

**SCRIPTS:**  
**On The Description of**  
**Computer Animated Images**

by  
Luiz Velho

B.S. Industrial and Communications Design  
Universidade Estadual do Rio de Janeiro  
1979

SUBMITTED TO THE DEPARTMENT OF ARCHITECTURE  
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR  
THE DEGREE OF

**Master of Science in Visual Studies**

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1985

Copyright © 1985 Luiz Velho

The author hereby grants to M.I.T. permission to reproduce and  
to distribute publicly copies of this thesis document in whole or in part.

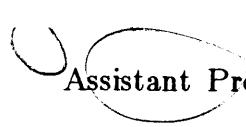
Signature of Author

  
Luiz Velho

Department of Architecture

May 10, 1985

Certified by

  
David Louis Zeltzer  
Assistant Professor of Computer Graphics  
Thesis Supervisor

Accepted by

  
Nicholas Negroponte  
Chairman, Departmental Committee on Graduate Students

MASSACHUSETTS INSTITUTE  
OF TECHNOLOGY

Rotch

MAY 31 1985

LIBRARIES

**SCRIPTS:  
On The Description of  
Computer Animated Images**

by  
Luiz Velho

Submitted to the Department of Architecture on May 10, 1985  
in partial fulfillment of the requirements for the degree of  
Master of Science in Visual Studies.

**Abstract**

The problem of specification of temporal transformations for Computer Animation production is investigated. Based on this analysis, an interactive animation language is developed which supports both procedural and key-frame animation. It is a flexible software environment for the design and prototyping of animation programs and interfaces.

The language is implemented in C within the UNIX operating system, and consists of C-like expressions, built-in functions, script and track constructs. There is also an escape mechanism to run UNIX commands. C-like expressions are the regular arithmetical, logical and control of flow operations. Built-in functions are C functions incorporated in the language. Scripts are *time programs* that are executed in parallel to generate animation. Tracks are *time variables* used to define dynamic animation parameters.

A small set of animation tools is also developed to exemplify the system's utilization. These include a three dimensional geometry model interface library, a spline library, and simple mechanics, collision detection and inverse kinematics functions.

Thesis Supervisor: David Louis Zeltzer  
Title: Assistant Professor of Computer Graphics

to my wife for her love and support

# Table of Contents

<b>Abstract</b>	<b>2</b>
<b>Table of Contents</b>	<b>4</b>
<b>List of Figures</b>	<b>6</b>
<b>Introduction: The Description of Computer Animated Images</b>	<b>7</b>
<b>1. Prospective History and Background</b>	<b>10</b>
1.1 Past, Today and The Future	10
1.2 Animation System Components	14
1.2.1 Object Modeling	14
1.2.2 Animation Design	16
1.2.3 Image Generation	17
1.2.4 Image Manipulation	18
1.3 Animation Control	20
1.3.1 Time and Temporal Aliasing	20
1.3.2 Animation and Simulation	21
1.3.3 Animation Abstractions	22
1.3.4 Animation Control Methods	23
<b>2. The Script Language</b>	<b>25</b>
2.1 Preliminary Definitions	25
2.2 Animation Specification	29
2.2.1 Script and Event Constructs	30
2.2.2 Track Structures	32
2.3 The Language Syntax	33
2.3.1 Data Definition	33
2.3.2 Expression and Control of Flow Statements	34
2.3.3 Animation Control Statements	35
2.4 The Language Interpreter	37
<b>3. The Computing Environment</b>	<b>39</b>
3.1 A Model of Interaction	39
3.2 Utility and Animation Tools	42
3.2.1 Input/Output	42
3.2.2 Object Modeling	42
3.2.3 Event and Track Manipulation	44
3.2.4 Collision Detection	45
3.2.5 Inverse Kinematics	45
3.2.6 Viewing and Display Control	46
3.3 Interfacing Tools	47
3.4 Examples	50

<b>4. Conclusion</b>	<b>61</b>
4.1 Summary	61
4.2 Extensions and New Directions	63
<b>References</b>	<b>65</b>
<b>Appendix A. Script Reference Manual</b>	<b>71</b>
<b>Appendix B. Animation Tools Programmer's Manual</b>	<b>75</b>

## List of Figures

**Figure 3-1:** Robot arm and bouncing ball animation sequence 60

## **Introduction: The Description of Computer Animated Images**

I think of a computer display as a window on Alice's Wonderland in which a programmer can depict either objects that obey well-known natural laws or purely imaginary objects that follow laws he has written into his program. [Sutherland 70]

Animation is an extremely complex art form. As a communication medium it involves our most important cognitive senses to present an artificially created reality. The problem is not just to move images around on the screen, but one of manipulating form, sound, space and time in expressive ways in order to convey a certain message. We want to create imaginary worlds and describe them so convincingly that the audience experiences the feeling of being there.

Until very recently, conventional animation required the generation by hand of every frame of a recorded sequence. This task was usually done by people with an artistic background, and the representational techniques involved were those of the traditional visual arts. With the advent of computers, they gradually were introduced in the field; first as a production tool, and more recently as a design tool. This has caused a revolution in the creation and generation of animated images, giving rise to new technical and aesthetic issues.

This thesis addresses the problem of animation design in a computer

assisted animation environment. There are many aspects of the problem to consider. Some, like three-dimensional modeling and image rendering, are well understood and will not be discussed in detail. Others, such as knowledge based animation, are just emerging and will be out of the scope of our work.

The main goal is to create a flexible software environment for animation design, providing the animator with tools for describing temporal phenomena.

Animation is the art of manipulating the invisible interstices that lie between the frames. The interstices are the bones, flesh and blood of the movie, what is on each frame, merely the clothing. [McLaren ]

An interactive animation language is implemented that allows the specification of dynamics at various levels, from automatic procedural simulation to guiding level animation. The language design focuses on the definition of mechanisms and abstractions to manipulate time events. It supports synchronized parallel processes - similar to those in ASAS [Reynolds 78] and MIRA [Thalmann 83] - , and multi-level instantiated processes.

An additional set of tools is also implemented. It includes hierarchical three-dimensional model interfacing functions, track functions, splines and mechanics functions. This makes possible the integrated use of procedural and key-frame animation.

A model of a man-machine interface for animation design is developed as an example of the actual use of the system. It consists of multiple interfacing programs, such as a motion editor and a text editor, that deals with different aspects of the animation.

While this work is restricted to three dimensional computer animation, the concepts investigated could be applied as well to other areas of the audio-



visual media. In particular, the fields of film and video editing, special effects creation, and multi-track videodisk authoring involve the description of multiple synchronized events similar to those studied here.

The structure of this thesis is as follows: Chapter One provides the preliminary background information for the project development. It consists of a brief history of computer animation in the entertainment field, an investigation of the main components of animation systems, and an analysis of the different types of animation control methods. Chapter Two describes the development of the project itself, including the statement of the project's goals and objectives, the definition of the software abstractions chosen for animation control, and the specification of the Script language - its syntax and semantics - that implement those abstractions. Chapter Three gives an insight of the project's application. It presents a model of user interaction, the development of a library of animation and interfacing tools, complemented by examples of animation scripts. Chapter Four is a concluding evaluation of the project. It summarizes the work realized and suggests possible extensions and future directions. The Appendices are intended to serve as the project's on-line documentation in the UNIX operating system environment. Appendix A is a reference manual of the Script language, while Appendix B is the definition of a library of animation and interfacing built-in functions, to be included in the local section of the UNIX Programmer's Manual.

# Chapter 1

## Prospective History and Background

### 1.1 Past, Today and The Future

The history of computer animation must be traced in the context of computer graphics as a major discipline. They share many points in their evolution, and several developments, both in hardware and in software, from areas such as computer aided design and flight simulation, have been incorporated as part of the computer animation technology. Although this is true in the most general sense, the history of computer animation is also the history of the individual efforts of a group of pioneers, whose confidence in the future of the activity made them, at each step along the line, push the existent technology to the limit, creating the reality of computer animation.

The work of Ivan Sutherland on the Sketchpad Drawing System, at M.I.T. in the early 1960's, is perhaps the origin of modern computer graphics [Sutherland 63]. It was possible to identify in the very first years of computer animation, during the 1960's, three major trends. They can be classified by the type of driving technology, as well as by the type of application. They are: analog computer animation, motion control animation and two-dimensional key-frame animation.

Analog computer animation makes use of analog computers to modulate a TV camera input video signal, generating in real time a corresponding distorted image. This technique was responsible for introducing computer

graphics to television. High costs and lack of versatility restricted their further use, and gradually a new generation of digital devices took their place.

Motion control animation uses analog or digital computers to control a full range of animation equipment. The operation of recording devices, like animation stands or optical film printers, could be coordinated with the motion of recorded subjects, like artwork or mockups, to achieve a variety of effects. This technique became the standard tool in the motion picture industry for special effects and graphics in feature films and television advertisements.

John Whitney Sr was the first researcher of motion control [Whitney 81] He explored the idea of "light paintings", that helped make things like streaking and slit scan effects an integral part of the film vocabulary. The National Film Board of Canada (NFB) was another important developer of motion control animation systems. They created the first computer controlled animation stand, commercialized afterwards by Oxberry.

Two-dimensional key-frame animation, makes use of digital computers to interpolate through time images drawn by the animator. The main representatives of this category are the Genesys System developed by Ronald Baecker at M.I.T. [Baecker 69], and the NFB's Animation System developed by N. Burtnyk and M. Wein at the National Research Council of Canada [Burtnyk 73]. The vector images generated by those early systems were considered applicable only for experimental films, of which, the 1964 Cannes Festival award winning, "*La Faim*", by Peter Foldes is the best example.

Later on, with the emergence of raster displays, the New York Institute

of Technology (NYIT) Computer Graphics Laboratory, and other companies - ACME Cartoon and Hanna Barbera - developed more advanced systems to be used in commercial cartoon animation [Catmull 79, Christopher 82, Rivlin 82].

The second age of computer animation, beginning in the mid 1970's, is characterized by shaded graphics and three-dimensional objects. The extensive research in image synthesis carried on during the 1960's and early 1970's began to bear fruit, and its results were applied to the Entertainment Industry. Utah University's Computer Graphics Group, headed by David Evans and Ivan Sutherland, was one of the most important of these research centers. There, a whole generation of computer graphics researchers and practitioners were educated. The group developed some of the basic modeling and rendering techniques used in three-dimensional animation.

Few commercial production companies were active at that time. They were responsible for bringing computer animation to the entertainment market and to the general public. In a symbolic way their effort culminated with the film TRON, a Walt Disney production directed by Stephen Lisberg in 1981, which involved most of those pioneering companies, and represented the major breakthrough of computer graphics into the mass media. TRON was the first feature film to make significant use of Computer Graphics. In order to create its 15 minutes of purely computer generated images and almost 200 computer generated environments it took the coordinated work of MAGI Synthavision, Information International Inc., Digital Effects and Robert Abel and Associates.

Other important research and production centers were the NYIT's Computer Graphics Laboratory and the Ohio State University's Computer Graphics Research Group. The cooperation of artists and scientists in these groups was the key factor for the well-balanced development and creative use

of Animation Systems.

Today, we are living in a time of transitions. The personal computer revolution and the consequent reduction in the prices of hardware in general, have made computer graphics widely available.

The television industry relies more and more on equipment like character generators, painting systems and digital video effects generators to enhance its programming. Real time video animation is already possible, in a limited fashion. The transformation towards a fully automated digital television plant is just beginning [cbs 84], and computer graphics will certainly play an important role in the process.

