel that the editing process is hindering his creative talent by constraining him to work in too structured a way and presenting a confusing image while the keyframe is partially edited.

What is needed is a method by which the artist is able to draw freehand but is prompted as to what needs to be drawn. In many cases it would be helpful if the animator could see the un-edited keyframe in the background as he worked (although it would also be a good idea if this could be turned off to allow him to view his new keyframe uncluttered by this background detail). Building on this idea, a better method might be one in which the computer merely displayed the preceding keyframe and the artist drew the next keyframe on top of it—just before each line-chain was drawn the corresponding line-chain in the other keyframe could be highlighted to ensure that the artist is aware of the correct drawing order. This would be a more natural way of editing when there were large numbers of complex changes between keyframes. As far as I am aware this method is not used in any existing computer-assisted animation system.

A problem with using a graphics tablet is that many artists find it much less easy to use than paper and pencil—this is partially a matter of adapting to a new medium but there are, in fact, real advantages in using a pencil (such as the ease of producing smooth curves and lines of differing thickness and "*weight*"). The surface of a tablet is typically much more slippery than paper and the stylus must be held more upright than a pencil. The wire from the top of the stylus may also get in the way. If artists are to be persuaded to accept the tablet as a drawing medium it is worth experimenting with styli which do not require wires (transmitting information by use of infra red radiation, for example), and tablets with different surface textures.

(b) A slightly different style of working is for the animator to produce the original artwork on paper, then have someone trace it using a graphics tablet. After the artist had produced the original drawing, someone would number the line-chains so that when the drawing was fixed to the surface of the tablet (making sure that all drawings were held in exactly the same position), the operator need only draw over the lines in the order specified by the numbers to arrange that the lines in the different keyframes correspond with each other correctly. This method allows the artist to sketch using the medium he prefers but necessitates two extra menial tasks (viz. line numbering and tracing) when compared with sketching directly onto

the tablet. Laborious as it may seem, this technique has found favour during the making of a number of computer animated cartoons.

(c) Rather than using a tablet and stylus to input drawings, it is possible to use a scanner to automatically digitise an image produced on paper and copy it to the computer's memory. This is equivalent to photocopying the frames onto cels in the traditional process. However it may be difficult to automatically interpolate between keyframes which have been digitised in this way. First it is necessary to use image enhancement techniques to infer where the lines are in the digitised image, and second to guess which lines should interpolate to which. This is a very difficult problem if the keyframes are significantly different.

That is not to say that digitisation of original artwork is not a useful tool. It causes major problems only if we desire to use the digitised images for automatic in-betweening. If **all** of the artwork is produced by hand, it may be possible to digitise the drawings and use the computer to fill the outlines with colour. This is discussed in more detail below, when we consider computerising the painting process.

In-betweening

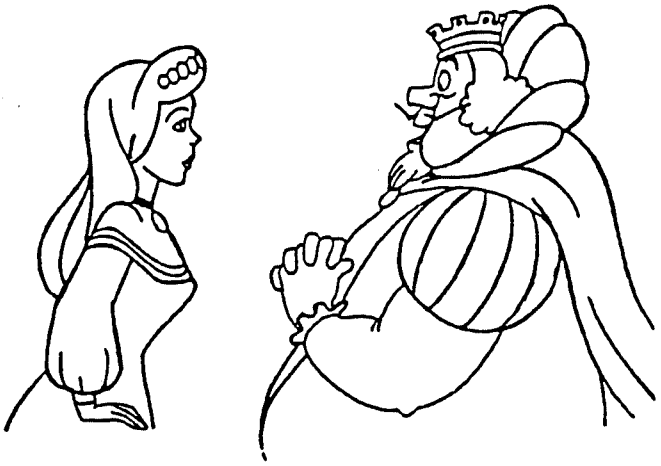
This is the part of the animation process which provoked the most attention in the early days of computer animation. It is one of the more automatic stages in the process, and, as such, seemed the ideal candidate for computer assistance. Many extravagant claims were made as to how its automation would revolutionise the entire world of animation. It was claimed that computers were going to replace whole teams of expensive in-betweeners and increase productivity whilst decreasing production costs; thus allowing even small animators to produce character animation of Disney quality.

The reality has turned out to be somewhat different. Producing in-between drawings automatically has turned out to be much harder than anyone originally anticipated. There are several reasons why this is true. In particular, it is normally the case that the objects to be animated are in reality three-dimensional: that is to say the images that the animator produces are merely two-dimensional projections of the three-dimensional scene he imagines in his head. The objects he depicts are recognisable to the human in-betweener—who can therefore infer a lot of the information which is missing from the two-dimensional image. This is not true of the computer which attempts to take over the in-betweener's job.

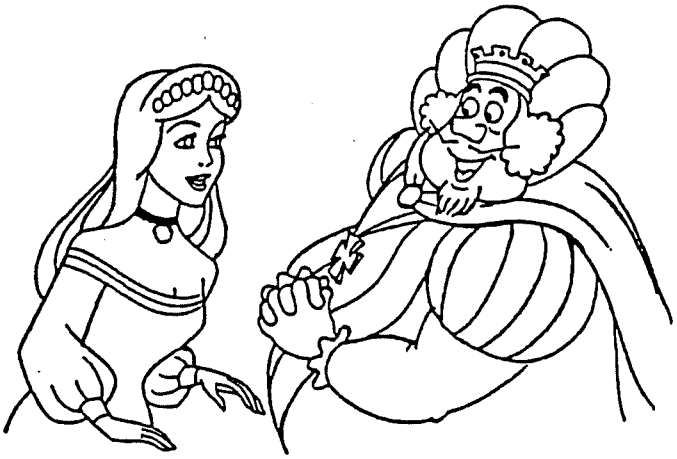
Let us consider the problem (highlighted by Edwin Catmull in his paper [*Catmull, 1978*]) concerning how to deal with parts of an image which are not visible in one keyframe, but appear in the next (or vice versa). For example, the first keyframe in figure 2.3 shows a human face in profile (notice that only one of the eyes is visible), the second keyframe shows the same head, but this time rotated towards the camera—how should the automatic in-betweener deal with the sudden appearance of the second eye?

The simple answer is that the computer cannot deal with this situation without some form of extra information being provided by the animator. The human in-betweener, on the other hand, knows quite a lot about human facial features and can easily produce suitable in-betweens. He is using his knowledge

The first keyframe



An in-between frame



The second keyframe

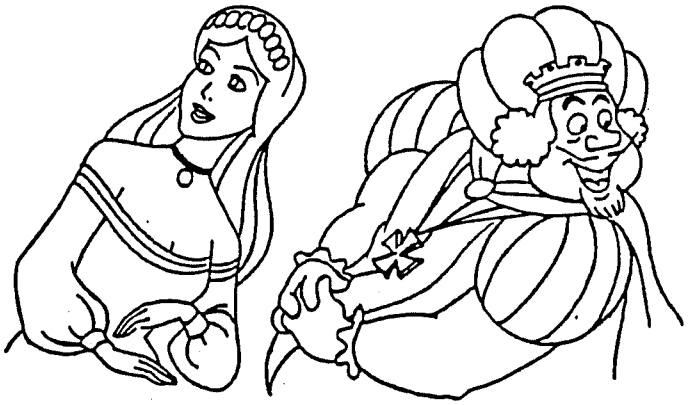


Figure 2.3: One problem encountered when producing in-between drawings of three-dimensional objects when working from two-dimensional keyframes is that objects (such as the king's eye) may be present in only one keyframe. When and how should this object enter any in-between frames? This example is taken from [*Catmull, 1978*].

about the real-world appearance of the objects being animated—knowledge which cannot be deduced from their two-dimensional projections alone.

Similarly, problems occur if a figure bends his arm between keyframes: the human in-betweener is aware that the forearm should remain of constant length and pivot at the elbow whereas the computer may well simply interpolate linearly between the start and end positions of the arm—with amusing results.

This problem is not confined only to complex three-dimensional objects; if two keyframes depict a simple rectangle at different rotations, interpolation between the vertex positions will not produce the effect of a rotating rectangle, the rectangle will change size as the in-between frames are produced (see figure 2.4). What is needed is an understanding of the geometry of a rectangle—i.e. that angles remain constant and sides do not normally change length (although in the **projection** of the rectangle, neither of these is necessarily true).

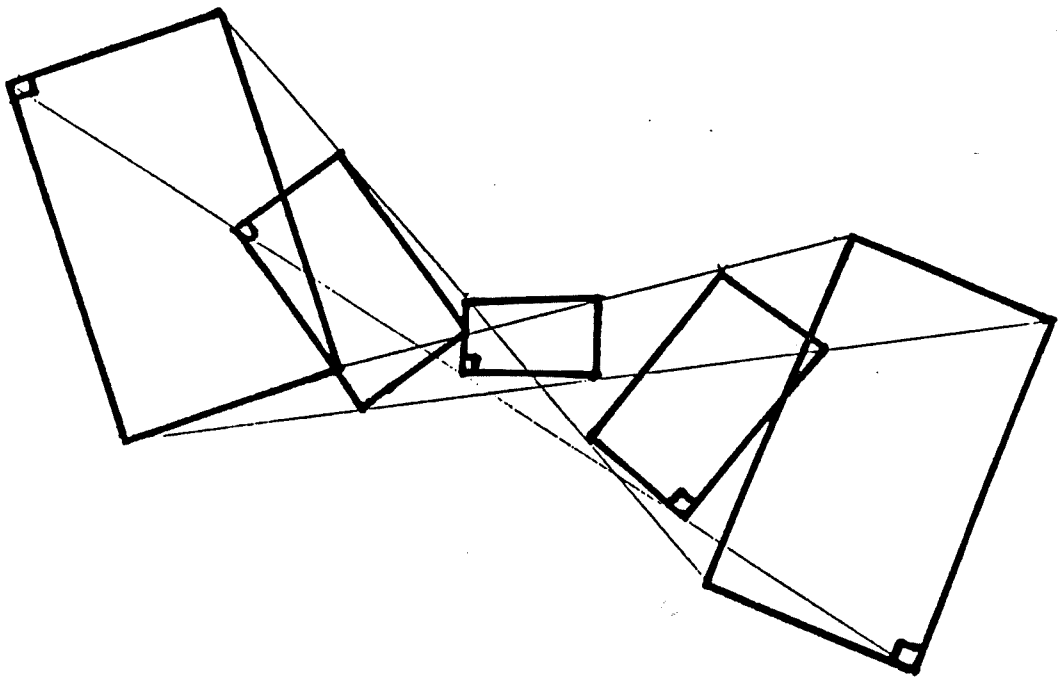


Figure 2.4: The inadvisable use of linear interpolation to produce images between keyframes depicting a rectangle at different rotations. Note that the rectangle appears to change size rather than simply rotating.

The above problems (and others like them) mean that computer-assisted character animation can never be a totally automatic process. It will always be necessary for the animator to spend some time helping the computer around the problems. Proposed methods of doing this include:-

- edit in the missing information (as some sort of hidden lines);
- break the objects to be animated into overlays such that no part of an overlay obscures another part of the same overlay during the sequence to be animated;
- use skeletal drawings (see section 4.4.1);

- do not attempt to animate three-dimensional objects using two-dimensional keyframe animation techniques, instead define three-dimensional objects with which the computer can work directly.

None of these suggestions is by any means a completely satisfactory solution, and they all increase the time needed to be spent on each frame by the animator (or his assistant), thus losing much of the advantage which automatic in-betweening was supposed to bring.

I do not mean to imply that computer in-betweening is never useful. It has been used to great effect in such films as *Hunger* by Peter Foldes [*Foldes, 1974*] in which the transformation of objects into other objects produces a very interesting and unusual effect. For example, in this film, instead of a man getting into a car to drive home, he actually transforms into a car himself. The transformation is very simple for a computer to perform since it merely involves linear interpolation between the corresponding lines in the two images. It would not be as easy for a human in-betweener since he has no real-world experience to help him understand the complex transformations and deformations involved. However, this technique, although eye-catching at first, quickly becomes repetitive and boring; it does not represent a very significant addition to the keyframer's character animation repertoire.

Keyframe animators should, I believe, make themselves aware of the possible contribution which computer in-betweening can make to their art, but not expect it to be the universal solution to their problems—nor should they feel threatened by it.

Line test

If the frames in an animated sequence are available to a computer system in such a form that the computer can display them quickly in succession on a graphics monitor, this display could take the place of a conventional line test. To produce a successful line test the computer must be capable of displaying the frames in "real time"—meaning that each frame is shown on the computer monitor for the same length of time as that frame will be displayed in the final film. *Double buffering* (showing one image while producing the next) should be used so that only complete images are presented to the viewer, thus recreating the effect of a conventional line test as closely as possible.

The main purpose of a line test is to check the quality and dynamics of the motion. It is therefore very important that the computer should be capable of keeping up with the correct display rate. This may be achieved either by predrawing a large number of images into the computer's graphics memory and then displaying these images one after another, at the correct rate; or having a display that is capable of drawing the images fast enough to keep up. It is important for the computer to keep track of the actual time which has elapsed since it began to display the sequence, since it can then calculate which frame in the sequence should be displayed at the current instant in time and omit (or re-show) any frames as necessary for the display to keep up with real time. This is especially true if each frame takes more than one "frame time" to be drawn. However the drawing speed should be such that not many frames need be omitted, since if too few frames are displayed each second, the quality of the

animation will be degraded to such an extent as to make the line test useless.

Now that better performance graphics workstations are coming onto the market (and at a reasonable price) this is an area in which use of the computer may become more common. However it is only feasible if the frames in the animated sequence are available to the computer—either because they have been produced using the computer, or because they have been digitised ready for painting by computer.

Xeroxing and inking

If the frames are stored on a computer system it may well be possible to produce the final animation direct onto film or videotape without any need for acetate cels. Thus this stage becomes redundant. If it is not feasible (or desirable) to write images onto film or videotape directly, it may be possible to connect a plotter to the computer and use it to draw onto the acetate cels which can then be painted by hand.

Painting

Contrary to initial expectations this is the stage of the keyframe animation process which has seen the most benefits from computer assistance. Computer in-betweening has been found to be much more difficult than originally anticipated; so much so that many computer-assisted animation systems make no use of it and have become instead *"scan and paint"* systems. In these systems all of the artwork is initially produced using pencil and paper, then the drawings are digitised using a scanner and "painted" at a computer graphics workstation. This painting process is most commonly achieved on a frame-by-frame basis, by selecting a colour from a "palette" displayed on the screen and indicating the areas which should be filled with this colour, probably using a graphics tablet and stylus. This has obvious speed advantages when compared with the traditional hand painting method. However it is important that the images to be painted are carefully drawn and well digitised so that the lines bordering each of the areas to be filled make a continuous boundary without any gaps at all. If there are any breaks in the boundary the colour may leak out and cover the surrounding areas. If this were to occur it may make it necessary to begin "painting" this frame all over again, thus wasting much of the time to be gained by use of the method. It is therefore very important to avoid this situation if at all possible. It is worth expending a little extra processing power on image enhancement to ensure that the lines do not contain tiny breaks after the images are digitised rather than forcing all of the "cleaning up" of images to be done totally by hand.

Hanna Barbera Studios are one of the most prolific producers of animated cartoons. They use the keyframe animation process. The cartoons they produce are aimed at the American television market and an important consideration in much of their work is that it should be of acceptable quality, but produced as quickly and cheaply as possible. This has lead them to study cost cutting very carefully. Consequently, their films contain more "cycles" (cyclicly repeated action) than those produced by many other animators, and, although most of their keyframes are produced by American artists, the majority of the

in-betweens are produced in the Far East where wages are lower. They have also had to consider where computerisation can help to increase production speed and reduce cost. They have decided that the way in which the computer can be of most benefit is to use a "scan and paint" technique; i.e. the original artwork is produced using pencil and paper, then digitised and electronically painted a frame at a time. They also maintain a computerised exposure sheet in order to automatically assemble the correct images together for direct writing to video tape.

A possible way to speed up the painting process even more would be for the computer to deduce how subsequent frames should be painted after the artist has painted one. It should be possible to use frame coherence to deduce, with reasonable accuracy, that if an area is filled with a particular colour in one frame, the corresponding area should be shaded the same colour in the next frame. A possible reason why this technique has not been widely adopted lies with two of the phases used in the previous sentence: *reasonable accuracy*—if the system makes too many wrong guesses, it may be more troublesome to correct its mistakes than to paint each frame individually in the first place; and *corresponding area*—it may be difficult for the computer to determine which area corresponds with which, and determine how to shade the area concerned.

Final Photography

Computer controlled rostrum cameras have been in use for several years. They allow camera movements to be performed under computer control with less possibility of error than with a human operator. The camera controller may possibly be able to take instructions directly from the exposure sheets if they are held in a suitable form on the computer.

Editing and Post-production

Computer generated special effects and editing techniques are already considered useful in a variety of situations; they may yet take on increasing importance. The use of video effects generators can, for example, increase the range of cross-dissolve and wipe effects over and above those achievable by optical means.

The next chapter describes my own implementation of a two-dimensional computer-assisted keyframe animation system. The discussion of its implementation and use will hopefully help in evaluating the effectiveness of computer-assisted keyframe animation.

Chapter 3

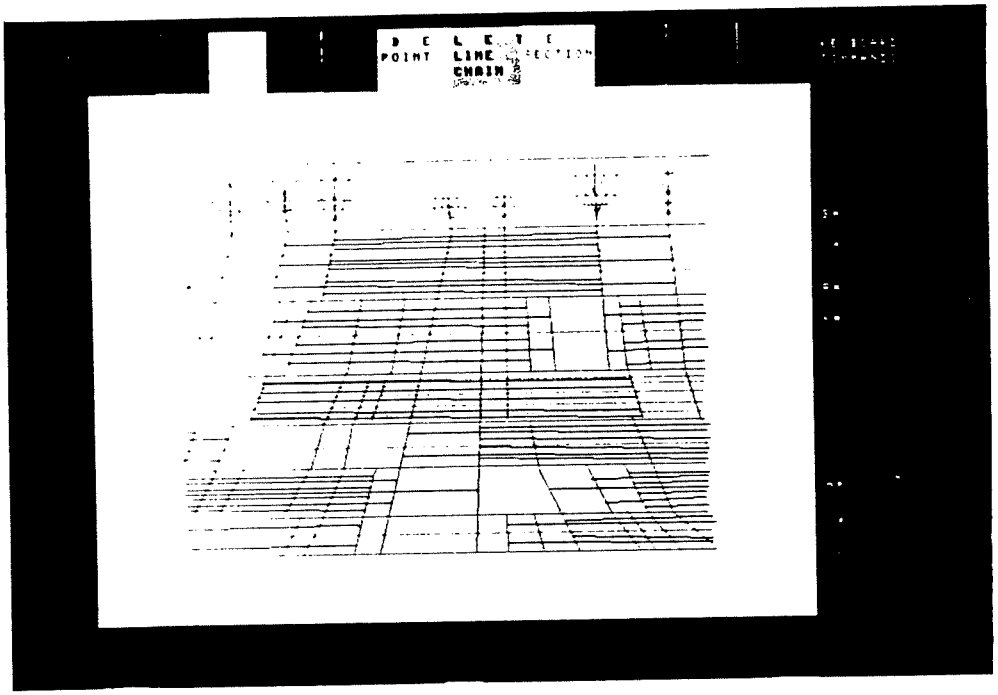
KAS—a Computer-Assisted Keyframe Animation System

As mentioned in the previous chapter, keyframe animation seems an obvious starting point from which to begin an experiment with computer animation. It is also the computer animation method which is likely to be most readily accepted by conventional animators embarking upon their first excursion into the realm of new technology. For these and other reasons, it was decided to produce a new two-dimensional keyframe animation system without too much reference to existing systems in the hope that producing it would force me to appreciate the shortcomings of the method and perhaps begin to prompt the process of developing better approaches. It was also hoped that the experiment would provide a catalyst for meeting professional animators and provoking discussion with them about the way that they believed animation should be progressing and how the computer could be of use to them. In this way it was hoped to obtain a clearer understanding of what traditional animators found difficult, and the kinds of animation they would like to produce—given the right tools. Then, perhaps, an animation system could be designed which allowed the animator to exercise his artistic flair in the way *he* wanted, rather than constraining him to work in the way I, as a computer scientist, thought he should work.

Before beginning to write the programs in earnest, I was introduced to a free-lance animator from Glasgow, named Lesley Keen. She had been commissioned by Channel 4 television to produce a short animated film based on the work of the Swiss artist Paul Klee. The film was to be called *Taking a Line for a Walk.* Lesley thought that it would be interesting, and in keeping with Channel 4's policy of innovative television, to use computer animation for some of the sequences. We therefore agreed to work closely together and hopefully both learn something of the methods and problems of each other's discipline.

3.1 KAS—a Keyframe Animation System

In consultation with Lesley, a suite of programs was produced to perform two-dimensional keyframe animation. I called this suite of programs *KAS.* This animation system will be described briefly below.



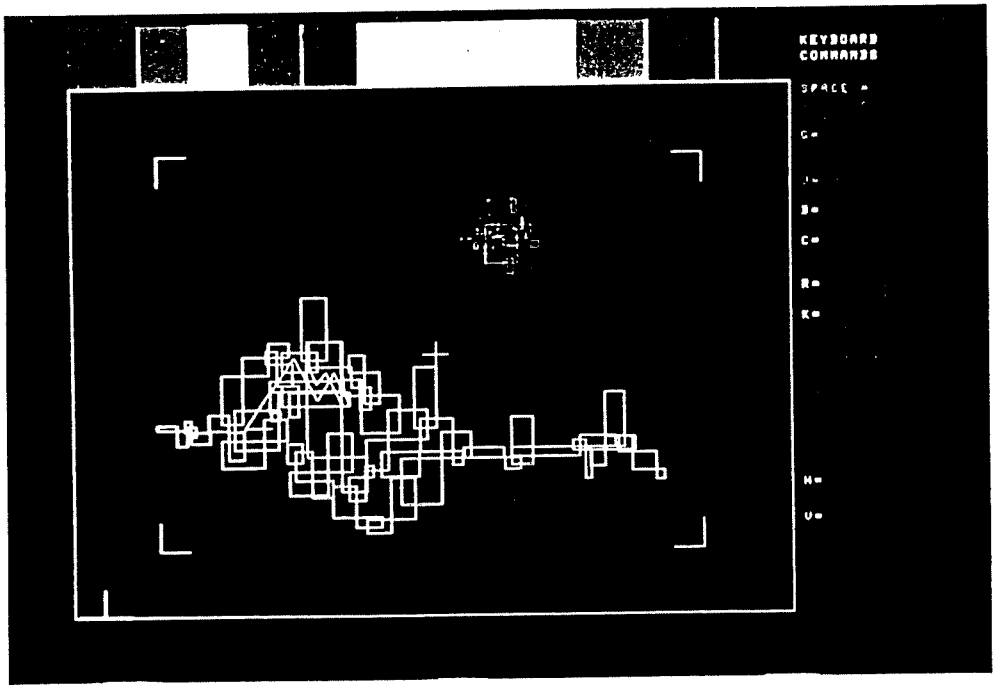


Figure 3.1: KAS screen layout. The user is given the option of either a black or white background on which to draw.

The suite consists of five main programs, written in BCPL [*Richards & Whitby-Strevens, 1979*] under the Tripos Operating System [*Richards et al., 1979*]. The host processor was a Motorola 68000-based machine and the graphics display hardware consisted of a Sigma T5684. Graphical input was achieved via a graphics tablet and stylus (see Appendix I for a more detailed description of the excellent hardware and software support environment provided at Cambridge University Computer Laboratory).

There now follows a brief overview of the five programs in the suite.

3.1.1 Keyframe editing

Images to be used as keyframes (pictures of significant events during the animated sequence) are entered into the computer using the tablet and stylus. Keyframes may either be drawn free-hand, or pencil sketches can be attached to the tablet's surface and traced. Once entered into the computer these images may be edited, using a variety of methods, until correct. They may also be edited to form other keyframes. Alternatively, these other keyframes may also be entered free-hand or by tracing.

In this program, commands are given either by the use of single key-strokes at the keyboard, or by selecting items from an on-screen menu. The selection of menu items is performed by moving a cursor (controlled by use of the graphics tablet and stylus) into the menu area and pressing the stylus down onto the tablet. The menu area consists of a row of coloured boxes, above the main drawing area (figure 3.1 shows the screen layout as the program is running). To the right of the drawing area is a list of some of the commands available from the keyboard. All commands are available by use of the keyboard, but only the most common ones are also available from the menu. This is partially due to lack of space, but is also an attempt to leave the screen area relatively uncluttered so as not to distract the user too much. The other alternative which was considered was to have the menu permanently attached to the tablet surface rather than displayed on the screen. This was rejected since it was felt undesirable that the animator should have to keep looking at the tablet rather than the screen when drawing freehand; and even if an image were being traced (in which case the artist would be concentrating on the tablet more than the screen), the paper containing the original image may well obscure the menu. Yet a third consideration is that it may well be desirable to use different menus at different times and it is very inconvenient to keep swapping menus attached to the tablet.

On entry into the program the animator is presented with a screen containing a rectangle representing a twelve-field; (as mentioned in section 2.2.1, many animators traditionally work on a piece of paper 12" x 8.75" which they refer to as a twelve-field). Inside this rectangle are marked the corners of a smaller nine-field and a cross to indicate the centre of the working area (the nine-field is of interest because when working in a twelve-field the significant action is

normally kept within the central 9" wide portion of the screen). If the animator has asked to edit an existing image, this will appear on the screen too.

Initially the program is in *"drawing mode"*. This mode is used for the input of new art-work. It is possible to draw freehand curves (actually a succession of short straight lines) by keeping the stylus in contact with the surface of the tablet as it is moved; or a sequence of longer straight lines, by lifting the stylus slightly away from the tablet surface when not marking the endpoints of the line segments. The position of the stylus above the tablet is always indicated by means of a cursor on the screen. Commands are provided to simplify the drawing of rectangles and circles, and the logical "pen colour" may be changed at any time by touching the stylus down in a "paint pot" in the menu area at the top of the tablet. Since lifting the stylus away from the tablet surface, then replacing it, will normally cause a straight line to be drawn between the two points, it is necessary to have some method of telling the computer when to leave a break in the line-chain (a sequence of connected line segments is referred to as a line-chain). The method decided upon was to insist that the user press the space bar on the keyboard in order to produce a break in the line.

In *drawing mode*, the only way to amend an existing image is to delete the most recently specified *point*, *line-chain* or *section* (set of line-chains) and redraw the relevant part of the image. It is possible to delete several such image elements—but only in reverse chronological order. More complex editing must be performed in *"edit mode"*. The major advantage of drawing mode is that all actions can easily be reversed by selecting the *"undo"* command from the menu displayed at the top of the screen (also available as command '*U*' from the keyboard).

Edit mode may be entered by selecting the *"edit"* option from the menu. This is the mode used for making more subtle alterations to the image, rather than merely deleting and re-drawing. It may be used simply to correct mistakes in an image; or to adapt an existing drawing to produce a related one, so that they can be used as successive keyframes in the development of an animated sequence.

In *edit mode* there are many ways of altering an image. The majority of these make use of two editing cursors (a cyan square and a magenta triangle) which can be manipulated by the use of the tablet, and keyboard commands. These editing cursors are used to delimit areas of action for the commands which alter the image. They perform a different function to the cursor which indicates the position of the stylus on the tablet, they are used only to indicate positions on the line-chains. Most editing commands are applied to the set of "*points*" (vertices) which lie "between" (in terms of order of drawing) the editing cursors.

Coordinates of *points* are the most important data produced by this key-frame editing program; *lines* are merely things which are used to connect points. It is therefore necessary for the artist to think in terms of manipulating the positions of points in order to alter the image.

Methods of altering the image include:-

- *Moving* (translating) a *line-chain*;
- *Moving a section* (set of line-chains comprising a logical image part);
- *Moving a set of points* (those points delimited by the editing cursors);
- *Rotating a set of points* (through a user-specified angle and about a user-specified origin);
- *Reflecting a set of points* (producing a mirror image in a user-specified mirror-line);
- *Collapsing a set of points onto a user-specified point or straight line* (useful for interpolation of sequences where images grow from nothing);
- *Deleting a set of points*;
- *Inserting points* (drawing);
- *Enlarging or Stretching the image defined by a set of points*;
- *Changing the colour of a line segment or line-chain*;
- *De-regularising an image* —sometimes a computer generated image can appear too regular, with lines too straight and angles too perfect; a command is therefore provided to turn a line-chain into a *"fractal polyline"* [*Mandelbrot, 1975*], in which each line segment is divided in two by the addition of a point at its centre. This central point is then randomly perturbed (with a user-specified maximum magnitude) in a direction perpendicular to the original line segment. In this way straight lines become more crooked, giving the impression of hand-drawing. The effect may also be used for more dramatic purposes. Consider, for example, producing a sketch of an imaginary island's coastline. Rather than spending time drawing each individual headland and cove, the artist can draw a crude outline of the island and use repeated application of the *de-regularise* command, with relatively large perturbations, to produce a suitably wiggly coastline.

Unlike *drawing mode*, not all actions can straightforwardly be undone in *edit mode*. This is because commands such as *'collapse'* destroy information. However, all of the simple transformations can be undone by using the *'move back'* command. This has the effect of reversing the most recent *'move'*, *'rotate'*, *'reflect'*, *'enlarge'*, or *'stretch'*. It works by remembering information such as angles, directions and origins, in order to be able to calculate the inverse transformation. There is an equivalent command, called *'move again'* (or just *'again'*), which simply repeats the most recent transformation. Both of these commands may be used repeatedly, if required. It is also possible to alter the part of the image which is affected by the transformation, by changing the position of the editing cursors before issuing the *'move back'* or *'move again'* commands. This makes it trivially simple to perform exactly the same operation on several parts of the image. Another useful feature is that the

information for the various transformations is held independently. This means that it is possible to reverse (or repeat on another part of the image) a complex transformation consisting of several simple transformations. The artist can, for example, repeatedly issue the commands: *'enlarge'*, *'again'*, *'rotate'*, *'again'* in order to apply a pre-defined enlargement followed by a pre-defined rotation to several image elements.

In *edit mode* the artist has the option of specifying that the un-edited image (as it was before entry to edit mode) should be displayed in the background during editing. Where the edited and un-edited images coincide—i.e. for those line chains which have not (yet) been moved—the display shows a dim version of the line's colour so that the artist can distinguish it from an image element which has already been edited. When a line has been moved it appears as a line at normal brightness and its old position is indicated using a faint purple line (see figure 3.2). This feature may be enabled and disabled at will.

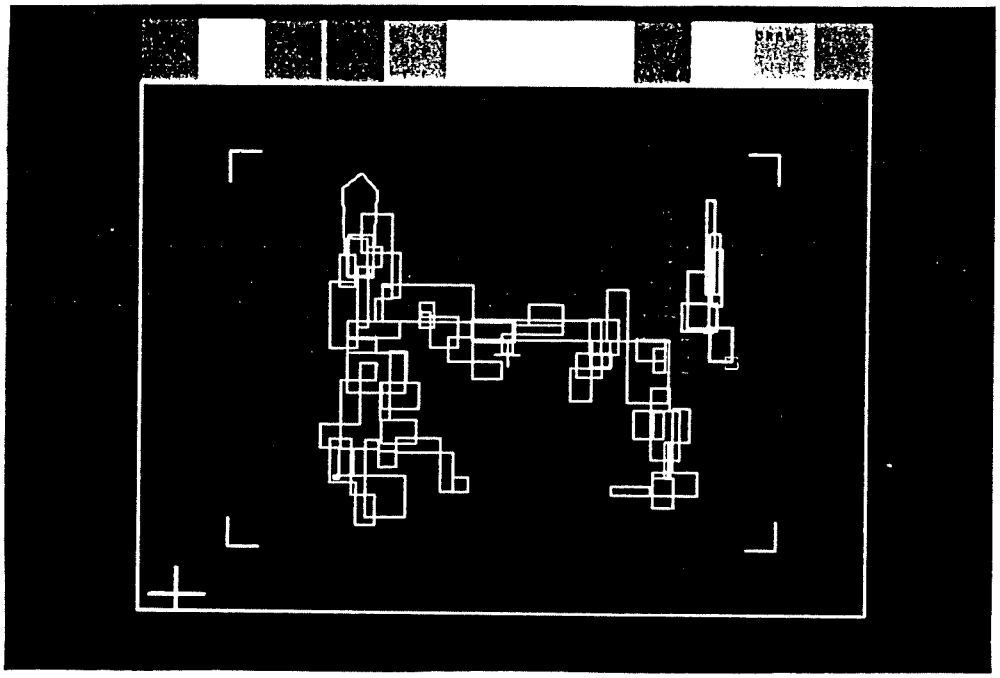


Figure 3.2: KAS in edit mode. Dull lines mean that that part of the image has not so far been altered during editing, bright colours show lines in new positions, and purple lines indicate the original positions of lines which have moved.

It is possible to perform accurate editing of fine detail by enlarging the image, making any corrections, and finally reversing the enlargement. It is also possible to move an object out of the viewing area (off-screen) ready for it to move back into view again during the animated sequence (the usable animation area is many times larger than the area visible on-screen).

This picture editing program is most commonly used for the production of keyframes. It can, however, also be used to edit individual frames in an animated sequence—thus allowing automatically generated frames (see description of in-betweening below) to be modified if necessary.

3.1.2 In-betweening

Automatic in-betweening has traditionally been considered to be central to the theme of computer-assisted animation. Without it much of the speed benefit of automating the animation process is lost.

On initial inspection, the problem appears simple: *given two or more keyframes which define an animated sequence, produce intermediate frames such that when the frames are displayed at a standard projection rate, the motion appears acceptably smooth and fluid, and lasts for the required length of time.* As mentioned in the preceding chapter, fulfilling these aims is by no means as straightforward as was originally presumed in the early days of computer-assisted animation.

The straightforward naive solution to the problem of producing the intermediate frames is simply to use linear interpolation between corresponding points in two keyframes. This was the approach which I initially implemented; however before this could be done there were several problems to be overcome. Some of the problems were easily solved—for example, how does one determine which points in the two keyframes should correspond? The obvious solution, and the one which was adopted, was to assume that the order of drawing of the line-chains is the same in both keyframes. This places an extra burden on the animator to conform to this restriction, but a little inconvenience at the time of keyframe production will almost certainly avoid many problems at later stages in the animation process. As an attempt at avoiding this restriction, I was initially attracted to the idea of allowing the animator complete freedom in his order of drawing and leaving the software the job of deciding which parts of the image should correspond. Unfortunately the solution to this problem is sufficiently difficult that not even a human being can be certain of determining the correct correspondence all of the time. The application of artificial intelligence techniques to enable the computer to make such decisions with any reasonable accuracy is a very interesting research topic in its own right, but I felt that it was beyond the scope of what was necessary for this initial experiment into the realm of computer-assisted animation. It would be possible to return to this topic at a later date if it appeared that the experiment warranted such substantial further development.

Once the computer has decided which line-chains should correspond (by whatever means), it is possible to interpolate one complete line-chain in the first keyframe to the equivalent line-chain in the second keyframe. However, linear interpolation is essentially a point by point algorithm, so what if the two line-chains consist of different numbers of points? This situation is quite likely to occur if the two keyframes have been drawn independently, rather than editing one keyframe to produce the second. The solution chosen was to ensure, before beginning interpolation, that the two line-chains contained the same number of points by adding extra points equally spaced along the length of the deficient line-chain (see figure 3.3). As interpolated frames are produced, any redundant points (for example, the central point of three collinear points) are removed in order to keep the amount of data to be plotted and written out to files to a minimum.

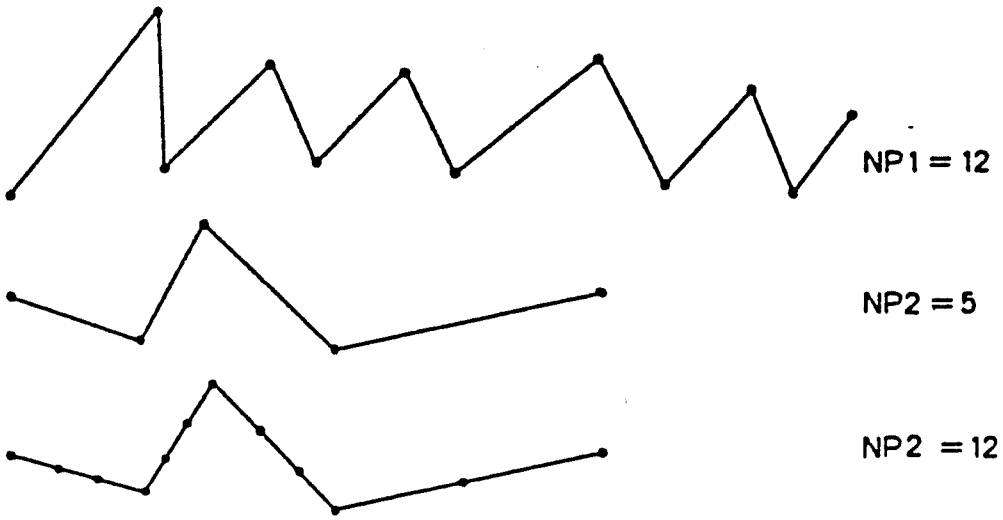


Figure 3.3: It is necessary to process line-chains so that they have the same number of points prior to interpolation.

A more serious problem occurs if the two keyframes contain different numbers of line-chains. Often the colours of corresponding line-chains in the two images give a clue as to which line-chains have been omitted (corresponding line-chains would normally be expected to have the same colour in both keyframes), so it may still be possible to make sensible assumptions about which line-chains should interpolate to which; but we are still left with the problem of how to invent line-chains to interpolate to the extra image components in the keyframe with the surplus. Several solutions to this problem were considered before a somewhat arbitrary decision was finally made.

- One possible solution would be to assume that extra line-chains should grow from (or vanish to) a single point (perhaps the point at their geometric centre), thus giving the effect of travelling towards (or away from) the viewer.
- Another solution would be to divide some nearby line-chain into two so that its two halves could each interpolate to a separate line-chain.
- A solution which was initially favoured was to make any line-chains which were missing from one keyframe occupy the same position as in the other keyframe (i.e. remain stationary during the interpolation). This has the advantage that if part of the image is intended to be static during the interpolated sequence it can be omitted from one of the keyframes. However it has the major disadvantage that one of the keyframes is altered in appearance by the addition of extra lines. This is not acceptable since it is the keyframes which define the motion and as such are exclusively in the artistic domain of the animator, and should not be visually altered in any way by the computer.

There are many more possible solutions to this problem, but none of them can be universally the most appropriate in all situations. There is no "correct" solution. If the animator wants to be in total control and does not want to be surprised by the animation, he should not produce keyframes which contain different numbers of line-chains. Nevertheless, I believe that it is important that the computer attempts to do something in this situation, since the animator may conceivably like the results, and even if he does not, he can discard the interpolated frames and edit one of the keyframes before trying again.

The solution finally chosen for implementation was to assume that any lines which are missing from a keyframe were intended to be "hidden" behind some other line-chain and should therefore be copies of those other line-chains. The problem came in choosing a suitable line-chain to copy. I decided to use the closest one since this would involve the 'new' image element crossing as little of the rest of the image as possible. This solution may not initially appear to be a very useful one, but it was in fact used several times to good effect during the making of the film *Taking a Line for a Walk.*

Now that the problems of determining which parts of the keyframes should correspond have been solved, it is necessary to decide how the interpolation itself should be performed. It has already been mentioned that simple linear interpolation between pairs of keyframes does not normally give satisfactory results. Animation produced in this way appears too mechanical. There are very noticeable discontinuities at the keyframes as moving objects suddenly change speed and direction; this has an unsettling effect on the viewer. A second problem is that the motion between keyframes is totally regular: objects move between keyframes in a straight line with constant velocity. This is also an effect which is very rarely witnessed in the real world; it therefore appears unnatural in the majority of situations to be animated.

In order for animation to appear smooth and natural it is necessary for motion to be *continuous* through keyframes. This applies not only to the direction of motion; continuity of velocity (the first derivative of the motion) is also very important. It often surprises people to note how sensitive human beings are to discontinuities in motion, extending down to the second derivative (acceleration) and even the rate of change of acceleration.

The question remains of how this smooth continuous motion is to be achieved. Perhaps the most obvious solution is to use some form of automatic curve fitting through the corresponding points in all of the keyframes which define a continuous piece of action. But what are the most satisfactory curves for this purpose? Polynomials are much too unpredictable to be of any use; fitting a fourth degree polynomial through the corresponding points in five keyframes, for example, is almost certain to produce a curve which is very different from the intuitive one (see figure 3.4).

A better solution would be to use Overhauser curves [*Brewer & Anderson, 1977*], which are formed by taking the weighted average of quadratic curves passed through successive sets of three positions (see figure 3.5). However, even these are surprisingly difficult to control. This is partially due to the fact that the theory behind their use only really applies to the case where it is possible to specify a Cartesian coordinate system such that the points to be interpolated have one coordinate which changes linearly.

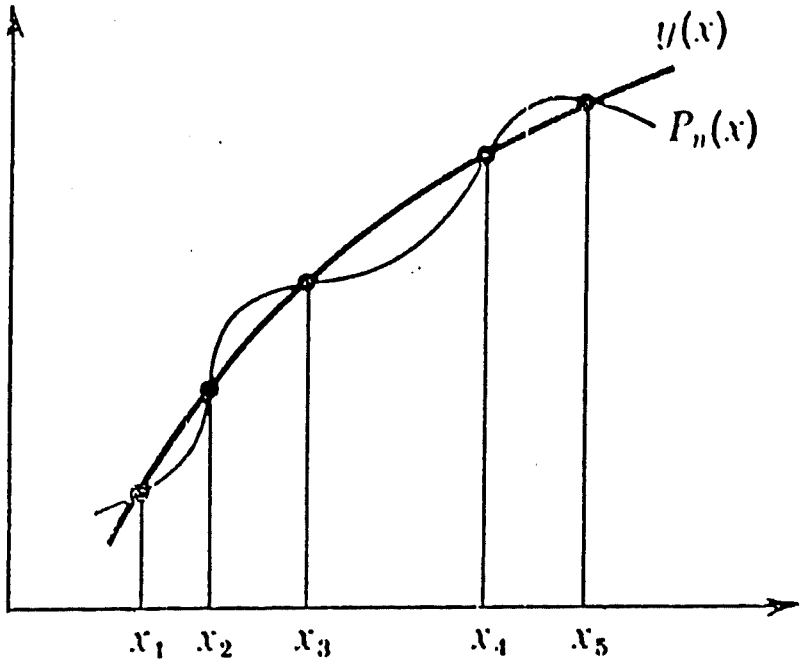


Figure 3.4: Fitting a fourth degree polynomial through five points representing the positions of a particular point in successive keyframes. The ideal curve is shown as a thick line but the polynomial approximation contains undesirable undulations (this example is taken from [*Hamming, 1971*]).

