THE UNIVERSITY OF CALGARY

An Integrated Kernel for Computer Animation

BY

Michael Chmilar

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

January, 1990

ISBN 0-315-61690-3

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled, "An Integrated Kernel for Computer Animation," submitted by Michael Chmilar in partial fulfillment of the requirements for the degree of Master of Science.

Supervisor Brian L. M. Wyvill
Computer Science

Graham M. Birtwistle
Computer Science

Bruce A. MacDonald
Computer Science

Lawrence O. Sinkey
Environmental Design/Academic
Computing Services

Michael R. Williams
Computer Science

Date 1990-01-08

Abstract

Computer animation systems often fail to meet the requirements of their users. Animators find that they are unable to create the results they desire, or that the task is too difficult. In this thesis, the methods used to produce three-dimensional modelled animation are examined, as well as their implementation in animation systems. Some flaws in the system designs which cause animators to be hindered unnecessarily are discussed, and a set of general criteria or guidelines for system design is proposed.

A design for an animation kernel which adheres to the criteria is presented. The close integration of the data structures that store model and animation data is proposed as a solution to many of the flaws. This integration allows the kernel to be extensible, script and interactive interfaces to be used, high-level motion control to be added, and model and animation data to be specified in a unified format.

Examples are provided to illustrate the success of the kernel design in satisfying the criteria.

Acknowledgements

I would like to thank Brian Wyvill, my supervisor in this effort, for his guidance, and for letting me go skiing.

The other members of the computer graphics group, Charles Herr, Jeff Allan, Zoran Kačić-Alesić, Anja Haman, Angus Davis, and Chris Bone, have influenced my ideas and, due to the diversity of their work, made me take a wider view in my design. They, along with other graduate students and researchers in the department, have also helped make my stay as a graduate student a very enjoyable one.

Special mention must be made of the graphics and shred god, David Jevans, who was my cohort in creating *Lumpy's Quest for Sev*, and who has taken me skiing and to Sev frequently. He also fulfills the role of bass god in our band. His ideas and encouragement are appreciated.

David Hankinson provided the crazy symbol table code for the parser, took me to Sev, and listened to and appreciated my ideas.

Acknowledgement must also be given to the twins of evil, Bruce MacDonald and Graham Birtwistle, whose comments forced me to write a better thesis.

A scholarship from the Natural Sciences and Engineering Research Council of Canada supported me while undertaking this research.

Finally, I give special thanks to my parents for putting me up, and putting up with me, during my studies.

Contents

List of Figures

Chapter 1

Introduction

The design of a kernel for three-dimensional computer animation systems is investigated in this thesis. Existing animation systems are seen as too inflexible in the set of tools that they provide to their users. The users are not satisfied with the systems: it is often difficult to perform a particular modelling or animation task with them; sometimes it is impossible. These problems can be attributed, in part, to the design of the user interface of a system, or because the system is being used for applications to which it is not suited. However, many of the problems are due to deeper flaws in the design of the systems such as the techniques chosen for modelling objects and controlling motions, the data structures used, the lack of integration of the various processes involved in creating an animated film, and the lack of extensibility of the system, that would allow it to be used for new applications. The kernel design presented in this thesis overcomes these problems, or provides the potential to overcome them; it forms a solid basis on which computer animation systems for many applications can be built. An initial paper on the design of this system is published as [Chmilar 89].

1.1 Statement of Thesis

The close integration of the data structure used to store descriptions of three-dimensional (3D) models in a scene and the data structure that stores motion data

provides the ability for animators to access and control all possible degrees of freedom in the scene. The integration mechanism is effected via parameters which are specified for instances of the basic data elements within scene descriptions. The values of parameters are controlled over time to cause changes to occur in the scene. Each parameter represents one degree of freedom.

The integration mechanism provides these advantages:

- Degrees of freedom may be controlled from many sources, such as low and high-level motion control techniques and user interaction.
- The communication of data through the parameter mechanism allows new scene data types to be added to a system with superficial modification. The degrees of freedom of the new data elements are immediately available for manipulation.
- The changes in parameter values may be monitored so that, between two frames in an animated sequence, only those data elements which have parameter values that are altered need to be re-evaluated.
- The processes of specifying models and animation are unified.

The kernel design presented in this thesis illustrates how the integration mechanism is central to the provision of the following capabilities:

- Interchangeable use of script and graphical user interfaces.
- The ability to extend the set of tools made available to animators by adding new modelling primitives, attribute types, transforms, motion interpolation functions, and motion control techniques to the system.

- Exploitation of temporal coherence to avoid performing unnecessary recalculations between animation frames.