The motion picture industry is also experiencing transformations. Lucasfilm Ltd. and Digital Productions are conducting research in the areas of electronic film printing and very realistic image synthesis with very good results. In the future, we can expect computer generated animation to be an integral part of the filmmaking tools, and the audiences won't even be aware that the viewed imagery was generated by computers.

## **1.2 Animation System Components**

Before analyzing in detail animation control methods, it is enlightening to situate them within the context of a global animation system.

There are Three essential stages in the process of generating animation, and a optional fourth post-processing stage. They correspond to the tasks of object model design, animation design, image generation and image manipulation [Zeltzer 84a].

### **1.2.1 Object Modeling**

Object model design or modeling is the description of the environment and objects to be animated. The data base with all the information necessary to generate the scene is created in this stage. Modeling procedures and object characteristics may vary widely with model types, and even with the rendering algorithms adopted.

Object models are classified according to the model structure and the shape representation scheme used to describe the object's geometry. Additional attributes can be associated with objects in order to describe their non-geometric features.

The standard form of structuring three-dimensional models is by means of a hierarchical tree of affine transformations that selectively scale, rotate, and position primitive objects. In this way complex objects can be assembled by instancing simple ones, and suitable structures for articulated objects can be created.

Boundary Representations (B-Rep) and Constructive Solid Geometry

(CSG) are currently the most widely used and understood representation schemes for solid objects [Requicha 80]. These two geometric representations are by no means the only ones for generating primitive objects. In the search for greater realism in image synthesis new models have been created that are more appropriate for the description of other non-geometric objects. Some examples are procedural models used to create several types of plants [Smith 84a], stochastic models used to represent terrain and other natural irregular phenomena [Fournier 82], and particle systems, a method for modeling ephemeral objects such as fire, clouds and water [Reeves 83].

Each type of model requires procedures for creating, editing and accessing the representation of objects. We shall see, also, that display functions are intimately connected with object descriptions, and that some models may need special rendering programs.

Several techniques exist for creating primitive objects. They define an input language, and range from the direct digitization of orthogonal views to the specification of a few parameters internal to the model. In the case of boundary representations, very common procedures are the translational and rotational sweep of a 2D set through space, and the surface reconstruction from contour lines.

Additionally, other attributes, such as color, texture or bump maps, may be created and assigned to objects. In general, they are related to the surface's characteristics and play an important role in depicting realism in the scene. The object modeling system should provide a coherent set of functions for primitive object generation, consistency checking, model assembly and manipulation.

### 1.2.2 Animation Design

Animation design or scripting is the description of the objects' temporal transformations. In this stage all the information necessary to animate the scene is generated. The way in which this information is specified, depends on the control modes and interfacing techniques used, and also, in part, on the object's parameter to be transformed.

Control modes can be classified as interpolated and algorithmic animation. They correspond, somehow, to the subtle difference between data and programs.

Interpolated Animation specifies a sequence of data elements describing the state of the scene at successive points in time. In general, all the details must be specified, and although the animator has complete control of the transformations, complex animation is usually difficult to describe. Animation systems based on interpolated control can be subclassified further according to the input methods. Motion tracking, score-based, and key-frame systems are some examples. In motion tracking systems, actual movements recorded in real time serve as the input for the animation. In score-based systems, movements are described in an alphanumeric choreographic notation, very much like a musical score. In key-frame systems the state of the scene is described at key frames, and the inbetween frames interpolated automatically by the system.

Algorithmic animation specifies, through a set of procedures, the rules that regulate transformations in the scene. This is a powerful method that can solve complex animation problems, but, on the other hand, usually requires intensive software development. Animation systems based on



algorithmic control, according to the degree of abstraction supported, range from a general programming language enhanced for animation to a task level knowledge-based system<sup>1</sup>.

One point to emphasize is that animation systems may describe temporal transformations through parametric procedural models, controlled by a small number of interpolated input parameters, breaking, in this way, the division line between control modes. In the next section, we will investigate animation control systems in greater detail.

### **1.2.3 Image Generation**

Image generation or displaying is the creation of each frame in the animation sequence, for the purpose of previewing or recording. Previewing is a very important feedback element for animation design, and near real time output rates should be assured by efficient algorithms and/or special purpose hardware. Since full-quality image generation is rarely feasible in real time, simplified display mode are used for previewing.

The image generation pipeline is divide into four main processes: scene description, scene traversal, viewing and rendering.

Scene description is the process of updating, at each frame, the scene's model by the animation programs. The transformation of objects gradually takes place, as the sequence is being generated.

---

<sup>1</sup>Zeltzer classifies animation systems as guiding level, animator level and task level. In general, guiding level systems may use interpolated control, possibly combined with parametric algorithmic models. Animator and task level systems, both make use of algorithmic control, allowing different degrees of abstraction.

Scene traversal is the process of walking through the scene's hierarchical tree of objects, generating a concatenated modeling transformation matrix for each primitive object. The modeling matrix positions, orients and sizes the objects in the environment.

Viewing is the process of actually transforming objects, from their local coordinate system to the standard view volume, as they were seen through the lens of a synthetic camera. During this process, objects are clipped against the boundaries of the viewing pyramid and just those in the field of view rendered. This process can be intermixed with the scene's traversal for efficiency reasons, and the subtrees of objects whose bounding volumes are completely outside the viewing pyramid, or smaller than some minimum projected screen area are respectively pruned or culled, being discarded from further consideration.

Rendering is the most variable display process. The trade-off of image quality vs. generation time usually decides the choice of methods. The image rendered may vary from a simple "wire frame" view to a high resolution anti-aliased shaded picture with textures, transparency, reflections, refractions, softshadows and special effects. All non-geometric attributes are processed by the rendering functions in the context of an illumination model, that calculates the intensity values for each pixel on the screen. Scan conversion or ray tracing, shading and visible surface calculations, central operations in this process, are applied in different ways on the various rendering methods.

#### **1.2.4 Image Manipulation**

Image manipulation is an optional final stage in which a variety of

digital image processing functions are applied to the resulting animation frames. This post-process includes: image enhancement, image transformation, and image compositing.

Image enhancement functions alter the picture intensity values. Color correction, diffusion, tinting, highlighting, defocusing, edge enhancement and hand touchups are some useful operations.

Image transformation functions change the image space geometry. Translation, rotation, scaling, and different types of projection are standard operations.

Image compositing functions are used to combine several images into one single picture. Separate foreground images can be layered on different backgrounds, and special effects, such as wipes, fades and dissolves can be generated to combine animation sequences.

Post-processing is necessary for two reasons. First, more and more we find that complex three-dimensional animation should be divided into separate elements, which are independently rendered and, then, composited. Secondly, the combination of live action and computer-generated animation is an indispensable requirement in both feature films and television advertisements.

It is important to note that, for proper image anti-aliasing, pixel information must include an additional coverage channel, besides the regular red, green and blue channels. [Porter 84]

As a final note, it should be stressed that object modeling and image synthesis are very extensive subjects, beyond the scope of this thesis. They are described briefly here just as contextual references of a global animation system.

### **1.3 Animation Control**

Animation control systems, as we have seen, can be classified according to the controlling and interfacing methods provided. In this section, we will try to identify the main principles that regulate animation control modes, as well as the software abstractions that determine user interactions in animation systems.

#### **1.3.1 Time and Temporal Aliasing**

The computation process of animation has as its final objective generating the changing state of the scene through time. The model description is updated as the time goes by: new objects may be created, and existing objects may be transformed or deleted from the scene.

Because we are working in the digital domain, this process occurs at discrete instants in time. Consequently, time resolution or granularity is presupposed, and although we may be picturing continuous phenomena, changes take place at time boundaries.

To avoid temporal aliasing we have to consider in our computations all actions that have happened during each elapsed time interval. For display purposes we not only need to calculate the right positions at the right times, but we have also, to integrate the projected screen images of moving objects. This operation, known as motion blur, has been, in the last years, very actively researched. [Cook 84, Korein 83, Potmesil 83]

### 1.3.2 Animation and Simulation

Animation, as noted by Zeltzer [Zeltzer 84b], can be seen as a process of simulation in its most general sense. This reinforces the idea that we are, in fact, creating behavioral mechanisms in order to materialize imaginary universes on the screen. It is only a perfect comprehension of the inner workings of these universes, that will enable us to produce good animation. In its various instances, this understanding will tend either towards a more artistic intuitive view, or to a more scientific rational approach. The animation system should be designed to encourage this multiplicity of approaches, amplifying the user's actions in an integrated way.

Simulation, well established as a discipline of computer science [Franta 77], only recently has been incorporated into animation systems. Some important concepts in simulation applications are the distinctions between continuous and discrete simulation, and the simulation of kinematic and dynamic models. The difference between continuous and discrete simulation is related to the nature of the modeling solutions. Continuous simulation specifies the problem as a set of simultaneous, time-dependent equations, while discrete simulation describes it as a coordinated sequence of events in time. The simulation of kinematics and dynamics refers particularly to the modeling of motion. Kinematics models directly the descriptions of movements, and dynamics models the systems of forces that causes these movements. The simulation paradigm requires that the current values of the attributes of objects to be accessible and modifiable. If dynamics simulation is used, previous values, rates of change, and/or differentiable attributes may also be required.

It is important to keep in mind, though, that Computer Animation is not concerned with the pure simulation of physical reality. The use of simulation techniques in animation systems is merely an amplifying mechanism to enhance the user's creative power.

### **1.3.3 Animation Abstractions**

Users interact with the animation system by means of a network of abstractions that, ultimately, makes up the repertory of actions supported by the system. Taking the simulation paradigm another step further, we realize that the animation system's design is a matter of creating the world of software abstractions that will constitute the creative substance of animation. This is certainly the most complex issue regarding animation systems, and there is no formula applicable to it. Only the iterative process of research, together with the the accumulated animation's heritage, can bring out powerful and effective abstractions.

The world of the symbolic can be dealt with effectively only when the repetitious aggregation of concrete instances becomes boring enough to motivate exchanging them for a single abstract insight. [Kay 84]

Animation abstractions relate to three areas where the user is called up to an effort of design; object modeling, animation modeling, and the creation of animation itself. We have already discussed briefly three-dimensional modeling, and won't go further into the subject for a while. Suffice it to say that object modeling defines the geometric entities that will be manipulated with the animation software to generate animation. Animation modeling and creation are tasks that can be intermixed in one single step, or completely

separated into two independent ones.

Animation modeling establishes the functional abstractions of the object's dynamic transformations. It can be as simple as the direct mapping in motion tracking systems, or as complicated as the synergy of simulation machines in knowledge-based systems. These abstractions are created with the animation system's software tools, and define levels of interaction for controlling the animation parameters. Animation, when not described directly, will be usually produced by applying these functional abstractions to objects, in the context of an interfacing program, that provides adequate control parameters and effective feedback, facilitating the full expression of animation ideas.

Animation abstractions are influenced by several cultural factors and scientific concepts. The notions of time, velocity and acceleration are particularly important for the design of animation. Film language and the motion picture traditions also contribute largely to the way animation is conceived and produced.

#### **1.3.4 Animation Control Methods**

Animation can be represented as a sequence of concurrent synchronized temporal events. These events are the functional entities to be manipulated. They are prototypes of temporal transformations that can be instanced to perform actions when and where this is required. Such events are often perceived as a series of independent actions. If a group of events is interdependent, it is seen, instead, as one single event that represents those correlated actions. Temporal events, among other attributes, may be:

continuous or discrete, unique or periodic, delimited or open-ended.