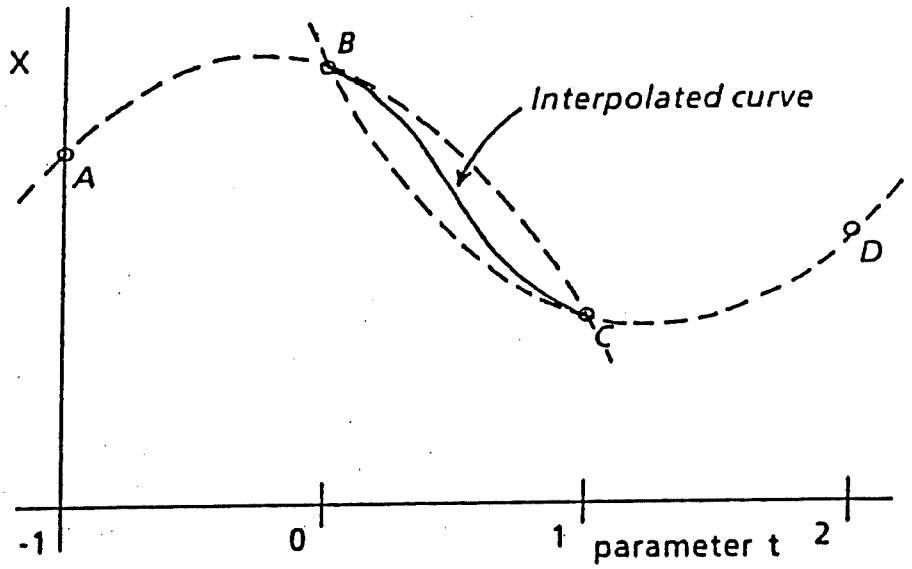


Figure 3.5: Generating the Overhauser curve between points B and C.

Other contenders for the choice of curve include various types of cubic spline (see [*Catmull & Rom, 1974*] or [*Kochanek & Bartels, 1984*]), but these too are difficult to control reliably. It should be noted that it would be necessary to use an interpolating spline rather than an approximating one, since it is essential that the image positions specified in the keyframes are achieved. This would not be the case if approximating curves were used.

An altogether simpler solution would be to specify the interpolation curve manually. It would then be the animator's responsibility to define precisely how the image should move. There would be no need to consider the mathematics underlying this motion; all that would be necessary would be to draw the desired curves on a graphics tablet.

If this solution is to be implemented it is necessary to consider the question of how to specify which parts of the image should follow which curve. At one extreme we could consider specifying the path to be followed by each individual set of corresponding vertices. However, this would involve the animator being aware of which were the corresponding vertices (and where any new vertices would be generated if a line-chain contained too few points); it is therefore totally impractical. But the question remains as to how we should specify that a curve is valid for more than a single set of points. How too can a curve be used for points which are not in the same spatial orientation as the relevant points on the motion curve? This final question is very important since it will normally only be the case that several points follow exactly the same path if those points comprise a rigid object which has merely undergone translation between keyframes.

It was decided to simplify the problem by insisting that only two keyframes are considered at a time and only one curve should be specified; this curve is adapted (translated, stretched and reflected) as necessary to suit each pair of corresponding points to be interpolated.

In order to specify a curve, the two keyframes in question are drawn superimposed on the screen (in different colours for ease of differentiation), and a sample interpolation path is drawn between any convenient pair of corresponding points (or without reference to any points at all). Once a sample path has been specified the interpolation progresses by considering each pair of corresponding points in turn. For each pair the sample path is stretched, translated and reflected as necessary until the ends of the path match up with the positions of the current point in the two keyframes. All that then remains to be considered is the specification of the timing information. How long should the transition between keyframes take? Should the velocity be constant; or should the objects be accelerating or decelerating (or some combination of the three at different positions along the path)? The timing is specified in a very similar way to the path definition—by hand. Tick marks are made along the interpolation path at points which correspond to the in-between frames in the sequence. In this way both the length of time taken (the number of frames) and the distance moved between consecutive frames (hence the velocity and acceleration) are all easily specified. This method of motion definition is in many ways similar to Baecker's p-curves [*Baecker, 1969b*] except that with p-curves only a single keyframe is produced—the curve explicitly defines the

motion—whereas with my approach the curve is stretched, translated and reflected as necessary to cause it to fit between each pair of corresponding points in a pair of keyframes (see figure 3.6).

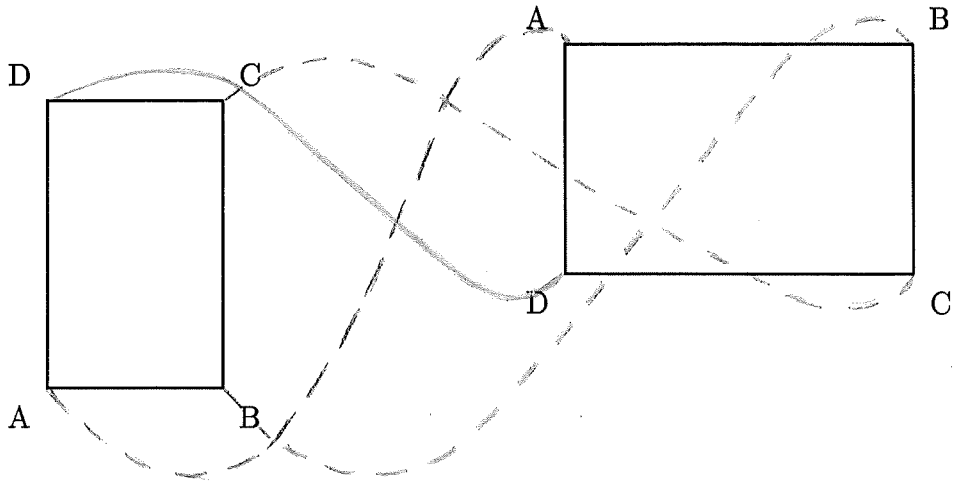


Figure 3.6: The way in which different parts of an image can follow different paths due to the single sample path being stretched and/or reflected to fit between each pair of corresponding points. The two rectangles represent the same object in two keyframes; the solid line represents the motion definition path drawn by the animator; the other curves are automatically derived from it.

Although the above description refers only to hand-drawing of interpolation paths and timing marks, it is also possible to make use of an interpolation path and/or timing information from a library of such curves and timing data. The library facility is also of use in attempting to achieve continuity of motion at keyframes. The interpolation curve used between a pair of keyframes may be saved in the library and displayed on the screen when the next pair of keyframes is being considered. This allows the new sample path to be drawn with reference to the shape of the previous path, allowing the artist to attempt to achieve continuity.

The above method of working may on first consideration appear too restrictive and to rely too much on manual intervention, but it was in fact found to be extremely versatile and simple to use during the making of *Taking a Line for a Walk*. This is particularly true since the method's main drawback—that an adaptation of the same curve is used for an entire image—is easily overcome. It is a simple matter to split a keyframe up into several separate images (in much the same way as transparent overlays are used in conventional animation) and animate each part independently, then re-combine the interpolated images using the superimposition program described below. Thus a different curve may potentially be specified for each separate logical image element.

As far as I am aware this simple but effective manual approach to the specification of the interpolation path and timing information is not used by any

similar computer-assisted animation system. This is slightly surprising since in my experience this method of working is particularly acceptable to traditional animators since it makes use of their traditional skills of hand and eye and is actually quite similar to the way in which animation may be produced by hand—a main animator may well produce similar curves to explain to an assistant the type of motion to be produced. There are several possible enhancements to the method as implemented which may add extra versatility and ease of use, but these will be discussed later in this chapter rather than at this juncture.

3.1.3 Superimposition

As mentioned above, it is possible in *KAS* to decompose keyframes into a set of disjoint parts, analogous to the overlays used in conventional cel animation. These separate image elements may be animated independently of each other. The sequences of frames produced may then be superimposed to produce the final composite sequence. In order to achieve this effect it is merely necessary to quote the names of the files containing the various sub-sequences of frames, and the name of a new file to hold the complete sequence.

Normally the various sequences to be superimposed will contain the same number of frames, and be intended to begin and end simultaneously. The superimposition program provided by *KAS* however is more general purpose in its design. If it is the case that the durations of the sequences differ, any sequence which ends early will have its final frame repeated for the remainder of the sequence (i.e. that object will remain stationary until the entire composite sequence finishes). It is also possible to specify that one sequence of frames should commence after the start of another sequence. In this case the initial frame will be repeated until this sequence should commence. All timing information is prompted for on entry to the program. Consider, for example, the case of two animated sequences which are to be combined into a single sequence: one sequence consists of 50 frames and depicts a bird initially sitting in a tree, then flying away; the other consists of 40 frames and depicts a flower growing; let us imagine that it has been stated that the flying bird sequence should begin after 20 frames of the flower sequence. In the composite sequence the bird will initially be stationary for 20 frames while the flower begins to grow, then the bird will fly away as the flower continues to grow for the next 20 frames, finally the bird will continue to fly for a further 30 frames while the flower remains stationary in its fully grown position.

An arbitrary number of sequences may be combined in this way. This is an advantage over overlays in conventional animation where only a small number of acetate cels (typically five) can be laid on top of each other before the background image becomes noticeably degraded.

The superimposition technique is also useful for certain special effects, such as superimposing the same sequence of frames on itself several times with a one frame offset between sequences, thus giving a multiple image effect.

3.1.4 Displaying animated sequences

Having produced an animated sequence, the next logical step is to check it for timing and smoothness of motion. If this aim is to be fulfilled the animation system should be capable of replaying a pre-calculated sequence of frames in real time (normally 25 or 24 frames per second). The usefulness of this stage will be greatly enhanced if it is also possible to replay the sound-track in synchronisation with the pictures; (sound-tracks are usually recorded before the pictures when producing an animated sequence).

Unfortunately it was not possible to fulfill these aims in a satisfactory manner using the hardware available for this experiment. The Sigma graphics terminal has a very limited graphics memory so it is not possible to draw several frames into its memory and display them one after another by manipulating the colour palette or other such means. It is also connected to the host computer via a 9600 baud serial line so it is infeasible to send all of the characters necessary to draw a frame during one frame interval (even when using the "abbreviated graphics mode" which involves the transfer of approximately half the characters needed to draw the same picture using the standard graphics mode).

Given that it is not normally possible to display a sequence of average complexity in real time, the viewing program allows the user to decide whether

1. every frame should be displayed (regardless of the amount of time taken to draw each frame), or

2. only selected frames should be shown in order that the dynamics of the motion should be approximately correct.

If the latter display method is chosen the program keeps track of the elapsed time since the first frame of the sequence was displayed in order to decide which frame should currently be on the screen. As soon as a particular frame has been displayed the computer checks which frame should be displayed at the current time and transmits the drawing commands for that frame. In this way (assuming each frame takes approximately the same amount of time to draw), the timing of the sequence should be almost as in the finished film.

Regardless of whether every frame is displayed, or only selected ones, the frames are *double-buffered* (i.e. one frame is displayed while the next frame is being prepared in an off-screen area of memory); it is then possible to switch to the next frame only after it is fully drawn so that the viewer is never presented with a part-drawn image. This was achieved on the Sigma T5684 by dividing the 4-bit deep graphics memory into two 2-bit buffers and drawing into one while the other was displayed. Since these two buffers were only two bits deep, it was only possible to display four colours simultaneously. This meant that the colours were not always as specified in the original keyframes, but this was felt to be less important than the double-buffering.

It is often helpful to employ both of the display techniques mentioned above on a given sequence since between them they can give the animator quite a good indication of both the dynamics and the smoothness of the motion. The results obtained are not as useful as a conventional line-test, but the fact that they are obtained so much more quickly and easily compensates, at least partially, for any shortcomings.

3.1.5 Producing hard-copy output

Once the animator is satisfied with the sequence produced, he will need some means of incorporating it into his film. There are many possible ways of achieving this:-

- the most satisfactory methods involve direct writing of the finished sequence onto movie film or video tape. Unfortunately, equipment capable of doing this (such as a Dicomed plotter—which writes a high-resolution colour image directly onto movie film) is very expensive and was not available for this project.

- A cheaper alternative is to use the video output direct from the graphics terminal to write to either a video recorder which is capable of single frame operation, or one of the (relatively cheap) film hard-copy devices designed for this purpose. Both of these alternatives were open to me; however the method suffers from serious drawbacks. In particular, the quality of the images produced on the graphics terminal is not acceptable for serious film or broadcast use—the Sigma graphics terminal has a resolution of only 768 x 512 pixels; various aliasing effects would be very visible at this resolution [*Crow, 1977*]. Manifestations of these aliasing effects would include the familiar "*jaggies*" (the stepped appearance of lines which should appear straight) and "*crawling*" around the edges of slowly moving objects, due to the perimeter lines taking on different aliases in different frames. There are several well known solutions to the aliasing problem [*Crow, 1981*]. They all attempt to overcome deficiences in spatial resolution by using greater depth (colour) resolution—the colour of each pixel (picture element) is determined by considering a suitably weighted average of the various colours which partially cover it. The Sigma terminal is capable of displaying up to sixteen colours simultaneously, chosen from a palette of 4096. If sixteen shades of grey, ranging from black to white, were chosen it should be possible to display a monochrome image with at least a modest attempt at anti-aliasing; a colour image, however, is out of the question without different graphics hardware.

  Since the animation produced for *Taking a Line for a Walk* was intended for television use, it was decided that the image quality achievable by this method with the available hardware was not acceptable.

- An even more straightforward alternative is simply to photograph the screen of the graphics terminal using a single frame movie camera. This suffers from all of the above drawbacks and the additional problem that the image quality will be even worse than that obtained by the previous method.

- A fourth alternative, which at first appears totally unacceptable but was in fact the chosen solution, is to produce output on paper. This idea also suffers from very serious drawbacks, not least the fact that a plotter will be kept busy for a considerable length of time while hundreds of yards

of images are produced. It is also the case that the plotter output will almost certainly not be suitable for direct use in the final film. It will, however, be possible to trace the frames produced in this way—just as the pencil sketches would be traced in traditional animation—and thus produce an image of identical quality to that produced totally by hand. It was this consideration which finally tipped the balance in favour of this method, since it was essential for *Taking a Line for a Walk* that hand- and computer- animated sequences could be freely mixed, without any obvious joins.

The output produced consisted of a series of frames, each plotted at a size specified by the animator—usually 12" x $8\frac{3}{4}$" (the animator's *twelve-field*). It was then a simple (but tedious) matter to line up an acetate sheet on top of each frame using the registration marks provided, and paint over the outline. The colours produced by the plotter are the same as used in the original keyframes so it was possible to use colour coding to aid the painter in his task.

This solution to the problem of hardcopy output is by no means totally satisfactory. If it were felt in the future that it would be desirable to perform the inking and painting stage by hand, it would not be acceptable to use the plotter again. A more sensible alternative would be to photograph each frame in one of the ways mentioned above and use back-projection to produce an image on the drawing surface of a light box ready for tracing.

3.2 Lessons learnt during the making of the film "Taking a Line for a Walk"

There is a well known saying that "the proof of the pudding is in the eating". This applies to software packages just as much as it applies to culinary matters—it is only when a package is used in a real life situation that it can be guaged whether it is a useful tool, or a hindrance to the user's creative talents.

I consider myself very fortunate to have known Lesley Keen, a trained animator who was not only willing to experiment with the animation package I produced, but actually wanted to use it to make a film. She was also involved from an early enough stage to be able to influence the development of the package. It was after much discussion with Lesley, and with some idea of the demands which would be made of the package, that much of the software was written. That is not to say that the software was strongly influenced by the subject matter, but the provision of certain features which were not essential for the making of *Taking a Line for a Walk* was given a lower priority than might otherwise have been the case. In particular the "*image painting*" side of things was not initially dealt with since *Taking a Line for a Walk* contains no filled areas.

In actual use the software performed well. It was, of course, necessary for the animator to adapt to a slightly different method of working, but this was not a major problem. In particular, the use of a graphics tablet and stylus as opposed to pencil and paper needed a certain amount of perseverance to master. It took Lesley approximately two days to become used to working

with the picture editor and in-betweening software. After that time she was familiar with how each command worked so that she could specify exactly how the animation should proceed. This is what the software had been designed to do: to give the animator total control over the animation produced. However, somewhat unexpectedly, Lesley soon complained that the animation was "*too predictable*". She said that she was already very good at producing animation over which she had total control—she was looking to the computer for something different: *she wanted to be surprised by what the computer did* in the hope that it would extend the range of effects she could produce. In seeking to achieve this aim, she attempted to add a certain amount of uncertainty to her sequences by breaking some of the rules concerning how line chains should correspond in different keyframes. By drawing the line-chains in a different order, or having different numbers of line-chains in the two keyframes she was able to see what effect it had on the animated sequence. If the sequence produced was not acceptable (as it very often was not) it was possible to either discard it as a failure or edit it until a more suitable sequence was produced. This editing could be performed either on the individual frames of the sequence, or by adjusting the keyframes. I found this a very surprising method of working; it was however used in the development of at least two sequences used in the film. In one such sequence a collection of straight lines, representing furrows in a field, collapsed into a series of small circles. The intermediate animation, quite unexpectedly, appeared very much like musical notation on a page, as small circles with short tails crossed horizontal lines. This idea was expanded upon to create the finished sequence. In another sequence a "city" was formed out of a swirling "snow storm" which would have been very difficult to animate without specifying a random initial frame and an eccentric interpolation path.

It was found that more complex interpolation could be undertaken with the aid of the computer than could practically be attempted by hand. This was partly due to the increased speed of production of in-between frames which allowed more time to be spent on specifying the motion. It was also because the computer is not confused by complex images; it is always aware of which point is which and how the interpolation should progress. Thus "messy" abstract images are no more difficult for the computer to deal with than simple translations of rigid objects. The same is not, however, true for the human in-betweener. He finds it relatively simple to produce in-between frames if he understands the objects which are being animated and how they move, but very much more difficult if the images are more abstract and only take on recognisable forms intermittantly. It is therefore true that human- and machine-in-betweening each have their strengths and weaknesses. It was fortunately possible to exploit both to their best advantage in the making of *Taking a Line for a Walk.*

I do not mean to imply that it would not have been possible to improve upon the keyframe editing and interpolation programs in any way. On the contrary, there are many minor improvements which could have been made. For example, the method of specifying the timing information by placing tick marks on the interpolation path was not always the most easily used technique. It was often difficult to specify the timing correctly first time. Certain features

of the method can be considered advantages when the animation produced is required to fit in with hand animation; in particular timing specified in this way contains irregularities due to operator inaccuracy—but this is not always desirable. It would often be preferable to define timing information by drawing a curve of distance travelled against fraction of time between keyframes and specify the number of frames to be produced independently, rather than trying to specify both pieces of information at the same time. It would then be possible to build up a more useful library of timing curves which could be used however many frames are to be produced between the keyframes.

3.3 "Painting" Animated Sequences

The version of *KAS* used in the making of *Taking a Line for a Walk* was capable of producing only outline images; there was no concept of filled areas. This was perfectly acceptable for this particular film since the film consisted only of line drawings. Obviously this will not be true in the general case. It was therefore decided to add a painting program to the suite as soon as time permitted. In fact the implementation (based on my original specification) was undertaken by Mark Chapman as an undergraduate project, as part of the Cambridge University Computer Science Tripos [*Chapman, 1984*].

It was intended that the program should be more automatic than simply to allow individual frames to be coloured by a simple operator-controlled process of "pointing and filling". I wanted it to be possible to paint an entire sequence of frames in a way that could be inferred automatically from the way that one of the frames of the sequence was coloured. The process must also be reasonably robust and capable of essentially error-free operation. A useful measure of such a system is the amount of operator time necessary per frame. This operator time includes not only the time needed to specify how an image should be shaded, but also any time necessary to correct mistakes. It is pointless having a program which can colour frames automatically if it colours them incorrectly, thus necessitating a large amount of *touch up* time.

The problem arises of how the painting process should be carried out. It is worthy of note that cartoon-style animation (the style most often attempted with this kind of animation software) traditionally uses large areas of flat colour; there is very little attempt at shading to give depth to the images. The most obvious solution, therefore, is to allow the user to specify that a certain enclosed area should be filled with a single solid colour—this is referred to as flood filling. A *seed point* is specified and the currently selected colour spreads out from this point, covering all neighbouring areas which were initially the same colour as the point (*pixel*) on which the seed point was placed. The colour stops spreading out when it reaches an area which is some colour other than this initial background colour. Thus, it will fill the entire area within the boundary but not spread outside it. One of the major disadvantages of the method is that the boundary must be a clearly defined continuous solid line. If there are any breaks in the boundary the colour will "leak out", possibly obliterating much of the image on the screen. A significant amount of user time may therefore be taken up with ensuring that there are no gaps in the perimeter line of areas to

be filled, and correcting mistakes which occur due to this not being the case. A further drawback is that the method is not well suited to the automatic colouring of a sequence of frames since the area to be filled is not defined in terms of the line-chains contained in the image database; the area to be shaded is obvious to the human viewer but it is very much less obvious to the electronic artist. It is probably the case that the perimeter of the area is defined in terms of parts of several line-chains, and the portions used may well vary from frame to frame. It is by no means a simple matter to automatically generate seed points for different frames in a sequence—it is not sufficient to interpolate the seed point at the same time as interpolating the image to be shaded since the object to be filled may change shape and cause the seed point to fall outside the area, with disastrous results.

For this reason it was decided to insist that the boundary of the area to be shaded should be defined in terms of the the line-chains produced by the keyframe editing program. Any number of complete line-chains may be indicated by positioning a cursor near them and issuing the *select* command. These line-chains have their end points joined together to form a closed shape, which is then filled with the currently selected colour. Since the vertices and line segments which comprise the boundary are known, a *boundary filling* algorithm [*Ackland & Weste, 1981*] is most suitable for performing the shading. Unfortunately the Sigma graphics terminal does not provide firmware support for boundary filling. To perform the boundary fill by software would have resulted in a very large amount of traffic along the serial line between the host computer and the graphics terminal, since each horizontal span of pixels to be drawn would have required several characters to be sent. This would have made the colouring process unacceptably slow. It was therefore reluctantly decided to make use of the Sigma's *flood filler* which is implemented in firmware. The Sigma's flood filler is actually not precisely as described above. Rather than colouring all pixels which were initially the same colour as the seed point, this filler spreads out from the seed point until it meets a pixel which is a specified boundary colour. This enables a boundary filler to be simulated much more easily since the background of the area to be filled may be arbitrarily complex and still require only a single seed point. The use of a flood filler obviously re-introduces many of the problems mentioned above; in particular the necessity of determining a seed point which is guaranteed to be within the area to be filled. It may even be necessary to find a whole range of seed points since the perimeter line-chains may well cross either themselves or each other thus producing several different closed shapes to be filled. It was felt, however, that any extra processing necessary to find suitable seed points would be more than compensated for by the speed increase achievable.

The graphics terminal used for this program was the Sigma T5688, an eight plane version of the display used for the rest of the suite of programs. Having eight bits per pixel means that it is capable of displaying up to 256 colours simultaneously; these may be chosen from a palette of 16 million possible colours. It was decided, however, to allow a maximum of 32 colours to be used per sequence for filled areas. This was so that only five of the eight planes needed to be reserved for the display of filled areas. Two of the remaining three planes

were used to hold the (four colour) outlines produced by earlier stages of *KAS*, and the final plane was used to highlight selected line-chains. The Sigma allows individual planes or sets of planes to be read- or write- selected; it is also possible to arrange the lookup table in such a way that if the bit in the "highlight plane" is set, that pixel should appear in the highlight colour. Thus highlighting an image component merely involves writing a 1 into the relevant position in this plane leaving the colour information in the remaining seven planes untouched. Removing all highlights is easily achieved by clearing the highlight plane, with no need to redraw any of the main image.

Colouring of line-chains and filling of areas is always carried out in the currently selected colour. This colour is selected from a palette displayed on the screen. There is a default palette but, if desired, any palette entry can be edited using a variety of means—such as mixing already defined colours or by a system which displays the current proportion of red, green and blue in the colour and allows these values to be incrementally altered (it is possible in this way to specify any colour which can be displayed by the terminal). Colour editing may be performed either before using a colour for the first time, or afterwards if the effect is not quite right. To allow the effect on the finished picture to be judged the colours in the palette may be manipulated while a quarter size version of the shaded image is displayed in one corner of the screen. This is possible since the Sigma T5688 has a drawing area which is four times the size of the screen. Normally only a quarter of this area is displayed at a time but it is also possible to display the entire area at a reduced size. Since the "painting" and "palette" areas are in physically separate areas of the graphics memory it is possible to instantly display either one of them, or indeed both of them. Figure 3.7 shows the screen organisation.

Having painted one frame in the sequence (normally a middle one, so that if there are different numbers of line-chains in the keyframes this will have been compensated for), the other frames are automatically painted in a similar way. Thus, if the first shading operation is to fill the area bounded by the seventh line-chain with the colour red for example, the same operation is performed on all other frames in the sequence. The order of shading is defined to be that specified whilst colouring the first frame so there is no confusion as to which areas should appear to be "in front of" other areas. If necessary this drawing order can easily be altered by editing the "*operations file*"—an automatically produced textual file containing a description of the palette and the colouring operations to be performed on each individual frame in the sequence.

The shading information stored in the *"operations file"* refers to line-chain and colour numbers, together with seed point coordinates, on a frame by frame basis. Any graphical editing of individual frames is also stored in this file, so that local changes in shading can be made to individual frames. One possible way of using the system is, therefore, to use the automatic shading option to colour those parts of the image which remain consistently shaded throughout the entire sequence but finish off any extra shading by hand, a frame at a time. Once the shading is complete it is this file which enables the finished sequence to be replayed. The sequence cannot be presented in real time but double buffering (using the remaining two unused quarters of the terminal's drawing

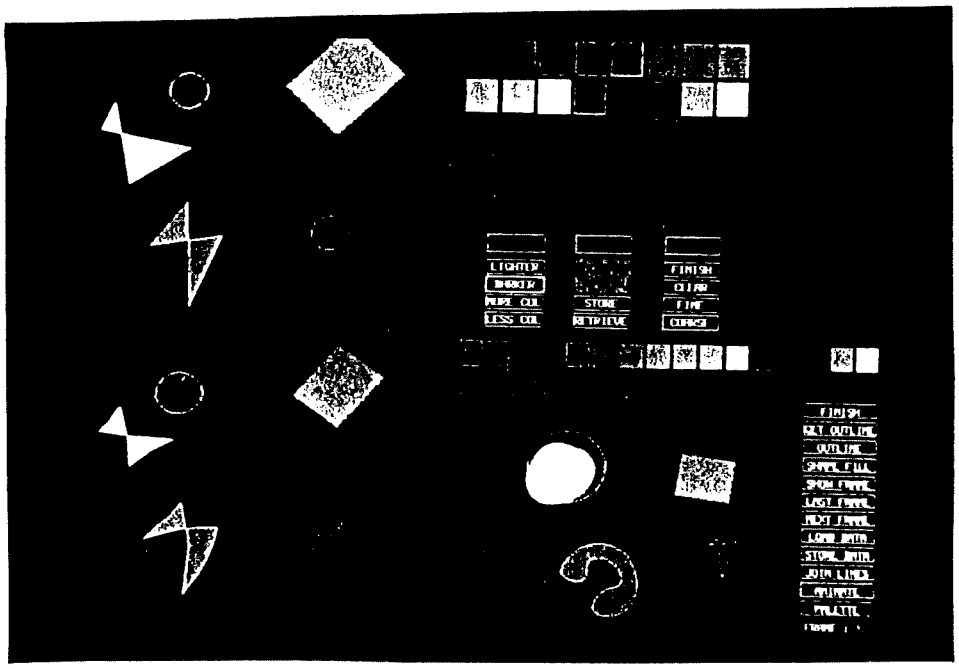


Figure 3.7: The four working areas used during image painting (courtesy of Mark Chapman). Normally any one of the areas would fill the screen but here they are shown at one quarter of normal size. The "painting area" is shown bottom right, the "colour mixing area" is top right; on the left are the double buffers used to display completed sequences.

area) means that the animator is never presented with a partly-drawn picture.

The creation of filled areas is a very important improvement to any animation production system. With such a facility it becomes possible to produce cartoon-style animation directly on the computer screen. The method, described above, of using only entire line-chains in the specification of boundaries is somewhat restrictive since it forces the animator to think ahead at the keyframe production stage and be aware of how the areas produced will be shaded. This was found to be the main drawback of the method. In practice the automatic colouring of frames was used for only parts of the image, with the rest of the shading finished off by hand on a frame by frame basis. I still believe, however, that automatic colouring of frames is a useful facility although it would be easier to use if a method were found to allow any continuous boundary to be used for filling, rather than just those defined by complete line-chains.

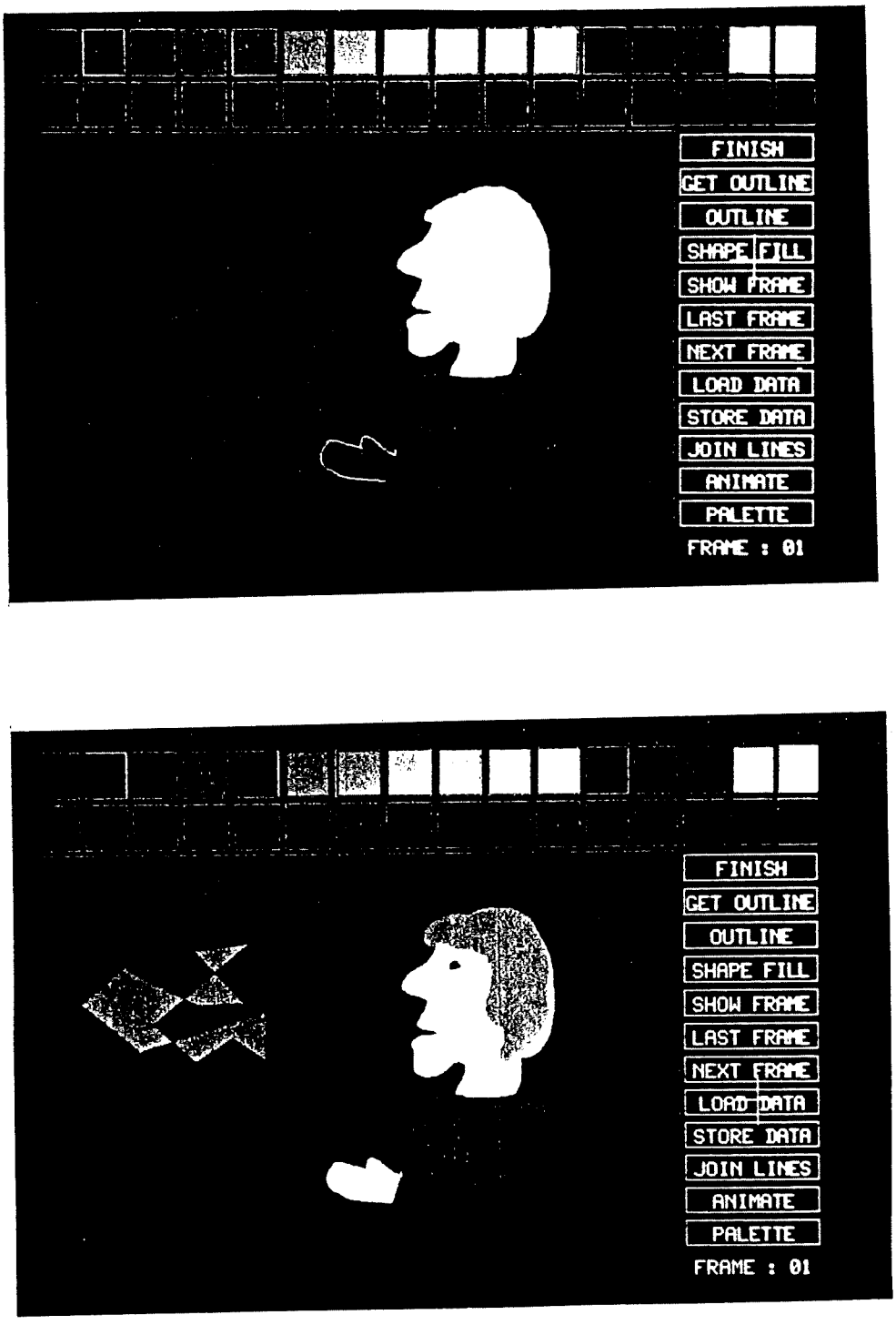


Figure 3.8: A simple example of painting an outline image.

Chapter 4

Computer Animation (review of allied work)

The previous chapter described an attempt to produce a simple two-dimensional keyframe animation system. In this chapter consideration is given to a range of animation systems developed by other people over the years. The purpose of this consideration is two-fold: firstly it will serve as a short, selective history of the development of computer animation, and secondly a discussion of these systems will help to illustrate their relative strengths and weaknesses. The next three chapters of this thesis will then describe my thoughts and experiments regarding a new approach to the definition of motion in computer-animated sequences.

Before considering individual systems it is worth pausing briefly to consider the types of problems they are trying to solve, since it is by no means true that each of the systems is attempting to solve the same problem; it is therefore not surprising that the solutions may be very different in philosophy and approach.

Even if we restrict ourselves to consideration of computer-animation systems in which the computer takes a hand in the production of the images to be used (thus ignoring such things as computer-controlled rostrum cameras and, perhaps, the majority of analogue systems too) there are still a large number of possible objectives for a piece of software designed to aid the animator in his task.

At one end of the spectrum there are systems which merely allow the digitisation and "painting" of hand-produced drawings, leaving the most artistic part of the work to be performed by traditional methods.

At the other end of the spectrum the complete sequence may be generated entirely within the computer: the computer is fully aware of the passage of time and is able to manipulate complex objects in time and space. These systems tend to rely on motion definition being performed by some form of graphical programming.

Between these extremes is a wide variety of different tools to aid the animator in his task, varying widely in the amount of input needed from the artist.

4.1 The Early Days at Bell Labs

Much of the earliest work in the field of animated computer graphics was performed at Bell Telephone Laboratories (Bell Labs). The first film to be given the label of "computer animation" was a four minute sequence produced by E. Zajac in 1961 [*Zajac, 1964 and 1966*]; it was called *Two-gyro gravity-gradient attitude control system.* As its name suggests, it was not an artistic film telling a fictional story but more a graphical form of output from a mathematical computer program. It opened up a whole new medium of expression for computer scientists and it was not long before others at Bell Labs had realised the potential of the medium for artistic expression. In the years up to 1967, Bell Labs produced a dozen animated films, by such pioneering computer animators as K. Knowlton, S. Van Der Beek, and F. Sinden.

Van Der Beek joined Bell Labs after some initial work at the University of Texas; his first film at Bell Labs was *Poemfield* produced with Knowlton in 1964. Sinden's earliest work was *Force, Mass and Motion* which he produced in 1966 [*Sinden, 1967*]. It was, however, Ken Knowlton who became the most influential force at Bell Labs in the field of computer imagery.

Most of the films produced during these early years were generated using one-off programs written in standard computer languages such as FORTRAN or PMACRO [*Alexander and Huggins, 1967*]; but Knowlton quickly recognised the need for a language aimed specifically at the production of computer animated films and set about developing one [*Knowlton, 1964 and 1965*]. The language was called BEFLIX; it was used for the first time in 1964 for the production of a film called *A computer technique for the production of animated movies* which was intended to explain the use of the language for film production.

Using BEFLIX it is possible to directly manipulate a matrix of 252 x 184 three-bit "pixels", representing eight grey-levels. Images are built up using statements which colour specific pixels or groups of pixels. These images can then be animated by sending electronic signals (waves) to perform image distortions.

After his initial experiments with BEFLIX, Knowlton developed a second animation language called EXPLOR [*Knowlton, 1970*]. It was implemented as a library of functions and subroutines to be called from a FORTRAN program. The name is an acronym for *EXplicit Patterns, Local Operations and Randomness.* It was intended for use in developing abstract moving patterns and mosaics—subroutines are provided which allow the placement of primitive shapes within images but do not specify their colours precisely, allowing for a certain amount of random colouring. For example,

```
CALL PUT (X,Y,W,H,PCENT,N)
```

means use colour `N` to shade `PCENT` percent of the pixels within the rectangle with centre at `(X,Y)`, width `W` and height `H`. The remaining pixels are left unaltered. Repeated calls of such procedures allow abstract images to evolve in a gradual, pseudo-random manner. EXPLOR has been used to make over twenty films—mostly on the theme of abstract moving patterns, as in *Pixillation* produced by Ken Knowlton and Lilian Schwarz in 1970; however *Olympiad,*

produced by Lilian Schwarz in 1971, is an exception to the general style in that it makes more attempt to tell a story.

4.2 Picture-driven Animation

While work was going on at Bell Labs to develop computer-animation languages, Ronald Baecker was undertaking research towards his Ph.D. at the Massachusetts Institute of Technology (M.I.T.) [*Baecker, 1969*]. He believed that graphical interaction should play a major role in the development of an animated sequence. In particular, he believed that the definition of the image transformation should itself be in graphical form; this gave rise to the name "picture-driven animation" [*Baecker, 1969b*]. In order to experiment with these ideas Baecker produced an animation system which he called GENESYS. GENESYS includes a simple language to define which image parts are associated with each other, but the images themselves and all motion and timing information is specified graphically using a locator device.

The method of working is as follows:-

1. Inform the system that a new film is being developed using the command `FORMMOVIE <name>`.

2. Issue the `FORMBACKGROUND` command then use the locator device to sketch the background in front of which the animation will take place.

3. Define the shape(s) of the various image elements to be animated, using the `FORMCEL` command. This command allows a range of shapes (positions) to be associated with a class name; for example:

```
FORMCEL #1 IN CLASS BODY
    (the artist draws his first representation of
       what a BODY looks like)
            :
FORMCEL #1 IN CLASS LEGS
    (the artist draws his first representation of LEGS)
            :
FORMCEL #2 IN CLASS LEGS
    (a picture of LEGS in a different position)
            :
```

4. More complex objects can be constructed using the `BIND` command. For example:

```
BIND BODY, LEGS
```

   The reason for `BIND`ing objects together is to allow them to be manipulated as a single entity.

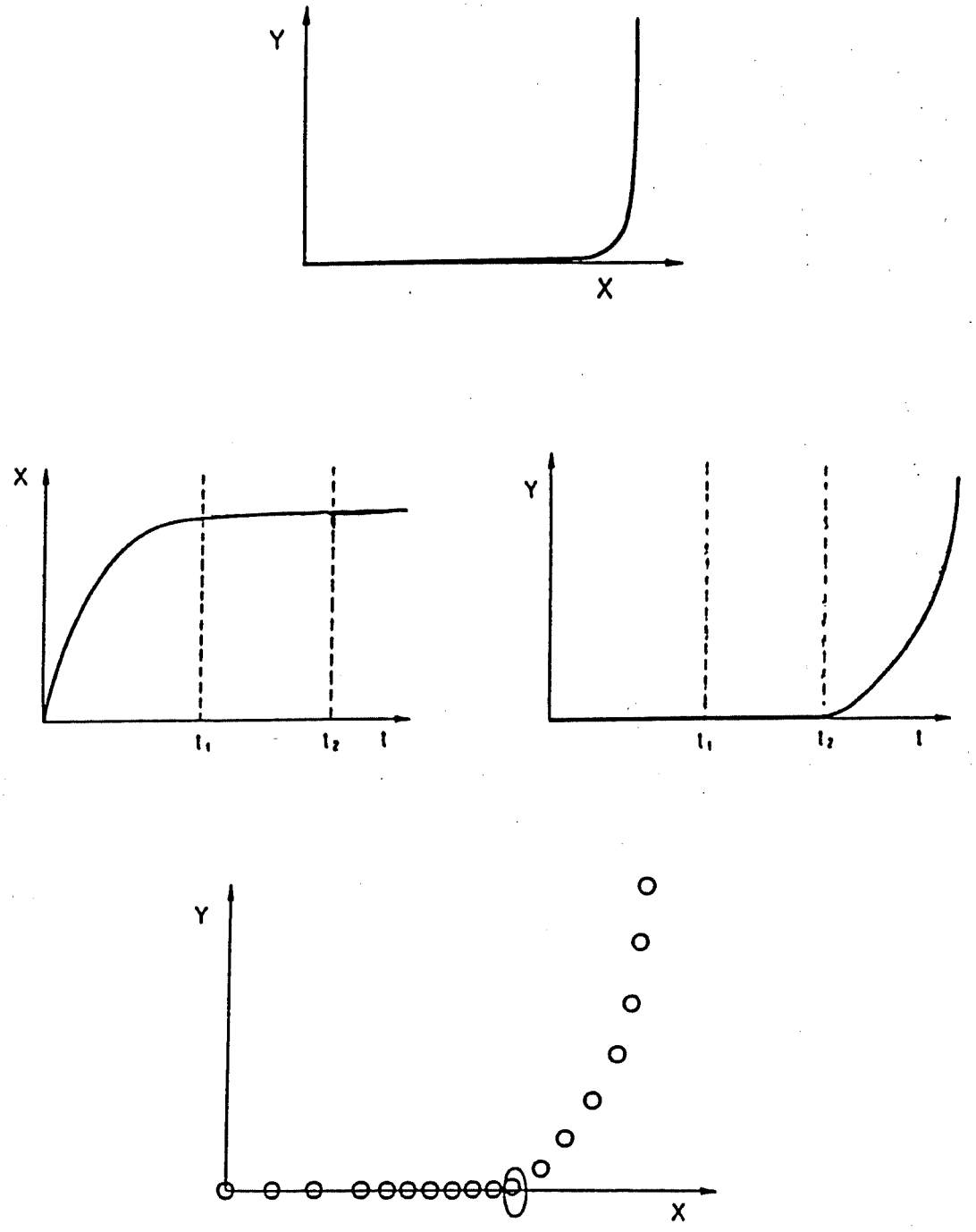


Figure 4.1: The use of p-curves to define motion. The situation to be described is that a car turns left at a T-junction. The top graph shows the car's trajectory but includes no timing information. The graphs on the middle row define the x- and y- coordinates of the car with respect to time. It is now possible to determine that there is very little motion between times t_1 and t_2. The information contained in each of the first three graphs is combined to form the p-curve, shown at the bottom; the marks along the curve represent the instantaneous positions of the car sampled at equal time intervals.

5. Having defined the shape of an image element, the motion is specified by means of a novel mechanism which Baecker calls p-curves. These curves allow both the positional and timing information of the motion of an image element to be specified using a single curve (see figure 4.1).

   P-curves are curves along which are placed marks at equal time intervals to show the object's position in each individual frame. The placement of the marks can be achieved by a variety of means including, for example, the artist tapping on a button to indicate the required rhythm for the motion.

   Note that p-curves are not used for in-betweening (as described in the preceding two chapters)—the shape of the image undergoing motion remains fixed as it moves.

6. Having got the path and timing of the global motion correct the artist is then free to specify the motion local to the object (such as the movement of legs) by selecting a sequence of image shapes to be displayed (one for each frame in the animated sequence). Thus it may be that in frame five an animated dog should be in a position defined by the p-curve, and its leg positions should be as specified in cel `#2` (as specified in the `FORMCEL #2 IN CLASS LEGS` command). In frame six it may well be that the legs should be in position `#3`.

4.3 Early Attempts at Modelled Animation

There are various ways of classifying animation systems; a commonly used classification is into *"computer-assisted"* and *"modelled"* animation.

The term "computer-assisted animation" is usually used to refer to automation of the traditional keyframe production technique. The computer is simply a tool for the animator to use to facilitate the conventional animation process; frames are electronically painted and new frames are generated either by hand or by mathematical interpolation. The computer knows only the positions of the various lines in the image; it is not given a higher level definition of the objects being animated.

In "modelled animation", on the other hand, the animation software is in some sense "aware" of the objects being animated and is able to manipulate these higher level representations to produce the new frames. Modelled animation systems generally work in three-dimensional space and use some form of animation language to define how objects should move (a *script*). More complex modelled animation systems may allow *actors* to be defined. Different people use the term "actor" to mean different things, but a commonly used definition (provided by [*Hewitt, 1971*]) is that actors are *"objects which can send and receive messages"*. These messages may originate directly from a script or from other actors in response to messages which they themselves receive. For example, when an actor in the guise of a car is told to move forward it may send messages to each of its wheels (sub-actors) telling them to rotate by a suitable amount. These systems aim to leave the animator free to worry about

the overall aims of the animation without having to build the more tedious animation details into the script.

ANIMATOR

ANIMATOR [*Talbot et al., 1971*] is considered by many to have been the first modelled animation system. It works in only two-dimensions but it does allow image components (consisting of points, lines and circles) to be associated together and given names and then manipulated as a single unit. The system contains a motion definition language which allows the animator to specify movements and transformations for these named objects by constructing operators based on motion primitives such as rotation, translation and change of scale.

MOP

Not long after this, a post-graduate student at the University of Utah, named Edwin Catmull, developed a modelled animation system which he called MOP (MOtion Picture language) [*Catmull, 1972*]. It was designed for the production of shaded three-dimensional sequences containing objects defined as hierarchical sets of polygonal facets. These objects had hidden surfaces removed by the Watkins method [*Watkins, 1970*] and were smooth shaded using the Gouraud algorithm [*Gouraud, 1971*]. With the benefit of hindsight, MOP may be considered an early attempt at creating an actor system, since the motion language allows the mechanics of motion to be defined using mathematical formulae or by using lookup tables. For example, the MOP statement:

```
40,99 W ROTATE "WHEEL", 2, 90
```

means "rotate the wheel around axis 2 starting from a rotation of 90°, in 60 frames (40 to 99) using table `W`". The detail of the motion (speed of rotation, acceleration, etc.) may be altered by modifying table `W`. Catmull made an animated sequence depicting the movement of a human hand to illustrate the usefulness of the system. The hand in this sequence has a somewhat polygonal silhouette and appears to be made of plastic (Gouraud shading is not capable of realistically representing skin characteristics) but the movement of the fingers is extraordinarily life-like. The sense of realism is aided by the fact that non-linear rates of change are easily modelled using this system.

4.4 Keyframe Animation

Computer-assisted keyframe animation has been discussed in the preceding two chapters; now its origins and development will be described.

4.4.1 The work of Burtnyk and Wein

Two of the earliest exponents of computer-assisted keyframe animation were N. Burtnyk and M. Wein of the National Research Council of Canada. It

was their paper [*Burtnyk and Wein, 1971*] which introduced the principle of performing the in-between calculation by computer. They incorporated their work into a keyframe system which they called MSGEN [*Burtnyk and Wein, 1971b*]. Burtnyk and Wein themselves used the system to produce several films including a well-known short sequence entitled *Running Cola is Africa* which depicts a runner transforming into a Coca-Cola bottle which in turn transforms into a map of Africa; but it was the well-known artist Peter Foldes from the National Film Board of Canada who produced the most impressive films using their system. Foldes began with *Metadata* in 1971 and culminated with the superb *Hunger* in 1974, which received numerous prizes at international film festivals and was even nominated for an Academy award. In these films Foldes makes great use of the interpolation facilities provided by MSGEN to produce unusual effects which would be difficult to produce by hand. In particular, objects are continually undergoing metamorphosis so that rather than a man sitting down in an armchair, for example, he actually transforms into the chair. The images produced by this system are strictly two-dimensional line drawings (so if three-dimensional objects are depicted, lines which should be hidden are not removed) but this did not matter for films such as *Hunger*; indeed it actually added to the sense of difference, and excitement at a new technique.

MSGEN uses a form of linear interpolation to produce its in-betweens, so many of the problems described in the previous two chapters are encountered; however the system does allow an individual interpolation timing rule to be applied to each separate image component so that some parts of the image can progress between keyframes at constant velocity, while others are accelerating and yet others are accelerating then decelerating. Indeed it is possible to apply these timing rules independently in the x and y directions thus allowing a limited range of curved interpolation paths.

Skeleton Animation

Despite the success of such films as *Hunger*, Burtnyk and Wein were quick to realise the deficiencies of the linear interpolation technique and decided to do something about it. The obvious solution was to allow arbitrary interpolation paths rather than limiting the artist to linear interpolation, but they felt that it would still be too restricting to insist that there should be only a single interpolation path per image element, so they suggested a somewhat different solution. Rather than expecting the interpolation software to produce the desired motion, they decided that the artist should produce a much larger number of keyframes so that even relatively crude interpolation techniques would produce motion that was close to what the animator intended. Since it is not realistic to expect the animator to produce large numbers of detailed extra frames, they invented the technique of skeleton animation [*Burtnyk and Wein, 1976*] to reduce the amount of drawing required to an acceptable level.

Using this technique, the animator produces large numbers of keyframes for a crude representation of the object to be animated—a "stick man" for example. The artist also provides a single drawing to define how the completed frames should be automatically produced from the skeletal representation. In this way the motion can be accurately controlled by the animator, without the need to

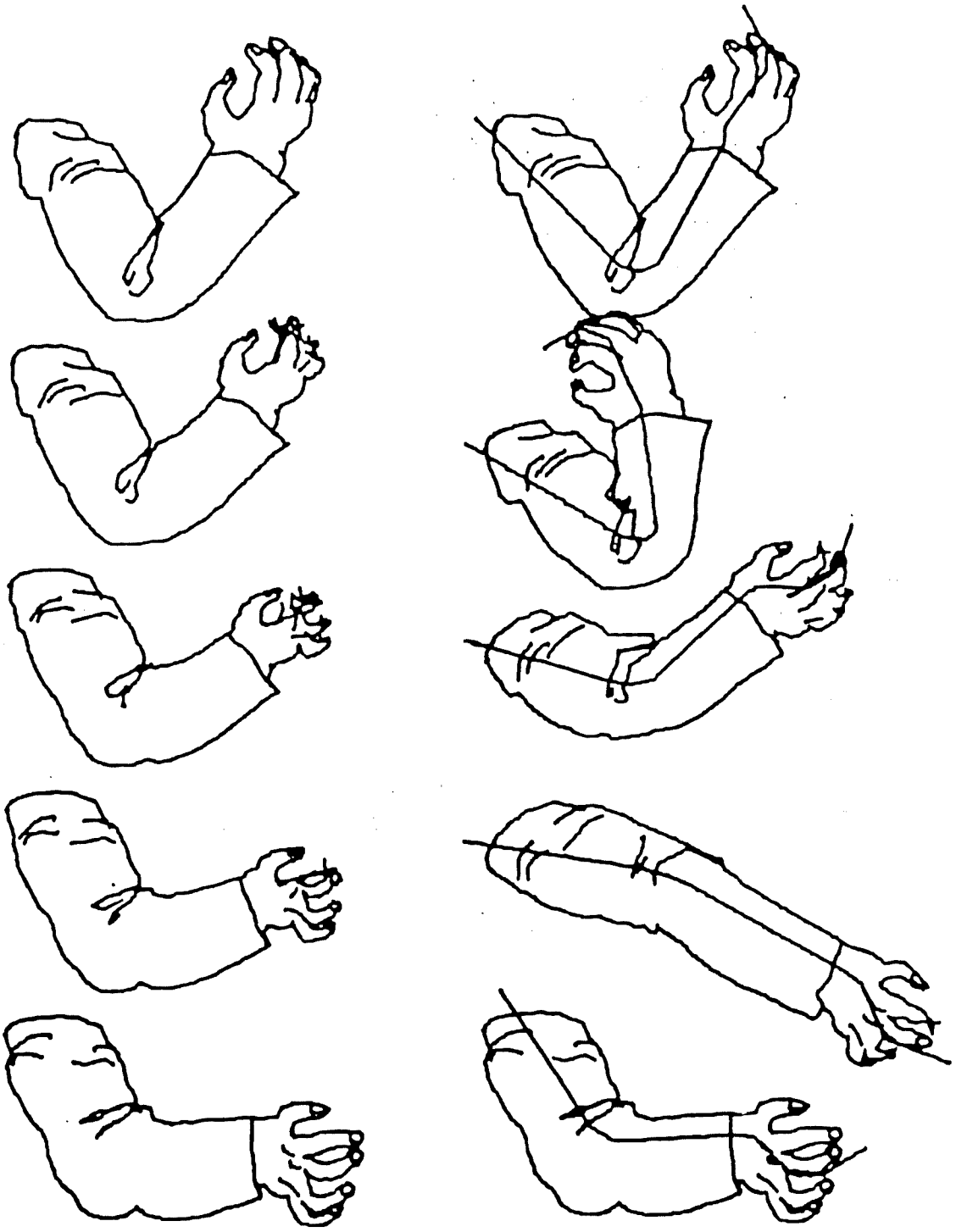


Figure 4.2: Illustration of the skeleton animation technique for computer keyframing (taken from page 52 of [*Magnenat-Thalmann & Thalmann, 1985*]). On the left is the result obtained by straightforward linear interpolation between the initial and final keyframes—note the deformation of the fingers. On the right is the result obtained by using the skeleton technique (in this case without any interpolation at all). More keyframes are drawn but the animator needs only to define the position of the representative lines, the detail is filled in automatically thus necessitating even less drawing than that needed to perform the interpolation.

produce large numbers of artistically good keyframes. Consider, for example, the animation of a bending arm as shown in figure 4.2.

4.4.2 ANTICS

ANTICS [*Kitching, 1973*] is an animation system which includes keyframing amongst its methods of motion specification. Only linear interpolation is used but there are facilities to "smooth" and "taper" the motion in the time dimension in order to cushion the stopping and starting (this is traditionally referred to as *"easing in"* and *"easing out"*).

ANTICS is a large system which has been developed over many years and now has a wide range of facilities—such as a high quality painting program—which were not envisaged when the idea for the animation system was originally conceived. The system incorporates a range of methods for motion specification and allows sequences to be built up by the superimposition of several independently animated image elements which may have had their motions defined in a variety of different ways. Possible methods for specification of motion include:-

- *Keyframing*;
- *Tagged and Scaled movement*, where the movement of one object is linked to the movement of another, such as a person's eyes following the flight of a bird;
- *Wave movement*, which causes picture elements to undergo periodic vibration—it may be linked with other motion for more complex effects;
- *Curve movement*, which allows picture elements to move along paths described by mathematical functions;
- *Constrained Random movement.*

There is also a range of image manipulation commands, such as *zoom*, which give access to television-style special effects.

4.4.3 CAAS 2

One of the most successful keyframe animation systems, in terms of artist acceptance and quality of results, is the *Computer-Assisted Animation System* (now in its second incarnation and known as CAAS 2) developed at the New York Institute of Technology's Computer Graphics Laboratory (NYIT). The system consists of three major components designed to assist the conventional animation process:-

1. **Tween** [*Catmull, 1979*] is a two-dimensional package to facilitate the production of keyframes and in-betweens. It runs on a VAX 11/780 computer attached to a vector graphics terminal. The artist uses a tablet and stylus to produce keyframes; then the computer calculates in-betweens by mathematical interpolation. The interpolation may be controlled by drawing graphs of x or y against time (or making use of standard graphs

from a library)—this makes *easing in* and *easing out* quite straightforward.

Experience has shown that, even with this system, it is difficult to create the required subtlety of non-linear motion without producing very large numbers of keyframes. For this reason, in normal use the computer is typically used to generate only one or two frames for each artist-generated keyframe. However, the system is easy to use and very responsive (including good line-test facilities), and has therefore gained widespread approval from artists who have used it—indeed it is still in use today.

A major area in which Tween scores over other similar systems is in the fact that it maintains an exposure sheet (as described in section 2.2.1) on the computer. This allows timing to be straightforwardly adjusted by simply re-assigning a keyframe to a different time slot in the exposure sheet and having the system recalculate the in-betweens. There are several other advantages in having an electronic exposure sheet, since it is the exposure sheet which is the master definition of how the animation should be produced. Holding it on a computer allows the computer to control all aspects of the animation, such as sound synchronisation and gathering together the correct background and foreground images for each frame for direct output to film or video tape.

2. **SoftCel** [*Stern, 1979*] is a system designed to be the electronic equivalent of the traditional animation process of copying pencil drawings onto acetate cels and painting them. The line drawings are entered into the system either by scan-converting the vector images produced by Tween, or by using a television camera to digitise conventionally-produced pencil sketches. In either case the result (after any necessary automatic image enhancement) is an anti-aliased line drawing held in an eight bit frame-buffer.

   Four planes of the frame-buffer are reserved for shades of grey for the anti-aliased outlines. Anti-aliasing is necessary in order to avoid artifacts such as jagged edges since the display resolution is only 512 x 480 pixels. The remaining four planes of the frame-buffer are used to hold the colours for area filling. Enclosed areas can be filled using the *tint-fill* algorithm [*Smith, 1979*] which effectively flood fills from a seed point to the *centre* of the perimeter lines (remember that these lines are anti-aliased—so they are wider than a single pixel). Since the perimeter lines and the colours for filling are in separate planes of the frame-buffer no information is lost where they overlap; this allows the video lookup table to be arranged to display a colour computed to be the mixture of the two overlapping shades—this has the effect of anti-aliasing the filled area.

   A major disadvantage of the SoftCel system is that only fifteen colours are available for filled areas (four bits are capable of holding sixteen different values and one of these is reserved to be a transparent background). This restriction can be overcome by breaking an image up into several parts for later overlaying (each overlay can use fifteen colours of its choice), but this obviously necessitates extra work. When the images (including a

background image) are finally overlaid (as defined in the exposure sheet) they are written into a 24-bit frame-buffer with eight bits for each of the primary colours, red, green and blue; so there is effectively no restriction on the number of colours available at that time. The final image may then be transferred directly to film or video tape.

3. **Paint** [*Smith, 1978*] is a painting system, developed by Alvy Ray Smith, which is used to paint backgrounds in front of which the animated action should take place. As with most painting systems it makes use of a tablet and stylus to perform the actual drawing; a range of menu options are available to enable the system to simulate a variety of different styles of painting, such as normal painting, air-brushing, smearing and rubber stamping. The results can be anti-aliased against an existing background. It is also possible to fill enclosed areas with colour (using *tint-fill*, described above) and define shapes and colours of brushes to speed up the painting process—for example, if it is necessary to include hundreds of daffodils in a picture, the artist could simply define a daffodil as a brush and put down hundreds of copies of it.

4.4.4 Keyframe animation using moving point constraints

Two of the problems with computer-assisted keyframe animation systems are that interpolation is normally performed using exactly two keyframes at a time; and it is usually only possible to specify a single interpolation path for all of the image elements in a keyframe (without separating the image elements into different overlays and superimposing later). Bill Reeves attempted to overcome these restrictions by enabling the animator to specify what he terms *moving point constraints* [*Reeves, 1981*].

Reeves' method works as follows: the animator displays a series of two or more keyframes on the screen simultaneously and draws an explicit path for a specified point (known as a *moving point*) to follow during interpolation. Tick marks are made along the path to represent the positions of the point at equal time intervals. The user can specify as many of these moving points as he desires in order to accurately control deformation of a line-chain; the motion of intermediate points in the line-chain (which are not specified as moving points) is computed as a "smooth blend" of the motion of their neighbouring moving points. In order for this algorithm to work, it is necessary for there to be moving point curves specified for the end points of every line-chain; if the user does not specify these points as moving points it is necessary for the system to invent a suitable curve based on what it knows of how other points on the curve have been constrained to move (see figure 4.3).

Reeves suggests several methods of performing the "smooth blending" referred to above, but concludes that the most satisfactory is a method which effectively treats the area delimited by the position of the curve in two keyframes and two moving point curves as a Coons patch [*Coons, 1974*] and computes in-between positions for the curve by taking two-dimensional projections of lines ruled on this three-dimensional surface. However, even this algorithm can produce undesirable results under certain circumstances (see figure 4.4).

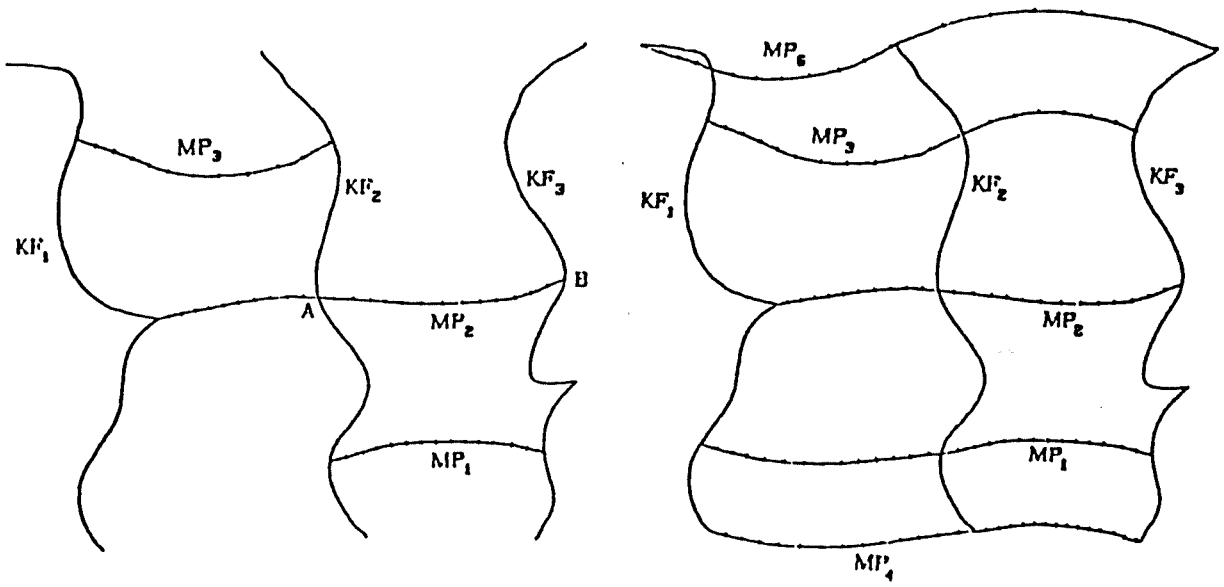


Figure 4.3: Keyframing using moving points. The diagram on the left shows a typical user specification of how a particular line chain should move. Its positions in three keyframes (KF1, KF2 and KF3) are shown together with three user specified moving point curves (MP1, MP2 and MP3). The diagram on the right shows the automatically completed network.

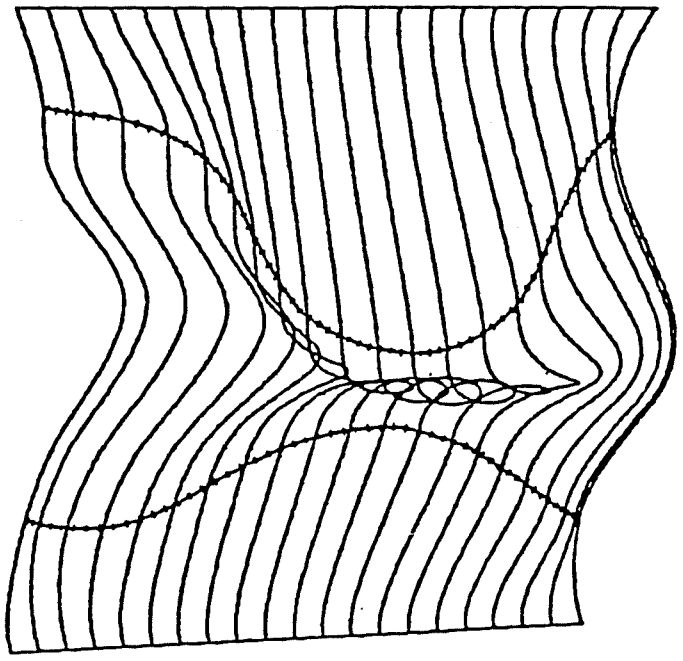


Figure 4.4: Interpolation using Coons patches. The lines on the left and right of this diagram represent the positions of a particular line in two keyframes, The four lines across the diagram represent moving point constraints. Three patches are therefore used to control the interpolation. The contortion of the interpolated lines as they cross the middle patch are caused by the algorithm's excessive attempt to achieve continuity of curvature and twist across patch boundaries.

4.5 Animation at Ohio State University

Over the years, the Computer Graphics Research Group at Ohio State University has been one of the most dynamic groups in computer animation research. Their emphasis has always been on modelled animation systems. Real-time production or previewing has also been a high priority. Several animation systems have been developed there under the direction of Charles Csuri, who has also been active in the commercial field through his company Cranston-Csuri Productions.

4.5.1 GRASS—a three-dimensional real time system

Csuri developed a real-time animation system on the IBM 1130 as early as 1970 [*Csuri, 1970 and 1974*]. However it was Tom De Fanti, one of Csuri's Ph.D. students, who in 1972 developed a real-time, three-dimensional, modelled animation system which could readily be used by a computer novice. De Fanti called his system GRASS, "the *GRAphics Symbiosis System*"; he later extended the system for commercial use [*De Fanti, 1976*].

GRASS was written entirely in assembler and ran on a PDP-11/45 with a Vector General graphics display. Much of the speed of the system was due to the fact that the Vector General maintains a three-dimensional display-list which it is able to manipulate using special purpose hardware to produce translated, rotated or scaled images in real time—but only in the form of monochrome line drawings without hidden lines removed.

The GRASS language has essentially two types of primitives:-

1. **objects** consisting of user-defined lists of vectors in three-dimensional space. These objects may be given names and grouped together in a tree structure. They are potential image components, but at the time of definition there is no need to know whether this object will be displayed for a particular frame, and if so at what orientation—that is specified elsewhere in the script (program). Using this system, up to 60 image components may be displayed simultaneously.

2. **commands**—which may be further classified into:-

   (a) commands which leave the vector list unaltered—translation, rotation and scaling, for example. These commands are executed by making use of the Vector General's display list manipulation hardware, so may be performed very rapidly.

   (b) commands which cause the display list to be altered—such as clipping, smoothing or windowing.

Commands may be grouped together into a script which defines the motion of the objects during the life of the animation. The language allows conditional execution and the definition of macros.

4.5.2 ANIMA, ANIMA II and ANTTS

ANIMA [*Csuri, 1975*] is an animation language developed by Charles Csuri in 1975. It supports three-dimensional real-time animation.

ANIMA II [*Hackathorn, 1977*] is a non-real-time three-dimensional colour animation system written in assembly language, which runs on a PDP-11/45 attached to a Vector General graphics display equipped with a joystick, buttons and dials.

ANIMA II provides the animator with the following main facilities:-

- **An interactive geometric modeller** which allows the user to build complex objects, by positioning geometric primitives in space and specifying their colours. Internally the objects are represented as polygonal surfaces, but the user does not need to be aware of this.

- **An animation language** with which to script the motion. Complex motions must be broken down into simple changes in time and space. For example:

  ```
  set position <name> <x>,<y>,<z> at frame <frame number>
  change position <name> to <x>,<y>,<z> from frame <start>
                    to frame <stop>
  ```

  which causes the object with name `<name>` to move in the obvious way.

  Other types of motion can be specified by replacing the word `position` by such words as `rotation`, `size`, `shape` or `path`.

  As we can see, the language supports instructions which can remain active over a specified time interval; the use of several such instructions allows several independent processes to take place in parallel—loops and conditional statements are unnecessary.

  When the script is complete a preprocessor works out exactly where each object should be for each frame; then the scene is rendered by taking account of the user-specified viewpoint and lighting. However, the shading algorithm calculates only one shade for each polygonal facet; no attempt is made to perform smooth shading.

  The final outcome of the interpretation of the script is an animation file, which may be further interpreted to produce an image on some form of display device.

- **Methods of viewing and recording frames.** A "line test" can be produced on the Vector General display (without colour or shading) but final sequences are recorded onto video tape for later display. To aid them in this task, the Csuri group have developed a method of converting the animation files to NTSC video signals in real time.

Although quite advanced for its time, ANIMA II had significant drawbacks particularly in terms of the quality of the final images; the system did not

perform smooth shading, and there were aliasing problems. It was also not possible to have transparent objects in the animation, and these were something with which the Graphics Research Group were keen to experiment.

For these and other reasons a new system called ANTTS (ANimated Things Through Space) was developed [*Csuri et al., 1979*]. This is a very complex three-dimensional animation system which aims to overcome the problems of ANIMA II—such as dealing with many types of image component rather than just polygonal surfaces, and allowing sequences to be edited after production. A major objective was to enable the animation of high complexity scenes containing hundreds of thousands of polygons (or their equivalent). The shading algorithms used are much more comprehensive than those used in ANIMA II and include texture mapping and rendering of smoke, hair and clouds.

The use of the term *texture mapping* in their system is slightly different to the conventional use of the word in the field of computer graphics: texture mapping is generally used to mean the mapping of a two-dimensional image onto a smooth three-dimensional surface [*Blinn & Newell, 1976*] or the simulation of wrinkled surfaces by perturbing the local surface normals as the surface is displayed (*bump mapping*) [*Blinn, 1978*]—in either case the actual surface is left unaltered; it is only the illusion of texture which is added. In ANTTS on the other hand it is assumed that highly complex objects are being rendered, so it is possible to actually change the colour and orientation of particular facets to create a "real" texture.

4.5.3 The Skeleton Animation System

SAS, the Skeleton Animation System, was developed by David Zeltzer in 1982 [*Zeltzer 1982 and 1982b*]. It is a three-dimensional animation system aimed at accurately reproducing the motion of human figures—but only considering the underlying joint motion, not taking account of muscles, soft tissue, clothing and so on. It has been used to produce extremely realistic motion of a skeletal figure. The main interface to the system is via SAL, the Skeleton Animation Language.

SAS is very much dedicated to a single purpose; all necessary information regarding the mechanics of motion is built into the system so there is very little way in which an animator can influence the style of walking, for example—he can only decide when the figure should walk, jump, turn and so on. The system is capable of a limited form of adaptive motion—the skeleton is aware of its own position and those of some of its surroundings and is therefore capable of adjusting its walk to take account of slightly uneven surfaces.

A short example will illustrate how adaptive motion is catered for in SAL. The command:

```
bend left_knee to 45 z 15 until (left_heel touch)
```

calls for a non-weight-bearing rotation of the left knee about the local z axis to an absolute angle of 45°, taking 15 frames to do so; however the rotation should stop immediately if the left heel touches the ground.

4.6 Analogue Systems

There are several systems (used principally by television companies—or at least for animation aimed at the television market) which may be considered to produce "video effects" (of the type associated with television programmes such as *The Kenny Everett Television Show*) rather than what is presented in the majority of this thesis as "computer animation". The difference lies in the fact that the computer is not responsible for producing any of the original images, it merely manipulates them—using either an analogue computer or a hybrid computer containing both analogue and digital components. This manipulation may take such forms as rotating, scaling, translating or cropping the image, or altering the colours. More complex systems may allow the source image to be decomposed into a number of sub-parts each of which may be manipulated independently. The source of the picture to be manipulated may either be a conventional live television picture or the output from some form of video graphics generator, such as a *Quantel Paintbox*.

Two of the earliest analogue animation systems were developed in the United States; they were called SCANIMATE and CAESAR [*Honey, 1971 and 1971b*]. They have been used extensively in the production of station identification logos and commercials. In fact, CAESAR (which stands for Computer Animated Episodes using Single Axis Rotation) breaks across the boundary between analogue video effects and keyframe animation systems; since, although essentially an analogue system, it is also capable of performing in-betweening.

A more recent system is the *Quantel Mirage* from Britain which is used primarily to provide interesting mixes between shots on such programmes as *Top of the Pops*. It is a very powerful system which allows an enormous variety of mix and wipe effects between sets of live images. The British television 625 line standard specifies that each television image should consist of 576 visible lines and have an aspect ratio of 4:3. It is therefore necessary for each line to consist of 768 pixels in order to make the pixels square; *Mirage* allows each of these pixels to be treated independently of all others and so allows a practically unlimited range of special effects to be produced, given the correct programming. Unfortunately really spectacular effects do require quite a lot of setting up effort, which partially explains why the range of effects used is more limited than might be expected.

4.7 Three-dimensional Keyframe Animation

At the New York Institute of Technology's Computer Graphics Laboratory it was decided to progress from two-dimensional keyframing to the use of keyframes to define motion for three-dimensional objects, particularly human figures. The result was a system called BBOP, developed by Garland Stern [*Stern, 1983 and 1983b*], which runs on an Evans and Sutherland Picture System attached to a VAX 11/750. Three-dimensional keyframing is a form of modelled animation since the software needs to be aware of a high-level definition of the object to be manipulated, but the more familiar script for the definition of motion is replaced by the use of keyframes. The objects to be ani-

mated are defined as articulated tree-structured models composed of polygonal or quadric surfaces. Key poses are defined by interactively selecting a joint to be manipulated then using a three-dimensional joystick to bend that joint to the required position; (there are no constraints on joint movements so there is nothing to prevent the animator bending a knee forwards, for example). Having positioned all joints as required for a pair of keyframes, interpolation of joint angles is performed to produce the in-between frames. This gives very acceptable results for objects composed of rigid parts articulated at specific joints, but it is important for the animator to avoid impossible motions since the software gives no help with this. By careful choice of key poses it is possible to allow the system to produce quite large numbers of in-between frames with perfectly acceptable results.

The system also includes comprehensive shading software with anti-aliasing and texture mapping. It is being used in the production of a fully computer-generated feature film called *The Works* which has been under development for several years at NYIT. If the short sequences so far seen are anything to judge by, *The Works* will almost certainly be one of the most sophisticated computer animated film so far produced; however I suspect that there is no real intention ever to complete this film, it is more a "continuously updated show reel". Another superb sequence produced using BBOP was *Catherine's Wheel* for the BBC2 *Horizon* programme, in which a computer-generated dancer is mixed with live action.

The main difficulty in developing animated sequences using BBOP is defining the articulated object to be animated in the first place. To date the majority of objects which have been animated have been existing digitised objects which have then had articulation added. The animators at NYIT have found this to be a less than totally satisfactory way of working as it is usually very difficult to add joints to an existing figure; it would be much better to design an articulated model in the first place.

Building on the experience of BBOP a more complex three-dimensional animation system called EM [*Hanrahan & Sturman, 1984*] is now under development at NYIT.

4.8 Actor-based Animation Languages

It was Carl Hewitt of M.I.T. who first introduced the term *actor* ([*Hewitt, 1971*] and [*Hewitt et al., 1973*]); he defined an actor as an object which can send and receive messages. All elements of a system are actors, and the only activity possible in the system is the transmission of messages between them. Customisation of the system is achieved by telling the various classes of actors how to respond to the messages they receive—what actions to perform, messages to send, and items of information to remember or forget.

4.8.1 DIRECTOR

Kenneth Kahn developed an animation language called DIRECTOR [*Kahn, 1976*], based on Hewitt's actor types. In a DIRECTOR program, actors and the

```
TO DEFINE.FLOWER                          --LOGO command
10 ASK OBJECT(MAKE FLOWER)                --flower creation
20 ASK FLOWER(REMEMBER SIZE 10)           --flower size
30 ASK FLOWER(REMEMBER DRAW               --to draw a flower using
              USING DRAW-FLOWER)            a LOGO procedure
END

TO DEFINE .SEED
10 ASK SOMETHING(MAKE SEED)               --seed creation
20 ASK SEED (IF RECEIVE?SEED(START)       --start message and
            THEN DO.SEED.THING:?SEED)     --appropriate procedure call
END

TO DO.SEED.THING:SEED                     --procedure for handling seeds
10 LOCAL A FLOWER                         --local flower name
20 ASK FLOWER (MAKE A.FLOWER)
30 ASK A.FLOWER (APPEAR RIGHT 90)         --turtle commands
40 ASK A.FLOWER (APPEAR FORWARD
                 (*100 (RANDOM)))
50 ASK A.FLOWER (APPEAR LEFT 90)
60 ASK A.FLOWER (PLAN:SHOW IN 10 TICKS)--the message ASK A.FLOWER
                                          (SHOW) will occur after
                                          A.FLOWER has received
                                          10 ticks
70 REPEAT 15 (ASK A.FLOWER                --ASK A.FLOWER (GROW 10)
              (PLAN:GROW 10 AFTER 2         will be called 15 times,
               MORE TICKS))                 2 ticks after the last
80 ASK SEED (PLAN:ASK (SEED (MAKE))       --creation of another seed that
    (START) AT THAT TIME)                   must start at the same time
                                            as the last thing scheduled
90 ASK A.FLOWER (PLAN:HIDE
        AFTER 60 MORE TICKS)
END
```

Figure 4.5: A DIRECTOR script with comments. The situation to be animated is a garden in which seeds are born, wait, grow into flowers, create new seeds, continue growing and finally die.

messages they can handle are specified. Most actors are capable of responding to messages which would be understood by a LOGO turtle [*Papert, 1970*], but it is also possible to specify other messages that they can understand. There are two pre-defined actors called `SOMETHING` and `OBJECT` which initially exist in order to allow the user to ask for other actors to be created; the difference between the two is that any actors created as a result of a creation message to `OBJECT` are capable of behaving as a LOGO turtle, whereas actors created by `SOMETHING` are not (in DIRECTOR, when actors are created they inherit all of the message response properties of their creator).

Figure 4.5 presents an example script taken from Kahn's paper.

4.8.2 ASAS: The Actor/Scriptor Animation System

ASAS [*Reynolds, 1982*] is an extensible, procedural, LISP-like programming language for animation and graphics developed by Craig Reynolds at M.I.T; it was later incorporated into the Digital Scene Simulation System at Information International Inc. (Triple I). It allows objects, virtual cameras and light sources to be defined, and provides operators, such as *forward, backward, left, right, zoom-in, zoom-out, cw* (rotate clockwise), and so on. These operators effectively provide a three-dimensional extension to LOGO's turtle graphics with which to manipulate objects defined elsewhere within the script/program. This is a very powerful system which has been used to great effect in the making of films such as *TRON*; but, like most actor and script systems, it is aimed at the computer programmer rather than the artist. Assuming that the artist is not a competent programmer, his artistic control must always be one step removed from the tool since he must work with a programmer who interprets his ideas into code.

4.8.3 Work with actors and scripts at the Université de Montréal

Over the last few years a large amount of research into the actor/script environment has been undertaken at the University of Montreal, under the direction of the husband and wife team Daniel Thalmann and Nadia Magnenat-Thalmann. They first received widespread recognition in 1982 when their 13 minute film *Dream Flight* was shown at the Siggraph '82 Film Show. The main novel feature of *Dream Flight* was that it was a three-dimensional, computer-generated film which told a fictional story. Before that time nearly all films had either had no story-line at all, merely demonstrating the technical achievement of producing moving pictures by use of a computer; or had been made to demonstrate some technical or scientific point. *Dream Flight* took 14 months to produce—by writing a script (program) in a language called MIRA-3D [*Magnenat-Thalmann and Thalmann, 1983*] which is a graphical extension to Pascal.

CINEMIRA

After the experience of MIRA-3D, the next step was to produce a language which was targeted specifically at three-dimensional animation, rather than

just three-dimensional graphics as MIRA-3D had been: this language was called CINEMIRA [*Thalmann and Magnenat-Thalmann, 1984*]. CINEMIRA introduced the idea of *animated types*—for example, an *'animated INTEGER'* is an integer whose value depends on time (it has a law built into its type definition to define how the value should change with time). Animation can be developed by using animated variables to control the motion of actors.

As well as animated INTEGERs it is also possible to animate REALs and VECTORs. Consider, for example, the definition of a vector which is initially at point (12,45,13) and starts moving at time 15 at a constant velocity of (4,5,8), finally coming to rest at time 25 (times measured in frames and positions measured in arbitrary but consistent units). This motion is expressed in CINEMIRA as follows:-

```
type MYVEC = animated VECTOR;
             val <<12,45,13>> .. UNLIMITED;
             time 15..25;
             law <<12,45,13>> + (<<4,5,8>> * (CLOCK - 15))
             end;
var Fred: MYVEC;
```

where:

`UNLIMITED` avoids the need to specify an unnecessary value.
`CLOCK` refers to the current time.

Type definitions are allowed to be more general purpose than this. It is possible to define a number of formal parameters after the words `animated VECTOR` which may then be given actual values by use of an `init` statement, as follows:

```
type VEC = animated VECTOR (StartTime, StopTime: INTEGER;
                                Position, Velocity: VECTOR);
           val Position .. UNLIMITED;
           time StartTime .. StopTime;
           law Position + (Velocity * (CLOCK - StartTime))
           end;
var Jim: VEC;
init Jim (15,25,<<12,45,13>>,<<4,5,8>>);
```

Variable Jim may then be used to control the motion of an animated object.

There are two special abstract animated data types in CINEMIRA—the *actor* and *camera* types. The definition of *actor* types includes a definition of the visual appearance of the actor (written in MIRA-3D) as well as the definition of the actor's motion (perhaps making use of locally declared variables and types). *Camera* types include definitions of a camera's position, orientation and centre of interest. When the script is compiled, its execution causes pictures to appear on the screen as if the actors were viewed from the given camera positions. This is a very powerful system, and a wide range of effects are possible but it has the disadvantage of being difficult for non-computer-scientists to use.

MIRANIM

In an attempt to make modelled animation accessible to the non-computer-expert artist, the Thalmanns developed MIRANIM [*Magnenat-Thalmann et al., 1985*], an extensible, director-oriented three-dimensional animation system. This system has three distinct parts:-

1. **BODY-BUILDING** is a three-dimensional object modelling and image synthesis system which allows a variety of different types of objects to be created out of

   (a) elementary objects such as polygons, spheres, and regular polyhedra

   (b) ruled surfaces including cylinders, cones and surfaces of revolution

   (c) parametric surfaces and patches specified in any of a variety of ways, including Coons' and Bézier's methods.

   Each component of an image has a name and the artist issues commands to position these components in three-dimensional space and in relation to each other. For example:

   ```
   PUT VASE ABOVE TABLE
   ALIGN VASE TABLE XZ
   ```

   positions a `VASE` above an already positioned `TABLE` and aligns the local coordinate systems of the two objects in both the x and z directions—the y coordinate will naturally be different since the vase is above the table.

   BODY-BUILDING also provides commands to perform rendering using either constant shading or smooth shading by either the Gouraud [*Gouraud, 1971*] or Phong [*Phong, 1975*] algorithms.

2. **ANIMEDIT** is an artist-oriented three-dimensional animation system which allows an artist to take on the role of director and build up a script by interactively issuing commands to actors. It also allows the interactive control of cameras and lighting.

   The animator issues commands from a list of known commands; for example, '`MOVE Car MyVec`' may mean that the actor called `Car` should move according to the definition contained in the animated variable `MyVec`. Animated variables are declared in much the same way as in CINEMIRA, described above. A major advantage of the system is that the list of available commands is not fixed; it is possible for a programmer to add a new command by writing a procedure in a language developed for the purpose, called CINEMIRA-2.

3. **The CINEMIRA-2 sub-language** is a simplified derivative of CINEMIRA which allows new commands to be programmed and linked into the ANIMEDIT code. For example, if the animator wishes to use a command `WHEELIE` which causes an actor such as a motor bike to pick its front wheel off the ground and go along on only one wheel, a programmer

could write a CINEMIRA-2 procedure to perform this action then link it into the ANIMEDIT code—at which point it would become accessible to the animator, together with its arguments such as speed of travel and height and duration of wheel lift. This extension process is not particularly quick since it needs the original CINEMIRA-2 code to be converted to Pascal by a pre-processor, then compiled and linked with the existing ANIMEDIT code; it is however relatively straightforward since it is only necessary to write a procedure and link it in for it to be directly available within ANIMEDIT.

I believe that the main disadvantage of this approach is that the artist is still required to be aware of such things as animated variables, and produce a textual description of the motion which is more like a programming language than a conventional script.