Animation control systems are classified in relation to the ways in which the user can specify these events, and to the types of abstractions provided for manipulating them. The major distinction, as we have seen, is between interpolating and algorithmic animation. In the former, the problem is the reconstruction of single signals from sparse samples in time, while in the latter, the concern is with the synthesis of a composite signal through algorithmic procedures. The fundamental difference, here, is that algorithmic animation explicitly specifies the relationships that constitute a complex signal, and in this lies the source of its power.

In interpolating systems, events are represented by tracks that correspond to sequences of time samples or marks. For efficient access and modification, tracks are usually implemented as doubly-linked lists. Different types of splines and interpolating functions are provided for controlling the track's characteristics. Key-frame systems have as an interface, interactive motion editors, that allow the association of tracks with the object structure, the synchronization of parallel tracks, and the manipulation of marks and interpolating functions.

In algorithmic systems events correspond to processes that embody groups of transformation actions, described in a programming language. The capabilities of the system will depend, mainly, on the language's features, and on the existing software tools. Mechanisms for start-and-stop parallel processes at arbitrary instants in time, and for interprocess communication should be provided to support concurrency and synchronization. Other layers may define new abstract entities and manipulating operations, assembled in an integral way for the creation of animated images.



## Chapter 2

# The Script Language

### 2.1 Preliminary Definitions

Computer animation systems are combinations of software and hardware elements for production of synthetic moving images. The animation software itself is actually a small part of a larger computing environment. The consequence of this fact is that part of the animation system's effectiveness will depend on the degree of integration among the components of this environment. The animation software, as a central element in the system, must be able to communicate with others software modules, to share data objects with them, and to have efficient access to hardware resources.

The animation system, as a film production tool, is expected to face a wide range of demands, as different projects come and go. To meet most of the possible requirements, the system has to be versatile enough to adapt itself to each specific situation. This suggests that we must see it as an evolving organism, and design it to be flexible and extensible. The animation software should be modifiable, it should be easy to change existing modules and to add new ones. The software modules and the interface between them should be designed in such a way that once a module is incorporated into the system, it can be used to solve a general class of problems, not only the ones that originated it. This will make the system grow in power, being perfected as it evolves.

The animation system is a means to produce synthetic moving images. It should facilitate the creative expression of its users by providing a rich set of mechanisms to interact with the medium. These mechanisms must allow the user to conveniently manipulate animation parameters. The user interface has to be compatible with the need of experimentation inherent in the creative process, being adaptable to the specificity of each individual parameter. The user may wish to describe operations at different levels of detail, and/or to have alternative ways to perform the same operation.

Finally, and most importantly, the animation system must be effective and efficient. To be effective it has to materialize the right abstractions to deal with temporal events and their interactions. These have to be implemented through a coherent set of mechanisms, and the system's efficiency must be assured by fast algorithms and good use of the resources available.

In summary, the animation software should be integrated, flexible, extensible, interactive, effective and efficient. The Script system was created in conformity to these characteristics. The main point to be underlined is that we didn't intend to develop a complete set of software modules for animation, rather, our objective was to design an open mechanism by which these modules can interact in the context of animation design and generation.

The development strategy is to implement this basic mechanism along with an initial set of animation modules, and leave the system to be expanded with time and use. The class of animation problems addressed are those related to the synchronization of independent parallel processes through time. As a consequence, interdependent processes are considered atomic units in the system, and all dependencies have to be handled inside these units. The

general problem of time interdependency represents a big step in complexity that is left for further research.

The animation software is implemented in the C language within the UNIX operating system environment. The UNIX system is suited to our purposes, as it provides the program and data interactions required.

The major piece of software developed is an interactive script interpreter that act as a coordinating mechanism for animation design and generation. To meet the extensibility requirements, it has to be programmable and be able to run other programs in the system. The interpreter itself must be modular and easily modifiable, allowing the redefinition of syntactic and semantic rules.

In order to facilitate user interaction and to permit a continuous development of flexible animation tools and interfaces, the system is designed to allow three levels of operation with increasing control of dynamics specification. At the first level, the user creates animation with the help of existing interactive animation programs, written in script language and coordinated by the script interpreter. At the second level, the user interactively defines an animation description by writing and refining a program in script language. The program could be added latter to an animation library for posterior use. At the third level, the user writes a program in C language to be incorporated in the system as a built-in function.

The operation in levels gives flexibility in the description of animation, allowing problems be approached with the right type of tool. Large animation projects also can be developed from basic functions to high level interfacing

procedures. For this mechanism to work, the frontiers between levels must be transparent, and the user should be able to move freely between them.

Another point is that rather than attempting to define an all-purpose user interface, the utilization of multiple interfacing programs, specific to each type of animation problem, is adopted as a model of interaction.

Lastly, to exemplify the system's utilization, a small set of animation tools is also developed. These include a three-dimensional geometry model library, a spline library, and simple mechanics and collision detection functions.

## 2.2 Animation Specification

The script language integrates algorithmic and interpolating animation control, the two most common approaches to animation description, while being compatible with the criteria of extensibility and flexibility.

The animation control mechanism built into the language is designed to support synchronized independent events, similar to the actors in ASAS [Reynolds 78] and MIRA [Thalmann 83]. The notion of parallel processes, in which these mechanisms are based, is by no means new in computer science. Most of the current operating systems, and several programming languages, such as Simula [Nygaard 68], Smalltalk [Goldberg 83], and Modula [Wirth 77], allow this type of concurrent control structures.

In comparison with ASAS and MIRA, Script differs mainly in three distinct aspects. The first is related to the programming base language. Script is implemented in C, following its main characteristics, while ASAS and MIRA are based respectively in Lisp and Pascal. The second is concerned to the way events are handled: Script allows the definition of nested parallel processes, that is supported neither by ASAS nor by MIRA. This closure property is very important, because it makes possible to break complex animation problems in simpler ones, easier to program and mantain. Third, the Script language provides a more general scheme for the description of dynamic animation parameters. In ASAS this is restricted to piecewise cubic curves with selectable degree of continuity at joints, and in MIRA, to arbitrary interpolations of end values. In Script tracks constitute an open mechanism for temporal parameter definition, that can be used even beyond the scope of the language.

The language consists of C-like expressions, built-in functions, script and track constructs. The C-like expressions are the regular arithmetical, logical and control of flow operations. Built-in functions are C functions incorporated in the language. Scripts are *time programs* that are executed in parallel to generate animation. Tracks are *time variables* used to define dynamic animation parameters.

### **2.2.1 Script and Event Constructs**

Script constructs are the primary element for animation specification in the language. Scripts are static algorithmic descriptions, that can be instanced into dynamic events to perform animation actions through time. They are prototypical, in the sense that one script description may originate several instances of distinct events.

Events are the run-time instantiation of scripts that model their temporal and algorithmic properties. They are composed of a body of instructions and a private memory that registers the uniqueness of each instance in relation to the others. Events can generate other events, defining a hierarchy of procedural instances implicitly ordered and synchronized by activations at time boundaries, during their active periods.

Once a script is started, the event instance lasts until explicitly stopped, being activated for every time interval. If an event is stopped, its dependent sub-events are stopped as well. Synchronization between events is guaranteed by the parallel activation of events at time interval boundaries. Each event has incorporated in its memory a local time, that is automatically updated by these activations. The activation consists of the time update, the evaluation

of all expressions in the event body, and the recursive activation of dependent sub-events.

The periodic activation<sup>s</sup> of the event hierarchy is originated at the highest level of the event tree. Time acts as streams of updating rates that flows through events at each<sup>t</sup> activation cycle. In this way instantiated scripts are played, in very much the same manner as film projectors or videotape players. It is possible to define playback functions such as forward, stop, and reverse by controlling the rate of change of time activations at the highest level. Although in the current implementation time is represented as a single number, in future extensions the time structure will be constituted by a time instant and rate of change. This will allow continuous variable sampling and differential compression or expansion of event durations. Time modifiers could also be defined with this mechanism. They would be intermediate elements between event levels that would alter the incoming time rates for the downstream levels. The scheme would allow control of relative time changes between events.

The discrete nature of time activations requires some conventions being made to avoid temporal aliasing. Scripts have to account for total changes during activation intervals whenever necessary. If motion blur is to be implemented, the displaying software must be prepared to integrate the screen images of objects affected by those changes.

This control mechanism also implies that events have to be independent, because continuous changes are resolved only at interval boundaries.

Intercommunication of events is possible, in a limited way, by means of global variables. Although this solution is clearly insufficient for complex

event interconnections, it does provide a coarse level of communication, adequate for simple interactions. A more effective scheme, using a send/receive convention with time stamps that would allow more sophisticated interactions, is planned for future extensions.

### **2.2.2 Track Structures**

Tracks are data structures that hold the necessary elements for describing time variable parameters. They constitute a flexible mechanism that accommodate the needs of various different parameter types. Tracks are specified by a list of values and a set of manipulating functions. Each type of track may have different configuration, as well as, functions assigned to it. Track values at particular instants in time are accessed through some of these functions. Several techniques can be used to generate them, and, in general, but not necessarily, there will be some kind of interpolation, such as spline, parabolic or linear functions. Values of the derivatives of interpolating curves can also be obtained, if required, by dynamic equations. Other functions will perform various manipulating operations. The basic ones are: insertion, deletion and modification of elements; and access of the first, last, previous and next elements in the track structure.

Script algorithmic descriptions, event control mechanisms, and track data objects, make up the set of primitive animation entities in the language. Together they form the basis upon which other animation abstractions can be built, extending the repertory of operation within the language.



## **2.3 The Language Syntax**

The script language syntax, with few exceptions, is patterned after the C programming language. This section describes the overall structure of the language with great emphasis in its animation constructs. A complete specification of the language syntax can be found in the Appendix A - The Script Reference Manual.

### **2.3.1 Data Definition**

The basic data objects in the language are real and string variables, track and event structures, functions and scripts.

Track structures are doubly linked lists of marks manipulated by a set of built-in functions. Marks, for greater flexibility, are constituted of a variable list of floating point values, that can have different meanings according to track types. All operations on tracks are performed indirectly by built-in functions, that define the interpretation of marks, and return track values as appropriate. The only directly accessible element of a track structure is its name. Tracks are static storage type, and may be declared exclusively at the highest syntactic level. A track pointer type is also defined providing a convenient track representation inside functions and scripts.

Event structures are references to instantiated scripts. They uniquely identify script instances, and are the link through which operations may be performed on them.

Functions and scripts are the primary procedural objects in the language. In fact, scripts can be abstracted as temporal functions that will be active for

some period of time. Their declaration has a similar form, composed of procedure name, procedure parameters, local variables, and procedure body. The procedure name declaration consists of a type keyword - either function or script - and the procedure identifier. The procedure parameters declaration, like in the Pascal programming language, consists of a list of data type declarations enclosed in parenthesis. The declaration of local variables consists of a list of data type declarations, placed between the procedure heading - name and parameters - and the procedure body. The procedure body consists of a list of statements enclosed by braces. Actual parameters are passed to procedures by value. Function variables are automatic, and function recursion is allowed. Script variables are local to each instance, and last while that instance exists. Script instances have a predefined time variable *t*, that is updated to the current local event time at every activation.