4.9 Commercial Computer Animation Production Houses

The majority of animation systems mentioned so far are the fruits of university research; it is, however, naturally the case that the majority of computer-animated sequences seen in films or on television are produced by commercial production houses. Some of the software used to produce this animation has grown out of academic research but much of it has been developed within the companies concerned. Some of the most impressive animated sequences yet seen have been produced by these production houses—at least partially due to the huge budgets available to some of them.

The two feature films with the greatest use of computer animation to date have been *Tron* and *The Last StarFighter*. Other films have also included computer generated sequences—*Futureworld* was an early example, and more recently *Star Trek II, The Wrath of Kahn* and *Star Wars VI, Return of the Jedi* have included impressive computer-animated sequences.

Tron included computer-generated sequences by MAGI (Mathematical Applications Group Inc.), Robert Abel and Associates, Information International Inc. (who were also responsible for a 40 second sequence in *Futureworld*) and Digital Effects. Digital Productions was largely responsible for the sequences in *The Last StarFighter*; the sequences for *Star Trek II* and *Return of the Jedi* were produced at Lucasfilm.

4.9.1 Work at Lucasfilm

Very active research is currently being undertaken at Lucasfilm into a wide range of areas of computer graphics and animation including such things as *fractal landscapes* [*Fournier et al., 1982*], *particle systems* [*Reeves, 1983*], *compositing* [*Porter & Duff, 1984*] and *motion blur* [*Cook et al., 1984*]. Some of these techniques have already been used in the production of feature films and presumably many more fruits of their research will be used later. The

research team has the great advantage of access to very large amounts of money (generated as profits from Lucas feature films) to aid them in their work.

The most technically advanced sequence produced at Lucasfilm so far is a Disney-style cartoon entitled *André and Wally B.* The foreground animation was developed by a Disney animator using Lucasfilm's three-dimensional keyframe animation system. Great use is made of motion-blur—the result of which is a **very** smooth animated sequence which has fooled at least one conventional animator into thinking that the film was being projected at a considerably faster rate than 24 frames per second. However such sophistication does not come cheaply; the two minutes of foreground animation took a month to produce on a Cray super-computer. The action takes place in a forest containing thousands of individual trees represented using particle systems. Rendering of the backgrounds took *three months* on a network of *thirty* VAX 11/750s. Although technically superb it is obvious that until cpu power becomes much cheaper, or rendering algorithms improve dramatically similar sequences are unlikely to become common.

4.10 What Next?

With the ever increasing demand for new and better animation systems, fuelled mainly by the advertising and feature film industries, the future of this rapidly changing subject looks very bright indeed.

I believe that there is still a significant way to go before the subject reaches maturity; this can only bode well for new and exciting developments in all fields of animation research. The next three chapters of this thesis present my own work in the field of motion development.

Chapter 5

Using Simulation in the Animation Process

Much of the emphasis of my work so far described has been on two-dimensional animation, particularly keyframe animation. It has been shown that this type of animation is not entirely suited to computer assistance. Three-dimensional animation is, perhaps, a more promising area for exploration.

To a first approximation three-dimensional animation may be considered to be a three-stage process;-

- get the motion right;
- decide on suitable "camera angles";
- add light sources, and render the finished product.

The remaining chapters of this thesis concern themselves mainly with the first of these stages. Some attention is paid to the second item, but details of rendering are not considered in any depth.

5.1 Specifying Three-dimensional Motion

As mentioned in the previous chapter, there are several possible methods of specifying three-dimensional motion. The two which are most commonly used are:-

1. a three-dimensional version of the *keyframe* technique, and
2. the *actor and script* approach.

5.1.1 Three-dimensional keyframing

The main problem with two-dimensional keyframing is that the only way of describing the motion is by means of flat images. This causes too much information to be lost. A three-dimensional approach attempts to overcome this problem by giving the animation software more information about the objects

to be animated: in particular, information about physical appearance and articulation. These systems generally deal with the animation of three-dimensional objects composed of rigid sections connected together at *joints*. Instead of interpolating vertex positions, as in two-dimensional keyframe animation, it is usually *joint angles* which are interpolated. Thus to animate the action of a human figure bending his arm it would merely be necessary to specify the initial elbow position and the final elbow position then interpolate the angle of the elbow—the lengths of the arm sections would remain constant, as expected. There is, however, nothing inherent in the method to stop the animator specifying impossible motion, such as the elbow bending sideways or backwards; and nothing to prevent objects passing 'through' each other.

5.1.2 The Actor and Script approach

In this method a textual description of the objects to be animated and their associated motion is produced. The description language is normally a development of a conventional computer programming language which has been tailored to provide the types of commands which might make sense to a traditional director. Thus commands will be provided which enable the animator to specify what objects are composed of, what their initial positions are, and how they should move. There will also be commands to define the position of the virtual camera observing the scene. This '*script*' is then compiled, or interpreted, and an animated sequence is produced. If the sequence does not turn out as desired, the text is edited and re-compiled. This process continues until a satisfactory outcome is achieved.

5.2 Using Simulation to Define Motion

Drawing on the experience of *Taking a Line for a Walk*, I decided to attempt a somewhat different solution to the problem of specifying how objects should move. One particular aim was to satisfy the request which Lesley Keen had repeatedly made while using *KAS*, namely that the animator should not always be in explicit control of the animation but should sometimes be surprised by the motion produced. By giving the software not only a description of what an object looks like, but also a set of rules to define how it should behave under the majority of circumstances, it should be possible to set up an initial situation and then leave the animation to proceed with only minimal operator intervention to guide it towards the desired conclusion. In these circumstances it is the computer (with its simulation rules) which is making many of the decisions, particularly as regards the interaction between objects.

Let us consider the application of this approach in the animation of the following example.

5.2.1 A street-scene example

The situation to be animated consists of a street scene, complete with a road, pavement, shops, people, dogs, cars and bicycles. Each of the objects in the scene has a set of rules associated with it which determine how it should behave (in isolation); there are also rules which define certain interactions between objects.

For the human beings there would be rules to define how a person walked, for example. This would include a large amount of detail about the mechanics of walking. Length of stride, amount of arm swing, forward velocity and other such parameters would be under animator control but the animator would not need to worry about the actual process of putting one foot in front of the other—this would be dealt with by the simulation rules.

Similarly dogs, cars and bicycles would have rules defining their motion. For a bicycle, for example, it would be arranged that the animator need specify only such things as velocity, acceleration and gear ratio; the simulation rules would ensure that the wheels and pedals rotated at suitable rates, and the bicycle leaned over at a realistic angle when going round corners. This would allow the animator to make high level decisions about where bicycles should move without worrying about the tedious details of pedal rotation etc.

The next step up in the hierarchy of simulation rules would be composed of rules concerning the system as a whole; these would deal with the interaction between objects. There may be rules, for example, which state that people should always attempt to remain on the pavement (unless explicitly instructed otherwise), and that they should try to avoid bumping into lamp posts and other people. Thus there must be rules to cause messages to be passed to the walking human figures to inform them that they are approaching another object which should be avoided. Similarly bicycles should probably attempt to remain close to the kerb and should not pull out at road junctions unless there are no cars coming.

Given an initial situation, the animation can proceed completely automatically according to the rules governing the simulation until either the animation reaches its logical conclusion (or a situation not allowed for in the rules), or the animator intervenes to alter the course of the action. It is by the selective breaking of rules that the animator is able to steer the animation towards its desired conclusion. In this way, the animator is responsible for the creative input which guides the course of the animation, but is relieved of much of the tedious detailed motion specification.

5.2.2 The production of a new animated sequence

To define the motion for a new animated sequence the animator must begin by defining the objects which are to be animated; (in what follows these animation objects will sometimes be referred to as *actors*, due to their similarity to Hewitt's definition of actors in [*Hewitt, 1971*]; the term is in no way intended

to imply any information as to whether the object is animate or inanimate). Object definition may be performed in a variety of ways. Perhaps the most general purpose method is by use of a specialised description language, as is often used with the actor and script technique. It may be the case that some of the rules of interaction are also specified at the same time.

Using these languages it is normally possible to set up a hierarchy of actors within an object; for example an actor such as a car may have four sub-actors representing its four wheels. The simulation rules to be built on top af this "actor" definition will then consist of definitions of responses by the actors and sub-actors to various input stimuli. The simulation may either be very simple, consisting perhaps only of rules to define speed of wheel rotation in terms of the car's forward velocity; or they may be very complex—using sophisticated artificial intelligence techniques to allow decisions to be made by the car about such things as choice of lane when approaching red traffic lights in an attempt to minimise waiting time, for example.

The definition of rules may be performed in a variety of ways. Ideally objects should be interactively *"taught"* how to behave; but to be realistic, it is likely that much of the definition will be performed via the keyboard, and may even take the form of procedures in a computer program.

Having specified the animation objects and their rules of motion, the next question is how does the animator view and interact with the system? Remember that it is only the motion which is being developed at this time; *"camera angles"* are not decided upon until the next stage in the three-stage process (mentioned earlier) of developing the final animated sequence. At this initial stage the animator needs to specify only how and where objects should move, not what the final animated sequence should look like. The animation system should therefore be aware of the physical appearance of each object not only in the final sequence but also for this, the motion definition, stage. This second representation is likely to be much cruder than the final image since the most important requirement at this stage is that it can be drawn quickly so that the animator can get a feel for the dynamics of the motion.

It must then be decided from what position or direction the motion will be observed during motion development. It may be desirable to have several different angles of view in order to allow the animator to check various components of the motion independently. These various views may either be presented simultaneously or separately. If several views are used, it is likely that different representations of the object will be used for different viewing directions. Only a simple stylised image is needed when planning the broad interaction between objects, but a more detailed representation would be needed to determine whether the walking style of a human figure (length of stride, amount of arm swing) is as required.

It is often the case that one of the more useful viewing directions is from directly above, since many situations to be animated consist of objects moving around on a (relatively) flat surface. In the "street scene" example above,

although the main "actors" (the people, dogs, cars and bicycles) are three-dimensional, much of their movement is parallel to the surface of the road; it is therefore possible to gain a good appreciation of this aspect of their motion and the relative positions of objects by viewing from above. This allows simple two-dimensional representations of the "actors" to be used. A car, for example, may be represented by a coloured rectangle—perhaps with an arrow on the front to represent the car's velocity.

The use of a *"bird's eye view"* and simple two-dimensional representation of the animated objects without any attempt at realism should enable the motion to be displayed in real time. Animation is essentially the effect produced by displaying a series of still images (frames) in quick succession; by "real time" I mean 24 frames per second (for film use) or 25 f.p.s. (for video). If the time taken to perform the simulation means that real time display is not possible, the animator should be able, at any time, to review the frames produced at full speed.

Even if frames can be generated in real time, it is still essential that previously generated frames can be re-generated at will, not least because all of the relevant information from each frame will be required at the camera definition and rendering stages later. Another reason why it is necessary to be able to return to an earlier frame is that it is in this way that the animator is able to influence the way the motion proceeds (see below). Usually the most straightforward means of allowing these earlier frames to be regenerated is by storing all of the information required for each frame's regeneration in the computer's memory (or on disc if there is insufficient memory space). It is likely that the status of each object in each frame can be described in a compact way in terms of its current position in space, velocity, state (running, walking, jumping), leg positions (assuming it has legs) and so on. This compact representation should enable a relatively large number of previous frames to be kept in main memory in order to aid rapid replay.

If at any time the animation develops in an undesirable way, the animator should be able to immediately halt the production of more frames and skip backwards through the frames already produced until an acceptable previous position is reached. From this position the animator may alter such factors as necessary in order to guide the system towards the required goal. This guidance may take the form of the animator explicitly controlling an object's movements for a few frames; or it may simply be necessary to change some parameter, such as its acceleration, then leave the system to continue with the simulation unaided. A third possibility is for the animator to issue some pre-defined command which the simulation model understands in order to change a large number of parameters at once (or over a period of time)—for example, issuing the command "**run**" (and specifying a speed) to a stationary human figure may cause that figure to go through realistic changes of stance and acceleration in order to reach the required terminal running velocity. In a somewhat more sophisticated system it may be possible to issue commands (pass messages)

such as *"go to the kitchen"*, and leave the system to use artificial intelligence techniques to determine that it is necessary for the actor receiving the message to first of all stand up before locating and using the relevant door.

As the animation recommences from a previous position all of the unacceptable future frames become invalid and are discarded (perhaps to a checkpoint file). If the animator later decides that the system's original attempt at motion was preferable to the amended one, these discarded frames may either be recovered from a checkpoint file (if a suitable one exists), or (assuming the simulation is deterministic) by returning to the frame before that at which the first change was made and allowing the simulation to proceed from that point free from animator intervention. The animator may go through this guidance process many times during the production of the final sequence of movements, each time hopefully refining the motion, until eventually a satisfactory sequence is produced.

This method of working is less constraining than the actor and script method since the animator is able to interact with the situation graphically and he receives immediate visual feedback to his commands without the need to wait for a "script" to be edited and re-compiled. The method therefore encourages a more adventurous, experimental approach since the animator can quickly and easily move backwards and forwards through the generated frames discarding anything he doesn't like and keeping anything he does. It is the animator who is in overall artistic control of the sequence but some of the ideas for motion may be suggested by the animation system, rather than being explicitly specified by the animator before the work of producing the animated sequence commences, thus partially satisfying Lesley Keen's request for a less predictable system.

5.3 Potential Advantages and Disadvantages of the Use of Simulation in the Animation Process

A big advantage of the simulation/animation approach is that it is interactive; the animator continuously views the state of the system on the computer monitor and interacts with it. This interaction may be in any of several ways. Ideally it should **not** be via a textual interface; the animator should provide input by means of buttons, dials, mice, graphics tablets and any other input devices at his disposal, only occasionally having to resort to use of a keyboard for input of data or commands. I believe that it is important to give the artist as much freedom of expression as possible, using as many input methods as possible since, in my experience, artists feel restricted if forced to interact with a graphical situation in a non-graphical way. It is likely that more unusual means of graphical input, such as foot pedals, will be found to be useful in certain circumstances. The aim is to allow the animator to use these input

devices to alter the course of the animation by changing such things as the sizes of objects, or their speeds and directions of motion. In some ways this is similar to a trainee pilot's interaction with a flight simulator, except that in the animation situation, the interaction devices are being used to influence how objects in the environment move, as opposed to how an aeroplane moves through the environment. Another analogy is with the way that a musician would play a musical instrument such as an electronic organ—keys and foot pedals are used to affect the sound being produced, just as the animator uses the input devices to change the pictures being generated. On an organ there are also knobs and switches which initiate complex sequences of notes in the same way as the animator should be able to issue commands (perhaps by means of a menu and mouse) to cause objects to begin complex motions, such as walking.

Let us now compare and contrast this simulation/animation technique with the keyframing and scripted approaches.

Three-dimensional keyframing, like simulation/animation is a very interactive technique but an advantage of simulation/animation is that it allows much easier control of complex motion (such as walking) thus rendering unnecessary much of the interaction needed to develop an animated sequence by keyframing. The amount of animator control provided by a keyframing system is both a blessing and a curse: with such a system it should be possible for the animator to define motion precisely but the problem is that there may be too many parameters for a human being to control efficiently.

The *actor and script* approach, on the other hand, is essentially non interactive; the input to the system is via a textual script which must be interpreted or compiled to produce the animated sequence. As with simulation/animation, it is possible to build high level commands into the actor and script system, but it is not easy to write a script which deals with interaction between objects since it is difficult for the animator to be aware of all clashes and inter-relationships between objects when writing the script. By way of illustration, Rob Cook from Lucasfilm often recounts the amusing story that during the making of the feature film *Star Trek II; The Wrath of Khan* there was a sequence (produced using scripting techniques) where a landing shuttle was supposed to fly through a ravine in a fractal-generated mountain range; in fact as the spacecraft approached the mountains it became obvious that it was going to miss the ravine and hit a mountain. This problem was overcome by manually altering the shape of the mountain, as the ship approached, in order to produce a suitable ravine for it to pass through; however this would not have been necessary if the spacecraft had been able to adjust its path as necessary in order to avoid obstructions.

One drawback when using simulation to produce animated sequences is that it relies on the provision not only of a set of objects to be animated but also rules to guide the motion. The specification of animation objects is a significant problem in keyframing and scripting too, so there is very little difference in that respect; but setting up simulation rules may be a time-consuming and complex

task, significantly affecting the time taken to set up a situation for use in simulation/animation. Keyframing makes no use of simulation rules to guide the motion; all movement is specified explicitly—so obviously no set-up time is needed. The actor and script approach may allow 'rules' to be set up to describe motion, and interactions between objects, but these rules are likely to be less comprehensive than those used for simulation/animation—with actors and scripts the animator is generally expected to provide more explicit control of motion; setting up these rules is therefore likely to be a less arduous task than for simulation/animation.

As mentioned earlier, the simulation rules should ideally be specified graphically—the animator should teach an object how to walk, for example—but in practice it is likely that some of the rules will have to be entered textually, perhaps even as procedures in a computer program. However, once specified, it is possible that objects to be animated and their simulation rules will be of use for more than a single animated film. Indeed it may be possible to build up whole libraries of such objects and simulation rules which can be called upon as necessary during the production of later animated sequences.

In the next two chapters experiments to test the idea of incorporating simulation into an animation system are described, with the object of assessing the usefulness of the technique.

Chapter 6

Snooker—a simple application of simulation to the animation process

Once it had been decided to implement an animation system based on the idea of the animator interactively steering the animation towards a satisfactory conclusion rather than explicitly scripting or keyframing the action, the next problem was to decide on a suitable experiment to test the idea. It was felt to be impractical to attempt a full, general-purpose, three-dimensional animation system immediately—but the subject for the experiment (and its subsequent implementation) had to satisfy a range of criteria:-

- It had to be a situation which would be difficult to animate by conventional means (both by hand, and with computer assistance).
- It also had to be sufficiently well-defined and understood to not cause too many implementation problems.
- The user interface of the final implementation should be sufficiently good for the method to be judged on its merits, without allowance having to be made for a poor interface restraining the artist's creative talents.
- The images to be worked with should be sufficiently simple to be generated in less than 1/25 second on the graphics hardware available (the Rainbow Display, see Appendix I), thus allowing the dynamics of the motion to be judged during the development process.
- It need not be general purpose since an initial experiment need only exercise the idea in a specific situation; generality can come later if it appears that the idea will be useful in the general case.

One situation which presents itself to mind, given the above criteria, is the motion of balls on a billiard table—more particularly the game of snooker.

Balls on a billiard table comprise a sufficiently simple universe for a first experiment. It is a closed system and the rules of motion are reasonably well understood and easy to implement. It is also a simple situation to draw—if

viewed from above, the table remains static between frames, and only the balls move. The balls need only be represented as circles, so the amount of drawing effort necessary to produce each frame is minimal.

It was therefore decided to produce a piece of software to aid an animator in the animation of a game of snooker. The program was intended to take over as much of the animation work as possible, but also provide the user with powerful tools to allow the game to be tightly controlled if so desired.

6.1 An Introduction to the Game of Snooker

Below is a brief description of the rules of the game, for those who are not familiar with it, followed by an overview of the laws of motion which need to be modelled.

6.1.1 The rules of the game

Snooker is a game for two players which is played using twenty two balls, $2\frac{1}{16}$ inches in diameter, on a green baize table whose playing surface is twelve feet by six feet one and a half inches. Surrounding this playing surface are *cushions* consisting of angled pieces of rubber, covered in baize, which serve as barriers off which the balls can rebound. There are gaps in these cushions at the four corners and half way along the long sides; these gaps allow the balls to leave the playing surface into *pockets*—the balls are then said to have been *pocketted* or *potted.*

The twenty two balls consist of the following colours:-

- 1 white ball (the *cueball*);
- 15 red balls (worth 1 point each);
- 1 yellow ball (worth 2 points);
- 1 green ball (worth 3 points);
- 1 brown ball (worth 4 points);
- 1 blue ball (worth 5 points);
- 1 pink ball (worth 6 points);
- 1 black ball (worth 7 points);

These balls are set up at the beginning of the game as in figure 6.1.

At the start of the game, the fifteen reds are arranged in a triangular pattern; the six coloured balls are placed on their *spots*, and the cueball is placed anywhere in the "D".

The players take it in turns to hit the white ball with a thin, tapered length of wood, called a *cue.* The object of the game is to get the balls into the pockets. This must be done in a specified order: a player begins his turn by attempting to pot a red ball; if he is successful in doing this he then attempts

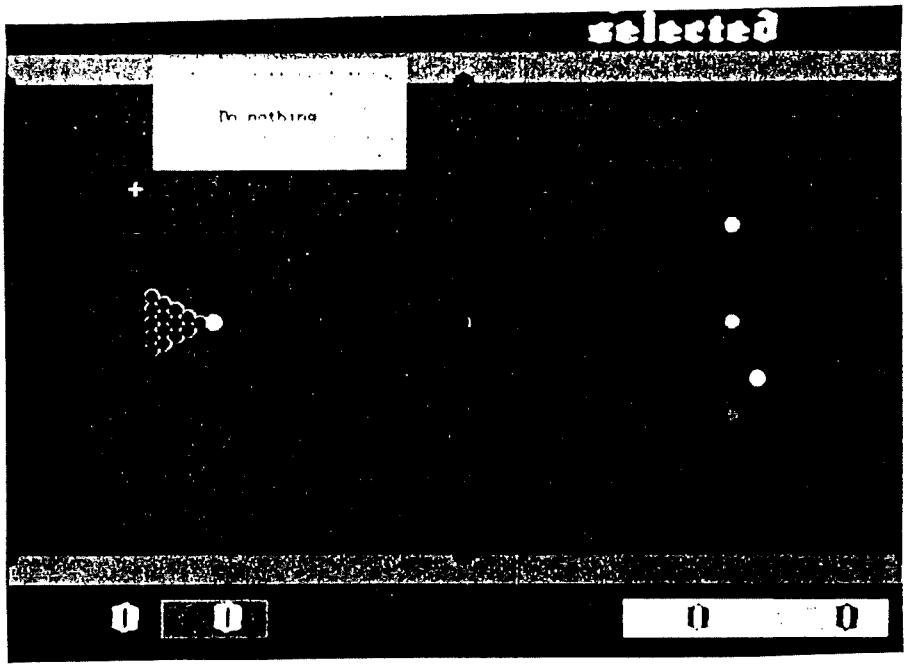


Figure 6.1: The situation ready for the start of a game of snooker.

to pot a non-red ball (a *coloured* ball). Red balls always remain in the pockets after being potted but coloured balls are replaced in their initial positions (*"on their spots"*) if there are still any red balls remaining on the table. The player continues to alternately pot red and coloured balls until he either fails to pot a ball or he hits or pots a ball of the wrong type. At this point the other player's turn begins.

Any balls which are potted add points to the player's score (according to the table of values given above). The number of points gained during a single turn is referred to as the player's *break*; (this is slightly confusing since the very first shot of the game is also referred to as "the break"). If a player makes a *foul shot* (hits or pots an illegal ball) penalty points are added to the other player's score. Once all of the red balls have been potted, the coloured balls must be potted in the order listed earlier (starting with yellow and ending with black). The only purpose of the cueball is to be struck by the cue and used to hit the other balls in an attempt to make them go into the pockets; it is a foul stroke if the white ball itself is ever potted. When all of the balls have been potted the game ends; the player with the higher score is the winner.

During the course of the game, if a player considers that he cannot realistically attempt to pot a ball he may decide to play a *safety shot*. This means that he will play a legal stroke but aim to leave the cueball in such a position that it makes it difficult for the opposing player to pot a ball. Ideally a player may be skillful enough to cause the cueball to come to rest behind another ball, so that it is difficult for his opponent to even hit a legal ball, much less pot one—this situation is referred to as a *snooker*.

6.1.2 Modelling the physics of the game of snooker

The equations of motion of the balls on a real snooker table are actually more complicated than might at first be imagined.

There are many factors which affect the motion of the balls; these include:-

- the friction of the table. This is complicated by the fact that the baize has a nap, i.e. the fibres all face up the table (away from the "D") which means that the frictional constant is different in the two directions; when hit diagonally across the table at very low speed, the effect of the nap is such that the ball does not even run in a perfectly straight line but along a slightly curved path.

- the quality of the cushions obviously affects the amount of energy that is lost on rebound and hence the rebound angle. The best quality cushions are made from many thin layers of rubber built up into the shape of a truncated triangular prism. Account must be taken of the fact that the cushions near the pockets are curved.

- collisions between balls cause changes in speed and direction. It is possible to work out the new velocities after collision using equations based on conservation of energy and momentum. Unfortunately it is not strictly true that energy is conserved, since some energy is lost by the balls during a collision (in making the sound of collision, for example); this may or may not be considered to be important.

- the way that the ball is struck by the cue is very important. A skillful player is able to impart a variety of types of spin to the cueball by hitting different parts of the ball with his cue. Hitting the cueball near the top will produce top spin, which encourages the ball to roll forward more than would be the case without spin; it also encourages the ball to continue moving in the same direction when colliding with other balls, thus reducing the angle of deflection. Back spin, produced by striking the lower half of the ball, has exactly the opposite effect. Side spin, produced by aiming the cue to the left or right of centre, may be used to affect rebound angles from cushions and at collision with other balls. In extreme cases it can even be used to cause the ball to swerve around an obstruction when getting out of a snooker.

Let us begin by considering the motion of a ball across the table, without considering collisions with cushions or other balls.

Sliding and rolling

Assuming that the ball slides across the surface of the table, and the friction between the ball and the table's surface remains constant, its velocity at any moment would be given by the equation:

$$\vec{V} = \frac{\vec{k}}{t + t_0}$$

where

t_0 = time taken for speed of ball to halve—a measure of the friction of the playing surface.
$\frac{\vec{k}}{t_0}$ = initial velocity
t = time
$\vec{V}$ = velocity

Unfortunately it is not true that the ball simply slides across the surface of the table. Assuming that the player has not imparted any spin to the ball, observations show that the ball does initially slide, but friction quickly causes the ball to start rolling. The graph of speed against time (figure 6.2) is therefore very different from the theoretical case.

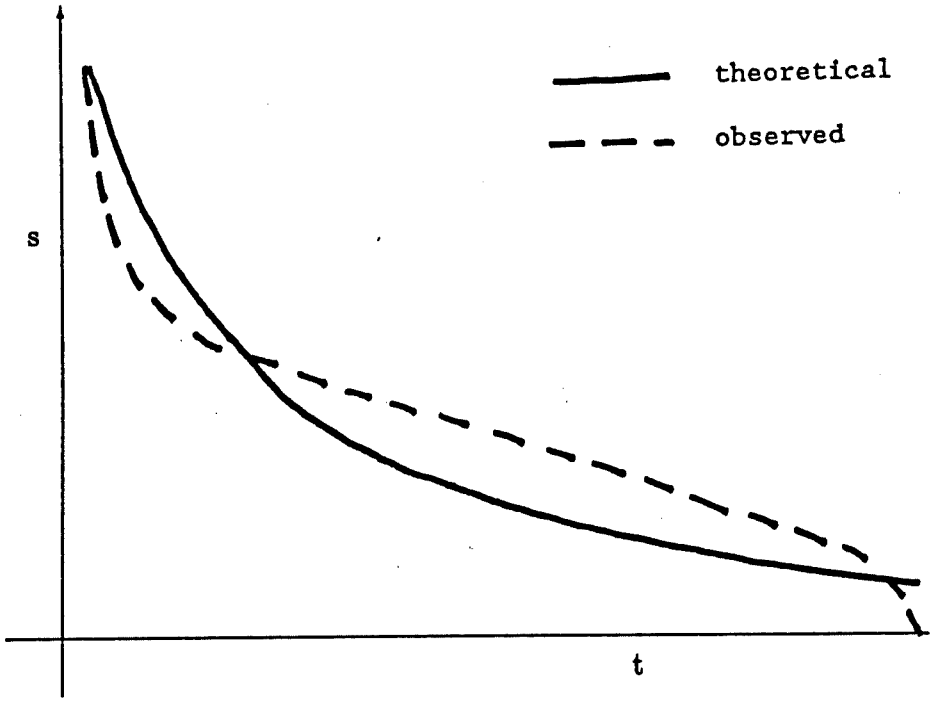


Figure 6.2: Graph of ball speed against time in both the theoretical and observed cases.

The shape of the very early part of the curve is similar in both cases since the ball is in fact sliding; but when the ball starts to roll, the change of speed is more linear. In the observed case there is also a final rapid down-turn in velocity just before the ball comes to rest.

As an approximation it was decided that the simulation would use a digitised version of the curve gained by observation, rather than attempting to find a more accurate mathematical model for the ball's motion.

Collision with cushions

Let us next consider the situation when a ball hits a cushion. The cushion exerts a force on the ball perpendicular to itself, so the component of the balls

velocity parallel to the cushion remains unaltered and (assuming the collision is perfectly elastic) the component perpendicular to the cushion changes sign as illustrated in figure 6.3.

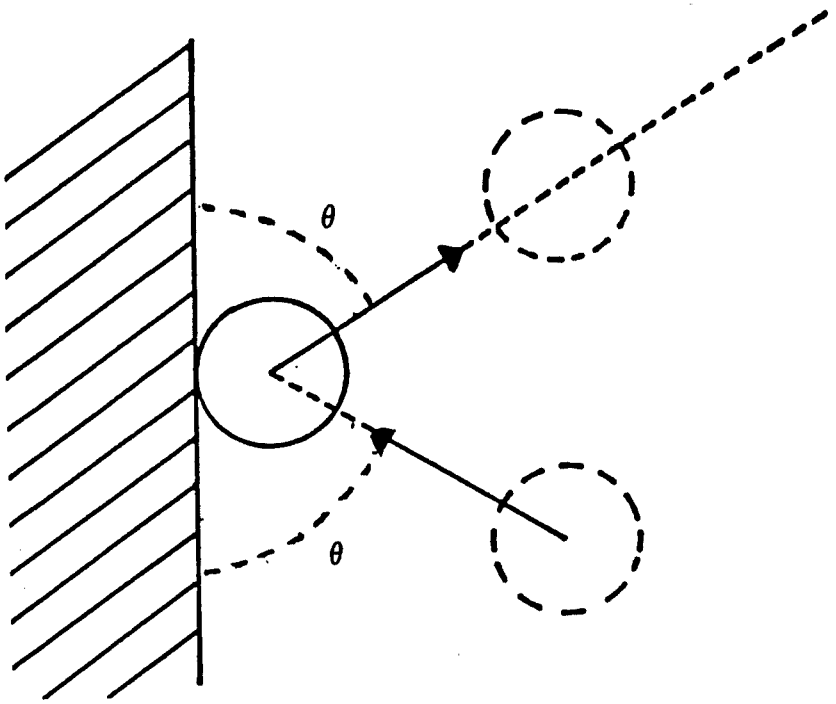


Figure 6.3: The collision of a ball with a cushion.

Since the collision is not perfectly elastic (some energy is used to deform the cushion, for example) the rebound angle may be expected to be slightly different from the incident angle.

Collisions between balls

An important part of the motion to be considered is when balls collide with other balls. For a perfectly elastic collision (without spin) the laws of conservation of energy and linear momentum may be applied. Thus

$$\tfrac{1}{2}mV_1^2 + \tfrac{1}{2}mV_2^2 = \tfrac{1}{2}mV_1'^2 + \tfrac{1}{2}mV_2'^2 \qquad \text{(i)}$$

$$m\vec{V}_1 + m\vec{V}_2 = m\vec{V}_1' + m\vec{V}_2' \qquad \text{(ii)}$$

where

m = mass of ball
$\vec{V}_1$ = velocity of ball 1 prior to collision
$\vec{V}_2$ = velocity of ball 2 prior to collision
$\vec{V}_1'$ = velocity of ball 1 after collision
$\vec{V}_2'$ = velocity of ball 2 after collision.

A vector written without an arrow over the top means the magnitude of the vector (so, for example, V_1 is the speed of ball 1).

Since the mass is the same in all cases, equations (i) and (ii) simplify to

$$V_1^2 + V_2^2 = V_1'^2 + V_2'^2 \quad \text{(iii)}$$
$$\vec{V_1} + \vec{V_2} = \vec{V_1}' + \vec{V_2}' \quad \text{(iv)}$$

Let

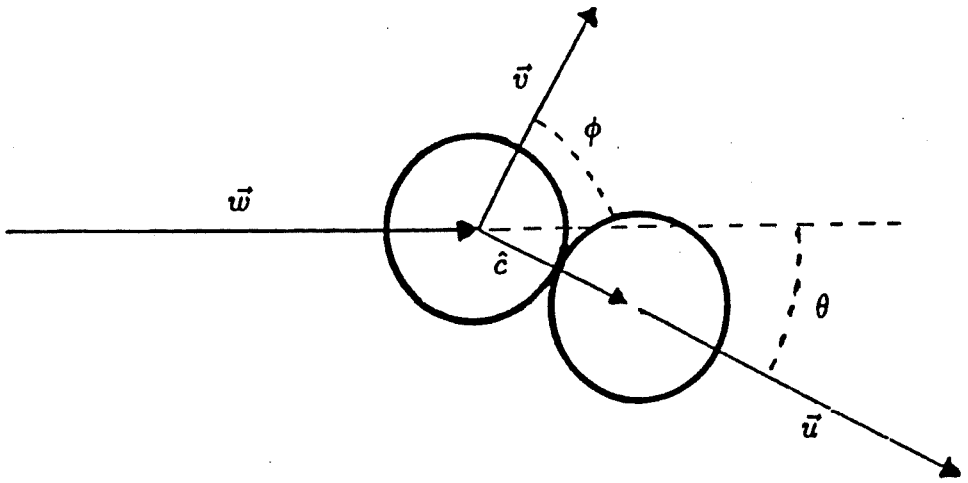
$$\begin{aligned} \vec{u} &= \vec{V_1}' - \vec{V_1} \\ \vec{v} &= \vec{V_2}' - \vec{V_1} \\ \vec{w} &= \vec{V_2} - \vec{V_1} \end{aligned} \quad \text{(v)}$$

so that all velocities are measured relative to the initial velocity of ball 1. In the calculations which follow, ball 1 may therefore be considered stationary prior to collision.

Thus equations (iii) and (iv) become

$$w^2 = u^2 + v^2 \quad \text{(vi)}$$
$$\vec{w} = \vec{u} + \vec{v} \quad \text{(vii)}$$

After collision, ball 1 will move off along the direction of the line of centres of the balls since this is the only direction along which any impulse is applied.



Let us consider velocity components in the direction of $\vec{w}$ and perpendicular to this direction. Linear momentum is conserved, so from the diagram

$$u\sin\theta + v\sin\phi = 0 \quad \text{and} \quad \text{(viii)}$$
$$u\cos\theta + v\cos\phi = w \quad \text{(ix)}$$

Rearranging (viii) and (ix) gives

$$v\cos\phi = w - u\cos\theta \quad \text{(x)}$$
$$v\sin\phi = -u\sin\theta \quad \text{(xi)}$$

Squaring (x) and (xi) and adding gives

$$v^2(\cos^2\phi + \sin^2\phi) = w^2 - 2uw\cos\theta + u^2(cos^2\theta + \sin^2\theta)$$

But $\cos^2\alpha + \sin^2\alpha = 1$ so

$$v^2 - u^2 = w^2 - 2uw\cos\theta \qquad \text{(xii)}$$

Subtracting (vi) from (xii) gives

$$\begin{aligned} 2u^2 &= 2uw\cos\theta \\ \Rightarrow u &= w\cos\theta \end{aligned} \qquad \text{(xiii)}$$

Using the well known definition of a dot product as $\vec{a}\cdot\vec{b} = ab\cos\varphi$ (where φ is the angle between the vectors $\vec{a}$ and $\vec{b}$) equation (xiii) becomes

$$u = \hat{c}\cdot\vec{w} \qquad \text{(xiv)}$$

where $\hat{c}$ is the unit vector in the direction of the line of centres (see diagram above for explanation).
We already know that ball 1 moves off in the direction of the line of centres so

$$\vec{u} = (\hat{c}\cdot\vec{w})\hat{c} \qquad \text{(xv)}$$

Substituting (xv) into (vii) gives

$$\vec{v} = \vec{w} - (\hat{c}\cdot\vec{w})\hat{c} \qquad \text{(xvi)}$$

But remember that we have been considering all velocities relative to ball 1; substituting (v) into (xv) and (xvi) gives our final result

$$\vec{V_1}' = \vec{V_1} + (\hat{c}\cdot(\vec{V_2} - \vec{V_1}))\hat{c}$$

$$\vec{V_2}' = \vec{V_2} - (\hat{c}\cdot(\vec{V_2} - \vec{V_1}))\hat{c}$$

This deals with the situation when only a pair of balls is involved. When a ball hits a ball which is in contact with other balls it is necessary to consider the effect on the touching ball or balls. This may be computed by performing the above calculation for each pair of balls, spreading out from the point of impact. After one pass through all of the pairs of balls, each ball will have a new potential velocity; however it may be the case that balls which are touching are attempting to move towards each other. It is therefore necessary to perform further passes through the pairs of balls performing the above collision calculations until no touching pair of balls is attempting to move closer together. This procedure is particularly important when the cueball makes first contact with the pack of reds at the beginning of the game, since it is important that the pack should split up realistically in this situation. Using the above method, very believable results are obtained which agree quite closely with results observed during a real game of snooker.

The effect of spin

Spin is difficult to model accurately so it was decided to use an approximation to simplify the calculation. According to this approximation, spin is considered as a separate component of the velocity—as well as having a forward velocity, the cueball is considered to have a (less significant) velocity component representing its spin. Top spin is considered as an extra forward component speeding the ball up, and back spin is a backward facing component slowing the ball down. For the sake of simplicity, side spin is ignored. The spin velocity component decays linearly with time and at a faster rate than the main sliding/rolling velocity component. Whilst moving across the table the effective velocity of the ball is taken to be the sum of the velocity components. At a collision the sliding/rolling velocity component is initially used to calculate the new velocities of the balls; then any spin component which the cueball may still possess is added to the velocity calculated for the cueball above (without any modification due to the collision); the cueball is then considered to have stopped spinning. This scheme appears to produce a believable approximation to the effects of spin at collision time. Figure 6.4 illustrates the outcome of similar collisions with and without back spin; in the case where the cueball has back spin note that the deflection angle is increased, as expected.

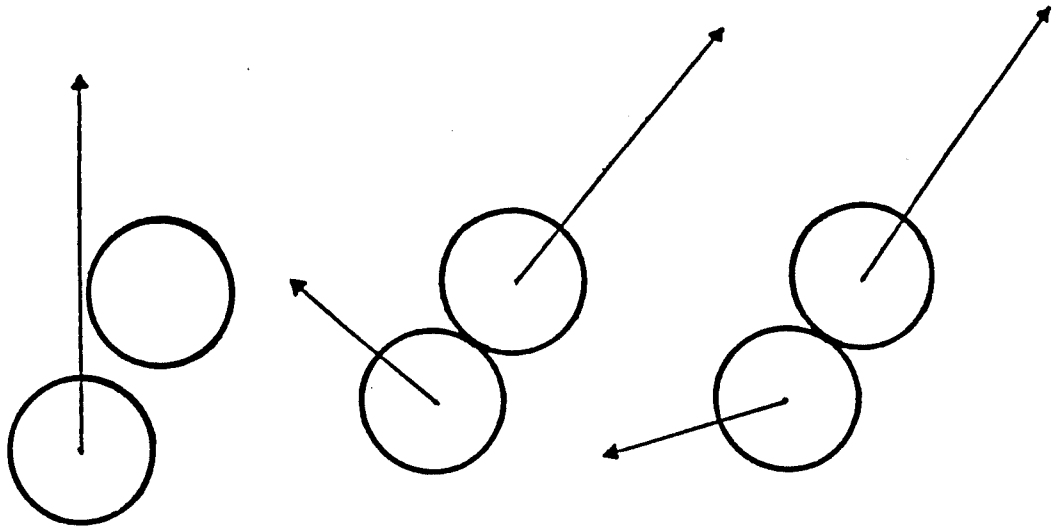


Figure 6.4: The effect of back spin during a collision between balls; (in this case a moving cueball hits another ball which is initially stationary). Figure (a) shows velocities prior to collision, (b) shows resulting velocities assuming no back spin and (c) shows velocities if the cueball had backspin.

The effect of spin in real life is considerably more complicated than is suggested above; for example, the fact that the ball is spinning will cause it to slide for a significantly greater distance than a non-spinning ball before beginning to roll; however such effects were not considered important for the purposes of this simulation.

6.2 The Motion Development Program

The software written to aid animators in the animation of a game of snooker has, built into it, the vast majority of the rules of snooker; these include both the rules of the game and the laws of motion. Thus the program is aware of such things as:-

- which player's turn it is;
- the current score;
- the current "break";
- which ball or balls it is legal for the current player to hit or pot (for example, when a player is trying to pot a coloured ball after potting a red, the cueball is initially allowed to hit any ball except a red one but once a particular coloured ball has been hit it becomes the only ball which may legally be potted);
- if, when, and where a coloured ball should be replaced on the table (*re-spotted*) after being potted—including the rules concerning where a ball should be placed if its own spot is covered by another ball.

The above knowledge is in addition to being able to produce a realistic approximation of how the balls should move under all forseeable circumstances.

Such rules, and particularly the scoring, may at first appear to turn the program into a kind of video game; however on further consideration it becomes obvious that all of the above features are of great use to an animator in planning a piece of animation. It would be disastrous, for example, if these facilities were not provided and, by mistake, the animator caused the wrong ball to be potted at some point during the game and continued the animation as though a foul shot had not occurred. By the time the mistake was noticed it may be too late to correct it.

Of course, all of this knowledge built into the system is only intended to be a guide to the animator. It may all be over-ridden at any time if the animator so desires. The most common thing to be adjusted is the motion of the balls; it often suits an animator's purpose to be told the correct motion but then "cheat" slightly by changing a ball's speed or direction of motion. Such cheating is best done during a collision, since then any lack of realism will be less noticeable to the eventual viewer of the animated sequence.

The method by which the animation progresses is for the software to keep a record of the state of the game every 1/25 second of simulated time (since the final display rate is going to be 25 frames per second) and also every time a collision occurs. The determination of when a collision occurs could have been achieved either by "*time stepping*" (moving the balls in a number of tiny steps and deciding whether a collision has occurred by checking whether any ball is

touching or overlapping another ball or a cushion), or by calculating mathematically exactly when the collision will take place. Mathematical calculation was chosen since it was felt that a more accurate answer could be obtained due to performing the collision calculations precisely at the point of collision as opposed to allowing balls to interpenetrate; it also avoided the problem of determining a suitable time between time steps; too small a time step would slow the simulation down; too large a step may allow balls to pass through each other due to the moment of collision being missed.

The calculation of when a ball will hit a cushion is quite straightforward: an initial check can be made as to whether the ball will cross a cushion during the next 1/25 second simply by using the ball's velocity to determine its expected position 1/25 second later. If this is outside the playing surface of the table it is a simple matter of clipping the balls path to the edge of the table (taking into account the fact that it is the edge of the ball which should hit the cushion, not its centre) to determine the time of collision.

Thus, considering only the component of the velocity which is taking the ball towards the cushion:

$$(c - r) = b + \lambda V \;\Rightarrow\; \lambda = \frac{(c - r - b)}{V}$$

where

c	=	position of cushion
r	=	radius of ball
b	=	position of ball
V	=	speed of ball towards the cushion
λ		is a measure of the time of collision.

The calculation of the time at which two balls collide is almost as straightforward. The calculation checks for the time at which the balls touch assuming they continue at their present velocity; (in this simulation, it is assumed that the way in which a ball slows down is for its velocity to remain constant for 1/25 second, then suddenly drop to the value for the next frame). If the balls collide there are usually two solutions to this equation, one for the actual collision and the other when the balls are touching after having passed through each other. It is obviously the former of these which is required; this will always be the solution which occurs at the earlier time.

$$d^2 = (x_2 + \lambda(\vec{V_2} - \vec{V_1})_x - x_1)^2 + (y_2 + \lambda(\vec{V_2} - \vec{V_1})_y - y_1)^2$$

where d is the diameter of the ball (the distance which the centres of the balls will be apart when they touch), x_1, x_2 are the x coordinates of the positions of ball 1 and ball 2, y_1, y_2 are their y coordinates, $\vec{V_1}$, $\vec{V_2}$ are the velocities of the balls and λ is a measure of the time of collision.

Let

$$\begin{aligned} V_x &= V_{2x} - V_{1x} \\ V_y &= V_{2y} - V_{1y} \\ x &= x_2 - x_1 \\ y &= y_2 - y_1 \end{aligned}$$

then

$$\lambda = \text{ minimum positive root of } \frac{-k_2 \pm \sqrt{{k_2}^2 - k_1 k_3}}{k_1}$$

where

$$\begin{aligned} k_1 &= V_x^2 + V_y^2 \\ k_2 &= xV_x + yV_y \\ k_3 &= x^2 + y^2 - d^2 \end{aligned}$$

If the balls do not collide the discriminant will be negative (i.e. complex roots mean no collision). If the only solutions are negative the balls are moving away from each other and do not collide.

Each moving ball is checked for collision with all other balls to determine the earliest collision time.

In addition to explicit rules to guide the motion, there are also a small number of constants which characterise a particular snooker table. These include such values as the coefficient of friction of the table's surface, the size of the table and balls, and the coefficient of restitution of the cushions. These have sensible default values, but they may be altered by the animator if desired. Interesting effects may, for example, be achieved by playing with very large balls, or altering the coefficient of restitution of the the cushions to greater than unity.

6.2.1 The user interface

The user interface was felt to be very important since one of the major reasons for the experiment was to evaluate the style of working. Practically all of the output from the program is graphical in nature although this is sometimes reinforced by the use of textual messages. Input is primarily performed using a mouse—commands, for example, are selected from pop-up menus; balls are selected by placing a cursor over them; and velocities are specified by indicating the length and direction of an arrow emanating from the centre of the ball. The current frame number is at all times visible in the bottom right hand corner of the screen; this is an important requirement of any practical animation system, since this is the method by which the animator is able to time sequences.

The program was implemented in BCPL [*Richards & Whitby Strevens, 1979*] on the Rainbow Workstation [*Wilkes et al., 1984*] under the Tripos operating system [*Richards et al., 1979*]. Rainbow is a prototype high performance graphics workstation developed at the University of Cambridge Computer Laboratory. Its most important feature is that it can dynamically map graphics memory onto the screen; that is to say that images stored in rectangular areas of graphics memory (up to eight bits deep)—known as *pads*—can

be displayed at any position on the screen. The visibility of an image or its position on the screen can be changed (effectively instantaneously) by what is essentially a change in the value of a pointer—it is not necessary to copy the image into a different area of graphics memory. This means that an arbitrarily complex rectangular image can be moved around the screen as quickly as a simple rectangle. It is possible to have these images obscure one another using a priority ordering, as in more conventional windowing systems; what is somewhat less usual is that, under certain circumstances, images can be made to interact with one another to give transparency or other effects. This facility was made great use of in this program; the image of a billiard table was drawn in one pad, two other pads with transparent backgrounds were used to hold the image of the balls (two pads were used, but only one was displayed at any one time—to enable double buffering), and a third transparent pad was used to hold a cursor. The Rainbow Workstation is described in more detail in Appendix I. Pop-up menus are trivially simple to implement using this system; a variety of pop-up menus and message areas were used. Figure 6.5 shows the screen layout as the program is running.

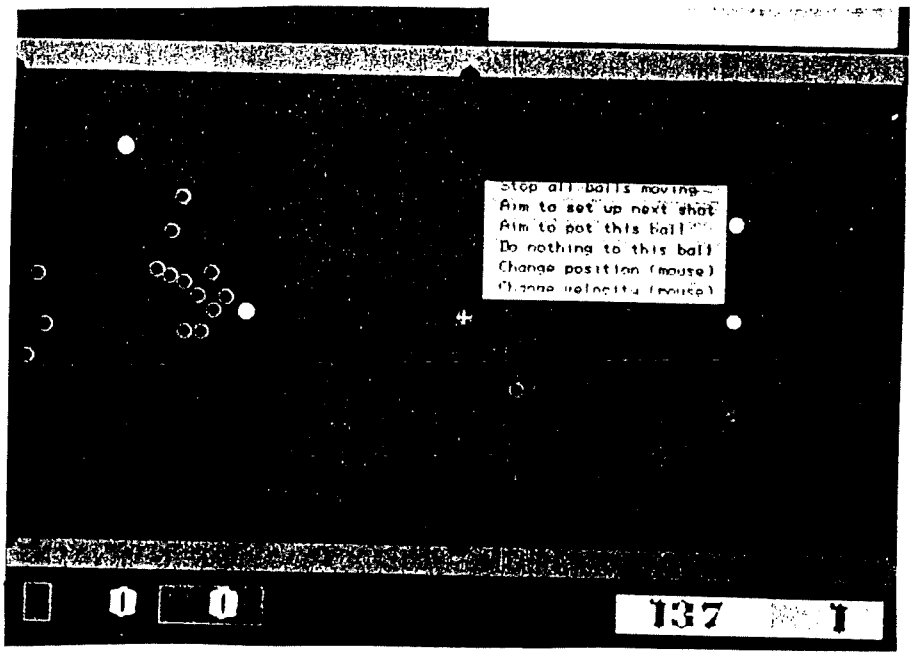


Figure 6.5: Screen layout as snooker program is running. This picture shows the orange menu obtained by pressing the middle mouse button when a ball is selected; a smaller green menu appears just to the right of the cursor if no ball is selected (see figure 6.1).

Some care was taken over the construction of the screen layout and the colours used. In particular, the pop-up menus were experimented with until a pleasing effect was obtained. It was found that the use of pastel colours for menus was particularly restful for the user; the two menus used were therefore

chosen to be an attractive pastel orange and a light, de-saturated shade of green. Each option is presented as a rectangle with a white border containing black writing on the orange or green background. As the (invisible) menu cursor passes over an item, that item is highlighted as black writing on a coloured background. The background colours are again all pastel shades and each colour is unique to its own menu item, thus the user can associate menu items not only with their position in the menu, but also with their colour. A selection is made by moving the mouse with the middle button depressed until the relevant item is highlighted; the middle button is then released, and the menu disappears. A message (written on a rectangle with the same coloured background as that selection's menu highlight colour) then appears in the top left hand corner of the screen to remind the user which item has been selected and to give him instructions for any further actions which may be necessary to complete the command initiated by this menu selection. This mechanism was found to be surprisingly useful; the user quickly gets used to the idea of particular colours being associated with particular actions—even if he does not read the messages every time, something in his subconscious is quickly alerted to the fact that something is amiss if a message with the wrong coloured background appears, due perhaps to his mistakenly selecting the wrong item from a menu. If the user intends to alter a ball's velocity, for example, but accidentally selects the command to change the ball's position instead, the fact that a pink message appears rather than a deep green one may be enough to cause him to pause before pressing any more mouse buttons.

An unusual feature of the orange menu is that it *scrolls*. The menu is larger than the portion visible on screen: as the user approaches the top of the visible menu it scrolls smoothly downwards to reveal the hidden entries (this causes the bottom few entries to disappear from view). If the user then moves the mouse downwards, the menu remains where it is until the menu cursor approaches the bottom of the visible menu, at which point the menu scrolls smoothly up again. This technique means that the menu obscures less of the screen than would otherwise be the case, but it has the disadvantage that the user cannot see all of the possible menu selections simultaneously. The scrolling effect is achieved quite simply on the Rainbow Display by moving a pad relative to its (smaller) clipping cluster (see Appendix I for explanation).

In this program an attempt was made to use mouse buttons in a consistent manner. A three button mouse was used. The right hand mouse button is used to mean "re-start simulation" (i.e. make the balls re-commence moving according to the simulation rules). The left hand mouse button is used to magnify mouse movements by a factor of four (i.e. if the user wishes to move the cursor from one side of the screen to the other, he can either move the mouse—with no buttons pressed—until the cursor reaches the required position, or he can hold down the left hand mouse button and move the mouse a quarter of the distance). The middle mouse button is used for everything else—in particular the middle button will cause a menu of options to appear if the user is not in

the middle of some action which has been initiated from the menu. The only exception to the above general rule occurs when the user wishes to amend an earlier frame in the animated sequence; having selected this option from the menu the left and right mouse buttons are then used to mean move backward and forward (respectively) through the already generated frames.

6.2.2 A typical animation session

When the animator first enters the program he is asked whether he wishes to begin a new game or restore the situation from an earlier session; this is to allow an animated sequence to be developed over a period of time rather than insisting that the animator should finalise the entire sequence in one sitting.

The animator is then presented with a picture of a billiard table viewed as an orthographic projection from directly overhead. All of the relevant features are visible; these include the cushions, the pockets, the "D" and the baulk line. The balls are initially arranged ready for the beginning of a game ("*the break*") unless the user has asked to reload an earlier session, in which case the balls are arranged as they were at the end of that sequence. This initial ball placement is, of course, only a suggestion by the system; the animator is free to alter the ball positions as desired; this may include moving some balls off the table to indicate that they have been potted. In the top left hand corner of the screen is a help area; this is the area used to give the animator hints about the options open to him and how to achieve them. The initial message informs the animator that in order to proceed further he should press the middle button on the mouse in order to obtain a menu of options. The menu he obtains depends on whether or not the small cyan cursor (moved by use of the mouse) was positioned above a ball when the middle mouse button was pressed. Placing the cursor above a ball in this way, prior to pressing the middle mouse button, is referred to as *selecting a ball.* Whenever a ball is selected a message appears in the top right hand corner of the screen to inform the user which ball has been selected and what its current position and velocity are; if no ball is selected a message informing the user of this fact appears instead. The reason for the different menus is that some options cause action to be applied to a specific ball; these options are therefore presented to the animator only when a ball has been selected. The selections available on the two menus are as follows:-

Menu when no ball has been selected (the green menu) :–

- Alter a different frame
- Exit from the program
- Do nothing
- Replay sequence at full speed
- Change the cueball's velocity

Menu when a ball has been selected (the orange menu) :–

- Change ball's position (by entering coordinates via the keyboard)
- Change ball's velocity (using keyboard)
- Stop all balls moving
- Aim to set up next shot
- Aim to pot this ball
- Do nothing
- Change ball's position (by indicating new position using the mouse)
- Change ball's velocity (by using the mouse to specify an arrow representing the magnitude and direction)

Each time the menus "pop-up", the option which is initially highlighted is *Do Nothing*; this means that if the middle button is tapped by mistake nothing disastrous will happen.

Having set up suitable initial positions for the balls the animator's next step is usually to change the cueball's velocity, simulating the effect of a player hitting it with his cue. This is achieved by selecting the bottom item from the green menu (the menu obtained by pressing the middle button on the mouse when the cursor is not positioned above a ball). This is such a common selection to be made that the user quickly gets used to pressing the mouse's middle button, moving the mouse downwards and releasing the middle button all in one smooth fluid movement. As the menu cursor is constrained to keep within the menu area, there is no danger of moving down too far.

Having made this selection a message appears in the top left corner of the screen to inform the user that he should specify a suitable velocity by indicating the end of a *velocity arrow*. Depressing the middle mouse button and moving the mouse gives rise to a magenta rubber-band line which is used to represent the magnitude and direction of the ball's initial velocity. A dotted extension to this line allows accurate aiming of a relatively short arrow (see figure 6.6); the value becomes fixed when the button is released. The next stage is to indicate the amount of top spin or back spin required. As mentioned earlier, spin is approximated by a hypothetical velocity component either in the same direction as the main velocity arrow or the opposite direction. The *spin velocity arrow* is specified by moving the mouse up (for top spin) or down (for back spin) while the middle button is depressed; the distance moved represents the magnitude of the spin component. This component is constrained to be along the line of the first arrow. Alternatively, if no spin is required, the user may simply press the right hand mouse button to cause the simulation to begin. As mentioned earlier, it is almost invariably the case that the user can press the right hand button on the mouse in order to abandon the current action and

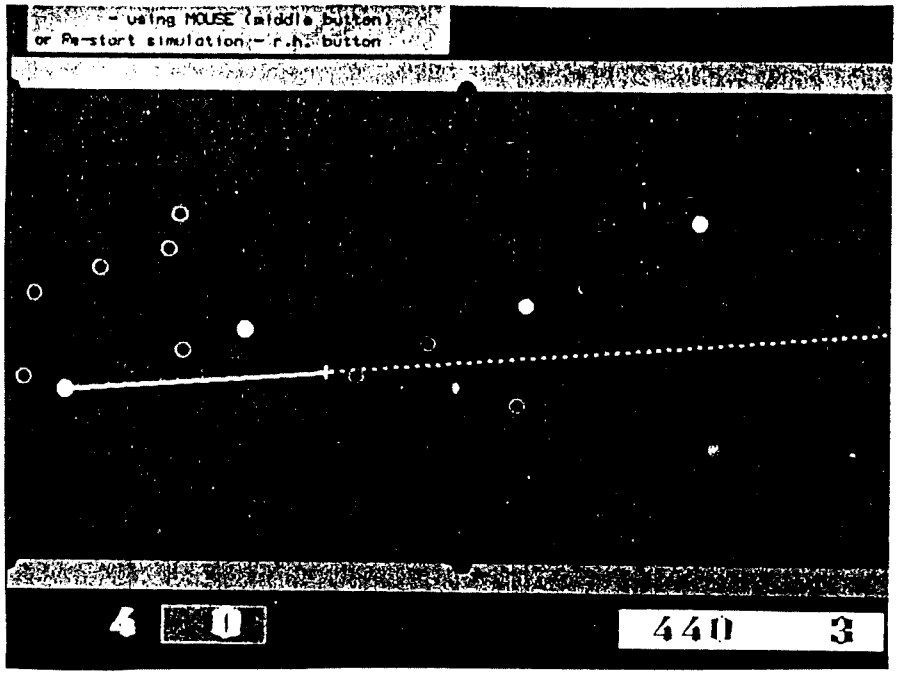


Figure 6.6: Setting the velocity of a ball.

re-start the simulation process. If the velocity specified is not satisfactory the whole process may be repeated by selecting the item from the menu again and repeating the process.

Once the simulation is started the balls move according to the laws of motion built into the software until interrupted by the animator pressing the middle button on the mouse. This may either be when all of the balls have come to rest, or at some earlier time if the animator wishes to alter the course of the animation in some way.

When the animator interrupts the simulation in this way, a menu appears in order to allow him either to alter some aspect of the current frame (such as the ball positions or velocities), or return to an earlier frame, make an adjustment to the situation in some way, then recommence simulation from that point.

Controlling the outcome of a shot

Let us assume, for the sake of illustration, that the animator is attempting to animate the very first shot in a game and wants the player to play a safety shot which leaves the next player in a snooker by causing the cueball to stop behind the yellow ball (after hitting the pack of reds). The way that the animator is likely to achieve this is by making an initial guess as to the cueball's correct initial velocity and position within the D, then starting the simulation to watch the outcome. It is almost certain that the outcome will not be as desired, but how should it be adapted? The answer to that question depends on how near the result is to being correct. If it is totally wrong (the shot was too slow and

ended up heading in totally the wrong direction, for example), the animator will select the top option from the green menu ("*alter a different frame*") and return to the very first frame by holding down the left hand mouse button. When moving backwards and forwards through the frames, velocity arrows are visible on all of the moving balls. This naturally includes the very first frame in the sequence so the animator is able to see what the velocity was on the previous attempt and adjust it and/or the ball's position in order to achieve a more satisfactory outcome. Once the outcome is not too badly wrong (it is up to the animator to decide when this is), the animator can decide to "cheat" in order to achieve the required goal. This cheating involves breaking the rules of the simulation in order to change the motion of the balls. The amount of rule bending which is desirable (or acceptable) is a decision made by the animator. The most satisfactory place to perform adjustments is at collisions since then any lack of realism will be less noticeable than if a ball suddenly changes speed or direction whilst rolling across the table (although, of course, the animator is free to bend the rules at any time—perhaps for comic effect). In our current example the most suitable time for making a large alteration is when the cueball hits the pack of reds since the mathematics is sufficiently complex at this point that a viewer is very unlikely to notice anything amiss. Other opportunities for slight adjustments are when the ball hits the cushions although here the mathematics is considerably simpler so a viewer is much more likely to notice that the speed or rebound angle is incorrect. Let us assume that the animator wishes to perform rule bending only at the instant of collision between the cueball and the pack of reds. The method he would use to achieve this would be as below.

The animator would initially select "*alter a different frame*" from the green menu and move backwards through the already-generated frames until the frame at which the collision with the pack occurred. The state of play is saved by the simulator both every 1/25 second of simulated time (a "*time frame*") and every time a collision between balls, or between a ball and a cushion occurs (a "*collision frame*"); a record of ball positions and velocities (after collision) is therefore certain to be available for every collision. During movement calculation or playback only the time frames are shown since these are the frames which will go to make the final sequence; however frames at which significant events occur, such as collisions or animator interventions, are very useful for "cheating", so a separate stack of these frames is maintained to aid the animator in his task. The animator can find the collision frame he requires (in this case the collision between the cueball and the pack of reds) either by replaying the entire sequence backwards (both time frames and collision frames), or by skipping backwards through the frames, considering only frames at which significant events occurred, until the relevant frame is found. The choice between these two methods may be made dynamically while moving backwards and forwards through the frames by use of a sub-menu which appears on the top right hand corner of the screen. Pressing the left hand mouse button causes the

program to step backwards one frame; pressing the right hand button causes the program to step forward one frame, and moving the mouse changes the definition of one frame to be either truly a single frame or a single significant event (holding down the left or right button produces a conventional "auto repeat" facility). Pressing the middle mouse button terminates the process and selects the current frame as the frame to be considered. All the time the user is moving through the frames the screen display of score, frame number, collision number and current player is kept consistent with the display. The program also keeps an internal knowledge of which balls it is legal for a player to hit and pot; this means that it is not possible to confuse the scoring algorithm by moving backwards in time then altering the outcome of a shot.

Having found the required frame the animator has several methods of rule bending at his disposal: he can simply change the velocity of one or more balls in an attempt to achieve the desired outcome, or he may try one of the more powerful and predictable approaches. There are two particularly useful commands provided on the orange menu, namely "*aim to pot this ball*" and "*aim to set up next shot*". They are used as follows:-

1. *aim to pot this ball* is used to ensure that a ball is aimed precisely at a pocket (the particular pocket is specified by pointing with the mouse). It leaves the ball's speed unaltered but modifies the direction to point towards the indicated pocket. It warns if the ball's current velocity is insufficient for the ball to reach the pocket, but no account is taken of possible collisions with other balls which may prevent the ball reaching the pocket.

2. *aim to set up next shot* is useful when a ball is required to end up in, or pass through, a particular position on the table. The position to be achieved is indicated, together with the speed the ball should have at that moment. If desired, the user may also specify a list of cushions to be hit on the way; the program then calculates the velocity which is necessary to achieve this outcome. Collisions with cushions are dealt with properly but no account is taken of possible collisions with other balls.

In our particular example it is the *aim to set up next shot* option which is of use. The user would select the cueball then indicate a position behind the yellow ball and state that the cueball should be stationary when it reaches that point; he may also, for example, specify that the ball should hit the left, bottom and right cushions (as viewed on the screen) in the process of the shot. The program would then calculate the required velocity and draw a dotted line to indicate the path the ball would take. The user would be told what the calculated velocity and the original velocity of the ball were and be asked to confirm that this change was acceptable before the old velocity was discarded. Reasons for not accepting the correction might include such things as the fact that the user considered that the change in velocity was too great or that there was a ball in the way of the indicated path (remember, no account is taken of

collisions with balls during the calculation although collision with cushions is properly dealt with). If the calculated velocity is accepted and the simulation re-started from this point, the motion of the balls will now be such that the cueball comes to rest behind the yellow having hit the specified cushions on the way. However, it may still be the case that the animator is not entirely happy with the sequence produced; it may be that several more changes to ball velocities are necessary before the sequence is totally correct.

Completing the sequence

More shots are built up in this way until the animator is satisfied with the complete sequence of events. At any time the animator can select the "*replay sequence in real time*" option from the green menu in order to view all of the frames produced (or any sub-range of them) at 25 frames per second. He can thus remind himself of the sequence produced so far, and judge the quality of the motion.

The animator is free, at any time, to leave the program and save the state of the sequence so far to a file (either the complete sequence or a subset of frames). This allows a sequence to be developed over a number of different sessions; the file is also used at later stages in the development of the final sequence (such as "setting up camera angles" and rendering). This checkpoint file is written in a compact textual format—keeping an incremental record of how balls are moving. The first twenty two lines in the file specify the initial positions of the twenty two balls, in some standard order; thereafter each line represents a single frame and contains details of the change in position of any balls which have moved since the last frame.

For example,

```
5000,6003
7000,7002
    :
    :
19000,3000
1 122 -119, 7 -2 1, 22 -12 13
1 119 -116, 22 -11 12
-1
```

This rather unlikely file begins by defining the positions of the twenty two balls at the start of the sequence; the next line says that balls 1, 7 and 22 (the cueball, the black and a red) have moved since the previous frame, and gives the amounts by which they have moved. The next line says that in the next frame only balls 1 and 22 are still moving. A negative ball number marks the end of the sequence. All movements are specified using an internal integer coordinate system which has 4096 units to the foot. The playing surface of a billiard table is twelve feet by just over six feet so ball positions can be packed into a single 32 bit computer word (16 bits for each of x and y).

The above description gives only a brief taste of the facilities available using this motion development program but it hopefully gives an indication of the normal style of working, namely:

1. Specify initial positions and velocities of balls in an attempt to produce the required motion;

2. if the results obtained using straightforward simulation from this initial position are not as desired select *"alter a different frame"* from the green menu and move back through the frames until a satisfactory situation is reached;

3. modify velocities as necessary in order to improve the motion—this may involve the use of commands such as *"aim to pot this ball"* and *"aim to set up next shot"*;

4. repeat from 2 above until a satisfactory result is obtained.

The next section describes the next stage in the animation process after motion development—the introduction of a virtual camera with which to view the scene as specified so far.

6.3 Setting up "Camera Angles"

Having developed the motion for the desired sequence of shots in a game of snooker, the next stage is for the animator to decide on suitable positions from which to view the scene; this is what I refer to as "setting up camera angles". During the motion definition process the snooker table was viewed from directly above in order to allow the motion development to be treated as a two-dimensional process, now it is desirable to view a three-dimensional representation of the table from more interesting positions in three-dimensional space.

I decided that a simple interpreted language would be the most appropriate means by which to specify the motion of a virtual camera. This choice was influenced by the desire to make control accessible to a non-computer-expert user whilst recognising that it is difficult to accurately specify complex three-dimensional motion in a graphical way; the use of a specially developed command language should give access to a wide range of effects in the simplest possible way. The result of interpreting a "camera script" would be a camera position, orientation and "focal length of lens" for each frame in the sequence. This would then be combined with the ball positions to produce a representation of each frame drawn in perspective, as viewed from the given camera position.

The interpreter for camera control language and the software to produce a simple perspective view of the resultant scene was initially implemented on the IBM 3081 mainframe computer by an undergraduate student named Andy Storey [*Storey, 1984*] before I implemented an enhanced version on the Rainbow Workstation [*Wilkes et al., 1984*]. The description below refers to the version implemented on the Rainbow Workstation.

6.3.1 The "camera control language"

The *camera control language*, CCL, was intended to provide a straightforward and powerful way for a naive computer user to control the motion of a virtual camera (otherwise known as an observer). Since it was aimed at the non computer scientist it was considered important that the language should provide a flexible syntax and good error messages.

Before considering the language in detail, let us first consider some terminology:

- the **observer** is the thing which is surveying the scene; it has also been referred to, above, as the "virtual camera";
- the **viewpoint** is the position or object at which the *observer* is directing its attention (the point which will appear in the centre of the frame in the final picture; the centre of interest)—this may be specified as an absolute position in space or relative to some known object such as a particular snooker ball or pocket on the table;

- a **camera** (of which up to ten are allowed) is a predefined *observer position*—it is possible, for example, to specify an observer position by saying "observer at camera 3";

- **focal length** is used to define the angle of view in the same way as in conventional photography—the shorter the focal length (i.e. the smaller the focal length number) the wider the angle of view and the more exaggerated the perspective. The choice of focal length numbers was designed to make sense to a person familiar with 35mm photography—thus a focal length of 50 corresponds to the view that would be seen using a 50mm lens (this gives a perspective view which is approximately similar to the human eye). 28mm will be recognised by photographers as a wide angle lens and 135mm as a telephoto lens—the software allows any focal length in the range 8mm to 2000mm to be specified.

The CCL interpreter processes two files:-

1. the *camera script* (written by the user), and

2. a file containing the position of all balls in each frame (produced by the motion definition software).

The output from the program is a file containing a list of viewing parameters specifying the observer position, viewpoint and focal length for each frame in the sequence—this will be used by later visualisation programs to produce perspective views of the scene.

Using CCL it is possible to specify any combination of stationary and moving observers, viewpoints and focal lengths operating over time periods specified using absolute frame numbers or relative to significant events such as balls hitting each other or being potted.

All coordinates can be specified in either of two coordinate spaces—the default being measured in metres from the centre of the table but the other being configurable by the user (so the user could, if he preferred, measure distances in inches from the top left pocket).

Each CCL command specifies a range of frames to which it applies and any other information regarding observer position, viewpoint and focal length which it wishes to change. Any values which are not specified for a particular frame keep the same value as they had previously (everything has a default value at the start of day).

Commands are delimited by colons (:); the sub-parts making up the command are separated by semi-colons (;). Sub-commands may be specified in any order. Anything which occurs before the first colon on the file is treated as a comment. Commands may be in any mixture of upper and lower case and layout is free format.

Before a formal description of the facilities of CCL, let us consider a short example of a CCL script.

```
:
from frame 1 until frame 33;
observer at camera 1;
viewpoint at (0,0,0);
focal length of 50
:
camera 3 at (7.3, -4.5, 1.2);
from frame 34 until cueball strikes;
observer moving from camera 3 - (0,1,0) to camera 3;
focal length from 30 to 135;
viewpoint at pink spot
:
until pink ball stops - 10;
viewpoint following pink ball
:
until pink ball stops;
observer following pink ball + (0.1, -0.1, 0.05);
viewpoint at top left pocket;
focal length of 28
:
```

This script is intended for use with a sequence which depicts the cueball being struck and hitting the pink ball (which was on its spot at the time). The pink ball then goes into the top left pocket.

From frame 1 to frame 33 we want a general view showing the whole table (from the default position of camera 1)

The second command begins by redefining the position of camera 3, then specifies that from frame 34 until the time that the cueball hits the pink ball, the observer should be moving smoothly from a point one metre away from camera 3 in the y direction (parallel to the short side of the table) to the camera 3 position. The centre of interest (viewpoint) should be zooming in on the pink spot. In fact the `from frame 34` part of the command is redundant since the default start time for a command is immediately after the preceding command.

The third command leaves the observer at camera 3 and the focal length as 135mm but causes the centre of interest to follow the pink ball (thus keeping the pink ball in the centre of the screen). This continues until ten frames before the pink ball stops moving (in this case the reason that it will stop moving is that it will have entered a pocket).

The final command produces a wide angle view from a few centimetres behind the pink ball looking towards the pocket as the ball approaches it.

This example gives a rough appreciation of the power of CCL. There now follows a more complete definition of the facilities available.

Time definition commands

```
FROM <event> [+/- <offset>] UNTIL <event> [+/- <offset>]
```

This command defines the time over which the observer, viewpoint and focal length commands take place. At its simplest it simply specifies frame numbers at which other commands start or stop taking effect; more complex use allows times to be related to significant events occurring to the balls (or some number of frames before or after such a time).

```
UNTIL <event> [+/- <offset>]
```

As above, except that the initial frame of the time interval is taken to be the current frame.

An event is defined as follows:

```
<event>   ::= <number>/FRAME <number>/CUEBALL STRIKES/
              ANY BALL STARTS/ALL BALLS STOP
              <ball> STARTS/<ball> STOPS/END

<offset>  ::= number of frames before or after specified event

<ball>    ::= CUEBALL/YELLOW BALL/GREEN BALL/BROWN BALL/
              BLUE BALL/PINK BALL/BLACK BALL/HITBALL

HITBALL   ::= ball most recently struck by the cueball
```

Observer and Viewpoint commands

The observer and viewpoint commands have identical syntax except the relevant keyword. Their effect is either to position the observer (observer commands) or direct the observer's attention to a particular point in space (viewpoint commands).

```
OBSERVER  AT <coordinate>
VIEWPOINT AT <coordinate>
```

These commands allow the observer or viewpoint to be positioned in space—where it remains fixed until the next command to reposition it.

```
OBSERVER  MOVING FROM <coordinate> TO <coordinate>
VIEWPOINT MOVING FROM <coordinate> TO <coordinate>
```

Using these commands the user can cause the observer or viewpoint to move along a straight line between the two sets of coordinates (the timing of the move is found from the current time command).

```
OBSERVER FOLLOWING <ball> [+/- <simple coordinate>]
```

This command allows a "ball's eye view" to be produced—the camera position remains fixed relative to the centre of the given ball.

```
VIEWPOINT FOLLOWING <ball> [+/- <simple coordinate>]
```

Use of this command means that the specified ball (or some position at a specified fixed offset from the ball) will remain in the centre of the field of view.

Coordinates are specified as follows:

```
<coordinate> ::= <simple coordinate>/<name>/
                 <name><op><simple coordinate>

<simple coordinate> ::= (<number>,<number>,<number>)/
                        (#<number>,<number>,<number>)
                  - without a # means that the coordinate is
                    in metres relative to the centre of the
                    table, with a # indicates user-defined
                    coordinates (see misc commands, later)

<number>      ::= any real number

<name>        ::= <ball>/<colour> SPOT/CAMERA <camera id>/
                  BOTTOM LEFT POCKET/BOTTOM RIGHT POCKET/
                  LEFT CENTRE POCKET/RIGHT CENTRE POCKET/
                  TOP LEFT POCKET/TOP RIGHT POCKET

<ball>        ::= <colour> BALL/CUEBALL/RED <num>

<colour>      ::= YELLOW/GREEN/BROWN/BLUE/PINK/BLACK

RED <num>     ::= a red ball defined by its relative closeness
                  to the top cushion when compared with other
                  red balls (i.e. RED 8 is the eighth closest
                  red to the top cushion). Reds at equal
                  distances are counted left to right. This
                  form of identification is necessary since
                  reds are otherwise indistinguishable.

<camera id>   ::= integer in the range 0 to 9

<op>          ::= +/-
```

Commands to set the focal length

```
FOCAL LENGTH OF <number>
```

defines a fixed focal length. `<number>` is a real number in the range 8 to 2000.

```
FOCAL LENGTH FROM <number> TO <number>
```

defines a change of focal length over the current time period; this simulates the effect of zooming in or out.

Miscellaneous commands

```
CAMERA <camera id> AT <coordinate>
```

allows the symbolic name "`CAMERA <camera id>`" to be given to the specified coordinate—where `<camera id>` and `<coordinate>` are as defined above. This is particularly useful for positioning a ring of virtual cameras around the table and switching between them without the need to remember explicit coordinates.

```
COORDS <origin> <factor> <factor> <factor>
```

allows the user to set up his own coordinate space if he does not like the default one. `<origin>` is the new position for the origin (expressed in the default coordinate space); the three `<factor>` values are the number of user units to the metre in the x, y and z directions respectively. To distinguish coordinates in user units from those in default units `#` should be the first symbol in the brackets.

6.3.2 Scene visualisation

As mentioned above, the output from interpretation of a *camera control script* is a file containing an *observer position*, *centre of interest (viewpoint)* and *focal length of lens* for each frame in the sequence. The positions of the balls for each frame in the sequence is available on a file produced by the motion development software. Using the information contained in these two files it is a relatively simple matter to produce a perspective view of each frame. The mathematics involved is explained in [*Angell, 1981*]—the observer and viewpoint positions are reasonably self explanatory; the focal length value is used as a measure of the distance from the observer to the perspective plane.

The perspective viewing software is intended only to give an indication of the view obtained using the specified viewing parameters, no attempt is made to produce a highly complex, realistic image—the main objective is to produce the pictures as quickly as possible so that the scene composition and motion dynamics can be judged. It is important, however, to ensure that positioning of image components is exactly as it will be in the finished sequence, so it was considered essential, for example, that the image should be properly clipped to

the viewing area. Balls are represented as solid circles of colour without any attempt at smooth shading to indicate their spherical nature; and the billiard table has no corner pockets since these are unnecessary details when visualising the effectiveness of camera positioning.

The program makes use of the transparency facilities of the Rainbow Workstation (see Appendix I) to provide double-buffering and enable only the necessary parts of an image to be re-drawn between frames (not re-drawing the billiard table, for example, if its representation remains the same in different frames). In this way, the user is presented with a moving sequence that is as near to the dynamics of the finished sequence as possible. A single virtual background pad is used; this does not need to occupy physical graphics memory since it is all one colour. Two pads are used to hold double-buffered images of the billiard table while two more pads hold double-buffered images of the balls. The lookup tables are arranged such that the balls appeared to be above the table, which is itself in front of the background—the parts of the pads which do not contain useful information appear transparent, so that the images in the pads 'below' them show through. At any one time the background pad and one of each of the table and ball pads are on display while the remaining table and ball pads are having new images drawn into them ready to be displayed when complete.

As mentioned above, if the viewing parameters remain constant between frames it is not necessary to redraw the table since it will not have moved; all that is necessary is to redraw the balls in their new positions. It was decided to redraw all balls (in decreasing order of depth) rather than attempting to redraw only those ones which had moved since the last time that buffer was used since the housekeeping involved in erasing balls and deciding which balls were either moving or newly exposed may take more time and trouble than erasing the entire pad and redrawing all of the balls (particularly since it is only with a static view that there would be any possibility of retaining any ball positions). A fast circle drawing routine was written in an attempt to speed up the drawing of the balls as much as possible, but even so it only proved possible to achieve an average frame rate of twelve frames per second—even with a static observer and viewpoint. If it became necessary to redraw the billiard table every frame due to the viewing parameters changing, the frame rate dropped to an average of three frames per second. Figure 6.7 shows results obtained using this program.

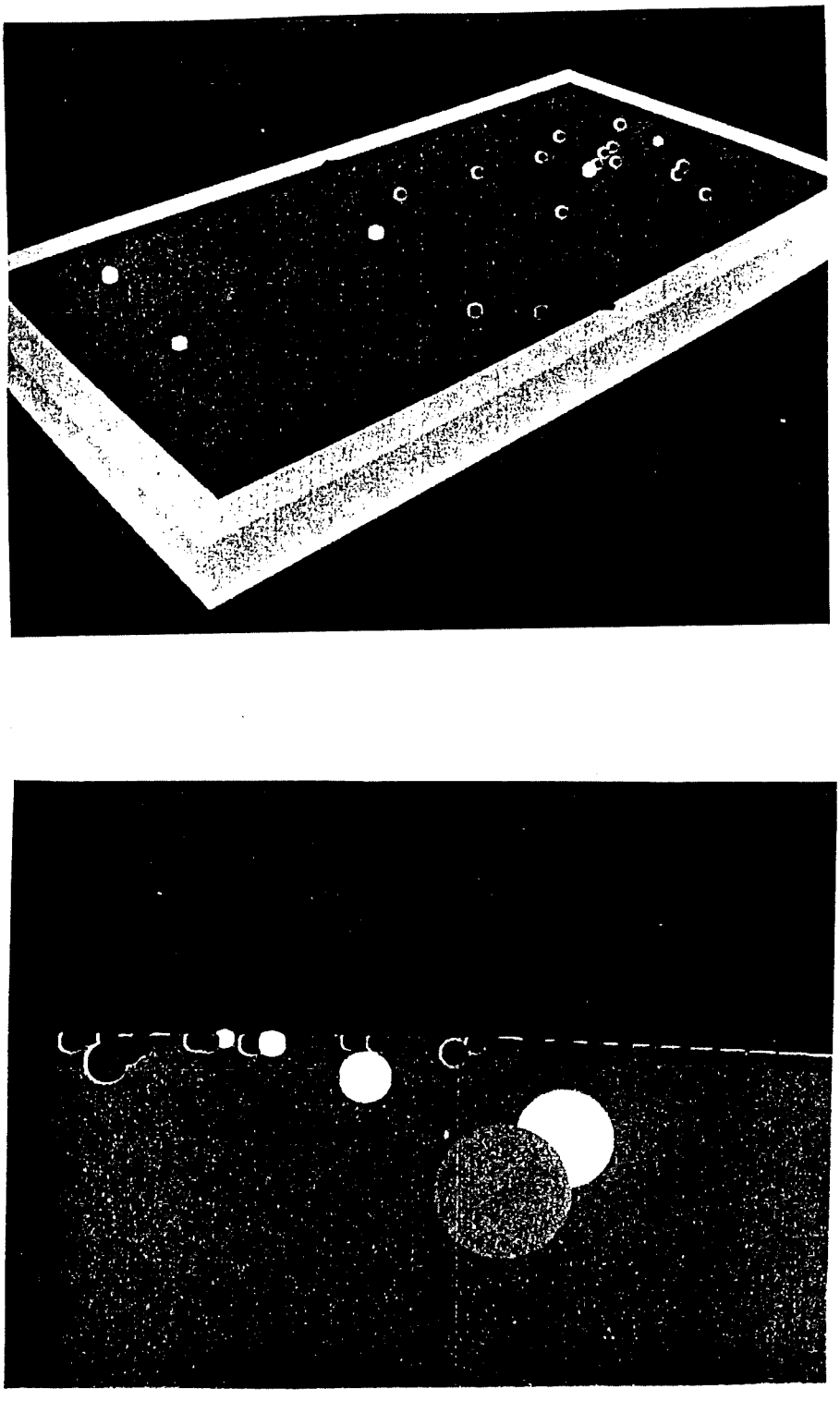


Figure 6.7: Images produced during camera definition.

6.4 Rendering Sequences Ready For Filming

After deciding on suitable motions for the balls on the billiard table and virtual camera positions from which to view the scene, the next stage is to produce high quality versions of the frames in the sequence. These images may then be compiled into a finished sequence by means of either a single frame video device or a pin-registered movie camera.

The production of a high quality image on a raster graphics terminal is an enormous subject in its own right which will not be dealt with here—what follows is a brief description of the way in which the problem was approached in this particular case. The actual program was implemented on the IBM 3081 mainframe with a Sigma T5688 graphics terminal [*Sigma, 1983*]. The specification of the problem and many of the suggested solutions were mine, but the detailed implementation was performed by Nick Bonvoisin as part of his undergraduate degree [*Bonvoisin, 1984*]. Only well known methods were used and no originality is claimed. Constraints of time and limitations of the available hardware meant that the implemented solution was not the best one possible under ideal conditions.

Some of the factors which were considered important for the production of high quality images are listed below.

- **perspective projection** to give an impression of depth—particularly as the projection changes when the position of the object relative to the observer changes. It was important that the view produced should agree exactly with the view predicted as part of the camera definition stage.

- **realistic illumination** makes a very major contribution to the visual realism of a scene. Most scenes are illuminated by one or more specific light sources (which for simplicity may be considered as point sources), as well as more general illumination due to such things as reflection of light from walls and ceilings. The illumination of objects by a light source gives rise to a *diffuse* illumination component and *specular highlights*; there will also be an *ambient* component due to background illumination. For a more detailed explanation please see one of the general computer graphics texts such as [*Foley and Van Dam, 1982*].

  There are several different algorithms aimed at modelling the reflectance of light from various surfaces but two of the most commonly used are *Phong shading* [*Phong, 1975*] and *ray tracing* [*Whitted, 1980*]. Ray tracing is an elegant algorithm which copes with the display of many types of object including mirrors and transparent objects; it also simplifies the generation of *shadows*. Phong shading, on the other hand, has the advantage of higher computational speed with adequate results for certain classes of object; in particular it deals well with objects made of plastic, or with plastic-like properties (see [*Cook and Torrance, 1982*] for explanation). Phong shading however is not capable of straightforwardly dealing with shadows or inter-reflections between objects.