### **2.3.2 Expression and Control of Flow Statements**

Script is essentially an expression language. Expressions are divided into arithmetic, logical, and assignment expressions. Arithmetic expressions are floating point type, and can be composed by combinations of primary expressions and operators. Primary expressions are references to variables, numerical constants, function calls, and parenthesized expressions. Valid arithmetic operators are the unary minus, multiplicative and additive operators, evaluated in that order of precedence. Unary operators group right to left, and binary operators left to right.

Logical expressions are also of floating point type, with the convention that *true* has the value of 1.0, and *false* the value of 0.0. Non-zero values are taken to be true. Logical expressions are combinations of primary

expressions, with relational, logical and negation operators. Unlike C, all operands in a logical expression are evaluated, and early termination is not used.

Assignment expressions group right to left, and may be numerical, string, event or track pointer. Only numerical types are allowed in general expressions, but all types can be passed as parameters to functions and scripts, and returned from functions as well.

The elementary actions of the language are specified by statements, that can be expressions, compound, control of flow and animation constructs.

Statement execution in functions or scripts is sequential, in the order they are listed. Control of flow statements provide means for the specification of other patterns of statement execution. Conditional, iterative and return constructs, make up a small, but sufficient, set of control of flow statements.

### **2.3.3 Animation Control Statements**

Animation control statements implements the mechanism for script instantiation and execution. Scripts are instanced with either a *play* or a *start* statement.

The *play* statement is to be used at the highest syntactic level, and is meant to generate animation sequences by instantiating the root script of an event hierarchy. It consists of a script call prefixed by the keyword *play*. The script call, like the function call, is denoted by the script name followed by parentheses containing a possibly empty list of expressions which constitute the actual arguments to the procedure.

The *start* statement is used in scripts to instantiate subscripts not at root level. It returns the event reference of that script instance, that can be assigned to an event structure. The general form of a start statement is the *start* keyword followed by a script call, and optionally prefixed by an assignment to an event variable.

Script instances are terminated by the *stop* statement. It has two forms: *stop* and *stop* followed by an event reference. The first one is intended for self termination, while the last terminates the subscript instance referenced by the event variable.

Except for the *play* statement, all other animation control statement should be used only in script descriptions.

As a final remark, it must be said that the script language is not intended to be a reinvention of the wheel. It does not attempt to be a modified copy of the C language. It is meant, instead, to complement C at a higher level for animation purposes, providing an interactive mechanism for the description of concurrent processes, features that C doesn't have. The consequence of this attitude, is that a convention to link C code as built-in functions is incorporated into the language. They behave as regular functions, and should be used whenever faster or lower level control is necessary.

## 2.4 The Language Interpreter

The script interpreter is an interactive program that accepts commands in the script language from the standard input and/or files, processing them to produce animation. It parses and executes valid statements, acting primarily as a mechanism for coordinating concurrent events.

The interpreter program, as most of the animation software, is written in C within the UNIX system environment, and was developed with UNIX tools, such as yacc [Johnson 75], and make [Feldman 79]. Its basic structure is similar to that of the expression language interpreter described by Kernigan and Pike in the book "The UNIX Programming Environment". [Kernigan 84]

The script interpreter is composed of a lexical analyzer, a parser/code generator, and a virtual machine interpreter. The lexical analyzer is a function that accepts an input stream of characters, and translates it into terminal symbols as requested by the parser. It also does some symbol table manipulation in order to install and recognize valid identifiers. The parser is produced by yacc from a grammar specifying the language syntax. Yacc outputs a LALR(1) parser, that together with error recovery functions and semantic actions form the translator/code generator program. A syntax-directed translation scheme is used to convert tokens supplied by the lexical analyzer into instructions for the virtual machine interpreter. Error handling guarantees that only valid code will be produced, and the early identification of syntax errors. This scheme is specially convenient, because it make easy the redefinition of existing syntactic and semantic rules, as well as the inclusion of new ones, facilitating language evolution.

The script virtual machine consists of the data structures functions that

implement primitive instructions in the language. The machine is a stack-oriented interpreter for arithmetic expressions. It recursively executes instruction functions that manipulate an evaluation stack, a context stack, and event tree structure. Numerical operands are pushed and popped onto the evaluation stack as expressions are processed. The context stack holds information related to procedure activation, and is used for function and event execution.

The event tree structure is a hierarchy of script instances, created and maintained by primitive coordinating functions. They execute, in parallel, events in the structure, updating it while scripts are being played.

Built-in functions are C functions returning floating point values, and are executed as primitive instructions in the language. There is also, an escape mechanism to run UNIX commands, that complements the interpreter interface to other modules in the animation system.

The interpreter calling sequence is the command *script*, and an optional list of file names. A character '-' in the list means that input is to be taken from the standard input stream.

## Chapter 3

# The Computing Environment

### 3.1 A Model of Interaction

The computing environment of an animation system is a collection of software and hardware with which users interact to produce moving images. All elements in the system contribute in some way to the accomplishment of this task, and as such, must be selected with this in mind. Great attention has to be paid, even to small details, because, for example, the lack of an appropriate input device, or the effects of an inefficient display program may compromise an, otherwise, well designed system. Every step in the production of animation requires some degree of user interaction, but design tasks depend essentially on it, and need to be based on a coherent model of interaction.

Animation design, for historic reasons, is often conditioned to stratified combinations of control modes and interfacing techniques. The two most common combinations are found in non-real-time text-mediated algorithmic systems, and real-time device-mediated interpolating systems. It is agreed that both have inherent advantages and disadvantages, but, at the same time there is a widespread belief that constituent elements are intrinsically associated with each other, while not necessarily so.

Several authors, however, have recently acknowledged a need for interpolated and algorithmic control modes under the same system [Fortin 83, Zeltzer 84a], and for an integration of control modes [Hanrahan 84, Zeltzer

85], because those approaches individually are unable to provide an effective interface for animation modeling.

This calls for a model of interaction that associates the power of algorithmic control with the high level of manipulating expression provided by interpolated animation, and that also, incorporates flexibility to describe a wide range of dynamic situations, and extensibility for creating animation abstractions.

We propose as a model of interaction, an open mechanism based on layers of functional abstractions and multiple interfacing processes, where the basic design cycle consists in the (re)formulation of solutions mediated by these processes, and the evaluation of some feedback results generated by them. This is essentially a trial and error process, in which the solution is arrived at through cycles of successive refinements, with eventual interruptions to build new tools for unexpected demands.

Functional abstractions are specified in script or C language, and developed with conventional programming tools, such as interpreters, compilers, text editors and debuggers. The interface between the script language and C facilitates the integration of levels in the development of animation tools. Layers of functional abstractions are created with those primitive algorithmic constructs, to define a hierarchy of animation entities that manipulate object parameters at different levels of detail. At the top level the user interacts with a complex of simulation machines associated to several interfacing processes. They generate the appropriate controlling parameters, and the animation is produced.

The script language supports the prototyping of these animation



interfaces, assembled from a set of pre-defined building blocks. Tracks are included as a main element for the general representation of control parameters, and specification of interpolated animation.

In the next sections we will describe some animation and interfacing tools, as well as, the development of animation examples using them.

## **3.2 Utility and Animation Tools**

The animation and utility tools developed establish a basic interaction protocol between the script language and other components of the animation system, and also, define a primitive set of animation control procedures. They are related to general input/output, three dimensional object modeling, event/track manipulation, interpolation, collision detection, inverse kinematics, viewing and display control.

These tools are C procedures, implemented as built-in functions into the script language. We will describe them here from a functional point of view, a detailed description of the procedure calls and parameters is given in the Appendix B.

### **3.2.1 Input/Output**

General input/output built-ins are functions that read numeric values from the standard input, write alphanumeric strings and/or formatted floating point numbers in the standard output, and process interrupt signals generated from the terminal. A function to execute a cshell command line is the escape mechanism to run other UNIX programs.

### **3.2.2 Object Modeling**

Three-dimensional geometric modeling built-ins constitute a standard interface to the object's data base. Objects are represented by a tree of modeling transformations, and primitive definitions, that describe respectively the object's hierarchical structure, and the object's elementary geometric shape and other properties. This two level representation is a flexible scheme that

allows the description of most types of object models, including hybrid models. The upper level contains only the general structural description, applicable to all types of models, while the lower level concentrates all the details of particular representation models. Additional kinematic information, such as joint motion constraints, may be associated with these higher level links for the description of articulated figures. Primitive object descriptions, besides the specification of a three dimensional solid shape, may include a variety of data related to surface characteristics and to other object's physical properties.

Currently, there are three types of object description files, each denoted by a file name suffix code. Compound object files (.obj) are textual descriptions of the general object structure, in a modeling language similar to *sdl* [Zeltzer 82] and *mat* [Lundin 82]. Their specification is divided in two parts: the first part naming each component of the structure, that are either a joint type or a primitive type, and the second part describing the structure itself. Joints have associated translation, rotation, and scaling transformations. Primitives are references to primitive object description files. The structure specification is a parenthesized list of joints and primitives, that originates the object transformation tree. Primitive object files (.prm) are lists of attribute-value pairs, declaring the various object's characteristics. Polygonal shape files (.shp) contains three-dimensional vertex coordinates and polygon data of the object's boundary description.

These data files are loaded by a function that parses object's descriptions, and creates an object symbol table. Objects are referenced by their names, that must be unique in the table. Temporary object instancing is accomplished by concatenating the object's names. Other functions have been defined to access object attributes, to manipulate object structure, and

to display the object themselves. Accessing functions retrieve and modify the values of object's attributes. Manipulating functions attach/detach objects to and from a given structure, and group objects into new structures. Display functions generate simple renderings of the objects, primarily for feedback purposes, since high-quality image generation was not a concern in this first version of the system.

### **3.2.3 Event and Track Manipulation**

Event and track built-ins are the manipulating functions for these animation data structures. The event built-in is an inquiry function that asks the state of script instances referenced by a given event variable. Track built-ins perform interpolation, access and regular linked-list operations on track structures. Track data files, identified by the suffix (.trk), are lists of marks constituted by a key time index and a variable number of associated values, that are automatically recognized and manipulated by track functions. Interpolating functions include linear, spline and parabolic interpolation. The spline method used is an implementation of Doris Kochanek's splines with local control of tension, continuity and bias parameters [Kochanek 84], that produces a very general class of interpolating cubic splines, allowing precise control of temporal transformations. The flexibility of the track data structure made possible a straightforward implementation of d-splines, by the inclusion of the three control parameters to key values at each mark. The manipulation of tension, continuity and bias is direct, and the implementation requires only functions to properly interpret these control parameters and to perform the interpolation. Parabolic interpolation is part of a set of simple Newtonian mechanics functions. It is intended to simulate situations in which

accelerated motion is necessary, such as fall under gravity and ballistic trajectories. Track data consists of one element describing the direction vector, the initial velocity and acceleration.

### **3.2.4 Collision Detection**

Collision detection is essential for the development of adaptive motion, a technique used in goal directed and constrained animation [Badler 79, Zeltzer 84b]. The implementation of general collision detection mechanisms, is a complex problem, currently under research [Boyse 79], which to be viable requires effective access to the scene's data base and efficient object intersection algorithms. Rather than addressing this more general issue, which would not be practical in the present context, we developed a very simple collision detection function that deals only with bounding spheres and planes. This is adequate for simple adaptive motion, while a more complete mechanism is not implemented.