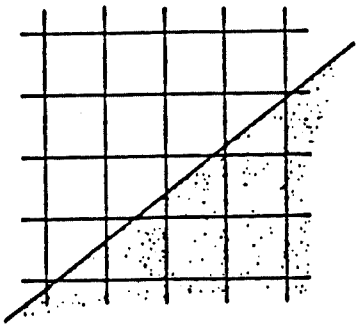
- **shadows** are useful not only to make a scene appear more natural, but also to add to the feeling of depth and to help to locate objects in space.

In our particular case shadows of the balls on the table would help to establish that the balls were in contact with the baize, rather than floating above it.

- **anti-aliasing** is extremely important if fine detail is to be displayed on a screen with limited resolution (the resolution of the graphics hardware used for this program was only 768 x 512 with 8 bits per pixel). In very simple terms, anti-aliasing is a filtering technique aimed at fooling the brain into believing that the picture is displayed at a higher resolution than is in fact the case. It does this by increasing the "depth resolution" of the image; a variety of shades are used to "smooth" the boundary between colours in such a way that if a pixel is partially covered by objects of two or more different colours, the shade displayed at the pixel is a weighted average of these colours[1]. Without anti-aliasing the pixel would be filled with only one of the overlapping colours (such as the one which covers the centre of the pixel), this gives a misleading impression of where the boundary is. Figure 6.8 illustrates the display of a boundary between a black polygon and a white polygon with and without anti-aliasing; notice the *"jaggies"* along the boundary in the straightforward aliased image and the shades of grey along the boundary of the anti-aliased version. The technique relies on the fact that our eyes cannot distinguish between size and brightness for small light sources so when the image is viewed from far enough away, our brain does not register that some of the pixels are a strange colour (neither black nor white); it appears to interpret grey pixels as smaller white pixels filling in the jagged edges of the black/white boundary. This is because our brain tries to organise fuzzy visual data into patterns: boundaries are accentuated and hard edges are the preferred interpretation. In order to obtain the very best anti-aliased results it is important to use gamma-correction (see [*Foley and Van Dam, 1982*]) when computing colours to be put in video lookup tables—to compensate for non-linearities in the display capabilities of computer monitors and our perception of intensity and colour.

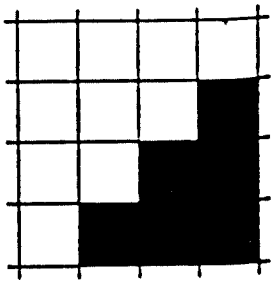
- **texture mapping** provides a way of adding surface detail to objects without increasing the complexity of the underlying model [*Blinn & Newell, 1976*]. This would be useful for the wooden parts of the billiard table to give the effect of grain, without which a brown rectangular slab does not look at all wooden (ideally this texture should also be anti-aliased). Another area in which texture mapping would be useful would be to add a subtle texture to the balls—representing "specks of dust" or "smudges of chalk"—which could move in such a way as to give the viewer visual clues to the fact that a ball was rolling, and in which direction. These specks would be particularly useful when a slow moving ball was seen in close-up.

[1] any colour for display on a graphics monitor can be described as the sum of red, green and blue components [*rgb*] so the weighted average may be produced by averaging the red, green and blue components of the various colours present at the pixel.



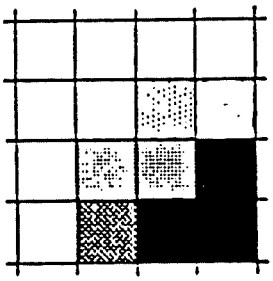
Polygon edge intersecting the pixel grid

1	1	1	1
1	1	1	0
1	1	0	0
1	0	0	0



Binary values in the vicinity of the boundary and the resulting image

15	15	15	15
15	15	13	7
15	12	5	0
10	3	0	0



Grey values in the vicinity of the boundary and the resulting image

Figure 6.8: Demonstration of anti-aliasing (taken from [*Wilkes & Wiseman, 1982*]).

All of the above techniques are important when producing a high quality still image but there are yet more considerations necessary to produce a high quality animated sequence:

- aliasing is not limited to static images; annoying though "the jaggies" are in a still picture, they become many times worse when that edge slowly moves across the screen in an animated sequence as the "steps" along the edge subtly alter between frames to give a *"crawling"* effect; this makes anti-aliasing of the individual frames even more important for animated sequences than for still images. In addition to the jaggies, aliasing manifests itself in a variety of other ways such as very small objects appearing to be the wrong size or shape (or even disappearing altogether) as their position affects their registration on the pixel grid; these effects are naturally much more noticeable if the object in question moves. In an animated sequence the positions of objects are sampled in both time and space; filtering ought, therefore, to be applied in the time dimension too; this leads to the idea of motion blur.

- motion blur, introduced in [*Korein and Badler, 1983*] and [*Potmesil and Chakravarty, 1983*], is a form of temporal anti-aliasing. It simulates the

action of a conventional film camera in which the shutter is open for an appreciable amount of time (approximately 1/50 second) each time it records the scene—so each frame does not represent an instantaneous snapshot of the world, but rather a blurred image depicting all of the positions that each object passes through during the time that the shutter is open. When applied to computer-generated images the effect is to make the motion appear very much smoother than is the case without motion blur. The most successful motion blur seen by me to date has been produced using a computationally expensive technique known as *distributed ray tracing* [*Cook et al., 1984*]. There are many problems associated with the production of accurate motion blur by less computation intensive techniques.

- it is necessary to be careful that any approximations ("tricks") used in the rendering of the various frames in an animated sequence cause consistent results in all of the frames of the sequence. For example, Phong shading is normally used to produce smooth shading of curved surfaces which have been approximated by polygonal facets: under certain circumstances, if such an object is slowly rotated during an animated sequence, the position of the highlights can flicker in such a way as to suggest the underlying polygonal surface—even though the boundaries between polygons have been smoothed away.

6.4.1 The implementation

The choice of rendering algorithms was influenced by a number of factors, but particularly:

1. The hardware and software environment within which the program was to be developed. It was to be written in BCPL and use a dumb eight plane graphics terminal (a Sigma T5688) attached to the university mainframe (an IBM 3081) via a serial line. There were also limitations on memory space and cpu time which could not be exceeded without penalty.

2. The time available in which to complete an undergraduate project meant that a totally comprehensive solution could not be attempted.

Bearing these limitations in mind it was decided to use the Phong model for computing diffuse, specular and ambient components for illumination of the balls. The balls are **not** represented using polygonal facets so there is no need to use interpolation of surface normals [*Phong, 1975*] to achieve smooth shading, instead the surface normals are calculated from the mathematical representation of a sphere. Multiple light sources were allowed—the most common method of lighting a billiard table is by suspending four light bulbs in a canopy above it, so it was thought desirable to be able to model this situation if requested.

The table surface itself and the cushions are rendered using constant shading for the sake of speed and ease of implementation—it would always be possible to implement a more subtle solution should it become necessary. The wooden surround of the table was also initially rendered using constant shading but with the firm intention to add wood texture and Phong shading (unfortunately, however, time ran out before these improvements could be added). It is worth noting that when using Phong shading it is sometimes assumed that the light source is at infinity in order to simplify the calculation (only the surface normals are changing from one pixel to the next); in this implementation this was not possible since the light sources are usually very close to the table and the direction to the light source is very definitely not constant—indeed in the case of the wooden surround, the surface normal remains constant and it is the directions to the light source and the observer which change.

Shadows are used to increase the realism of the illumination but it was decided that, for ease of calculation, only the shadows of the balls and cushions onto the table surface would initially be considered. For each light source the shadow of a ball on the baize is approximated by an ellipse of which one axis is the diameter of the ball and the eccentricity and position are related to the direction from the ball to the light source. The shadows of the cushions can also be calculated by considering the directions to the light sources. Other shadows are possible (such as shadows of balls on cushions) but these were not considered essential for an initial implementation, particularly since they are not usually apparent under normal viewing and lighting conditions.

Hidden surface elimination is performed by the painter's algorithm—i.e. image components are drawn in decreasing order of depth (a simple calculation since the objects to be drawn and many of their relationships to each other are known in advance). Assuming that the viewer is above the level of the playing surface it is possible to start the drawing process with the table legs, then progress to the playing surface and the shadows of the balls on it, followed by the cushions which are "behind" the playing surface, then the balls (from back to front) and finally the cushions and table edges closest to the observer. The image is drawn in perspective, using the viewing parameters produced by the "camera angles" program and ball positions produced by the motion development program. The image is clipped to the size of the screen so that anything off-screen or behind the observer does not appear.

Anti-aliasing is used to improve the image quality—all boundaries between colours are smoothed by computing a colour for the boundary pixels based on their coverage by the various colours meeting at the boundary. It is relatively simple, when drawing a ball, to compute the fraction of a pixel which is covered by the ball and average the ball's colour with the colour already present at that pixel position (remember that the images are created by drawing objects in decreasing order of distance from the observer). The Pitteway algorithm

[*Pitteway and Watkinson, 1980*] is used to anti-alias the edges of the polygonal objects such as cushions, wooden surround, legs and so on; these polygons are each shaded a single colour so it is theoretically a simple matter to draw all of the polygons using the Sigma's flood filler then anti-alias the boundaries between colours using the above algorithm. Unfortunately, a problem occurred due to the fact that the Sigma's firmware does not use Bresenham's algorithm to draw straight lines [*Bresenham, 1965*]; this means that the steps in the lines on the polygon boundaries do not agree with the positions predicted by the anti-aliasing algorithm, with the result that the colour inside a polygon sometimes appears on the outside the polygon's anti-aliased boundary (with unfortunate consequences). Given more time, this problem could easily have been overcome by avoiding the use of the Sigma's line-drawing firmware, but instead using software to draw lines for the polygon boundaries using Bresenham's algorithm (or perhaps by adapting the Pitteway and Watkinson algorithm to perform anti-aliasing along the lines produced by the Sigma).

As mentioned above, wood texture for the wooden parts of the table was not included due to lack of time; but a rudimentary representation of specks of dust on the balls was included.

Other problems which occurred during the implementation resulted from the fact that the Sigma is an eight-plane device—it is therefore only possible to display 2^8 (i.e. 256) colours simultaneously. This number is not really sufficient to produce an anti-aliased, full-colour image. This was particularly true since lack of picture storage space meant that it was infeasible to use anything but a fixed palette of colours—the alternative, given more memory space, would have been to calculate the image using a higher colour resolution and only decide after the image had been calculated which would the most useful 256 colours for that particular image (it is pointless, for example, to have colours which are only used to anti-alias a yellow/pink boundary if the pink and yellow balls never appear next to each other). Even using this approach it would often be necessary to approximate colours; some method would therefore have to be found to achieve continuity of approximation between frames so that objects didn't undergo annoying slight changes of colour during the sequence.

Shortage of time and implementation problems meant that the rendering software was never satisfactorily completed but the results obtained were nonetheless quite encouraging as demonstrated by figure 6.9.

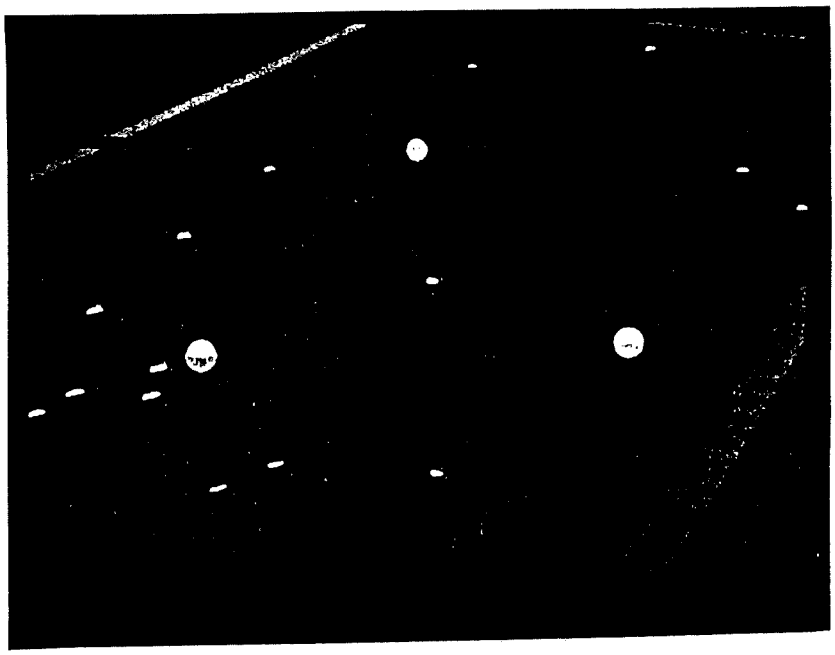


Figure 6.9: Shaded image of a snooker table. Note the multiple highlights and shadows under the balls.

6.5 Evaluation of the Snooker Experiment

The motion development experiment, described in the early part of this chapter, was a first attempt at testing the idea of using simulation to aid an animator in producing a piece of animation. As such it had its shortcomings—in particular the fact that the simulation rules were built into the program meant that the motion could not be customised by the animator quite as readily as would be expected in a more general purpose system. However, in the limited universe of games of snooker on a billiard table, the results of the experiment were encouraging—with the above software it is possible to define the motion of the balls for any game of snooker that the animator desires. The more realistic the game is intended to be, the easier it will be to animate since then the majority of the motion will be dealt with by the simulation software with only a limited amount of animator intervention necessary; certain types of unreality can also be easily added to the system, these include such things as different sizes of balls and different coefficients of restitution for the cushions. Even totally unrealistic games, with swerving balls for example, can be animated using this software since the animator has all of the necessary tools at his disposal to enable him to accurately control the motion of the balls in any way he desires—including for comic effect; however, in this case, the animator must take much tighter control of the animation, so many of the speed benefits are lost.

Despite having the simulation rules built into the program, it was still found possible to adapt the software to allow it to be used for simple animation of titles as opposed to games of snooker: all that was necessary was to change the rendering procedures so that the table became invisible and the (larger than usual) balls had letters of the alphabet on them. It then became possible to use the tools provided (particularly the *"arrange to set up next shot"* command) to arrange that the letters lined up to form words at opportune moments.

The use of a "camera control language" as described in section 6.3.1 proved to be a very acceptable solution to the problem of specifying viewing parameters since it gave accurate control in a very simple way. It is a relatively quick process to prepare a "camera script" and view the scene from the positions specified, then refine the script if necessary and repeat the process. An even better approach than that described would have been to combine the camera language interpreter and the perspective viewing program to produce an interactive system in which the user could issue a command and be presented with immediate visual feedback as to what the sequence would look like. This would provide a much better environment in which to develop camera motions.

The final rendering process was relatively uninspiring but did not really form part of the research; only existing techniques were used and the problems and solutions are well understood. The use of subtle "speck of dust" textures on the balls proved useful to suggest that balls were rolling rather than sliding.

The final outcome was that I decided that the results of this experiment into the realm of simulation for animation were sufficiently encouraging to suggest that further experimentation was worthwhile; these further experiments are described in the next chapter.

Chapter 7

Cellular Automata—a further use of simulation in the animation process

After the snooker experiment it was considered worthwhile to attempt to produce a slightly more general purpose system which would allow the animator to specify the rules to be used during the simulation rather than having them built into the software. However, it was felt that it would still be too big a step to develop a full general-purpose three-dimensional system, so it was decided to develop a two-dimensional system based around cellular automata instead. These automata are relatively simple for a user to program, yet are capable of producing very complex and interesting effects. A further reason for this choice was that it was also felt that it may be possible to build up intricate animation of larger objects by defining low level rules to describe the motion of simple component parts.

7.1 What is a Cellular Automaton?

In what follows, the term *cellular automaton* is used to mean a large, two-dimensional array of *cells* arranged on a rectangular grid, in which each cell is capable of containing a maximum of one *life-form* from a finite set of such life-forms; there are rules to determine which life-form should be in which cell at any particular moment in time (and also which cells should be empty), these rules normally take into account the type and distribution of living cells in the previous time period (each time period is normally referred to as a *generation*). Changes in cell contents occur in discrete time steps—all cells change value simultaneously. A cell with no life-form in it is referred to as a *dead cell*, the opposite of which is a *living cell*. Much of this terminology comes from the *Game of Life* invented by John Conway [*Gardner, 1970*] and studied by Conway and others in the 1970s.

7.2 The Game of Life and its Generalised Descendants

The "Game of Life" is an interesting mathematical game played on a rectangular grid as described above; the size of the grid is not fixed by the rules of the game but is assumed to be large. There is only one type of life-form in this game (so a cell is normally described simply as either *living* or *dead*) but there are a variety of rules to define how these life-forms can "be born" or "die". The player begins by setting up an initial configuration of living cells on the grid, then applies the following rules (or more usually watches while they are applied by a computer).

1. A cell will continue to live in the next generation if it has two or three living neighbours in this generation (by a neighbour I mean one of the eight cells immediately surrounding the cell in question).

2. A cell will be born in the next generation (i.e. change from being dead to being alive) if it has precisely three living neighbours in the current generation.

3. In all other circumstances the cell will be dead in the next generation (regardless of its current state).

These rules, simple as they are, can give rise to very complex and interesting effects; particularly since the perceived effect is not so much that individual cells are living and dying, but more that objects composed of groups of these cells are evolving—changing shape and moving across the grid. The most interesting effects are observed at the macro level rather than down at the level of the individual cells. The above rules support a large number of persistent patterns, some of which remain static, whereas others cycle on the spot, and yet others gradually creep across the grid as they cycle through a number of states. The best known example in this final category is the "*glider*" (many of the interesting shapes discovered during the study of the *Game of Life* have been given names); it goes through a repeated cycle of four states and moves diagonally across the grid as demonstrated in figure 7.2.

Naturally there is no reason why other rules cannot be used rather than those described above, but these rules, with their ideas of death caused by isolation or overcrowding, have been found to be particularly suitable for study since they do not cause undesirable population explosions or extinctions. This chapter describes experiments with cellular automata using more general rules, including having more than one type of life-form and allowing rules which consider neighbours more than a single cell away.

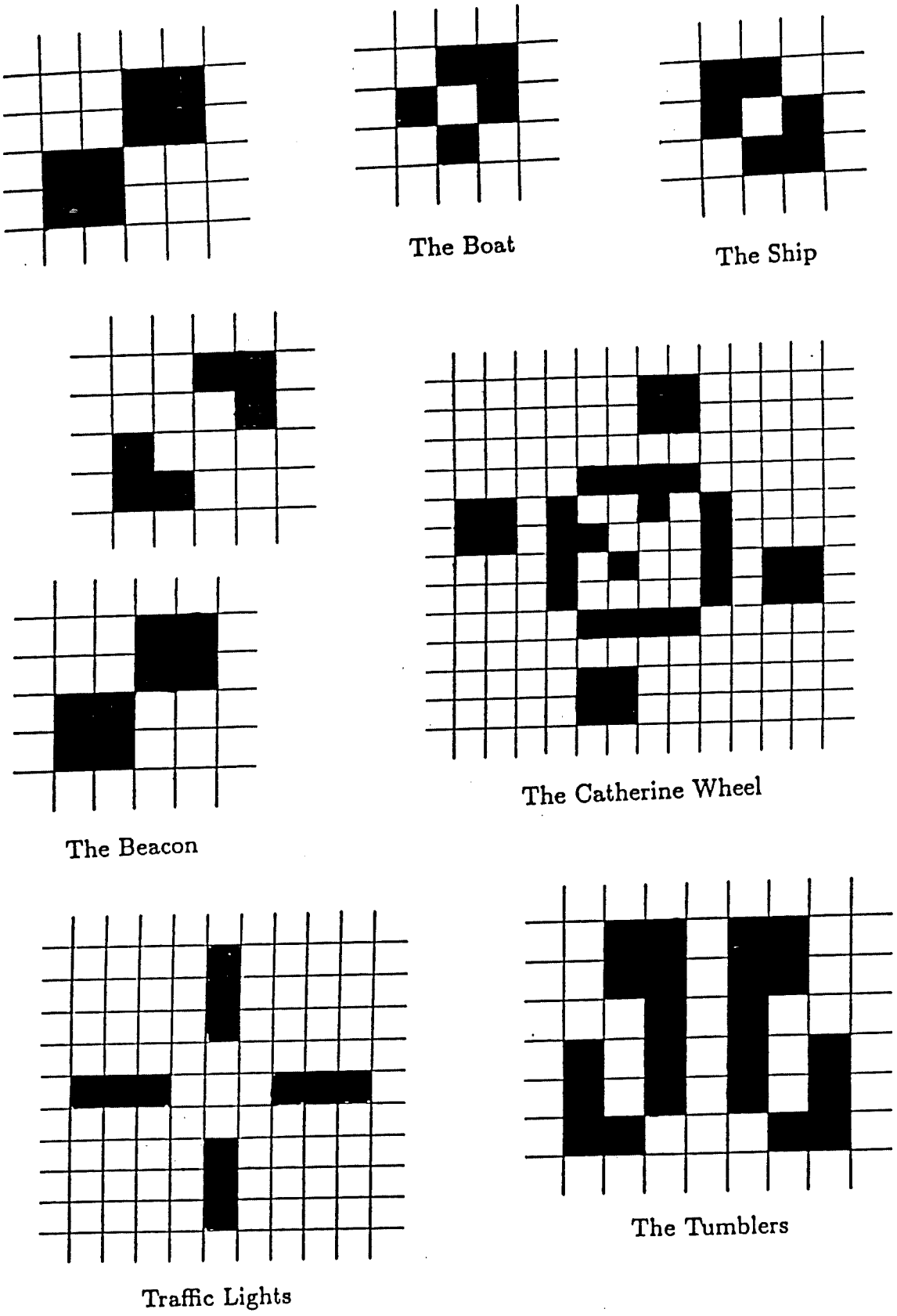


Figure 7.1: Examples of patterns from the Game of Life.

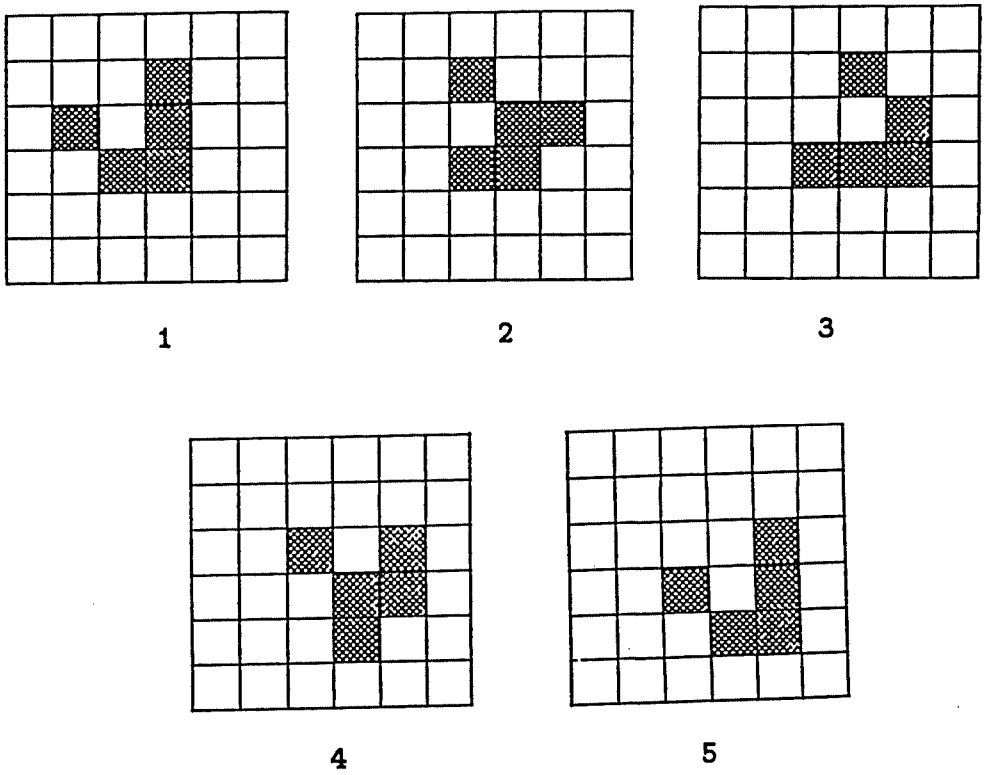


Figure 7.2: The evolution of a "glider" over five generations. Note that the configuration in the fifth generation is the same as the initial pattern but translated diagonally by one cell; this process continues ad infinitum unless acted upon by some external effect.

7.3 An Overview of the Software Developed for this Experiment

The programmable cellular automaton software developed for this experiment was implemented in BCPL on the Rainbow Workstation [*Wilkes et al., 1984*]. It made use of many of the novel features for which Rainbow provides assistance: such as pop-up menus, transparent overlays, double buffering and hardware-assisted panning (see Appendix I for more details of the Rainbow Workstation). Indeed, this program made more use of Rainbow's facilities than any piece of software before it; this made it necessary for me to develop several new techniques specifically for this program—including a very effective technique to create the illusion of panning over an effectively infinite plane. This technique involved mapping a large *pad* containing the image of part of the grid into a parent *cluster* four times, then using Rainbow's hardware assistance to pan this "super pad" relative to the screen. Newly visible parts of the image appearing at the edges of the screen were drawn on the relevant off-screen area of the pad just prior to it coming into view. This gave rise to much smoother panning of a full colour image than could be achieved using conventional bitblt techniques.

The program was intended to be quite general purpose with user-specified rules which could be straightforwardly defined and even altered during the course of the simulation. Cellular automata are machines which make decisions as to what value a cell should take on, based on the old value of the cell and the number and/or type of living cells nearby. Rather than allowing a life-form to inspect the grid around it in order to decide how to behave, it was decided to use a message passing mechanism such that living cells are allowed to pass messages to cells around them and after all messages have been sent, rules are applied to determine what value a cell should take on, based on the messages it received. This fits in with Hewitt's definition of an "actor" (see chapter 4). A practical reason for not allowing cells to inspect the surrounding area was that it would be very inefficient due to the necessity of considering every cell, not only the living ones (dead cells need to decide when to come to life). Using the message passing mechanism, only living cells pass messages, and it is only those cells which receive messages which need to consider their future value, all other cells will be dead next generation (a cell which wants to stay alive must send a message to itself).

7.3.1 Getting started

To enter the program the user types **ca** followed by a number of optional arguments. These arguments are `Log/k,RecordPansAndZooms/s,Checkpoint, Script/k,ReplayPansAndZooms/s`, where `/k` means that the argument is optional but if it is given the keyword must be quoted, and `/s` means that the argument is a switch—if it is given the value of the argument is `TRUE`, otherwise `FALSE`.

The `Log` keyword should be followed by the name of a file in which to store a record of everything which affects the outcome of the simulation. This will be of use if the user wishes to replay the sequence at a later date. In fact, since the only rules currently allowed are deterministic, all that is required is a record of any ways in which the animator intervenes, arranged in time order (the current generation number needs to be stored with each change); all other changes are easily obtained by applying the relevant rules. The list of animator interventions will include the following items:-

- any changes in simulation rules while the simulation is in progress;
- a list of all edits performed by the animator;
- a record of which patterns are loaded from the library, and where they are placed (see later);
- which areas are singled out for advancement while other areas remain static (see later).

If the user also quotes the switch argument `RecordPansAndZooms`, the list will also contain details of

- when the viewport was moved to a different position on the grid, and where it was moved to, together with
- when and how the magnification (zoom) factor was changed.

These final two items allow a sequence to be recreated exactly, so that it can be thought of as a piece of animation which may be replayed at will.

As well as the above information, checkpoint files may be created at intervals specified using the `Checkpoint` argument. These files have names formed by concatenating the name of the log file with the generation number; they contain a complete record of the state of the grid in order to allow the user to recreate a particular generation, or continue from the situation at the end of a previous session without having to wait for the entire sequence to be replayed from scratch.

The `Script` keyword is used to specify the name of the file in which the log of a previous session was kept. All of the user interventions performed during that session will be applied to the current run of the program, although the user will still be able to intervene to change the course of events still further if desired. If the switch `ReplayPansAndZooms` is quoted, any viewing information contained in the script will also be repeated. If a script is used, the user is asked at which generation he wishes to begin, and the software advances the state of the game to that point in time before producing any graphics (the latest applicable checkpoint file is read then the simulation rules are applied from that point until the correct generation is reached).

On initial entry to the program the user is presented with a grey and white grid 36 rows by 47 columns in size. This represents a small viewport onto the centre of a much larger grid over sixty five thousand cells square (i.e. containing over four thousand million individual cells); this is the playing surface. The user has a mouse and a keyboard with which to interact with the program. Graphics appear on a colour graphics monitor and text appears on a separate monochrome monitor. The view on the graphics screen may be panned at any time by moving the mouse whilst holding down its left hand button. The grid moves in the direction of the mouse movement. It was felt preferable that the grid should pan in this way (rather than using the analogy that the mouse is used to move the screen window over the grid), since it was thought to be counter-intuitive to observe that the grid moves in the opposite direction to the mouse.

The program is initially in *edit mode* which means that the user may place living cells onto the grid, in any desired pattern. This operation may be performed in several different ways (chosen from an editing menu obtained by pressing the middle button on the mouse). Figure 7.3 shows an example of

the screen complete with editing menu. Since there were five menu selections (including the one to leave editing mode), it was decided to construct the menu in the form of a cross so that simple movements in the four cardinal directions could be used to make a selection rather than needing to worry about the distance moved. A selection is made by pressing the middle mouse button then moving the mouse until the correct option is highlighted at which point the mouse button is released.

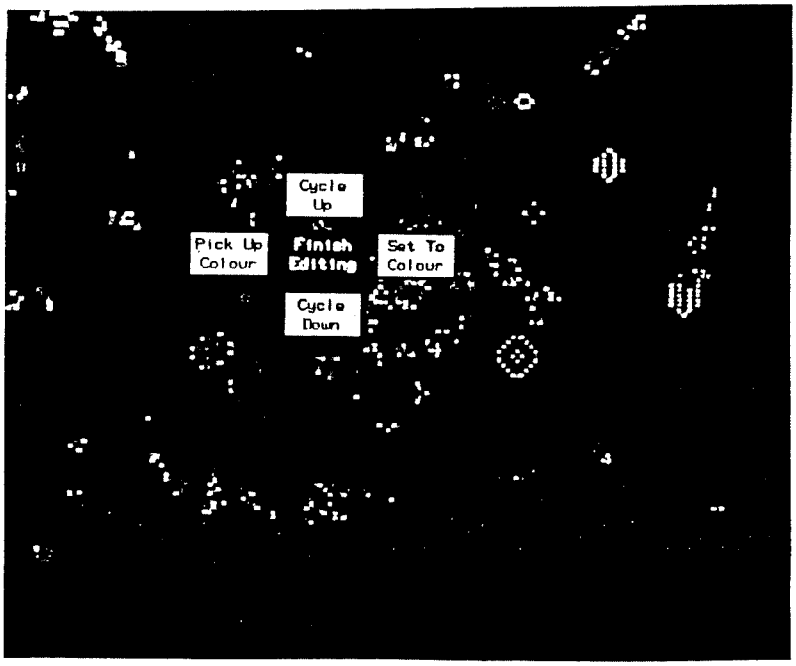


Figure 7.3: Screen layout during editing, including menu from which the editing method may be selected.

All of the editing modes make use of an editing cursor which may be moved by use of the mouse but which is constrained to remain within the visible area (remember that the grid may be panned by holding down the left hand mouse button whilst moving the mouse). The right hand mouse button is used to perform the editing of "life-forms" (each life-form has a number associated with it to indicate its type so this operation is often known as *changing the value in a cell*). The range of possible effects achieved by pressing the right hand mouse button whilst the cursor is above a particular cell is described below, but in all cases if the right hand mouse button is held down while the mouse is moved all of the cells which the cursor passes over are set to the most recently selected value. In editing mode there is a unique relationship between a cell's value and its colour and style of shading so that there can be no confusion as to which life-form is present in a cell.

There are four different methods of editing:-

1. **step up mode:** the value in a cell is incremented by one for each click of the right hand mouse button, thus clicking the right hand mouse button when the cursor is positioned over a dead cell will cause a life-form of type one to be created, further clicks will produce types two, three etc. Only life-forms for which rules currently exist can be selected in this way (to avoid any unnecessary cycling through redundant values). The values wrap round to zero (dead) again after the maximum value has been reached.

2. **step down mode:** as above except that the value is decremented by one each time rather than incremented.

3. **set to value mode:** a further menu appears to allow the user to select a particular value; this is then the only value which can be inserted into the array until another value or method of editing is selected (by means of the editing menu). This value will overwrite any other value except itself; if a cell containing the selected value is indicated again the effect is that the cell is reset to zero (dead)—this allows mistakes to be erased.

4. **pick up colour mode:** the current colour (life-form value) is selected by clicking the right hand mouse button in a living cell. The current value becomes whatever the value in that cell is and the cell remains unaltered. From that point onwards any dead cells which are indicated by the cursor when the right hand mouse button is clicked will be set to this value. Doing this in a cell which already contains the currently selected value will reset the cell to be dead. Selecting any other type of living cell will cause the colour in that cell to become the current colour. At all times there is an marker in the corner of the screen to indicate what the current colour is.

Having produced the required pattern of living cells, by whatever means, the user leaves editing mode (by selecting the middle item from the editing menu) and starts the simulation phase. Simulation can be performed in one of two ways:

- at the fastest rate that the software can perform, or
- a single step at a time in order that each generation can be examined before proceeding to the next generation.

These two modes of production may be toggled between by use of the menu which may be obtained at any time by pressing the middle mouse button (the options available from the menu will be described in section 7.3.2). Initially the software is in single step mode, and requires the right hand mouse button to be clicked in order to proceed to the next generation, but in either case the picture is double buffered so that only complete images are presented to the user—it is important that the cells should all change value simultaneously.

7.3.2 The options available from the main menu

When developing an animated sequence using this software, the animator can obtain a menu of commands at any time by pressing the middle button on the mouse. The menu layout is as follows:

Advance area	Centralise viewport
Alter display	Finish
Change rules	Step mode
Clear area	Auto mode
Statistics	Zoom 16
Do nothing	Zoom 8
Save pattern	Zoom 4
Load pattern	Zoom 2
Randomly populate	Zoom 1
Edit cells	Zoom subpixel

There now follows a brief description of the effect of each of the commands available from the menu.

Edit Cells

Edit mode is entered in order to allow the user to alter the contents of the cells on the grid, as described earlier. In this mode each different life-form is characterised by a different colour and/or style of shading; this allows the value in the cell to be inferred immediately by visual inspection. The same colours are usually used during the simulation too, but the user is at liberty to specify different colours for the life-forms as part of their rule definition—this is particularly useful if the user wants several life-forms to look the same but follow different rules; or if he wants certain living cells to be invisible (i.e. the same colour as the background).

Randomly Populate Area

It may be the case that the user is not worried about the precise positions of the living cells for a particular experiment, but only on the density of living cells and the relative densities of the various different life-forms; this option allows precisely this. The user is initially prompted to specify a rectangle to be populated; he does this by indicating two diagonally opposite corners of the rectangle using the mouse. Once the first corner has been fixed (by pressing the right hand mouse button), a "rubber-band" rectangle is drawn to help in the specification of the opposite corner. The grid may, of course, be panned when indicating these corners in order to define an area which is larger than the screen). The user is then prompted to specify the overall density of living cells to be produced in this area, followed by the relative densities of the various different life-forms. Random numbers are used to determine the value that should be placed in each cell within the area, according to these densities.

Save Pattern

This option allows the user to save a pattern of living cells for future reference. Having noticed that a particular pattern is of interest, the user can interrupt the simulation by pressing the middle mouse button to obtain the menu of commands. If he then selects this option from the menu, he is asked to define a rectangle (as described above) within which the pattern of interest lies, together with the name of a file in which to store it. The final stage is to indicate a *reference point* which will be used to define the position of the pattern when it is re-loaded. Any extraneous living cells which are not part of the pattern, but were included within the saved rectangular area may be tidied up by loading the pattern into a blank area of grid and manually deleting the unwanted life-forms before saving again.

Load Pattern

This option allows a pattern of cells from a library of previously-saved, interesting shapes to be loaded onto the grid at a given position and a particular orientation.

Over the years a very large library of patterns has been built up for the Game of Life, including such patterns as: *Gosper's glider gun* (a cyclical pattern which emits a new *glider* every thirty generations); *Walking spaceship factories* (patterns which move across the grid while emitting another type of small moving object, called a *spaceship*); and ultimately the *Megapuffer* (also known as the *Breeder*) which is a mammoth object, initially consisting of approximately two thousand living cells. The *Megapuffer* is composed of a collection of *trains*, *Walking glider guns* and *spaceships* which move sedately across the grid leaving debris in their wake; carefully orchestrated interactions between this debris, and streams of *gliders* (produced by the *Walking glider guns*) cause the debris to be converted into a row of *Gosper glider guns*, which in turn emit a triangular pattern of *gliders*.

The "load pattern" command allows such patterns to be loaded onto the grid simply by quoting the name of the file containing the pattern, and indicating the cell which the pattern's reference point should occupy (see description of "save pattern" above). The user is then asked for the desired orientation; (there are eight possible orientations: rotations by multiples of 90°, and the four obvious reflections). Multiple copies of a pattern can be loaded by indicating multiple reference points. Mistakes can be rectified by indicating the same reference point again; this causes the pattern to be erased from the grid.

Do Nothing

This menu option has no effect on the action of the automaton; it is provided as a safety device in case the user changes his mind about wishing to issue a command. It is the option which is initially highlighted when the menu appears so that if the middle mouse button is pressed and released by mistake nothing disastrous will happen.

Statistics

Selecting this option causes a table of interesting statistics to be produced. These consist of:-

- the current generation number;
- the total number of living cells;
- the number of living cells of each type;
- the position of the current viewport on the grid;
- the coordinates (expressed in cell positions) of the smallest rectangle which would enclose the entire population of living cells (so that the user can tell where the living cells are in relation to the current viewport);

Clear Area

This option allows all, or part, of the grid to be cleared of life-forms. The user is asked whether the entire grid should be cleared; if not he is asked to specify a rectangle (as described for "randomly populate") within which all life-forms should be removed.

Change Rules

This is the option used to change the rules guiding the motion once the simulation is underway. The user is prompted for the name of a file containing the textual description of the new rules (see later for the format of this file). Any life-forms which do not then have any rules associated with them will simply disappear next generation.

Alter Display Colour

Using this software it is possible to display each category of living cell in a different (possibly user-specified) colour; or show all living cells as white, on a black background. This means that in the *Game of Life*, for example, it is possible to have three different colours to represent the three possible ways that a living cell could be produced, as well as displaying the animation in the more traditional monochrome. The "alter display" menu option allows colour or monochrome display to be selected. Figure 7.4 shows a monochrome display depicting two *Gosper glider guns*.

Alter Display Frequency

This option is used to tell the system how often the display should be updated. Interesting effects may be obtained by missing out some of the intermediate generations whilst watching a pattern evolve. For example, in the Game of Life, the pattern called the *glider* cycles through four states in the process of moving one grid position diagonally; if the display is updated only every fourth generation the effect is of the *glider* shape being translated diagonally

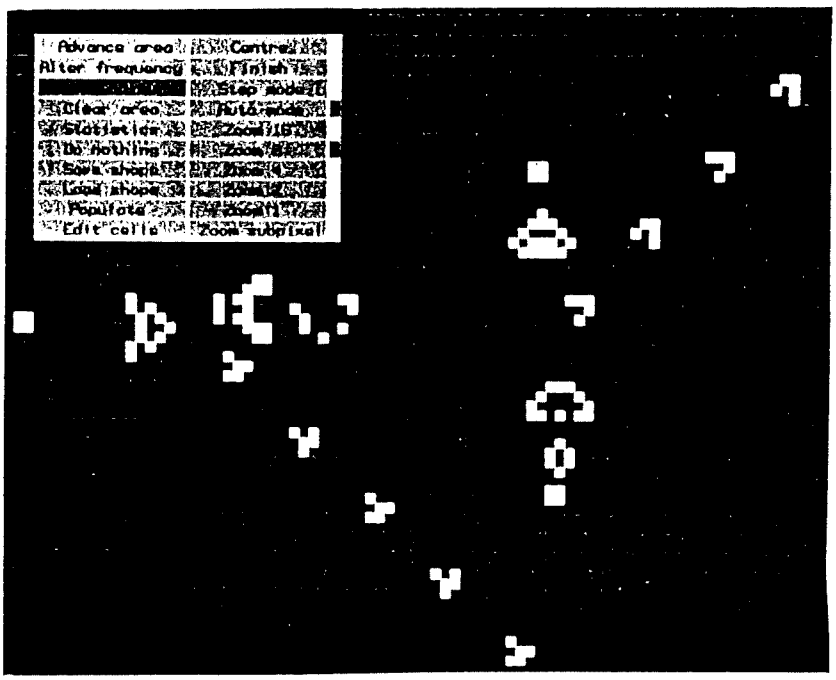


Figure 7.4: "Gosper glider guns" from the Game of Life.

without changing shape. It is sometimes the case that experimenting with different display frequencies will help to highlight the underlying effects of the rules currently guiding the simulation, without extraneous "noise" caused by viewing every generation.

Advance Area (Apply Rules only in a Localised Area)

This option allows different areas of the grid to develop independently of one another. Consider, for example, setting up a pattern for the *Game of Life* which consists of two *glider guns* which should be three generations out of phase with each other. One way to achieve this would be to load a single *glider gun* from the library, step forward three generations, then load the second one. An alternative solution would be to load both *glider guns* at the same time, but use this menu option to draw a rectangle around one of them (as described for the "randomly populate" command above), then step forward three generations (frames), by pressing the right hand mouse button three times; the rules of the game would only affect those cells within the rectangular box, anything outside the box remains unaltered and effectively does not exist as far as any cells inside the box are concerned. Now that the various parts of the grid are in the correct phase with each other the user can select the menu again (using the middle mouse button) and restart normal simulation.

Centralise Viewport

If the user has panned away from the main area of interest and now wants to return to the centre of population, this option will move the viewport so that it is in the centre of the area of living cells.

Single Step and Continuous Production (Step Mode and Auto Mode)

These options give the user the choice of either (i) explicitly controlling when the next generation should be displayed (by pressing the right hand mouse button); or (ii) allowing the program to display the patterns as quickly as they can be calculated. The currently active option is indicated by a marker to the right of the relevant menu option. In either mode the simulation process can be interrupted by pressing the middle button on the mouse in order to obtain the command menu.