### **3.2.5 Inverse Kinematics**

Inverse kinematics is a powerful way to control the movements of articulated objects. This technique, originally used in robotics [Paul 81], allows the determination of the internal joint angles of multilinked objects from the position and orientation of its terminal links. The solution to the problem consists in computing the angular joint velocities, using the inverse jacobian matrix derived from the object structure. If the structure has more than six joints, it is underconstrained, and the result, potentially redundant may present complications [Korein 82]. The use of pseudoinverse methods overcome these complications with the advantage of additional control over

the generic solution [Girard 85]. A library of object manipulating functions based on pseudoinverse kinematics is currently being developed, and will be included in future implementations. A simple manipulator for a 3 degree of freedom robot arm was also developed. This function, based on a direct geometric solution, is presented in Section 3.3.

### **3.2.6 Viewing and Display Control**

Viewing functions follow, with minor differences, the Siggraph Core standard, as described in [Smith 84b]. The virtual camera model is specified by the parameters viewpoint, viewnormal, viewup, viewdistance, viewdepth and viewing window. Additional mapping onto the physical display screen is specified by a viewport parameter. The viewing functions provide a more natural way to set these parameters, and generate the corresponding viewing transformation matrices. Lookat, Polarview and Camera are alternative ways to derive viewpoint, viewnormal, and viewup. Perspective, and Window similarly set up viewdistance, viewdepth and the viewing window.

Display control functions perform low-level operations related to graphic devices. A DEC/VT-125 vector terminal or a DEC/VS-100 bit map display are automatically selected according to the value of a shell environment variable.

### 3.3 Interfacing Tools

The user interface is one of the most critical elements in the animation system, mainly because it is the accessing channel to the system's capabilities. The interface, besides human factors considerations, has to be complete, in order to allow full use of the system's resources.

In algorithmic systems, like Script, users interact at different levels, and because it is extensible, new elements that require interfacing may be frequently added to the system. This means that a complete user interface, in this case, will actually be a meta-interface - a development mechanism for prototyping and refinement of animation interfaces. Furthermore, multiple interfaces may coexist, in the context of a multiprogramming system, sharing global data objects.

The methodology adopted here for the development of user interfaces is known as the building block approach [Foley 82, Green 82]. In this approach, interfaces are assembled from basic modules that implement common interaction techniques. Interfaces may also be created by the selective addition/modification of existing prototype interfaces.

Interaction building blocks are based on a screen oriented interfacing model, that divides the screen into logical areas, defining classes of interactive operations. Each logical area is associated with a major type of interaction technique, and have a predefined set of attributes that can be modified by altering default parameters. These include screen layout format, type of constituent elements, feedback conventions, and control structures. Active areas on the screen may be enabled or disabled during the course of interaction, determining visibility and interrupt status. Additionally, logical

input and output functions may be associated with interaction areas or used independently in the interfacing program.

Logical interaction areas are defined for menu selection, parameter control, prompt and data viewing.

The menu selection area displays a group of items, presented as text words or symbols, that can be selected by the user. Standard feedback practice is to blink picked items, and to invert the selected ones. Hierarchical menu selection is achieved by the control functions `push_selection` and `pop_selection`, and selection decisions by the function `wait_selection`. Selections can be made on an exclusive or inclusive basis.

The parameter control area displays a group of parameter representations, that can be changed to alter actual parameter values. Regular representation schemes include dials, sliders, buttons and plots.

The prompt area is a text display, usually used for feedback purposes or to instruct the user about the next steps of the interaction.

The data viewing area is a general purpose display area, where the data structure being manipulated by the interface can be presented, often requiring the use of general output methods.

Logical input functions, roughly correspond to the virtual input devices of the Core graphics standard. The defined types are button, valuator, 2-D/3-D locators and text. Input values can be accessed on a direct sampling or a wait event basis.

Logical output functions are preformatted presentation techniques, used to display common types of data, that can be associated with elements of display areas. Useful output format types include text labels, graphical



symbols, vertical and horizontal bars, scales, pie charts and 2-D plots.

Three main interfaces are used most of the time in conjunction with other interfacing modules developed with these tools. They are: the script interpreter itself, a script editor, and a track editor.

Emacs, a powerful screen-oriented text editor, is used as the script editor. Script files are read, modified and written back whenever necessary between script execution cycles.

The track editor, planned for future implementation, is analogous to the motion editors used in Mutan [Fortin 83] and Bbop [Stern 83], and supports full d-spline track manipulation. The editor maintain a list of tracks that can be accessed and modified interactively. Track data is displayed in a scrolling type window. Viewing modes are value vs. time, velocity, acceleration, and two or three dimensional combined track paths. Plotting style is either in continuous lines or dotted lines at equal parameter spacing. Editing operations manipulate from groups of tracks to single marks within an individual track. There are commands to merge tracks, to add, delete and modify marks, and to change mark values, as well as, tension, continuity and bias parameters. Mark intervals can be contracted or expanded, and marks can be moved relative to an interval without changing it.

### 3.4 Examples

In this section, simple animation scripts are presented to give a flavor of the language and to demonstrate the system's usage. Some examples are complete working scripts, while others are simplified versions of the actual ones, with the distracting details taken out for clarity in the presentation.

The first three exemplify the direct use of tracks and script descriptions to generate animation. They illustrate the simplest and most straightforward usage of the system's resources. The fourth and fifth examples show script structures that implement cycle and key frame functional abstractions. They reveal how effectively the language lends itself to the creation of new animation abstractions. The sixth and seventh examples demonstrate, respectively, procedural animation with adaptive motion and inverse kinematics control. They are combined in the last example to generate the animation sequence pictured in figure 3.1.

Example 1: A ball rotates while it bounces up and down. Rotational and positional parameters are controlled by linearly interpolating track values, that are described using the track editor. The main script groups and synchronizes the two movements. The builtin function *setobj* changes the specified attributes of the named object to the value of the third parameter.

```
track rot, jump;

script ballrot&jump()
  event r,j;
{
  if (t==0) r = start ballrotation();
  if (t==30) j = start balljump();
  if (stopped(r) & stopped(j)) stop;
}

script ballrotation()
{
  if (t==last(rot)) stop;
  setobj("ball"."rx".linear(rot,t));
}

script balljump()
{
  if (t==last(jump)) stop;
  setobj("ball"."pz".accmotion(jump,t));
}
```

Example 2: A cylinder rolls in one direction with its displacement calculated from its rotation. One track controls directly the cylinder rotation, while the cylinder position is derived from its radius.

```
track cr;

script cylroll()
  real rot, pos;
{
  rot = linear(cr,t);
  setobj("cylinder"."rx".rot);
  pos=pos+(rot*getobj("cylinder"."radius"));
  setobj("cylinder"."px".pos);
}
```

Example 3: A two section articulated arm performs a simple joint rotation. The parameter "s" is used to control the bending speed.

```
script bend(real s;)
    real r;
{
    r = t*s;
    setobj("joint2","rx",0.01745*r);
}

script db(real s;)
{
    if (t==0) start bend(s);
    if (t*s > 120) stop;

    cleardisplay();
    displayobj("arm");
    flushdisplay();
}
```

Cycles are among the most common animation abstractions. Example 4 is a one line prototype cycle script that uses the built-in function *stopped* to inquire the status of the script instance, starting it again when it finishes.

```
script cycle()
    event e;
{
    if (stopped(e)) e = start scriptname(arguments);
}
```

Example 5 is the basic structure of a script for interpolated animation. It could be used with an interactive program that simulates a key frame animation system. This program would coordinate the script interpreter and the track editor, maintaining a list of track-object parameter pairs, generating and playing such scripts.

```
track t1, ..., tn;      /* track list */
string o1, ..., on;    /* object list */
string p1, ..., pn;    /* parameter list */
event e1, ..., en;     /* event list */

script kf_animation()
{
    if (t==trackstart(1))
        e1 = start interpolate(t1,p1);

    if (interrupt()) kf_interface(t);
}

script interpolate(tkptr trk; string obj,param;)
{
    if (t==last(trk)) stop;
    setobj(obj, param, spline(trk,t));
}
```

Example 6: A ball bouncing inside a cubic space. The ball trajectory is calculated from an initial position and direction vector, using accelerated motion interpolation to account for the effects of gravity. A collision detection test determines the necessary trajectory reorientation whenever the ball is about to cross one of the cube boundaries. The functions *newtraj* and *collision*, not shown, were written in C for efficiency reasons. *Newtraj* modifies the trajectory track parameters for a new trajectory beginning at the collision point. *Collision* tests if there is any intersection between the ball trajectory and the six planes of the cube. It returns, if there is any intersection, a parametric value in the interval [0.0, 1.0] that corresponds to the intersection point of the ball trajectory and the closest plane. It returns 1.1 if no intersection.

```
track px, py, pz;

script inittraj(tkptr tx,ty,tz; real x,y,z,vx,vy,vz;)
    real gr; /* gravitational constant */
{
    gr = - 0.98;
    tkinsert(tx,0,x,vx,0.0);
    tkinsert(ty,0,y,vy,0.0);
    tkinsert(tz,0,z,vz,gr);
}

script bounce(string env,obj; real x,y,z,vx,vy,vz;)
/* parameters: environment, bouncing object,
 *             initial position, and velocity
 */
    real x0,y0,z0, /* current center */
          x1,y1,z1, /* new center */
          cp;      /* intersection param */
{
    if (t==0) { /* initialization */
        ldoobj(env); ldoobj(obj);
        attachobj(env,obj);
        inittraj(px,py,pz,x,y,z,vx,vy,vz);
        x0 = accmotion(px,t);
        y0 = accmotion(py,t);
        z0 = accmotion(pz,t);
    } /* calc new position */
    x1 = accmotion(px,t);
    y1 = accmotion(py,t);
    z1 = accmotion(pz,t);

    /* collision test */
    if((cp=collision(x0,y0,z0,x1,y1,z1))<1.1){
        newtraj(cp,px,py,pz,x0,y0,z0,x1,y1,z1);
        t = 1 - cp;
        x1 = accmotion(px,t);
        y1 = accmotion(py,t);
        z1 = accmotion(pz,t);
    } /* move the object */
    set(obj,"pxyz",x1,y1,z1);
    /* update curr position */
    x0 = x1; y0 = y1; z0 = z1;
}
}
```

Example 7 is a manipulator for a 3 joint robot arm. The internal joint angles are computed from the arm's link lengths and the three-dimensional goal position that is passed as a parameter to the function. The inverse kinematics solution is derived geometrically for this particular linked structure. The function *movearm* calls the manipulator and updates the arm joint angles.

```
/* l3.s - 3 link arm manipulating function */

real theta0, theta1, theta2; /* joint angles */
real l, l1, l2; /* link lengths */
real ax, ay, az; /* arm position in world coord */

function l3manipulator(real x,y,z;)
    real costheta2, d;
{
    /* translate to arm coordinate frame */

        x = x - ax; y = y - ay; z = z - az;

    /* calculate t0 - rotation to plane x,z */

        if (x == 0 && y == 0)
            theta0 = 0;
        else
            theta0 = atan2(y,x);

    /* test if outside reaching area */

        if ((d=sqrt(x*x + y*y + z*z)) > l) {
            x = (x/d) * l;
            y = (y/d) * l;
            z = (z/d) * l;
        }

    /* calculate t1, t2 - joint angles */

        x = sqrt(x*x + y*y);
        costheta2 = (x*x + z*z - l1*l1 - l2*l2)
                    / (2 * l1 * l2);
        theta2 = -acos(costheta2);
        theta1 = atan2(z,x) -
                atan2(l2*sin(theta2),l1+l2*cos(theta2));
}
```



```
function movearm(real x,y,z;)
{
    l3manipulator(x,y,z);
    setobj("part0","rz",theta0);
    setobj("part1","ry",-theta1);
    setobj("part2","ry",-theta2);
}
```