During simulation the next generation to be displayed is drawn in an off-screen buffer prior to actual display so that all cells appear to change value simultaneously when it is displayed; (this is known as double buffering). In auto mode the new generation is displayed as soon as it has been produced, but in single step mode the program must wait until the user presses the right hand mouse button. However, it should be noted that the program does not wait for this signal before performing the calculation and drawing of the next generation, only before displaying it. Of course, this unseen generation must be discarded and re-calculated if the user decides to edit same cells before proceeding to the next generation.

Zoom 16, Zoom 8, Zoom 4, Zoom 2, Zoom 1, and Zoom subpixel

These options are used to select the current "magnification". The number after the word Zoom indicates the length of side of a single cell measured in screen pixels. The resolution of the screen is 768x576 pixels and a band of 16 pixels to the right of the screen is reserved to facilitate smooth panning, the effective screen resolution is therefore 752x576 pixels—this means that there is room for 47x36 cells when each cell is 16 pixels square, for example; or 376x288 cells when they are 2 pixels square. At magnifications greater than or equal to 4 pixels square, a single pixel wide separation is left between adjacent cells (so at Zoom 16, for example, the cells are only 15 pixels square with a one pixel border on two sides). At resolutions of Zoom 16 and Zoom 8 the user is additionally given the option of having a grid displayed to enable him to distinguish the individual cell positions.

"Zoom subpixel" is used to fit even more on the screen than is possible with only a single pixel per cell—at single-pixel resolution only 0.04% of the cells on the playing surface are visible at any one time, this makes it very difficult to find isolated pockets of activity which may be occurring somewhere off screen. A wider view may be obtained by having several cells per pixel. When "Zoom subpixel" is selected the user is asked to specify how many cells each pixel should represent. The widest view allowed is to have 1024 cells per pixel, at this resolution approximately 40% of the playing surface is visible at any moment. Obviously individual cells are no longer distinguishable so colour is used to indicate density of cells within each pixel rather than the type of life-forms present.

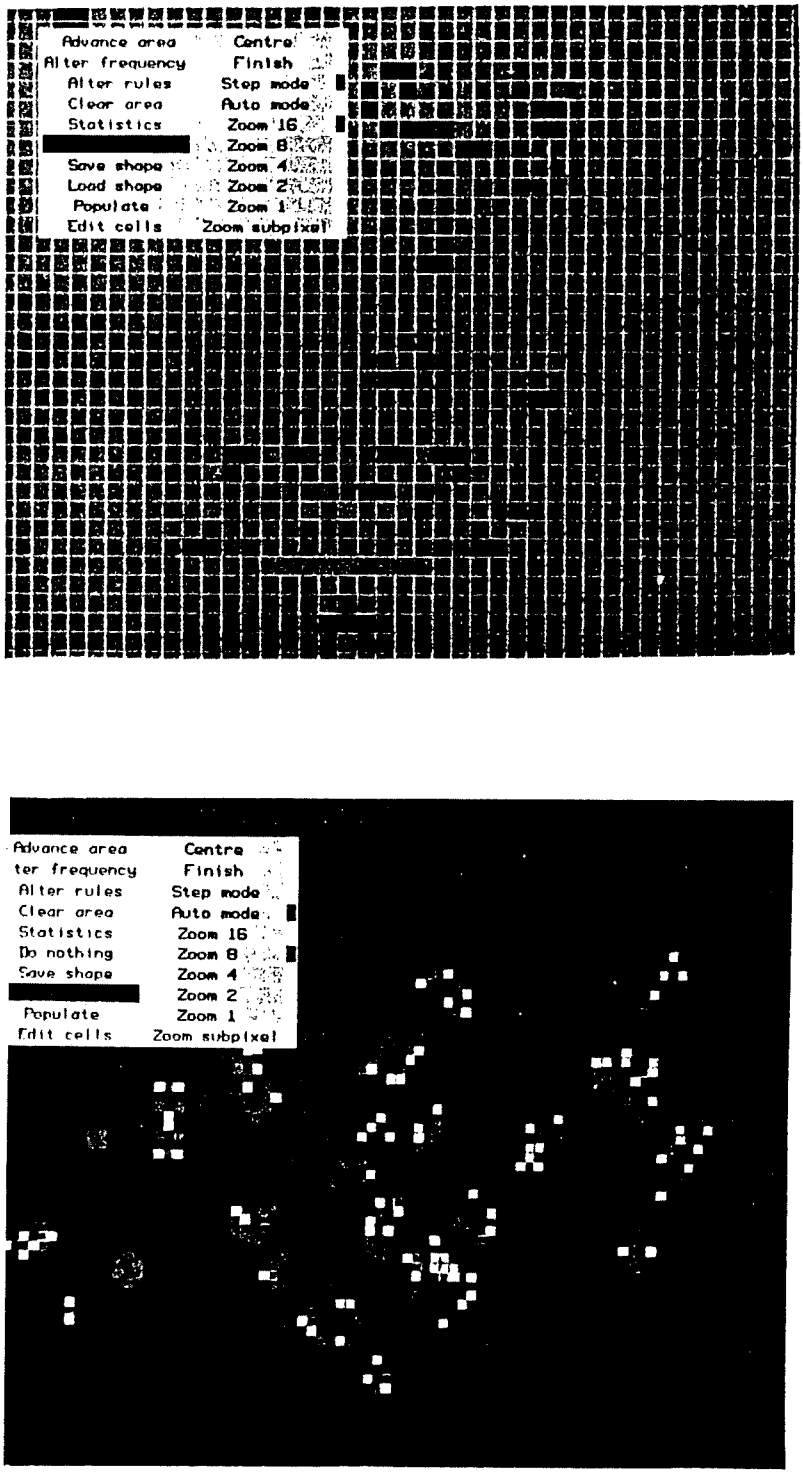


Figure 7.5: Cellular automaton screen images at zoom 16 (with grid) and zoom 8 (without grid).

7.4 The Method Used To Specify The Rules

One of the main motivations behind the cellular automaton experiment in the first place was that the user should be able to specify the rules to be followed without changing the program in any way. A method had therefore to be devised to allow the user to specify separate rules for each individual type of life-form (the software currently supports up to 40 types of life-form simultaneously but this number can easily be increased).

As mentioned earlier the software works by a message passing algorithm; each type of legal living cell must therefore have rules associated with it to specify what messages it should send, and to which of its neighbouring cells they should be sent. Different neighbours are likely to receive different messages. Messages may be sent to cells any distance away (the only limit being imposed by the amount of workspace available in the computer's memory—a row of cells can only be completely processed after messages from all rows which can possibly reach it have been sent). There must also be rules to interpret the messages that have been sent and act upon them.

The messages which can be sent are only of a limited range of types but they have been found to be perfectly adequate in a wide range of examples including some relatively complex ones. Each cell in the grid has, associated with it, a 32 bit message space. At the beginning of the calculation of each generation, each cell's message area is reset to zero. The messages which are sent specify how the bits in this message space should be affected. Messages are free to alter these 32 bits in any way they desire: they may, for example, treat them as a 32 bit integer and perform arithmetic operations on them; or treat them as a set of bits and perform boolean operations, such as **and**, **or** or **exclusive or**. In the *Game of Life*, for example, a living cell could send itself a message to set a certain bit in the message area to indicate that this was a living cell in the previous generation and also increment the "neighbour count" of all surrounding cells by one (see figure 7.6); after all messages had been passed, each cell's message area would contain a count of the number of living neighbours the cell had (together, perhaps, with a flag to indicate that the cell was alive in the previous generation).

When all messages that can possibly affect a cell have been sent, that cell's message area can be inspected, and rules can be applied to determine what value should be placed in the cell in the next generation, based on the messages received. Continuing with the example of the *Game of Life*, a cell will continue to live if it has a "neighbour count" of three (regardless of whether the cell was alive or dead last generation), or if it has a neighbour count of two and the flag is set to say that the cell was alive last generation. One of the many possible encodings of the complete rules for the Game of Life is given in figure 7.6.

The styles of animation possible using this system are not limited to "mathematical moving wallpaper" as the *Game of Life* may be thought to be. A human viewer is very good at observing motion on a global scale and identifying coherence between frames (or *generations* as they have been referred to in this chapter). This means that if a collection of cells maintains a similar overall

```
1     1 1 1 1     //Life-form 1 sends messages at most
                  //one cell above, one below, one to
                  //the left and one to the right.

COLOUR 00 FF 00 //A life-form of type 1 should appear green.
                  //  The three hexadecimal numbers respectively
                  //  define the intensity of the red, green and
                  //  blue components of the colour - any colour
                  //  which can be displayed, can be specified
                  //  in this way.
                  //This item is optional - if it is omitted,
                  //each life-form has a unique representation.

+1    +1    +1    //All eight immediate neighbours are sent the
+1   |#X10 +1    //message to increment the number in their
+1    +1    +1    //message area by one. The cell sends itself a
                  //message to OR the hexadecimal number 10
                  //(i.e. the binary number 10000) into its
                  //message area - #X indicates a hexadecimal
                  //number and #B is used for binary.

$                 //This terminates the message passing rules.
                  //The rules which follow define the actions to
                  //be taken when message areas are inspected.

#X12     1        //if the number in the message area (after
                  //all messages have been passed) is the
                  //hexadecimal number 12 (the flag is set to
                  //indicate that the cell was alive and the
                  //"neighbour count" is 2) the new value in the
                  //cell should be life-form 1

#B?0011 1         //if the message area contains 0...000011 or
                  //0...010011 (? means either 0 or 1) the new
                  //value at this position should be life-form 1
```

Figure 7.6: Rules for the Game of Life. In the above // introduces a comment—i.e. the rest of the line does not constitute part of the rules but is for information only.

shape and colour between frames but has (possibly) changed position slightly on the grid, the viewer is likely to see this as an object which has moved, rather than a set of individual cells dying and others being born.

During the course of this experiment many sets of rules were devised that allowed simple animations to be produced. Some of these were similar in style to the *Game of Life* in that they were developed as interesting mathematical exercises with no particular attempt at enabling the animator to guide the animation in a predictable way. Others attempted to give the animator a means of producing controlled motion at a level higher than the individual cells, in order to produce an animated sequence in which the viewer need not be aware of the mathematical nature of the production software.

Of the "mathematical game" types of rules, one of the most interesting was one which expanded on the rules of the *Game of Life* by adding the idea of "death by over-grazing" so that a cell could remain occupied for only a limited number of generations before it had to be vacated in order to regenerate for a few generations before being able to support life again.

There now follow a few examples from the second category of rules, i.e. those which attempt to produce controllable motion rather than "patterns".

7.4.1 A "maze runner" implemented as a cellular automaton

The following rules allow the production of animations of objects finding their way through an arbitrary maze. It uses the "keep one hand on the wall" principle. The walls of a maze are built up out of cells of type 1 (which are coloured green by default), these will remain static during the animation. A "runner" is then placed near the entrance to the maze, immediately adjacent to the "wall". This "runner" is made up of a "head" consisting of one cell of type 2 (red by default) and a "tail" cell of type 3 (blue by default); it moves "forward" one cell at a time in such a way that the red cell becomes blue, and the red cell "moves" to a cell adjacent to its old position, which is (i) immediately adjacent to a green cell (part of the maze wall) and (ii) not the cell which was originally blue—it is possible that there is more than one cell satisfying these requirements, in which case the runner will split and continue in both directions (runners which meet annihilate each other). The strategy adopted when writing the rules was therefore that:

- the green cells send themselves messages to stay alive
- the green cells send their eight immediate neighbours messages to inform them that they are next to a maze wall
- the red cells send messages to themselves to say that they should become blue next generation
- the red cells also send messages to their four abutting neighbours to tell them that they are potential new positions for the red cell
- the blue cells send messages to the neighbours of any possibly adjacent red cells.

After all messages have been passed the following rules are applied (strictly in the order listed):

- any cells which have been passed messages to say that they are part of the cell wall remain green
- any cell which received a message to say that it was previously a red cell should become blue
- any cells receiving "potential red cell" messages from both the red and blue cells and also a message to say that they are next to the wall of the maze (but no other messages) become red cells
- all other cells should be dead next generation

The rules are illustrated in figure 7.7 which shows the runner going round corners. Other interesting effects (worthy of a few minutes doodling) are observed when a runner approaches a gap in the maze wall (either one or two cells wide). Figure 7.8 presents a possible method of encoding these rules.

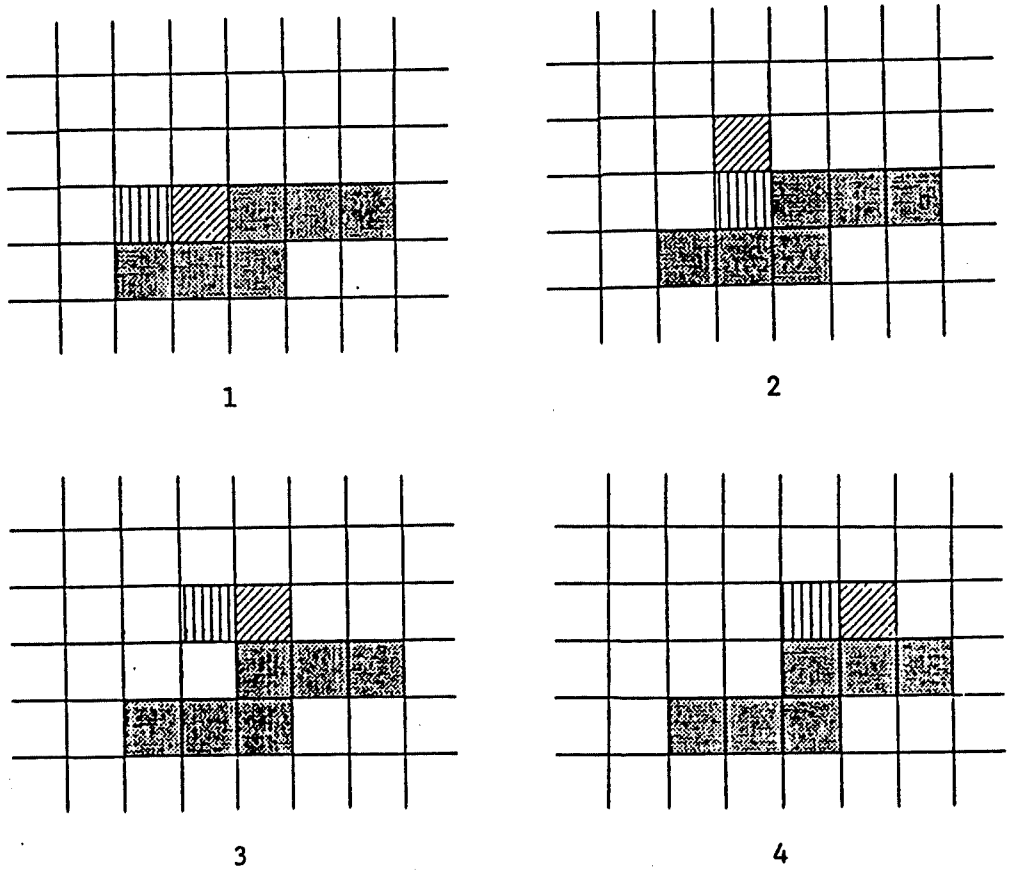


Figure 7.7: The 'movement' of a maze runner over four generations.

```
1                                   //messages sent by life-form 1
-1 1 -1 1                           //(representing the maze walls)

|1 |1 |1
|1 |8 |1
|1 |1 |1

2                                   //messages sent by life-form 2
-1 1 -1 1                           //(the "head" of the maze runner)

.     |2     .                      //a dot (.) indicates that no
|2    |16    |2                    //message should be sent to that
.     |2     .                      //neighbour

3                                   //messages for life-form 3
-2 2 -2 2                           //(the tail of the runner)
                                    //extend two cells in each
.     .     |4    .     .           //direction
.     |4    .     |4    .
|4    .     |32   .     |4
.     |4    .     |4    .
.     .     |4    .     .

$                                   //end of rules defining
                                    //messages sent and start of
                                    //rules defining interpretation
                                    //of messages received

#B1???    1                         //a cell receiving a message that
                                    //it was type 1 should remain so

7         2                         //a dead cell receiving messages
                                    //from each type of living cell
                                    //becomes a new "head"

R 16 23   3                         //a cell whose message area
                                    //contains an integer in the
                                    //range 16 to 23 inclusive (i.e.
                                    //cell was of type 2 and lesser
                                    //messages are irrelevant) should
                                    //become type 3 (this could also
                                    //have been encoded as #B10???)
```

Figure 7.8: Simulation rules for a "maze runner".

7.4.2 Animation of a Turing machine to recognise prime numbers

Another example of instructive animation which was produced using this system is one to demonstrate the operation of a three tape Turing machine which determines whether or not a number is prime. The number to be tested is written in binary as a horizontal pattern of 1s and 0s (with a 1 being represented as a cell of type 1, and 0 by a dead cell), a special cell is also needed at each end of the number to avoid confusion with spurious zeros. Three rows below the left hand end of this number is placed a cell representing the read/write head of the Turing machine in its initial state; the head initially moves to the right under the binary number copying the binary digits to the row above it as it goes. When it notices that it is directly below the cell which marks the right hand end of the number it reverses its direction of motion and checks to see whether the last digit in the number is zero—if so the number is divisible by two and is therefore not prime, so the machine enters a state indicating failure (for demonstration purposes this state was one in which the read/write head moved continuously left until it eventually collided with an area of invisible cells in the form of a cross, which then became visible to indicate that the number was not prime). If the last digit is not zero the number is odd and further investigation is needed to determine whether or not it is prime. The head begins by writing the binary representation of the number three into the second row above it (i.e. between the two copies of the number under test). The head then moves to the left hand end of the number ready to begin the main part of the determination process[1]. The situation at this point is illustrated in figure 7.9 using the number 25 as an example.

The next part of the process is for the read/write head to pass back and forth under the number, performing division by repeatedly subtracting the number on the middle tape from the number on the bottom tape. This continues until either the bottom number becomes zero (in which case the number is not prime), or the bottom number becomes negative. If the remainder is not zero, the head moves back under the digits copying them from the top row to the third row so that these two rows are once again the same. The head then increments the number on the second row by two, to produce the next odd number and the process of repeated subtraction begins again. Eventually either a remainder of zero occurs (in which case the number is not prime), or the number on the second row contains more than half as many significant digits as the number on the top row at which point the original number is declared

[1]Although the above description has been based on the idea of a read/write head reading values off the various rows of cells above it and making decisions as to where to move, what state to take on and what to write, based on what it has read, the reality of the implementation is somewhat different. The description is written in terms of what a viewer might consider to be happening, but in fact the head does not move at all but simply passes a message to an adjacent cell to say that it should become the head in the next generation; the state that the machine enters depends on what other messages that cell received from the three "*tapes*".

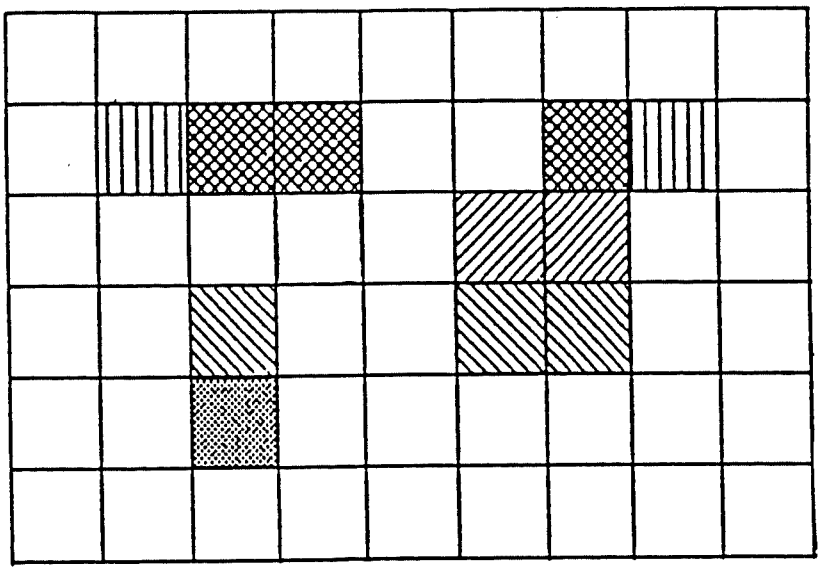


Figure 7.9: A cellular automaton simulating the action of a Turing machine to recognise prime numbers. The pattern of cells on the top line represents the (binary) number under investigation (25 in this case); the next row down contains a potential divisor (in this case the number 3) and the third row contains the result of repeated subtraction of one from the other. On the bottom row is the read/write head which passes backwards and forwards under the numbers performing the processing defined in the rules.

prime. The reason for stopping when the binary number on the second row becomes more than half as long as the number under investigation is that at that point the square of the number on the second row contains more digits than the number under investigation and so cannot be a factor of it (we have already considered all multiples of the middle number less than the square).

7.4.3 Illustration of the operation of electronic circuits

Several other experiments were performed, using different rules, in order to attempt to produce different styles of animation. One particularly interesting example concerned the simulation of stylised electronic circuits. Rules were devised which allowed cells to represent NAND, NOR, AND, OR and Exclusive OR gates; other cells represented the logic values 0 and 1 (both static and moving across the grid in various directions); finally there were cells which allowed the manipulation of moving streams of these 0s and 1s—turning streams through right angles, allowing them to cross over each other, eating up logic values to prevent them proceeding further, and so on. From these components, it is possible to build circuits and watch as streams of logic values pass through the circuit and are acted upon by the various boolean operators. These streams of pulses do not need wires to travel along, they simply need to be pointed in the right direction; strategically placed manipulation cells will take care of any adjustments—much in the same way as light is guided using mirrors. As these

streams of logic values pass by the sides of the cells representing the boolean operators they pass messages to those cells, which then decide which logic value to emit at their output(s). This message passing occurs when the logic values are immediately adjacent to the gate cells; each gate therefore has four potential inputs (although in practice it is difficult to use more than three of them due to the practical difficulties of having four input streams which do not interfere with each other). Four possible outputs are also provided for each gate, any number of which may be enabled by placing an enabling cell next to them; these outputs are positioned three or four grid positions away from the boolean gates to allow the input streams free access to the input positions next to the gates.

Quite complex circuits have been simulated using this automaton including a 4-bit adder which took eight streams of binary digits, representing successions of pairs of 4-bit numbers, as inputs and added them together to produce a stream of 5-bit results a few steps later. It is both interesting and instructive to watch the streams of pulses passing through the "circuitry" and being acted upon by the gates to produce the required result.

7.5 Implementation details

As mentioned earlier, the playing surface implemented for this cellular automaton program is over sixty five thousand cells square. It is likely, however, that only a small fraction of these cells will contain life-forms at any one time. Since it is obviously impractical to allocate an array with over four thousand million entries it is necessary to use sparse matrix techniques. An outline of the data structures and algorithms used is given below. Since it is desirable that the software should run as quickly as possible, all storage management is performed by the software in order to make it as efficient as possible.

Let us begin by considering the data structure used to hold the state of he grid at any moment. A record of the positions and types of living cells present in any particular generation is maintained by keeping a linked list of the rows which contain living cells, together with a list of positions of living cells within the row. This second list is actually organised as a simple vector (pointed to by the row links), whose entries contain the position and type of cells within the row. This saves space since pointers are not needed, and it allows the list to be traversed rapidly in either direction. Figure 7.10 shows an example of this data structure in use. Note that although a BCPL word contains 32 bits many of the values are stored in 16 bit fields (dibytes); the compiler used for this experiment allowed efficient access to these dibytes by use of the %% operator—for example, x%%0 refers to the 16 bits immediately following address x and x%%1 refers to the next 16 bits.

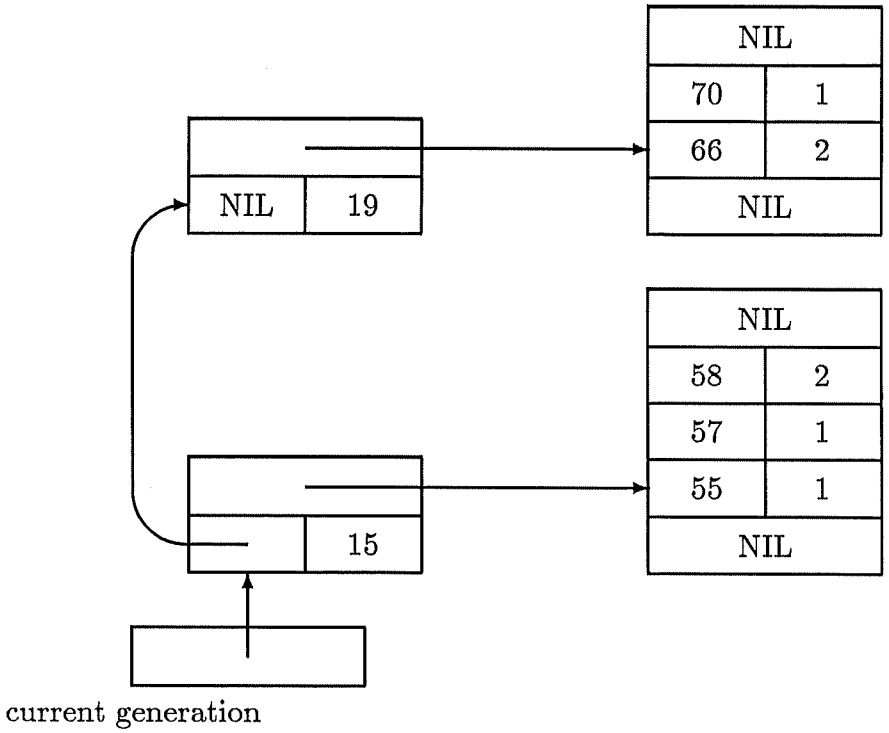


Figure 7.10:
The data structure used to represent the distribution of living cells in a particular generation. In this example there are cells of type 1 at locations (55,15), (57,15) and (70,19); cells of type 2 occupy positions (58,15) and (66,19).

At all times the currently displayed generation will be held in a data structure like figure 7.10. There will also be a similar data structure under construction for the next generation. When the time comes to display the next generation the *current* and *next* pointers are merely swapped over and the memory taken up by the old data structure is re-used.

Let us now consider the method by which the values for the next generation are calculated. The basic idea is that several lines of workspace conceptually start at the top of the grid holding the current generation and move downwards in unison a row at a time receiving messages from the cells in the row currently under consideration as they go. After all messages which can affect a particular line have been received the messages are interpreted, and the new values of living cells for that row in the next generation are written into the data structure. There must be one line of workspace representing the current row under consideration and a number of lines representing the previous and next rows on the grid; the precise number of lines of workspace needed depends on the furthest distance that messages can be sent above or below the cell sending them (as specified in the rules). Each time a row is processed the topmost line of workspace has its messages interpreted, after which all of its entries can be discarded. The lines of workspace are then cycled—the old topmost row thus appears as a (now empty) line at the bottom.

Let us consider the *Game of Life* as an example: a set of possible simulation rules was given in figure 7.6. Since messages are sent a maximum of one cell above or below it is only necessary to have three lines of workspace—one for the row under consideration and one each for the rows immediately above and below—I will therefore refer to these lines of workspace as *Above*, *Current* and *Below*. When beginning to process row **n** in the data structure for the current generation it will always be the case that *Below* is empty since no messages can yet have been sent to row **n+1**; *Above* and *Current* may contain entries due to having received messages during the processing of previous rows (*Above*, for example, may have received messages during the processing of rows **n-1** and **n-2**). Consider the following situation:-

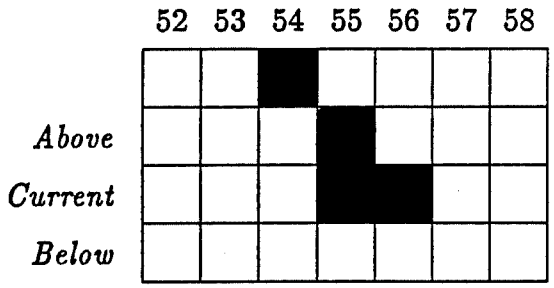


Figure 7.11 shows the situation just prior to processing the current row and figure 7.12 shows the situation immediately after processing the row; it is now the case that *Above* has received all the messages it ever can so the messages can be interpreted to determine the living cells for the next generation. The pointers are then changed so that the old *Below* becomes the new *Current*, the old *Current* becomes the new *Above* and *Below* is reset to NIL before the whole process is repeated for row **n+1**.

This method of working makes good use of the available memory space since only a minimal amount of workspace is needed to hold the messages from the currently active rows—the more obvious alternative of passing all messages for the entire grid prior to interpretation was rejected due to the amount of space needed for the linked lists, particularly due to the fact that many more cells are likely to receive messages than are likely to be alive in the next generation.

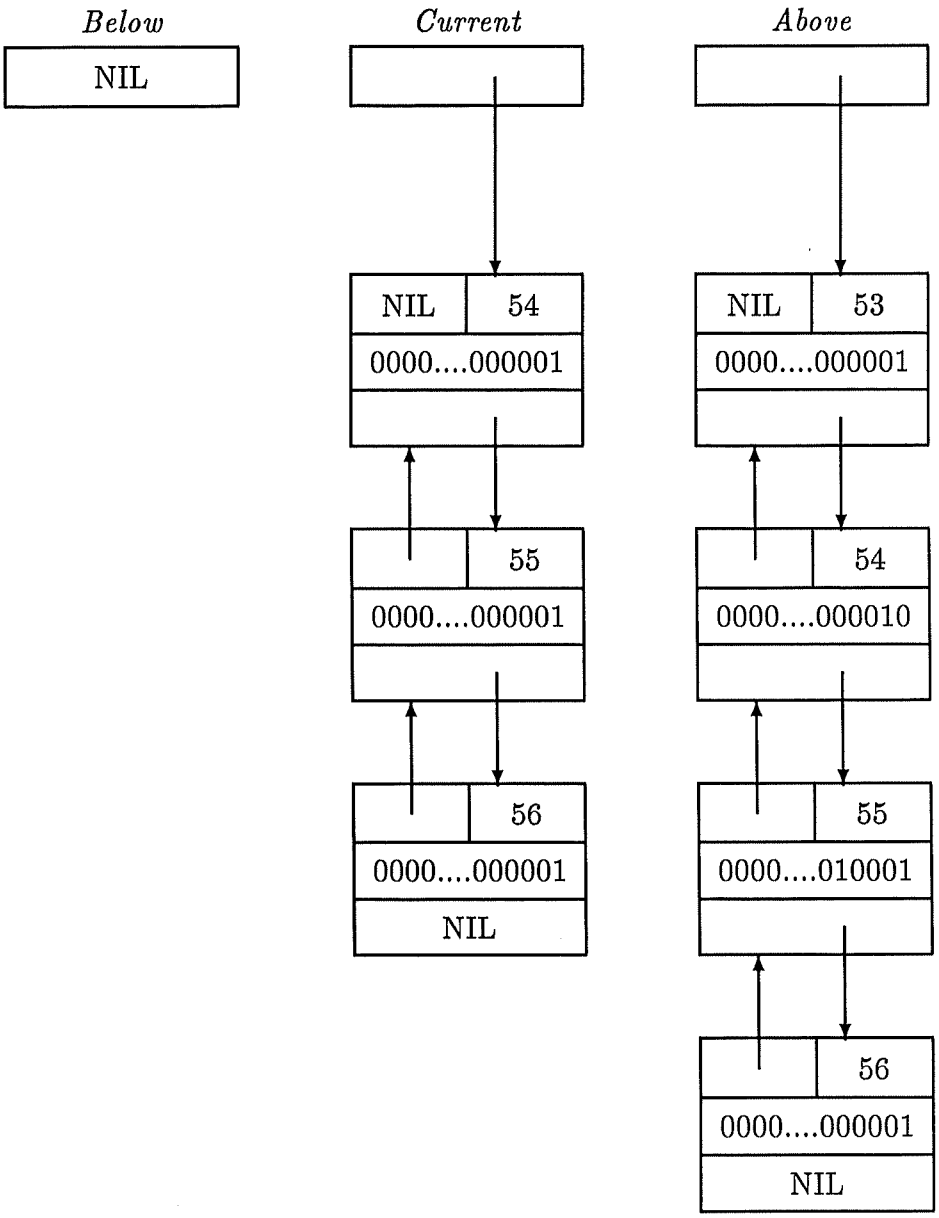


Figure 7.11: The state of the workspace immediately prior to processing a row in the current generation. Each entry in the linked list consists of a pointer to the previous entry, an x coordinate within the row, a message area and a pointer to the next entry.

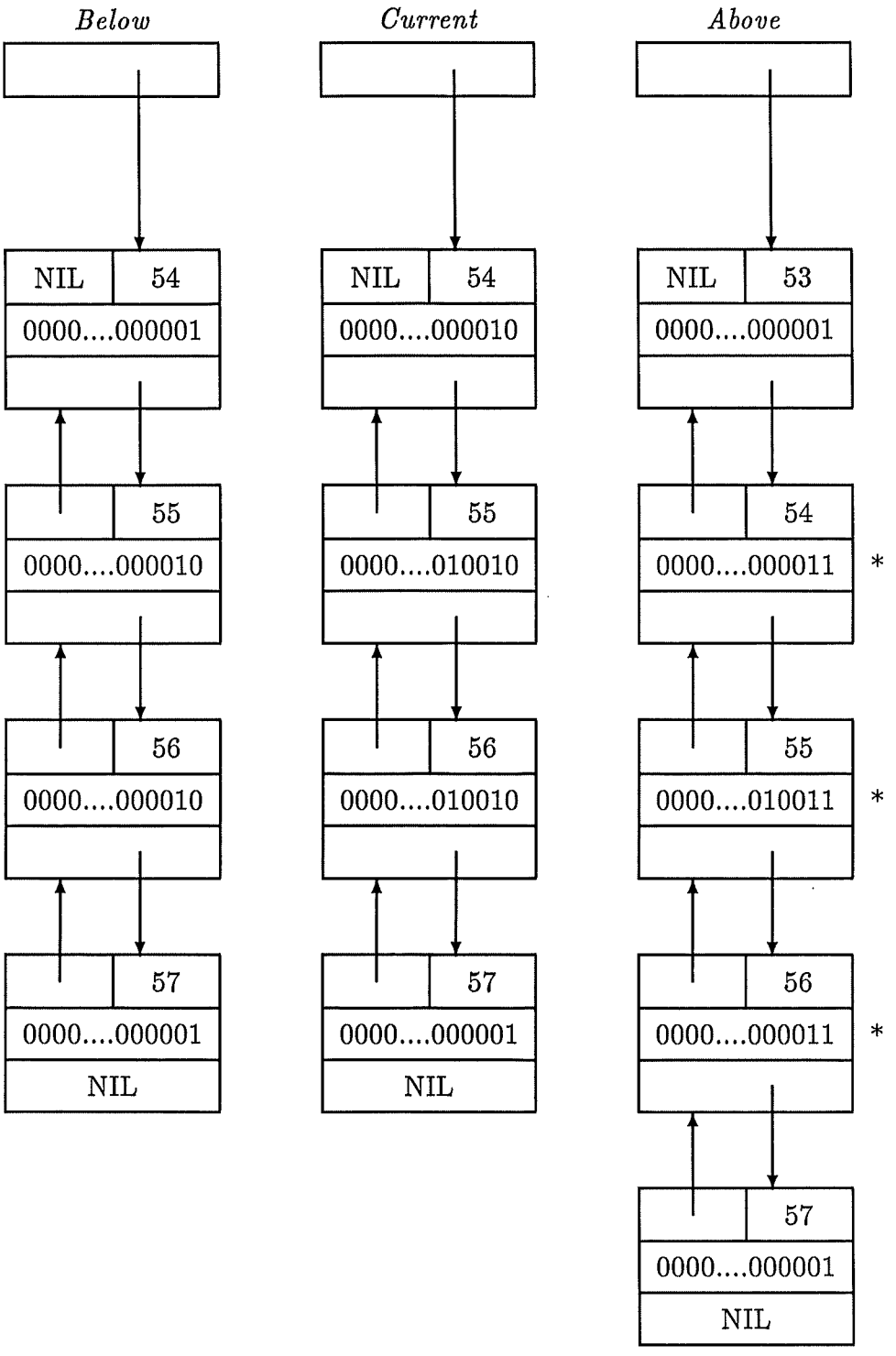


Figure 7.12: The state of the workspace after processing the row—this row is assumed to contain living cells at positions 55 and 56 within the row. The next stage is to interpret the messages received by the elements in 'Above'—those cells marked with an asterisk will be alive in the next generation.

7.6 Can Cellular Automata be used to Produce Larger Scale Animation?

It was hoped, when this experiment was initiated, that it would be possible to produce animation on a global scale by defining rules at the cell level. The rationale behind this hope was that there are objects in the real world whose behaviour may, to a large extent, be attributed to small scale interaction between subcomponents within the object. Soap bubbles are an example of one such object, the spherical shape that bubbles take on is the result of an attempt to minimise surface tension and surface area whilst maintaining a very small pressure gradient between inside and outside the bubble—it should therefore be possible to take a number of cells (representing a volume of air) surrounded by an arbitrary continuous boundary and define rules which attempt to keep the "air molecules" a certain distance apart, and in some sense look at the shape formed by the soap in the surrounding area of the bubble's surface in order to make local adjustments so as to minimise the surface area. Gradually the effects of these local adjustments would spread around the surface of the bubble and cause it to take on its characteristic shape.

This has turned out to be harder than originally anticipated. Some examples of animation at the macro level have been achieved—including such examples as *Gosper's glider gun* and my 4-bit adder; but it has proved difficult to consistently produce rules with interesting and predictable global effects.

7.7 Conclusion

An aim of this experiment with cellular automata was to develop an extensible artist-oriented system, one in which it was possible to adjust the rules guiding the simulation in order to guide the animation towards the desired outcome. I believe that this has been achieved since, when using the system described in this chapter, the animator has two distinct ways of affecting the course of the animation: by altering the patterns placed on the grid; or changing the rules guiding the development of the sequence. Neither of these operations places excessive technical demands on the artist.

From the results of the experiments described in this chapter it would appear that cellular automata are surprisingly well suited to certain styles of animation—particularly instructive sequences used to illustrate mathematical or scientific principles (such as my Turing machine and logic function examples)—but much less suited to animation of less regular objects since it is difficult to develop suitable rules such that application of these rules at the cell level gives rise to useful, meaningful motion on a larger scale with any hint of the subtlety and sophistication of traditional animation. A difference between these two types of globally-interesting animation is that in the case of Turing machines and logic functions, the individual cells are still important and may readily be identified by the viewer, whereas it is more difficult (with the soft-

ware as described) to produce rules such that it is only the global effect which remains consistent, the individual cells are unimportant. In some ways this second style of animation is quite similar to the original experiments with the *Game of Life*, since the interesting, moving patterns produced by this automaton metamorphose as they move, the individual cells do not persist. However, it would not have been practical to have expected an animator to have developed the rules for the *Game of Life* in order to produce an animation of a *megapuffer*—or even a simple *glider*.

Chapter 8

Review

> *... in real life mistakes are likely to be irrevocable. Computer simulation, however, makes it economically practical to make mistakes on purpose. If you are astute, therefore, you can learn much more than they cost. Furthermore, if you are at all discreet, no one but you need ever know you made a mistake.*
>
> John McLeod and John Osborn,
> *Natural Automata and Useful Simulations,* Macmillan, 1966.

This thesis began with a description of how animators traditionally perform the animation process and asked whether this process could be automated. I believe that it has been shown that the general answer to this question is "not in the way that was originally intended".

It was originally hoped that the computer would take the place of the *in-betweener* and produce the images for the frames between the significant events (key frames) drawn by the main animator. My own experience with *KAS* (described in chapter 3), and other people's experiences with similar systems have shown that this is not generally the case. The main reason for this is that the human in-betweener understands the three-dimensional qualities of the objects being animated, whereas the computer treats keyframes only as flat, two-dimensional drawings; it is therefore very difficult to animate three-dimensional objects in this way since too much information has been lost.

That is not to say that the computer is of no use to the traditional keyframe process. Great speed benefits can be realised by using the computer to electronically colour frames which have been digitised from hand drawings, for example. However, this does not address the central problem of computer animation—that of computer-generated motion.

The desire to use the computer for motion development has prompted a number of different attempts at solutions, each characterised by its own particular style of working. Almost all of these solutions have recognised the need to work with three-dimensional computer models in order to animate three-dimensional objects. The differences in philosophy and style come from different approaches to the creation and particularly the manipulation of these three-dimensional models. The creation of three-dimensional objects has not

been considered in this thesis and will not be considered now; motion development on the other hand is very relevant to the work presented here.

The approach to motion specification favoured at the New York Institute of Technology's Computer Graphics Laboratory is to extend the idea of keyframing to three dimensions; working with models composed of rigid sections articulated at specific *joints*, they allow the animator to position the object into key poses; positions between these key poses are generated by interpolation of joint angles. This is quite a powerful artist-oriented approach for certain classes of objects but large numbers of key poses are still needed in order to accurately control complex motion—such as a human figure walking.

The other major approach, exemplified by work at the Massachussetts Institute of Technology and the Université de Montréal, is the *actor and script* technique in which objects are manipulated by the use of textual descriptions of how they should move. This has the advantage that complex motions can be specified as "procedures" and then accessed by simple commands. However such systems tend to be more attractive to the computer scientist than the animation artist since the scripts bear more resemblance to computer programs than the kind of script that an animator may be used to. The technique also takes no advantage of the artist's traditional skills of hand and eye.

The main objective of my own research has been to evaluate a different style of working—an interactive system which is intended to be accessible to the animation artist and draw on his visual skills, but provide some of the power of the actor and script approach, including straightforward control of complex motions. The aim was to allow as much of the interaction as possible to be performed by graphical means—only resorting to textual descriptions and programming when absolutely necessary.

Two major experiments were performed to test this idea:

1. the movement of balls on a billiard table in the game of snooker; and
2. the development of motion for patterns of cells on a grid in the form of a cellular automaton.