The demo script combines the bouncing ball script and the arm manipulator function to produce the animation sequence shown in figure 3.1. The robot arm grasps the ball, lifts it up, and throws it against one of the room's walls. The ball hits the wall and bounces back successively until it stops by itself. The ball's initial trajectory, elasticity, as well as the gravitational factor can be controlled by manipulating the corresponding tracks and variables. The script *launch* guides the picking and throwing actions. The arm movement and ball position are linked by a single three-dimensional track, so that adjustments made on the track path affect consistently the whole action. The functions *loadscene*, *loadtracks*, *setview* and *display* perform preliminary set-up and image update operations.

```
/*      dm.s - demo script      */

track lx, ly, lz;

function loadscene()
{
    ldojb("room");
    ldojb("arm");
    ldojb("ball");
    groupobj("scene","room","arm","ball");
    setobj("arm","pxyz",0,-9,-10);
    ax = 0; ay = -9; az = -10;
    l1 = 10; l2 = 11; l = l1+l2;
    movearm(0,-9,11);
    setobj("ball","pxyz",0,0,-9);
}

function setview()
{
    initdisplay();
    initview();
    lookat(50,30,10,0,0,0,0);
    perspective(0.785398, 1.33333333, 1, 1e5);
}

function display(string obj;)
{
    cleardisplay();
    displayobj(obj);
    flushdisplay();
}

function loadtracks()
{
    tksld(lx,"lx");
    tksld(ly,"ly");
    tksld(lz,"lz");
    tkload(px,"px");
    tkload(py,"py");
    tkload(pz,"pz");
}
```

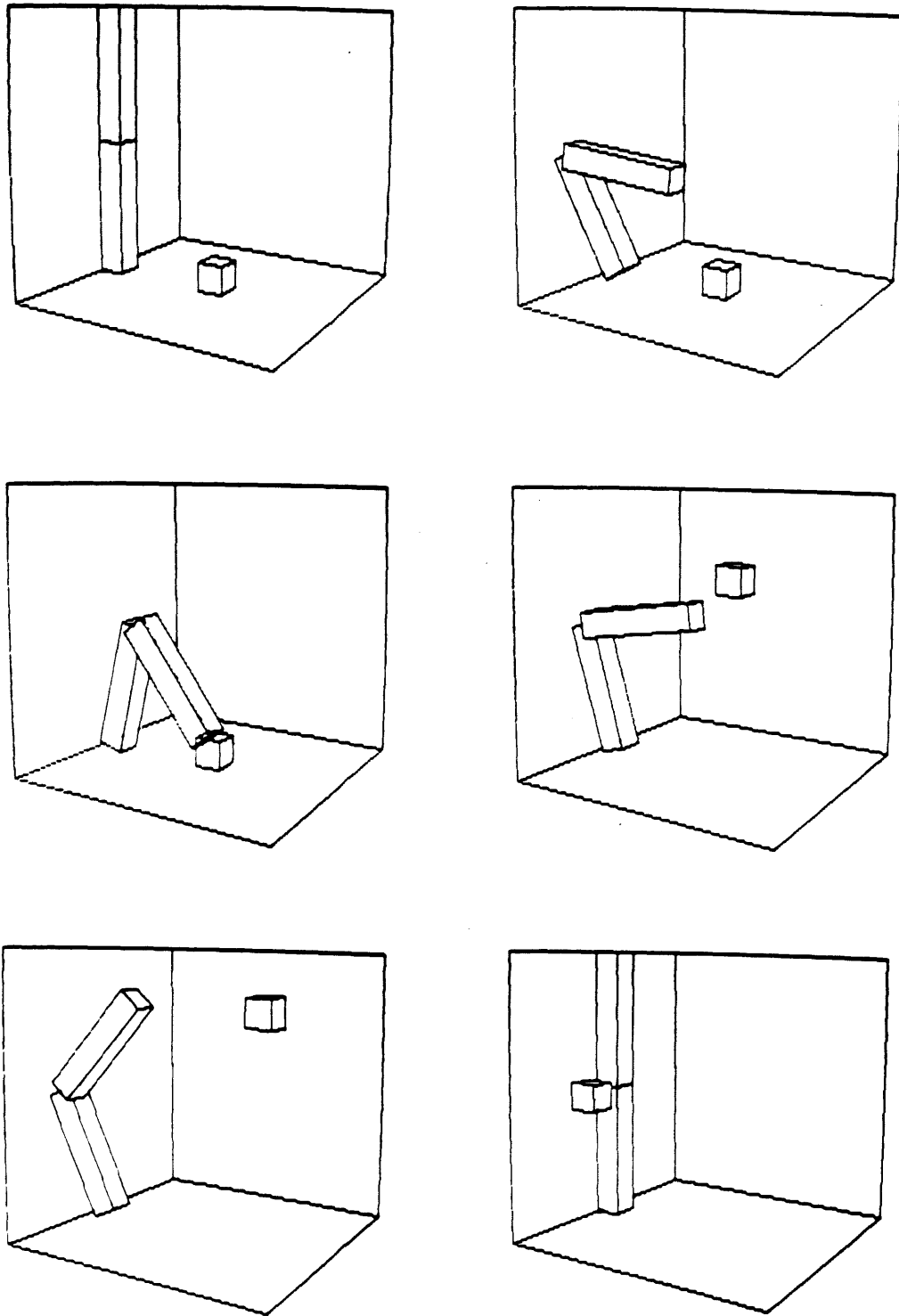
```
script launch()
  real x,y,z;
{
  x = spline(lx,t);
  y = spline(ly,t);
  z = spline(lz,t);
  movearm(x,y,z);
  if (t>tknext(lx,0)&&t<tkprev(lx,tklast(lx)))
    setobj("ball","pxyz",x,y,z);

  on (t == tklast(lx)) stop;
}

script demo()
  real x,y,z, vx,vy,vz;
{
  on (t == 0) {
    setobj("ball","pxyz",0,0,-9);
    start launch();
  }
  on (t == tkprev(lx,tklast(lx))) {
    x = tkget(lx,t,0);
    y = tkget(ly,t,0);
    z = tkget(lz,t,0);
    vx = x - tkget(lx,tkprev(lx,t),0);
    vy = y - tkget(ly,tkprev(ly,t),0);
    vz = z - tkget(lz,tkprev(lz,t),0);
    start bounce("room","ball",x,y,z,vx,vy,vz);
  }
  on (intr()) stop;
  display("scene");
}

loadtracks();
loadscene();
setview();

play demo();
```



**Figure 3-1:** Robot arm and bouncing ball animation sequence

## Chapter 4

# Conclusion

### 4.1 Summary

We have investigated the description of dynamic transformations in the context of a three dimensional computer animation system. Object modeling, animation design, image display and post-processing, four major components of animation systems have been studied in the formulation of a broader picture of the problem. The control methods and abstractions used in animation systems have been further analysed to establish the basis for the development of a flexible software environment for animation production.

We have proposed the integration of interpolated and algorithmic animation in a system that, based on the simulation paradigm, allows animation modeling in layers of functional abstractions, and its specification through multiple interfacing processes.

The central coordinating mechanism in the system is an interactive interpreter for a computer animation language - Script, that supports concurrent synchronized events, and track data structures. The Script language is intended to perform a double duty, in both the description of temporal object transformations, and the prototyping of animation interfaces, playing an important role in the realization of our model of interaction.

A small set of utility, animation and interfacing tools has also been developed to serve, respectively, as primitive functional elements for animation

specification, and as building blocks for interface prototyping.

The design philosophy behind the Script animation system was to create an open mechanism for animation production, that being flexible and extensible would evolve with day to day use. In this sense, the system is half-way completed. It is just a seed that needs the fertile soil of creativity to germinate and grow. Moreover, there is still a number of desirable features that have not been implemented yet.

## 4.2 Extensions and New Directions

The Script system, in its current state, has the minimum necessary features for the design of animation. We intentionally avoided cluttering its development with the details of desirable, but less fundamental features. Now, that the basic implementation is completed, it is time to consider possible extensions and alternative solutions.

Besides animation design, the areas of object modeling, image rendering and post-processing are virtually untouched, leaving a lot of space for improvements.

The Script language would benefit from the addition of several features, among them: a richer set of operators, such as the remainder ( $\%$ ), compound assignments ( $+=$ ,  $-=$ ,  $*=$ ,  $/=$ ,  $\%=$ ), and auto increment/decrement ( $++$ ,  $--$ ); string operations that could be implemented as built-ins, such as copy, compare and concatenation; a three dimensional vector data type; array data structures for the existing data types; and additional control of flow constructs like break and continue. The animation mechanism of the language could be enhanced with the addition of a two-element time structure, time modifiers, and a send/receive communication scheme.

In the area of object modeling, development of free format object's attribute definitions would be desirable, and also, the specification of motion constraints in the object structure.

The areas of image display and post-processing both require full implementation of high-quality image generation procedures. The definition of motion file formats would be necessary for non-real time processing.

The whole system would greatly improve with better virtual memory management facilities. Particularly, this would allow the dynamic linking of script built-in functions and more efficient manipulation of the various data base files.

New directions for work on the subject point towards the research of more sophisticated ways and resources for the description of complex animation problems. Some emerging topics that need to be explored include: collision detection, Newtonian mechanics simulation tools, object manipulators, and knowledge-based animation.

Collision detection is an important pre-condition for the feasibility of automatic motion planning strategies, and it is included in the broader category of scene analysis operations. The research in this area has yet to establish more effective and integrated models for these general operations, and particularly for collision detection. Mechanics simulation and object manipulators incorporate techniques from physics and robotics in the repertory of motion planning animation mechanisms. The challenge here is to translate those methods into suitable animation resources. Knowledge-based animation is a far reaching research area, that is still in its infancy. Its ultimate objective is to simulate the behavior of animated entities through representations and inference methods derived from artificial intelligence.

Finally, a qualitative change in the basic representation of temporal transformations, probably based on continuous simulation techniques, is foreseen in the near future. This would allow the accurate description of interdependent animation events.



## References

- [Aho 79] Aho, Alfred V. and Ullman, Jeffrey.  
*Principles of Compiler Design.*  
Addison Wesley, 1979.
- [Badler 79] Badler, N. and Smoliar, S.  
Digital Representations of Human Movement.  
*ACM Computing Surveys* 11(1):19-37, March, 1979.
- [Baecker 69] Baecker, R. M.  
Picture Driven Animation.  
In *Proceedings AFIPS Spring Joint Computer Conference*,  
pages 273-288. AFIPS, 1969.
- [Boyse 79] Boyse, John.  
Interference Detection among Solids and Surfaces.  
*Communications of the ACM* 22(1):3-9, January, 1979.
- [Burtnyk 73] Burtnyk, N. and Wein, M.  
Interactive Skeleton Techniques for Enhancing Motion  
Dynamics in Key-Frame Animation.  
*Communications of the ACM* 19(10):564-569, October, 1973.
- [Catmull 79] Catmull, Edwin.  
New Frontiers in Computer Animation.  
*American Cinematographer* :1000-1003, October, 1979.
- [cbs 84] Dickens, Bernard L.  
Transition from Analog to Digital Television Plants - A  
Broadcasters View.  
1984.  
Paper presented at the 126th SMPTE Technical Conference.
- [Christopher 82] Christopher, R.  
Digital Animation does Dallas.  
*Videography* :39-42, February, 1982.