Snooker

The snooker experiment incorporated a range of powerful tools to enable the animator to quickly and accurately develop an animated sequence depicting any game of snooker he may desire (in fact it was also found possible, with minor adjustments to the software, to adapt the animation to other environments with similar rules of motion to balls on a billiard table). A reasonably accurate simulation of the physics of the situation was built into the software, freeing the animator from the need to worry about making shots "look realistic" so that he could direct his attention to more artistic considerations. The motion predicted by the simulation is, of course, only intended as a guide for the animator's benefit—there is no compulsion on the animator to follow the simulation, he can interrupt the generation of frames at any time and alter positions and parameters in order to guide the animation towards the desired conclusion.

Facilities are provided not only to model the motion of balls after they have started moving but also to calculate a suitable initial velocity in order to achieve a particular objective—a simple form of *goal-directed animation.*

The use of immediate visual feedback and the ease with which actions can be undone—by returning to an earlier frame (with all of its associated state information) in order to alter the autcome—was intended to encourage the artist to adopt an experimental "what if ..." approach and develop an animated sequence by a series of refinements.

As an initial experiment, snooker was quite successful, but it certainly had significant deficiencies:

- the code to simulate the realistic motion of the balls was built into the software so the implementation of any changes which had not been envisaged when the program was written needed programmer assistance;

- the "world" was too simple—motion was limited to two dimensions (although the objects themselves were three-dimensional), and the only type of moving object was a ball. All balls followed the same simulation rules and reacted to the same fixed environment.

On the plus side the style of working was found to be quite agreeable and the program satisfied virtually every demand made of it in terms of ease of developing a sequence of shots; the tools provided seemed to be just what was required.

Cellular automata

The experiment with cellular automata was aimed at overcoming some of the shortcomings of the snooker experiment. In particular, it was intended to avoid having the rules of motion built into the software; a simple textual description is all that is needed to define a new set of rules (the rules can even be changed as the program is running). The experiment was also interesting as a use of *actors*[1]—the cells on the grid of a cellular automaton, as described in chapter 7, behave exactly as Hewitt actors.

The character of this experiment was more abstract than that of snooker, the aim was not so much to produce precise, naturalistic animation as to experiment with techniques which appeared to give a more random effect; this is what Lesley Keen (the professional animator with whom I had worked on the film *Taking a Line for a Walk*) had requested. Lesley claimed that the way in which a computer could be of most use to her was to throw up unexpected results since she was already good at producing carefully controlled sequences by conventional means.

That is not to say that the results produced using cellular automata are random, it is just that they are difficult to predict due to their complexity. Given sufficient thought, rules can be devised to produce particular effects but

[1]Hewitt defines an actor as an object which can send and receive messages and perform actions based on the messages it has received [*Hewitt, 1971*]—see also chapter 4

it can be very difficult to devise suitable rules—who, for example, would think of devising the rules of the *Game of Life* in order to produce the animation of a *glider* (see section 7.2). Other sets of rules, however, may have more obvious and predictable effects, as illustrated by the *logic gates* and *Turing Machine* examples in section 7.4.

It was hoped, when the experiment was initiated, that it would be possible to produce animations of quite complex shapes using simple rules applied to individual cells—the whole being greater than the sum of the parts. This has proved to be the case on a limited scale—*Gosper's Glider Gun* in the *Game of Life*, for example, is a considerably more complex object than its constituent parts, but attempts to produce artist-controllable animation on a large scale where each cell in the array becomes a single, semi-intelligent pixel on the final display (the animation of bubbles floating around the screen and bumping into each other, for example) have met with a singular lack of success. It may be that with more effort some headway could be made, but I believe that the problems are too great to make the technique useful in a general context.

A particular success of the program was that, as with snooker, the graphical interface was found to be straightforward and pleasant to use—the method of editing cells was found to be particularly useful and has since been adopted in other situations which require the editing of cells on a grid, such as editing founts for display on a raster graphics terminal.

Using simulation techniques, an animated sequence is guided towards its conclusion rather than having all motion precisely pre-defined. As the sequence progresses there may be rules which allow objects to interact with each other and their environment. Without these rules such interactions would have to be explicitly scripted or keyframed—this manual specification may not be easy. Using keyframes or scripts to produce an animation of a person skiing down a mountainside, for example, would be a difficult task since it would not be easy to ensure that the skier remained in contact with the surface of the snow; using simulation on the other hand makes the job relatively simple once a satisfactory simulation model of a skier on a mountainside has been developed. Of course the difficulties of producing a simulation model have to be weighed against the final ease of production of the animation, but it must be remembered that the simulation model may be of use for more than a single animated sequence.

I believe that the use of simulation can significantly simplify and speed up the planning of the motion for an animated sequence. If the animator wishes to have more precise control of the course of the animation, the rules guiding the motion may be discarded, leaving the animator to specify movements directly.

8.1 And what next?

As a next stage in the development of the cellular automaton experiment it may be useful to allow global message passing (broadcasting) rather than insisting that messages should only be passed to a cell's near neighbours. This would be useful in many situations; for example, to initiate a change of operating mode

once a particular condition had been satisfied. It could easily be implemented as follows:

- ascertain whether any cells in the current generation wish to broadcast messages to all cells; and if so, what the messages are;
- send all local messages, then
- send any broadcast messages identified above to all cells which received any local messages;
- finally interpret any messages received to produce the next generation.

Note that broadcast messages are only sent to cells which receive local messages; this helps to keep the number of cells receiving messages to an acceptable number (remember that all cells receiving messages need these messages storing and processing). The fact that only a limited number of cells actually receive messages intended for broadcast to all cells is easily justified on the grounds that for practical reasons of limitations of memory space it is inconceivable that a cell receiving **only** a broadcast message should be a living cell in the next generation—there are just too many cells on the grid for this to be acceptable. Note also that it is necessary to determine whether broadcast messages are needed prior to local message passing—this is due to the way that the grid is processed in order to produce the next generation (as described in the previous chapter): rows of the grid are processed from the top downwards and messages for line n are interpreted as soon as processing has progressed to line n+k+1 (where k is the maximum number rows higher in the grid that a local message can be sent). Thus some rows may already have been processed by the time the broadcasting cell is encountered.

It may also be advantageous in certain circumstances to allow random, or semi-random behaviour to be built into the rules guiding the cellular automaton. This could easily be achieved by amending the rules used to interpret the messages received in the following way:

rather than saying

```
10001      3
```

to mean that a cell receiving the message `10001` should become a life-form of type 3 in the next generation, we could say

```
10001      50% 3      40% 6      10% 0
```

to mean that having received message `10001` there is a 50% probability that the next life-form in the cell should be of type 3, a 40% probability that it should be of type 6 and a 10% probability that it should be of type 0 (dead).

The animator would also have to decide whether or not the random numbers used to make the choice should be repeatable. If the numbers are truly random different runs of the program starting from the same grid configuration will give different results; this contradicts the idea of being able to replay a sequence by using the log file as a script; it may therefore be the case that the animator

would prefer the 'random' numbers to be related to some re-producible condition such as the position of the cell on the grid (or some specified seed value from which to generate pseudo random numbers).

A further interesting extension to the cellular automaton experiment would be to provide a three-dimensional picture editor to allow cells to be represented in more complex ways. There is no reason why cells should be displayed as coloured squares; each different type of cell could be represented as some three-dimensional image (such as a chessman) whose motion was determined by the rules of the cellular automaton but whose appearance belied this fact. Indeed it may be possible, under certain circumstances, to have each cell as an animated object in its own right (a human figure for example), thus performing the animation on two levels—using cellular automaton rules to define movement and interaction between "actors" but with each actor undergoing its own independent motion.

I believe that the experiments into rule-based aids to motion development have been relatively successful and suggest that further experimentation is worthwhile. It should always be born in mind during this further research that artist control is an essential ingredient of any animation system that is to be anything more than a technical exercise. A common failing among computer scientists is to build systems which are packed with clever features but lack the necessary non-technical interfaces to allow them to be used to their artistic best. In the final analysis it is the quality of the animated sequence produced which matters. The use of interactive systems incorporating immediate visual feedback encourages choices based on aesthetic values rather than implementation considerations.

It should be remembered that the dictionary definition of the verb *to animate*, is to *"breath life into"*; current use of the term in computer graphics circles has reduced the meaning to *"create the illusion of motion"*—perhaps when artistic control is removed from scientists and given back to animators, their spark of artistic creativity can restore the two meanings to much the same thing.

References

Ackland B. & Weste N.
[1981] *The edge flag algorithm—a fill method for raster scan displays*
IEEE Transactions on Computers, Vol. C-30: pp. 41–48

Alexander S. & Huggins W.H.
[1967] *User's manual on PMACRO*
John Hopkins University

Angell I.O.
[1981] *A practical introduction to computer graphics*
Macmillan Computer Science Series

Baecker R.M.
[1969] *Interactive computer-mediated animation*
(PhD Dissertation). MIT, Project Mac-Tr-61

Baecker R.M.
[1969b] *Picture-driven animation*
Proc. Spring Joint Computer Conference, AFIPS Press,
Vol. 34: pp. 273–288

Blinn J.F.
[1977] *Models of light reflection for computer synthesized pictures*
Proc. SIGGRAPH '77: Computer Graphics, Vol. 11, No. 2:
pp. 192–198

Blinn J.F.
[1978] *Simulation of wrinkled surfaces*
Proc. SIGGRAPH '78: Computer Graphics, Vol. 12, No. 3:
pp. 286–292

Blinn J.F. & Newell M.E.
[1976] *Texture and reflection in computer generated images*
Communications of the ACM, Vol. 19: pp. 542–547

Bonvoisin N.J.
[1984] *Shading three-dimensional images for computer animation*
CST Project Report; Cambridge University Computer
Laboratory

Bresenham J.E.
[1965] *Algorithm for computer control of a digital plotter*
IBM Systems Journal, Vol. 4, No. 1: pp. 25–30

Brewer J.A. & Anderson D.C.
[1977] *Visual interaction with Overhauser curves and surfaces*
Proc. SIGGRAPH '77: Computer Graphics, Vol. 11, No. 2

Burtnyk N. & Wein M.
[1971] *Computer-generated key-frame animation*
Journal of the Society for Motion Picture and Television Engineers, Vol. 80: pp. 149–153

Burtnyk N. & Wein M.
[1971b] *A computer animation system for the animator*
Proc. UAIDE 10th Annual Meeting

Burtnyk N. & Wein M.
[1976] *Interactive skeleton techniques for enhancing motion dynamics in key frame animation*
Communications of the ACM, Vol. 19, No. 10: pp. 564–569

Catmull E.
[1972] *A system for computer-generated movies*
Proc. ACM Annual Conference: pp. 422–431

Catmull E.
[1978] *The problems of computer-assisted animation*
Proc. SIGGRAPH '78: Computer Graphics, Vol. 12, No. 3: pp. 348–353

Catmull E.
[1979] *New frontiers in computer animation*
American Cinematographer, October issue

Catmull E. & Rom R.
[1974] *A class of local interpolating splines*
Computer-Aided Geometric Design, Academic Press, San Fransisco

Chapman M.K.
[1984] *Colour mixing and "painting" for use in computer animation*
CST Project Report; Cambridge University Computer Laboratory

Cook R.L. & Torrance K.E.
[1982] *A reflectance model for computer graphics*
ACM Transactions on Graphics, Vol. 1, No. 1: pp. 7–24

Cook R.L., Porter T. & Carpenter L.
[1984] *Distributed ray tracing*
Proc. SIGGRAPH '84: Computer Graphics, Vol. 18, No. 3

Coons S.A.
[1974] *Surface patches and B-spline curves.*
Computer Aided Geometric Design, Academic Press

Crow F.C.
[1977] *The aliasing problem in computer-generated shaded images*
Communications of the ACM, Vol. 20, No. 11: pp. 799–805

Crow F.C.
[1981] *A comparison of anti-aliasing techniques*
IEEE Computer Graphics and Applications, Vol. 1, No. 1: pp. 40–48

Csuri C.
[1970] *Real-time film animation*
Proc. 9th UAIDE Annual Meeting, pp. 289–305

Csuri C.
[1974] *Real-time computer animation*
Proc. IFIP Congress '74, North-Holland: pp. 707–711

Csuri C.
[1975] *Computer animation*
Proc. SIGGRAPH '75: Vol. 9, No. 2: pp. 92–101

Csuri C., Hackathorn R., Parent R., Carlson W. & Howard M.
[1979] *Towards an interactive high visual complexity animation system*
Proc. SIGGRAPH '79: Computer Graphics, Vol. 13, No. 2: pp. 289–299

DeFanti T.
[1976] *The digital component of the circle graphics habitat*
Proc. National Computer Conference '76: pp. 195–203

Duff T.
[1983] *Computer graphics in the biggest box office hit: Return of the Jedi*
Proc. Computer Graphics '83, Online Conf.: pp. 283–289

Foldes P.
[1974] *Hunger: a 12 minute computer animated film*
National Film Board of Canada

Foley J.D. & Van Dam A.
[1982] *Fundamentals of Interactive Computer Graphics*
Addison Wesley

Fournier A., Fussell D., & Carpenter L.
[1982] *Computer rendering of stochastic models*
Communications of the ACM, Vol. 25, No. 6

Gardner M.
[1970] *The Game of Life*
Mathematical Games, Scientific American, Vol. 223, No. 4:
pp. 120–123

Glauert T.H. & Wiseman N.E.
[1985] *Real-time image combination*
Proc. MICAD '85: pp. 68–73

Gouraud H.
[1971] *Continuous shading of curved surfaces*
IEEE Transactions on Computers, Vol. C-20, No. 6:
pp. 623–629

Hackathorn R.
[1977] *ANIMA II: A 3-D color animation system*
Proc. SIGGRAPH '77: Computer Graphics, Vol. 11, No. 2:
pp. 54–64

Hamming R.W.
[1971] *Applied Numerical Analysis*
McGraw-Hill

Hanrahan P. & Sturman D.
[1984] *Interactive control of parametric models*
Proc. SIGGRAPH '84: Computer Graphics, Vol. 18, No. 3

Hewitt C.
[1971] *Description and theoretical analysis (using schemata) of PLANNER: a language for proving theorems and manipulating models for a robot*
(PhD Dissertation). MIT

Hewitt C., Bishop P. & Steiger R.
[1973] *A universal modular actor formalism for artificial intelligence*
Proc. International Joint Conference on Artificial Intelligence: pp. 235–245

Honey F.J.
[1971] *Artist orientated computer animation*
Journal of the Society of Motion Picture and Television Engineers, Vol. 80, No. 3: pg. 154

Honey F.J.
[1976] *Computer animated episodes by single axis rotations*
Proc. 10th UAIDE Annual Meeting

Kahn K.M.
[1976] *An actor-based computer animation language*
MIT AI Working Paper No. 120

Kitching A.
[1973] *Computer animation—some new ANTICS*
British Kinematography Sound Television Journal, Vol. 55, No. 12: pp. 372–386

Knowlton K.C.
[1964] *A computer technique for producing animated movies*
Proc. SJCC AFIPS Conference, Vol. 25: pp. 67–87

Knowlton K.C.
[1965] *Computer-produced movies*
Science, Vol. 150: pp. 1116–1120

Knowlton K.C.
[1970] *EXPLOR—A generator of images*
Proc. 9th UAIDE Annual Meeting: pp. 543–583

Kochanek D. & Bartels R.
[1984] *Interpolating splines with local tension, continuity and bias control*
Proc. SIGGRAPH '84: Computer Graphics, Vol. 18, No. 3

Korein J. & Badler N.I.
[1983] *Temporal anti-aliasing in computer generated animation*
Proc. SIGGRAPH '83: Computer Graphics, Vol. 17, No. 3: pp. 377–388

Magnenat-Thalmann N. & Thalmann D.
[1983] *MIRA-3D: A three-dimensional graphical extention of PASCAL*
Software—Practice and Experience, Vol. 13: pp. 797–808

Magnenat-Thalmann N. & Thalmann D.
[1984] *3D shaded director-orientated computer animation*
Proc. Graphics Interface '84, Ottawa

Magnenat-Thalmann N. & Thalmann D.
[1985] *Computer Animation, Theory and Practice*
Springer-Verlag

Magnenat-Thalmann N., Thalmann D. & Fortin M.
[1985] *MIRANIM: An extensible director-oriented system for the animation of realistic images*
IEEE Computer Graphics and Applications, Vol. 5, No. 3

Mandelbrot B.B.
[1975] *Stochastic models for the earth's relief, the shape and fractal dimension of coastlines, and the number area rule for islands*
Proc. National Acad-Sc. USA, Vol. 72, No. 10: pp. 2825–2828

Mandelbrot B.B.
[1982] *The fractal geometry of nature*
Freeman, San Fransisco

Moody K. & Richards M.
[1980] *A coroutine mechanism for BCPL*
Software—Practice and Experience, Vol. 10: pg. 765

Needham R.M. & Herbert A.J.
[1982] *The Cambridge Distributed System*
Addison Wesley

Newman W.M. & Sproull R.F.
[1979] *Principles of Interactive Computer Graphics*
McGraw-Hill

Papert S.
[1970] *Teaching children thinking*
Proc. IFIP World Conference on Computer Education, New York: pp. I/73–I/78

Phong B.T.
[1975] *Illumination for computer generated pictures*
Communications of the ACM, Vol. 18, No. 6: pp. 311–317

Pitteway M.L.V. & Watkinson D.J.
[1980] *Bresenham's algorithm with grey scale*
Communications of the ACM, Vol. 23, No. 11: pp 625–626

Porter T. & Duff T.
[1984] *Compositing digital images*
Proc. SIGGRAPH '84: Computer Graphics, Vol. 18, No. 3

Potmesil M. & Chakavarty I.
[1983] *Modelling motion blur in computer-generated images*
Proc. SIGGRAPH '83: Computer Graphics, Vol. 17, No. 3: pp. 389–399

Reeves W.T.
[1981] *Inbetweening for computer animation utilizing moving point constraints*
Proc. SIGGRAPH '81: Computer Graphics, Vol. 15, No. 3: pp. 263–269

Reeves W.T.
[1983] *Particle systems—a technique for modeling a class of fuzzy objects*
Proc. SIGGRAPH '83: Computer Graphics, Vol. 17, No. 3: pp. 359–376

Reynolds C.W.
[1982] *Computer animation with scripts and actors*
Proc. SIGGRAPH '82: Computer Graphics, Vol. 16, No. 3: pp. 289–296

Richards M.J. & Whitby-Strevens C.
[1979] *BCPL—the language and its compiler*
Cambridge University Press

Richards M.J., Aylward A.R., Bond P., Evans R.D. & Knight B.J.
[1979] *TRIPOS—a portable operating system for mini-computers*
Software—Practice and Experience, Vol. 9: pg. 513

Sigma Electronic Systems Limited
[1983] *T5688 User Manual*
Sigmex Limited

Sinden F.W.
[1967] *Synthetic cinematography*
Perspective 7, Vol. 4: pp. 279–289

Smith A.R.
[1978] *PAINT*
Technical Memo No. 7, NYIT

Smith A.R.
[1979] *Tint fill*
Proc. SIGGRAPH '79: Computer Graphics, Vol. 13, No. 2: pp. 276–283

Stern G.
[1979] *Softcel: an application of raster scan graphics to conventional cel animation*
Proc. SIGGRAPH '79: Computer Graphics, Vol. 13, No. 2: pp. 284–288

Stern G.
[1983] *Bbop: a system for 3D key frame figure animation*
SIGGRAPH '83 tutorial 7: pp. 240–243

Stern G.
[1983b] *Bbop: a program for three-dimensional animation*
Proc. Nicograph '83: pp. 403–404

Storey A.
[1984] *Setting up camera angles in 3-D animation*
CST Project Report; Cambridge University Computer Laboratory

Styne B.A., King T.R. & Wiseman N.E.
[1985] *Pad structures for the Rainbow Workstation*
Computer Journal, Vol. 28, No. 1

Talbot P.A., Carr III J.W., Coulter R.C.Jr. & Hwang R.C.
[1971] *Animator: an on-line two-dimensional film animation system*
Communications of the ACM, Vol. 14, No. 4: pp. 251–259

Thalmann D. & Magnenat-Thalmann N.
[1983] *Actor and camera data types in computer animation*
Proc. Graphics Interface '83: pp. 203–210

Thalmann D., Magnenat-Thalmann N. & Bergeron P.
[1982] *Dream flight: a fictional film produced by 3D computer animation*
Proc. Computer Graphics '82, Online Conf.: pp. 353–368

Watkins G.S.
[1970] *A real-time visible surface algorithm*
University of Utah Computer Science Dept.,
UTEC-CSc-70-101, NTIS AD-762004

Whitted T.
[1980] *An improved illumination model for shaded display*
Communications of the ACM, Vol. 23, No. 6: pp. 343–349

Wilkes A.J. & Wiseman N.E.
[1982] *A soft-edged character set and its derivation*
Computer Journal, Vol. 25, No. 1: pp. 140–147

Wilkes A.J., Singer D.W., Gibbons J.J., King T.R., Robinson P. & Wiseman N.E.
[1984] *The Rainbow Workstation*
Computer Journal, Vol. 27, No. 2

Zajac E.E.
[1964] *Computer-made perspective movies as a scientific and communication tool*
Communications of the ACM, Vol. 7, No. 3

Zajac E.E.
[1966] *Film animation by computer*
New Scientist Vol. 29: pp. 271–280

Zeltzer D.
[1982] *Motor control techniques for figure animation*
IEEE Computer Graphics and Applications, Vol. 2, No. 9:
pp. 53–59

Zeltzer D.
[1982b] *Representation of complex animated figures*
Proc. Graphics Interface '82: pp. 205–211

Appendix I

The hardware and software support environment at Cambridge University Computer Laboratory

The work described in this dissertation was undertaken within the Cambridge distributed system [*Needham & Herbert, 1982*]. At the heart of this system is a processor bank consisting of a large number of single user machines and a central file server. In effect most of these machines have only a single peripheral—a connection to the *Cambridge ring*; peripherals such as printers, discs and magnetic tape drives are provided remotely by the network. A user sat at any terminal on the network can run sessions on any one or more of these machines—having first asked for it (or them) to be allocated for his exclusive use. All files that the user may wish to access are available from all of the machines—the central file server means that there is no problem with creating a file on one machine and not being able to access it from another. Figure 1 shows the basic structure of this environment.

All of the machines in the processor bank run Tripos [*Richards et al., 1979*], an operating system which supports multi-tasking with message passing. Tripos is written in BCPL [*Richards & Whitby-Strevens, 1979*] and, until relatively recently, BCPL was the only high-level language which the system supported. All of the software described in this thesis was written in BCPL.

BCPL is a block structured language originally intended for systems programming and compiler writing. An unusual feature of the language is that it has only a single data type: the machine word. Any word can be treated as an integer, a boolean value, a bit pattern or a machine address—a very powerful facility but one which can cause untold problems for the naive user. Complex structures can be built up as *"vectors"* of words, but no bound checking is performed. On the plus side, BCPL allows constant values to be given names so as to make code more readable and allow constants to be changed simply by changing the value assigned to the name, rather than having to search through the code for every occurrence of the constant.

Another useful feature is that a BCPL program can be written as any number of separately compiled sections. Access to variables and procedures between sections is provided by the *global vector* which is a one-dimensional array of values which can be accessed by all sections. At the beginning of each section, names are associated with specific offsets in this vector; this allows each section to retrieve by name the values stored in the global vector. The names used are normally the same in each section, so rather than insisting that the same piece of code is duplicated in different sections, BCPL allows a suitable fragment of code to be imported into each section using the `GET` statement. If the name of a location in the global vector is declared as a procedure, the procedure's entry point will automatically be placed at the relevant location in the global vector. Any number of sections can access each location in the vector thus allowing inter-section communication.

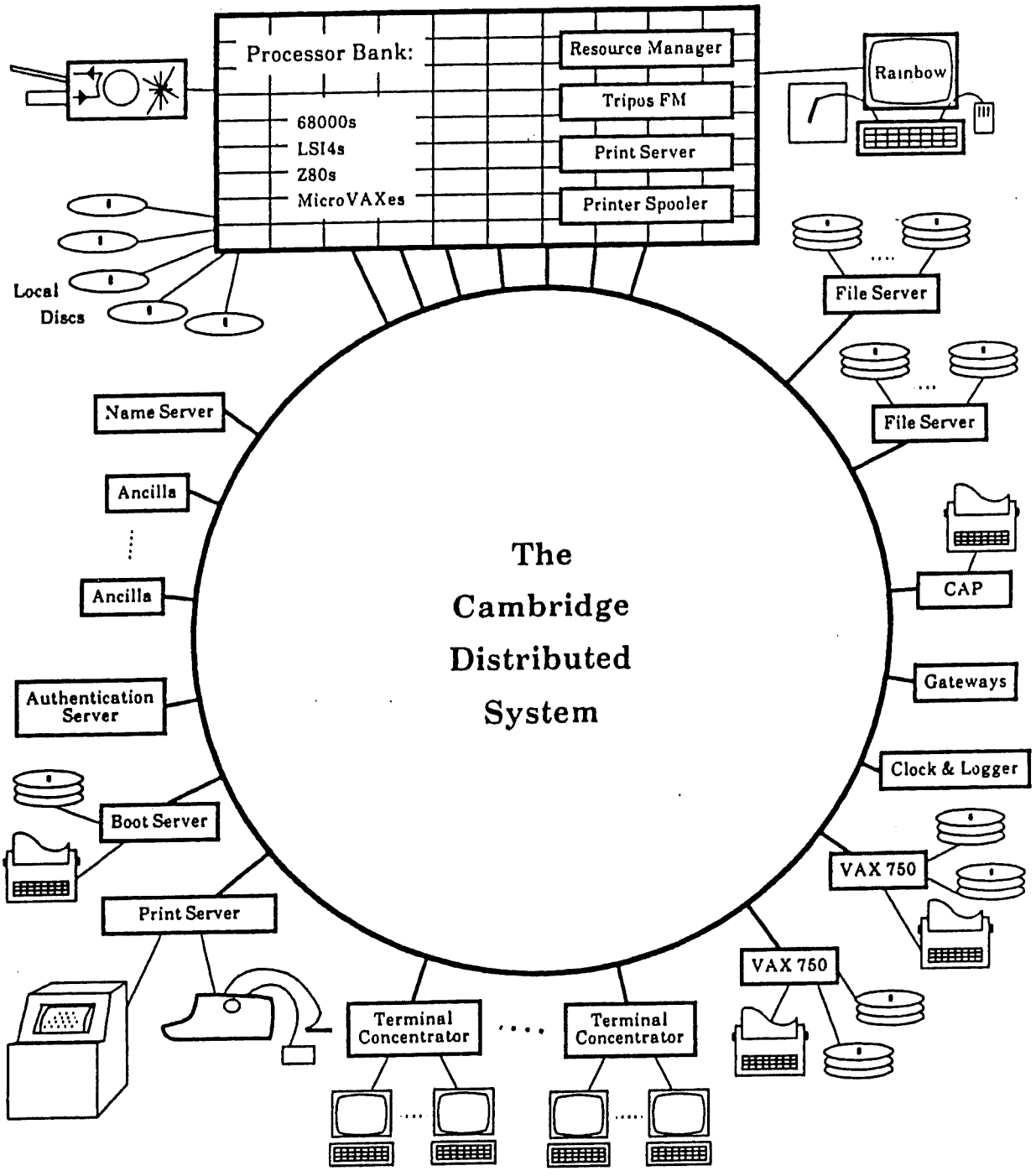


Figure 1: The Cambridge Distributed System (courtesy of Dan Craft).

The Rainbow Workstation

The Rainbow Workstation (henceforth known simply as *Rainbow*) is an experimental graphics workstation designed and built in the Cambridge University Computer Laboratory. Its main novel feature is that it provides hardware support for windowing by dynamically mapping graphics memory to video. Because windows are manipulated by this dynamic mapping mechanism rather than by use of block copying operations (*Bitblt*), full colour windows can be supported without speed penalty. In many ways Rainbow may be considered as a processor bank machine with extra peripherals in the form of graphics hardware, a mouse, tablet, keyboard and colour monitor; as with other processor bank machines it runs the Tripos operating system. The workstation's architecture is described in [*Wilkes et al., 1984*].

As has been mentioned, Rainbow dynamically maps graphics memory to video; the image that appears on the screen is made up of a mosaic of rectangular images, drawn from different areas of graphics memory. Which area of graphics memory should appear where on the screen is determined by a low-level screen description called the *band structure*; it is this description which is interpreted by the video processor.

User programs do not need to calculate band structure directly, instead they manipulate a higher-level screen description which is compiled into band structure whenever anything changes. This higher-level screen description takes the form of a rooted, acyclic, directed graph with the following features:-

- The root node represents the entire screen.
- Leaf nodes (known as *pads*) refer to actual areas of graphics memory. Pads can be specified to have any depth from zero to eight bits; this affects the number of colours they can display. A pad of depth zero is referred to as a *virtual pad* and may only display a single colour; it does not need to occupy space in graphics memory.
- The arcs which are used to map one node into another have a position and priority associated with them; the screen position and relative priority of subnodes depend on these values. It is also possible for arcs to be *turned off* or *turned on* effectively removing or re-introducing the sub-trees below these arcs.
- Internal nodes are referred to as *clusters*; they have a specified size and other nodes can be mapped into them at any position relative to their origin. Images which are mapped into a cluster are clipped to the size of the cluster (this size can be changed at any time). It is possible to change the position or visibility of an entire sub-tree merely by moving the parent cluster or changing its priority.
- There is no explicit restriction on how many times a node may be mapped into another node, so the same image can appear on the screen in several different places.

- The action when pads overlap is determined by the relative priorities of the paths by which they are mapped into the screen. If one priority is greater than the other, the one with the higher priority will obscure the other. If both have the same priority, the pixel values will be concatenated—giving access to a wide range of interaction effects (see below).

A library of procedures is available to help with the creation and destruction of pads and clusters, and to build and manipulate video lookup tables and the graph describing the screen structure. There are also procedures to draw lines, rectangles and polygons into pads.

Figure 2 shows a simple screen layout and the graph structure that must be set up to achieve it.

Rainbow's graphics memory is arranged in eight planes so each pixel can hold a maximum of eight bits of information—up to 256 colours chosen from a palette of sixteen million. That however is not the whole story—the video lookup table actually contains 4096 entries; each separate screen area (pad) can potentially access a separate area of the lookup table thus allowing an independent choice of colours in different pads. Indeed things are even more flexible than that: it is possible to arrange that where pads overlap (assuming they are in distinct planes of graphics memory and are mapped into the screen with the same priority) one pad does not simply obscure the other, but they are in some sense both displayed—the bits of the two images are concatenated to form a deeper pixel value which may be used to index into yet another area of the video lookup table. This facility is often referred to by the general name of *transparency*—although the range of re-colouring and combination effects possible is much more varied than the term transparency implies.

As one example of the use of transparency (suggested by Neil Wiseman) consider a red polygon which has been anti-aliased against a black background and drawn into a five bit deep pad (associated with a 32 entry video lookup table) containing a smooth range of colours from red to black. Consider also a three bit deep pad containing splodges of eight different colours (with colours specified in an eight entry lookup table). If these pads are created in distinct planes of graphics memory and mapped into the screen with the same priority, it is possible to specify that where they overlap eight bit pixel values should be formed by the concatenation of the five bit and three bit values—these pixel values should be used to index into a third (256 entry) lookup table. It is possible to choose the values in this lookup table to give the effect that the red polygon is totally opaque to the 3 bit pad, whereas the black background is totally transparent to it thus letting the splodges of colour show through; the anti-aliasing shades of dark red around the edge of the polygon can be arranged to mix with the colour in the 3 bit pad, so that the red polygon remains correctly anti-aliased regardless of background colour. The two images may be moved around independently in real time and remain correctly anti-aliased against each other without any need for any further computation except the automatic re-calculation of the band structure as the image changes.

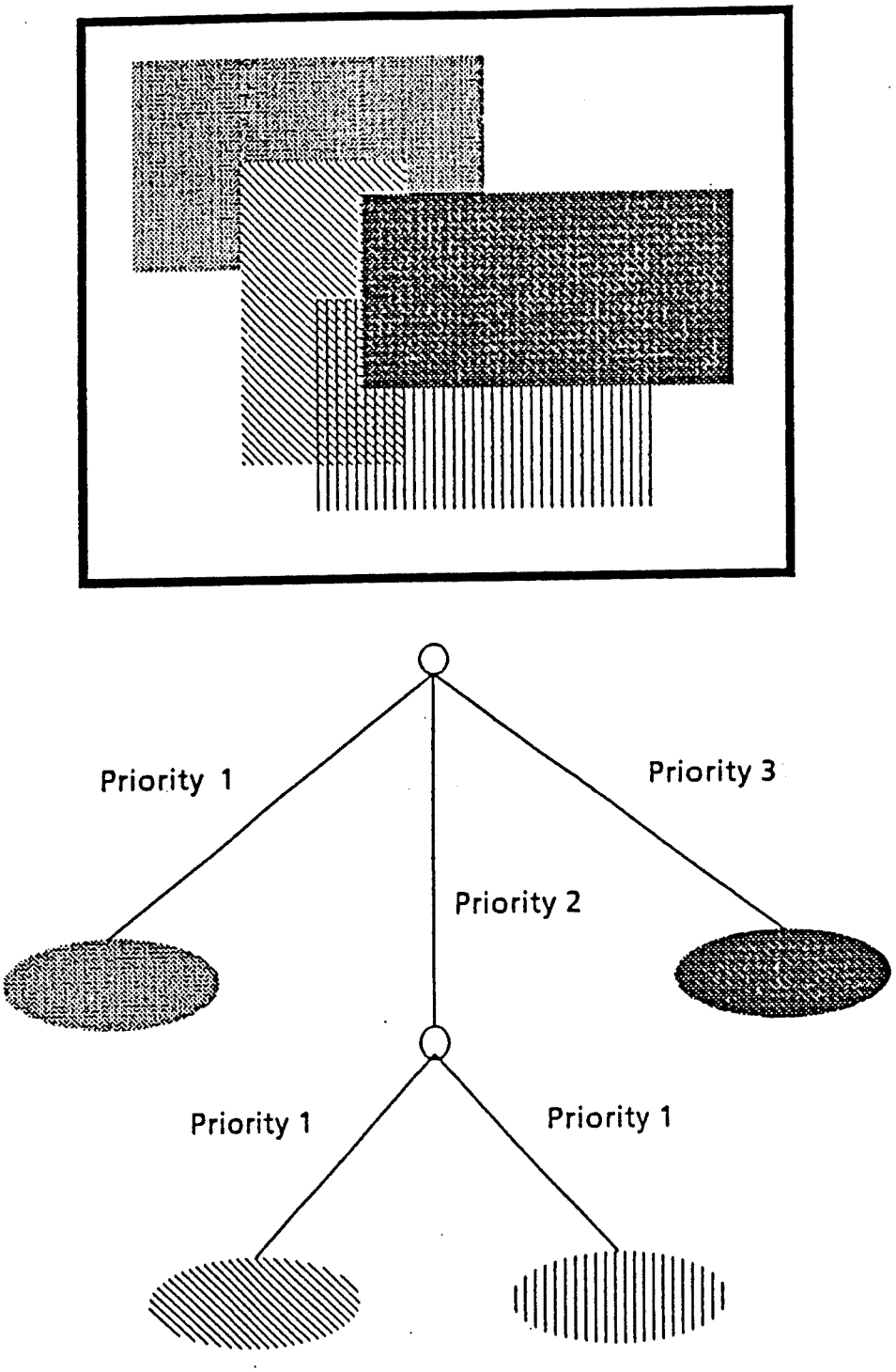


Figure 2: A sample screen and the associated pad structure (courtesy of Kathy Carter).

Another demonstration of transparency was designed and implemented by the author. It incorporates a three-dimensional representation of a shelf unit constructed from semi-transparent components and drawn on four separate pads which are able to move independently in real time. As the shelf unit slides apart (in a way designed to reinforce its three-dimensional appearance), the maintenance of realistic translucence is performed by the lookup tables.

More details of transparency effects are given in [*Styne et al., 1985*] and [*Glauert & Wiseman, 1985*].

References

Glauert T.H. & Wiseman N.E.
[1985] *Real-time image combination*
Proc. MICAD '85: pp. 68–73

Needham R.M. & Herbert A.J.
[1982] *The Cambridge Distributed System*
Addison Wesley

Richards M.J. & Whitby-Strevens C.
[1979] *BCPL—the language and its compiler*
Cambridge University Press

Richards M.J., Aylward A.R., Bond P., Evans R.D. & Knight B.J.
[1979] *TRIPOS—a portable operating system for mini-computers*
Software—Practice and Experience, Vol. 9: pg. 513

Styne B.A., King T.R. & Wiseman N.E.
[1985] *Pad structures for the Rainbow Workstation*
Computer Journal, Vol. 28, No. 1

Wilkes A.J., Singer D.W., Gibbons J.J., King T.R., Robinson P. & Wiseman N.E.
[1984] *The Rainbow Workstation*
Computer Journal, Vol. 27, No. 2

Appendix II

Glossary of terms

Aliasing

The result of discrete sampling of high spatial frequencies (such as those present at a sharp boundary between two colours) is that high frequencies become indistinguishable from lower frequency 'aliases'. This gives rise to such problems as "jaggies" along what should be smooth edges, "strobing" and "crawling" of edges in time (the temporal equivalent of jaggies in space), and the "wagon wheel revolving backwards" phenomenon.

Anti-aliasing

A most important and all-too-often neglected aspect of computer graphics the aim of which is to eliminate the problems mentioned above. It does this by a filtering process intended to determine a suitably averaged shade for each pixel based on the colours of all objects affecting this pixel.

Bitblt

The display on many window-based graphics terminals is maintained by copying the various visible image components to a special area of graphics memory from where it is displayed on the screen. This image must be updated whenever a window moves or changes priority. The process of copying large rectangular areas of image to the display area is known as bitblt (Binary digIT BLock Transfer). With some systems it is possible to perform logical or arithmetic operations on the image as it is transferred. The technique is most effective for monochrome displays. The alternative to bitblt is dynamic mapping—as described for Rainbow in Appendix I.

Compositing

Typically the images comprising each frame of an animated film are not created by a single program. It is often the case that various elements of the final images are computed by separate programs or separate applications of the same program at different times. These separate image elements are then merged together, a process known as *compositing*.

Double-buffering

A technique used when displaying moving images on a graphics terminal. The illusion of movement is lost if the viewer is able to see the changing image being drawn so instead two image buffers are used so that a completed image can be displayed while the next image is under construction in an off-screen area of graphics memory; once the new image is complete the buffers exchange roles.

Easing in, Easing out

Traditional animators often adjust the speed of motion as objects start and stop moving so as to produce a smoother effect. This is refered to as *easing* the motion (*out* as motion starts and *in* as it stops).

Editing

The task of splicing many pieces of film together to form a whole. Much more film is shot than will be used. The editor selects just those frames needed to create an effect, and the rest end up "on the cutting room floor". Obviously, a great amount of artistic control is exercised at this point in the filmmaking process. The lack of editing is generally the hallmark of "home movies".

Fractals

A branch of mathematics—recently investigated by Benoit Mandelbrot—which deals with curves and surfaces with non-integral, or fractional, dimension. In computer graphics applications, this relates to a technique for obtaining a degree of complexity analogous to that in nature from a handful of data points.

Framebuffer

An especially adapted piece of digital memory for storing a computed picture. Framebuffers are typically attached so some sort of video controller and display monitor for viewing the contents of the memory—assumed to be a two-dimensional picture.

Line-chain

A sequence of connected line segments drawn without lifting the 'pen' from the paper.

Patch

A piece of curvilinear surface, typically with four sides. Patches are attached together at their edges with at least first-order continuity to form complex three-dimensional surfaces. The edges of a patch are frequently described by polynomials or ratios of polynomials. For example, if both dimensions are described by cubic polynomials, then the patch is said to be "bi-cubic". If ratios of cubic polynomials are used, then the patch is a "rational bi-cubic" patch. Although difficult to deal with, one patch can take the place of hundreds of flat polygons and thus greatly reduce the size of a database.

Pixel

Short for "picture element". Refers to the smallest unit of display in a computed picture. A computed picture is typically composed of a rectangular array of pixels (e.g. 768 x 576). The number of pixels in one dimension is frequently used as the "resolution" of the picture (e.g. a 1000-line image).

Raster graphics

An image is displayed by scanning an electron beam in a series of horizontal stripes across the screen. The beam is switched on and off in order to excite the phosphor on the screen only where the image should appear bright. This scanning order is referred to as raster-scan. Colour images can be produced by having three electron guns (one for each of three primary colours, red, green and blue) which are arranged only to excite phosphor dots of their own colour. This is the arrangement used for ordinary television. The required electronics for a raster pattern are well-established, and hence not expensive. The most sophisticated computer graphics are produced this way but anti-aliasing is required. The opposite of raster graphics is vector graphics.

Spline

A piecewise polynomial with at least first-order continuity between the pieces. A mathematically simple and elegant way to connect disjoint data points smoothly. Splines can be either *approximating* or *interpolating*, the difference being that interpolating splines pass through their control points whereas approximating ones do not.

Texture

Any two-dimensional pattern which is used to add the appearance of complexity to a three-dimensional surface without actually modelling the complexity. Paintings or digitised photographs are frequently used. Whereas fractals actually add complexity to a three-dimensional database, textures do not.

Three-dimensional

In computer graphics this refers to the three spatial dimensions stored for each point of a model in a database, rather than to the perception of three dimensions as obtained, say, with two images and polarised glasses. Stereo graphics is possible, of course, but is only a subset of three-dimensional graphics.

Vector graphics

An image is formed from scan-lines orientated in arbitrary directions and drawn in an arbitrary order. Expensive electronics are required, but spatial anti-aliasing is not. Typical of this style of graphics are the "wire-frame" models which were considered synonymous with computer graphics in the early days.

Video lookup table

It is often the case that the numbers stored in graphics memory do not represent the colours of the pixels directly but are instead used to access entries in a lookup table in which the entries define the precise colours to be output to the monitor. This table lookup is performed at video rates just as the pixels are being output to the screen. This process enables a free choice of colours to be made for each image as opposed to having a fixed palette; it also means that whole areas of an image can be re-coloured simply by changing the value in the lookup table rather than having to alter the value stored at each individual pixel location.