- [Cook 84] Cook, Robert, Porter, Thomas and Carpenter, Loren.  
Distributed Ray Tracing.  
In *Computer Graphics*, pages 137-145. ACM Siggraph, July,  
1984.
- [Feldman 79] Feldman, Stu.  
Make - A Program for Mantaining Computer Programs.  
*Software--Practice and Experience* 9(4):255-266, April, 1979.
- [Foley 82] Foley, James and Van Dan, Andries.  
*Fundamentals of Interactive Computer Graphics*.  
Addison Wesley, 1982.
- [Fortin 83] Fortin, D., Lamy, J. F. and Thalmann, D.  
A Multiple Track Animation System for Motion  
Synchronization.  
In *Motion: Representation and Perception*, pages 180-185.  
ACM Siggraph/Sigart, April, 1983.
- [Fournier 82] Fournier, Alain, Fussel, Don and Carpenter, Loren.  
Computer Rendering of Stochastic Models.  
*Communications of the ACM* 25(6):371-384, june, 1982.
- [Franta 77] Franta, W. R.  
*The Process View of Simulation*.  
North-Holland, 1977.
- [Girard 85] Girard, Michael and Maciejwski, A.A.  
Computational Modeling for the Computer Animation of  
Legged Figures.  
1985.  
to appear in Proceedings of Siggraph 85.
- [Goldberg 83] Goldberg, Adele and Robson, David.  
*Smalltalk-80, The Language and Its Implemetation*.  
Addison Wesley, 1983.
- [Green 82] Green, M.  
Torwards a User Interface Prototyping System.  
In *Graphics Interface 82*, pages 37-46. NCGA Canada, May,  
1982.

- [Hanrahan 84] Hanrahan, Pat and Sturman, David.  
Interactive Control of Parametric Models.  
1984  
Course Notes: Tutorial on Computer Animation - ACM  
Siggraph 84.
- [Jankel 84] Jankel, Annabel and Morton, Rocky.  
*Creative Computer Graphics*.  
Cambridge University Press, 1984.
- [Johnson 75] Johnson, C. and Ullman, J. D.  
Deterministic Parsing of Ambiguous Grammar.  
*Communications of the ACM* 18(8):441-452, August, 1975.
- [Kay 84] Kay, Alan.  
Computer Software.  
*Scientific American* 251(3):53-59, September, 1984.
- [Kernigan 84] Kernigan, Brian and Pike, Rob.  
*The Unix Programming Environment*.  
Prentice-Hall, 1984.
- [Kochanek 84] Kochanek, Doris.  
Interpolating Splines with Local Tension, Continuity and Bias  
Control.  
In *Computer Graphics*, pages 33-42. ACM Siggraph, July,  
1984.
- [Korein 82] Korein, James and Badler, Norman.  
Techniques for Generating the Goal-Directed Motion of  
Articulated Structures.  
*IEEE Computer Graphics and Applications* :71-81, November,  
1982.
- [Korein 83] Korein, Jonathan and Badler, Norman.  
Temporal Anti-Aliasing in Computer Generated Animation.  
In *Computer Graphics*, pages 377-388. ACM Siggraph, July,  
1983.

- [Lundin 82] Lundin, Dick.  
Geometric Modeling - A Personal Orthodoxy.  
1982  
Course Notes: Tutorial on Computer Animation - ACM  
Siggraph 82.
- [Maxwell 83] Maxell, Delle Rae.  
Graphical Marionette: A Modern Day Pinocchio.  
Master's thesis, Massachusetts Institute of Technology, 1983.
- [McLaren ] McLaren, N.  
Quotation in permanent exhibition at the National Film Board  
of Canada.
- [Nygaard 68] Nygaard and Dahl Myhraugh.  
*The Simula 67 Common Base Language.*  
Norwegian Computing Center, 1968.
- [Paul 81] Paul, Richard.  
*Robot Manipulators.*  
MIT Press, 1981.
- [Porter 84] Porter, Thomas and Duff, Tom.  
Compositing Digital Images.  
In *Computer Graphics*, pages 253-260. ACM Siggraph, July,  
1984.
- [Potmesil 83] Potmesil, Michael and Chakravarty, Indranil.  
Modeling Motion Blur in Computer-Generated Images.  
In *Computer Graphics*, pages 389-399. ACM Siggraph, July,  
1983.
- [Reeves 83] Reeves, William T.  
Particle Systems - A Technique for Modelling a Class of  
Fuzzy Objects.  
In *Computer Graphics*, pages 359-376. ACM Siggraph, July,  
1983.
- [Requicha 80] Requicha, A.  
Representation of Rigid Solids: Theory, Methods, and Systems.  
*ACM Computing Surveys* 12(4):437-464, December, 1980.

- [Reynolds 78] Reynolds, Craig William.  
Computer Animation in the World of Actors and Scripts.  
Master's thesis, Massachusetts Institute of Technology, 1978.
- [Rivlin 82] Rivlin, R.  
The Arts.  
*OMNI* :32-34, January, 1982.
- [Smith 84a] Smith, Alvy Ray.  
Plants, Fractals, and Formal Languages.  
In *Computer Graphics*, pages 1-10. ACM Siggraph, July, 1984.
- [Smith 84b] Smith, Alvy Ray.  
The Viewing Transformation.  
1984  
Course Notes: Tutorial on Computer Animation - ACM  
Siggraph 84.
- [Stern 83] Stern, Garland.  
Bbop - A Program for 3-Dimensional Animation.  
In *NICCOGRAPH '83*, pages 403-404. Niccograph, Tokyo,  
Japan, December, 1983.
- [Sutherland 63] Sutherland, Ivan E.  
*Sketchpad, A Man-Machine Graphical Communication System*.  
PhD thesis, Massachusetts Institute of Technology, 1963.
- [Sutherland 70] Sutherland, Ivan E.  
Computer Displays.  
*Scientific American* 222(6):56-81, June, 1970.
- [Thalman 83] Thalman, D. and Magnenat-Thalman N.  
The Use of High-level 3-D Graphical Types in the Mira  
Animation System.  
*IEEE Computer Graphics and Applications* 3(9):9-16, December,  
1983.
- [Whitney 81] Whitney, John.  
Motion Control: An Overview.  
*American Cinematographer* :1220-1237, December, 1981.

- [Wirth 77] Wirth, Niklaus.  
Modula: A Programming Language for Modular  
Multiprogramming.  
*Software--Practice and Experience* 7(1):3-35, January, 1977.
- [Zeltzer 82] Zeltzer, David.  
Representation of Complex Animated Figures.  
In *Graphics Interface 82*, pages 205-212. NCGA Canada, May,  
1982.
- [Zeltzer 84a] Zeltzer, David, Gomez, Julian and MacDougal, Paul.  
A Tool Set for 3-D Computer Animation.  
1984  
Course Notes: Tutorial on Computer Animation - ACM  
Siggraph 84.
- [Zeltzer 84b] Zeltzer, David.  
*Representation and Control of Three dimensional Computer  
Animated Figures.*  
PhD thesis, Ohio State University, 1984.
- [Zeltzer 85] Zeltzer, David.  
Towards an Integrated View of 3-D Computer Character  
Animation.  
1985  
to appear in Proceedings of Graphics Interface 85.

# Appendix A

## Script Reference Manual

by  
Luiz Velho

### Abstract

Script is an animation language that integrates algorithmic and interpolated animation based on concurrent synchronized events. It has C style expressions and control of flow statements, C built-in functions, script and track animation constructs. Scripts are like time programs, and tracks like time variables used to describe animation.

### Introduction

The language syntax is patterned after the C language with a few exceptions. Comments, identifiers, operators, numerical and string constants are specified in the same way as in C. The following sections describe briefly the language's syntactic and semantic definitions.

### Variables, Tracks and Events

The basic data types are real and string variables, track and event structures. Real variables are double precision floating point values. String variables hold arrays of characters terminated by a null character. Track structure is a list of marks, manipulated by a set of built-in functions. All operations on tracks are performed indirectly by these functions (see track functions 3L). The only accessible element of a track is its name. Tracks can be declared exclusively at a global level. A track pointer type provides convenient track representation inside functions and scripts. Events are references to script instances, that uniquely identify them, being the link through which operations may be performed on them. These references are returned by the start statement. The data declaration grammar is:

```
data_decl:      REAL namelist;  
               | STRING namelist;  
               | TRACK namelist;  
               | TKPTR namelist;  
               | EVENT namelist;
```

## Functions and Scripts

Functions and scripts are the primary procedural objects in the language. Function variables are automatic, and function recursion is allowed. Actual parameters are passed by value to procedures. Scripts are instanced originating events that are executed in parallel, synchronized by periodic activations at time boundaries. Script instances have a private memory, as well as a predefined local time variable 't'. Activation consists of a time update, the evaluation of all the statements in the procedure body, and the recursive activation of dependent sub-events. Started instances last until explicitly stopped. When an event is stopped, its dependent sub-events are stopped as well. The procedure declaration grammar is:

```
proc_decl:      FUNCTION name ( param_decl )  
                localvar_decl  
                { stmtlist }  
               | SCRIPT name ( param_decl )  
                localvar_decl  
                { stmtlist }  
  
param_decl, localvar_decl:  data_decl_list
```

## Expressions

Expressions are combinations of primary expressions and operators. Primary expressions are numerical constants, reference to variables, function calls, and parenthesized expressions. Binary operators in decreasing order of precedence are:



```
^      exponentiation (right associative)
* /    multiplication, division
+ -    addition, subtraction
> >=  greater, greater or equal
< <=  less, less or equal
== !=  equal, not equal
&& ||  logical and, or (both operands
        always evaluated)
=      assignment (right associative)
```

Unary operators are the arithmetic and logical negation, respectively - and !. Function calls may be regular functions or C builtin functions. Arguments are a possible empty list of expressions separated by commas. Logical expressions follows the convention 1.0 (true) 0.0 false. Non-zero values are considered true. The expression grammar is:

```
expr:   NUMBER
        | variable
        | ( expr )
        | expr BINOP expr
        | variable ASSIGNOP expr
        | UNOP expr
        | function ( arglist )
        | built-in ( arglist )
```

## Statements

Statements specify the elementary actions in the language, that can be expressions, compound, control of flow and animation constructs. The statement grammar is:

```
stmt:   expr ;
        | { stmtlist }
        | RETURN 'expr' ;
        | IF ( expr ) stmt 'ELSE stmt' ;
        | FOR ( 'expr'; 'expr'; 'expr' ) stmt ;
        | PLAY script ( arglist ) ;
        | 'event = ' START script ( arglist ) ;
        | STOP 'event' ;
```

Note: quoted definitions are optional.

Expression statements are usually assignments or function calls. Compound statements group several statements that can be used in the place of one. Control of flow statements are the traditional *if then else*, and *for* loop, and have the same meaning as in C. The return statement is valid only in function definitions. Animation constructs implements the script execution mechanism. *Play* instantiates root level scripts. *Start* and *Stop* initiate/terminate subscript instances in a playing hierarchy. *Stop* alone is a self termination, otherwise it terminates the script instance referenced by the event variable.

## Appendix B

# Animation Tools Programmer's Manual

### NAME

Script - Interactive animation interpreter

### SYNOPSIS

```
script [ input file list ... ]
```

### DESCRIPTION

Script is an interactive interpreter for a computer animation language that integrates algorithmic and interpolated animation based on concurrent synchronized events. It has C style expressions and control of flow statements, C built-in functions, script and track constructs. Scripts and tracks are respectively like time programs and time variables used to described animation.

The input file list is read and interpreted in order. if the file list is empty, the standard input stream is read. A '-' in the place of a file name, also means the standard input. Script file names by convention have the suffix (.s).

### FILES

User/bin/script  
User/data/script/\*.s (script files)

### SEE ALSO

Script: On The Description of Computer Animated Images,  
M.I.T. Thesis by Luiz Velho  
Script Reference Manual

## NAME

read, print, sh, interrupt - script I/O builtin functions

## SYNOPSIS

```
read()
    returns number

print(format, variable list ...)
    string format; real/string variable list;
sh(command line)
    string command line;
interrupt()
    returns logical value
```

## DESCRIPTION

These built-in functions provide basic input/output operations in the script language, and an escape mechanism to execute UNIX commands from the interpreter. Read reads one floating point number from the standard input with scanf format. Print prints in the standard output a variable number of string and real arguments specified by format. Sh accepts a command line passed to the cshell for execution, and waits for its completion. Interrupt returns true if the signal SIGTERM have been sent, otherwise it returns false.

## FILES

User/lib/bltin

## SEE ALSO

Script: On The Description of Computer Animated Images,  
M.I.T. Thesis by Luiz Velho  
Script Reference Manual

## BUGS

the read function doesn't work well with interrupts.

## NAME

ldobj, displayobj, delobj, aliasobj, cpyobj, groupobj, attachobj, detachobj, setobj, getobj

## SYNOPSIS

```
ldobj(filename)
    string filename;
displayobj(objectname)
    string objectname;
delobj(objectname)
    string objectname;
aliasobj(oldname, newname)
    string oldname, newname;
cpyobj(objectname, newname)
    string objectname, newname;
groupobject(jointname, objectname)
    string jointname, objectname;
attachobj(jointname, objectname)
    string jointname, objectname;
detachobj(jointname, objectname)
    string jointname, objectname;
setobj(objectname, parametername, valuelist...)
    string objectname, parametername;
    real valuelist;
getobj(objectname, parametername)
    returns parameter value
    string objname, parametername;
```

## DESCRIPTION

These functions use object representations as described in 3D\_OBJECT\_MODEL\_DATA (5L) for compound and primitive objects and polygonal shapes. Compound objects are hierarchical structures of geometrical transformations and primitive objects. Primitive objects have a 3D solid shape and other associated attributes. Polygonal shapes are the only 3D shape type defined for primitive objects.

**FILES**

User/lib/bltin

User/data/obj/\*.obj, \*.prm, \*shp

**SEE ALSO**

Script: On The Description of Computer Animated Images,

M.I.T. Thesis by Luiz Velho

Script Reference Manual

## NAME

stopped, tkload, tksave, tkkey, tkinsert, tkdelete, tkprev, tknext, tklast

## SYNOPSIS

```
stopped(eventref)
    returns status; event eventref;
tkload(trackfile)
    string trackfile;
tksave(track,trackfile)
    trackref track; string trackfile;
tkkey(trackname)
    returns key value; trackref trackname;
tkinsert(trackname,key,valuelist...)
    trackref trackname; real key, valuelist;
tkdelete(trackname,key)
    trackref trackname; real key;
tkprev(trackname,key)
    returns value;
    trackref trackname; real key;
tknext(trackname,key)
    returns value;
    trackref trackname; real key;
tklast(trackname)
    returns value; trackref trackname;
note: trackref - track or tkptr
```

## DESCRIPTION

These functions manipulate event and track data structures described in TRACK\_DATA (5L). Stopped returns true if the script instance referenced by the event variable does not exist, otherwise returns false. Tracks are doubly linked lists of marks. track functions load track files, and provide basic linked list operations on them. The function tkinsert creates a new element if one with that time key value does not exist already. Track interpolating functions are described in TRACK\_INTERP (3L).

**FILES**

User/lib/bltin/\*

User/data/trk/\*.trk

**SEE ALSO**

Script: On The Description of Computer Animated Images,  
M.I.T. Thesis by Luiz Velho  
Script Reference Manual



## NAME

linear, mkspline, spline, accmotion

## SYNOPSIS

```
linear(trackname,timeindex)
    returns trackvalue;
    trackref trackname; real timeindex;
mkspline(trackname)
    trackref trackname;
spline(trackname,timeindex)
    returns trackvalue;
    trackref trackname; real timeindex;
accmotion(trackname,timeindex)
    returns trackvalue;
    trackref trackname; real timeindex;
```

Note: trackref - track or tkptr

## DESCRIPTION

These built-in functions interpolate between mark values generating a track value corresponding to a time parameter. linear does linear interpolation. spline does cubic d-spline interpolation with local control of tension, continuity and bias. mkspline has to be used to pre-process control parameters at initialization, and whenever they change before calls to the spline function. accmotion generates constant accelerated motion interpolation.

## FILES

User/lib/bltin  
User/data/trk/\*.trk

## SEE ALSO

Script: On The Description of Computer Animated Images,  
M.I.T. Thesis by Luiz Velho  
Script Reference Manual

## NAME

pscollision

## SYNOPSIS

```
pscollision(a,b,c,d,c0x,c0y,c0z,  
            c1x,c1y,c1z,radius)  
returns value:  
real a,b,c,d,c0x,c0y,c0z,  
      c1x,c1y,c1z,radius:
```

## DESCRIPTION

This function makes a collision test between a moving bounding sphere and a stationary plane. The input arguments are the plane parameters, the current sphere center coordinates, the future sphere center if no collision, and the sphere radius. It returns a value between 0.0 and 1.0, corresponding to the intersection of the sphere center trajectory with the plane. If there is no intersection it returns HUGE.

## FILES

User/lib/bltin

## SEE ALSO

Script: On The Description of Computer Animated Images,  
M.I.T. Thesis by Luiz Velho  
Script Reference Manual

## NAME

initview, lookat, polarview, camera, perspective, window, viewport, setbackface

## SYNOPSIS

```
initview()

lookat(vx,vy,vz,rx,ry,rz,roll)
    real vx,vy,vz,rx,ry,rz,roll;
polarview(vx,vy,vz,azimuth,pitch,roll)
    real vx,vy,vz,azimuth,pitch,roll;
camera(rx,ry,rz,nx,ny,nz,ux,uy,uz,deye)
    real rx,ry,rz,nx,ny,nz,ux,uy,uz,deye;
perspective(fieldofview,aspectratio,near,far)
    real fieldofview,aspectratio,near,far;
window(wl,wb,wr,wt)
    real wl,wb,wr,wt;
viewport(vl,vb,vt)
    real vl,vb,vt;
setbackface(bflag)
    real bflag;
```

## DESCRIPTION

These built-in functions define the interface to the viewing transformations. Initview sets up default values for all viewing parameters. Lookat locates the viewpoint at  $(vx,vy,vz)$ , the view normal direction by a vector from viewport to  $(rx,ry,rz)$ , and the view up derived from the roll angle about the view normal. Polarview places the viewpoint at  $(vx,vy,vz)$ , and derives the view normal and view up from the angles azimuth, pitch and roll. Camera has as arguments a reference point  $(rx,ry,rz)$ , the view normal vector  $(nx,ny,nz)$ , and the view up vector  $(ux,uy,uz)$ . The viewpoint is located at a distance deye from the reference point along and in the direction of the view normal. Perspective specifies the viewing pyramid assuming a centered window. The aspect ratio is the ratio of window half dimensions. The field of view angle is the full horizontal angle of the viewing pyramid. Window defines the viewing pyramid by its left, right, top and bottom corners. Viewport similarly defines the screen viewport location.

**FILES**

User/lib/bltin

**SEE ALSO**

Script: On The Description of Computer Animated Images,  
M.I.T. Thesis by Luiz Velho  
Script Reference Manual

## NAME

initdisplay, cleardisplay, flushdisplay, closedisplay, displayline

## SYNOPSIS

initdisplay()

cleardisplay()

flushdisplay()

closedisplay()

displayline(x0.y0,x1.y1)  
    real x0.y0,x1.y1;

## DESCRIPTION

These built-in functions perform basic display control operations. They select automatically between VT125 and VS100 terminal types, according to the environment variable DISPLAY. Initdisplay puts the terminal in graphic mode. Cleardisplay erases the screen. Flushdisplay sends out all buffered display commands. Closedisplay puts the terminal back in text mode. Displayline generates a line on the screen from (x0,y0) to (x1,y1). The coordinates range is 0 to 640 in the horizontal direction, and 0 to 480 in the vertical direction

## FILES

User/lib/bltin

## SEE ALSO

Script: On The Description of Computer Animated Images,  
    M.I.T. Thesis by Luiz Velho  
Script Reference Manual

## NAME

3d\_object\_model\_data - standard format for object descriptions

## SYNOPSIS

\*.obj

\*.prm

\*.shp

## DESCRIPTION

Three-dimensional objects are described by compound object files, primitive object files, and polygonal shape files.

Compound-object files (\*.obj) describe the object's hierarchical structure in a geometric modeling language, with the following grammar:

```
comp_obj:      decl struct

decl:  JOINT : trans rot scale
      PRIM ( primitive file ) trans rot scale

trans:  # NUMBER NUMBER NUMBER
      | nil
rot:    NUMBER NUMBER NUMBER
      | nil
scale:  $ NUMBER NUMBER NUMBER
      | nil

struct:  [ JOINT objlist ]

object:  PRIM
      | struct
```

Primitive-object file (\*.prm) is a list of attribute-value pairs. Currently the only valid attribute is the shape file name.

Polygonal shape files (\*.shp) contain three dimensional vertex coordinates and polygon data of the object boundary description, in the format:

- code (.shp)
- # of vertices, # of polygons
- Vertices [#vertices][3] (x.y.z)
- # of vertices/polygon
- VertexPointers [#vertex/poly]

## **FILES**

User/data/obj

## **SEE ALSO**

Script: On The Description of Computer Animated Images,  
M.I.T. Thesis by Luiz Velho  
Script Reference Manual

## NAME

track\_data - standard format for track descriptions

## SYNOPSIS

\*.trk

## DESCRIPTION

Track files are lists of marks, constituted by a key time index and a variable number of mark values in the format:

- code (.trk)
- # of values/mark
- Marks [#marks][#values/mark]

## FILES

User/data/trk/\*.trk

## SEE ALSO

Script: On The Description of Computer Animated Images,  
M.I.T. Thesis by Luiz Velho  
Script Reference Manual



## Acknowledgements

Nine years ago, I read for the first time about Computer Graphics. It was an article in the Scientific American magazine by Ivan E. Sutherland. I remember how deeply impressed I was about the author's ideas and the research environments he described. This was, for me, the beginning of a long way in the pursue of a career in this field, in which this thesis is a major achievement - and the realization of a personal dream.

I would like to express my gratitude to all who contributed to make this possible.

Special thanks to Noni Geiger, my wife, for convincing me to apply for a master degree in the first place, and to Rodolfo Capeto, my partner in animation projects, for his encouragement.

Thanks to David Zeltzer, whose work has been a major contribution to the computer animation field, and a invaluable source of inspiration to me. It was a privilege to have him as my advisor.

Thanks to Gloriana Davenport for her interest in the project and help in administrative matters. Thanks to Christopher Sawyer-Laucanno for proof-reading and style suggestions.

Thanks to Project Athena for the use of computing resources, and especially to Jim Gettys for providing the VS-100 display software.

Finally, thanks to CAPES for sponsoring my graduate education at M.I.T.