The animation kernel design presented in this thesis is motivated by a desire to create, or have the ability to create, a system which satisfies the following general criteria. A system which satisfies these criteria will serve as a good testbed for animation research, and as a production environment for many animation applications. It will be free of many of the problems that affect other systems.

- Access and control should be granted to all possible degrees of freedom within a modelled scene. This includes control over the position, orientation, and size of models, the attributes of models, and the shape of models. Access should be granted for interactive manipulation, low-level motion control, and high-level motion control.

- The system should be extensible. It must be possible to add new modelling and motion control techniques to the system, in order to satisfy the demands of users and applications. It should be possible to extend the system without disturbing existing facilities. New facilities should be made accessible to users in a manner which is consistent with old facilities.

- High-level and low-level motion control techniques should be available within the same working environment.

- Both scripting and interactive interfaces should be available. It should be easy to switch between them. Many styles of interactive interface should be possible, in order to satisfy different applications and user types. The script language

should be designed in such a way that novices can use it easily, and experienced users may describe very complex models and motions.

- Efficiency is always a concern in graphics systems. It is important for maintaining interactive speed and production rendering speed.

Some of these criteria are applicable to the design of any system or kernel. These particular criteria were formulated from a survey of the requirements of various applications of computer animation, and an examination of existing systems to see where they fail to meet these requirements. They are also influenced by criteria or guidelines specified in [Gomez 86], [Csuri 79], and the requirements of character animation as described in [van Baerle 86] and [Lasseter 87]. This will be discussed in greater detail in the following section, and in chapter 2.

The primary problem with the designs of existing systems is that they use a "reductionist" approach. This is the philosophy whereby a complex system is broken down into its component parts, and each piece performs one task in a process. In an animation system, this means that modelling and animation techniques are isolated. The problems that result from this approach will be illustrated in chapter 2.

The opposite design philosophy is "holism." Holism is the idea that all of the parts interact at such a low level that they cannot be dealt with properly in isolation. The kernel design presented here follows this philosophy. The mechanism for integrating the model and animation data is central to this holistic approach.

1.2 Computer Animation

An overview of computer animation is presented in this section. It describes the basic processes involved in the production of animation, its applications, and the role of an animation system. Some terms are defined. This background provides the context of the kernel's design, and shows some of the motivation for it. Readers who are familiar with computer animation may skip this section.

Computer generated animation is finding widespread application in many fields, including art and entertainment, manufacturing, and data visualisation for science and medicine. The style called *three-dimensional modelled animation* [Thalmann 85] is most commonly associated with the term "computer animation," although the computer is used as an aid in other styles. In 3D modelled animation, a scene is described using representations of objects, called *models*. The models may be moved around, or their shapes and attributes may be changed. Two-dimensional (2D) images of the scene are rendered, one for each frame of animation. The rendering algorithms are becoming increasingly sophisticated, producing very realistic images.

As with ordinary motion pictures, computer animation relies on the property that a rapid succession of incrementally varying static images appears to be fluid motion to the human eye. These images are called *frames*. If the animation is to be stored to film, 24 frames must be generated for every second of animation; for video, it is 30 frames per second (NTSC), or 25 (PAL, SECAM).

Modelled animation requires three processes to be performed: models of objects that comprise a scene are described; the motions of the models over time are specified; and the images for the frames are rendered and stored. [Ostby 89] states that this is

an attractive paradigm, because models are described only once, in complete detail, and the movements of the models are specified from frame to frame.

1.2.1 Modelling

Modelling is the process of building geometric descriptions of 3D objects from which images may be generated. Mathematical descriptions, including numbers, equations, and the relationships among them, are used to describe the shapes of objects; they may represent the surfaces of the objects, or they may contain additional information about objects, such as volume [Mantyla 88]. A model, then, is a representation of an object. The representation must contain enough information about an object to render and animate it.

A model may represent real or imaginary objects or even non-physical ideas or abstractions. A number of individual components may be required to model one complete object. A model may be constructed from components in a hierarchical fashion. The collection of models is a *scene.*

There are many techniques available for representing models. They include polygons, polygon meshes, bicubic spline patches, implicit surfaces, constructive solid geometry, and quadrics, among others. A general taxonomy of modelling techniques, taken from [Allan 88] is presented in figure 1.1. Allan evaluates a number of modelling techniques according a set of criteria: expressive power, precision and resolution, representational fidelity, geometric consistency, convertibility, space efficiency, computational efficiency, manipulation speed, applicability to animation, and applicability to rendering. The large variety of techniques is available because each has properties which are suited to particular applications.

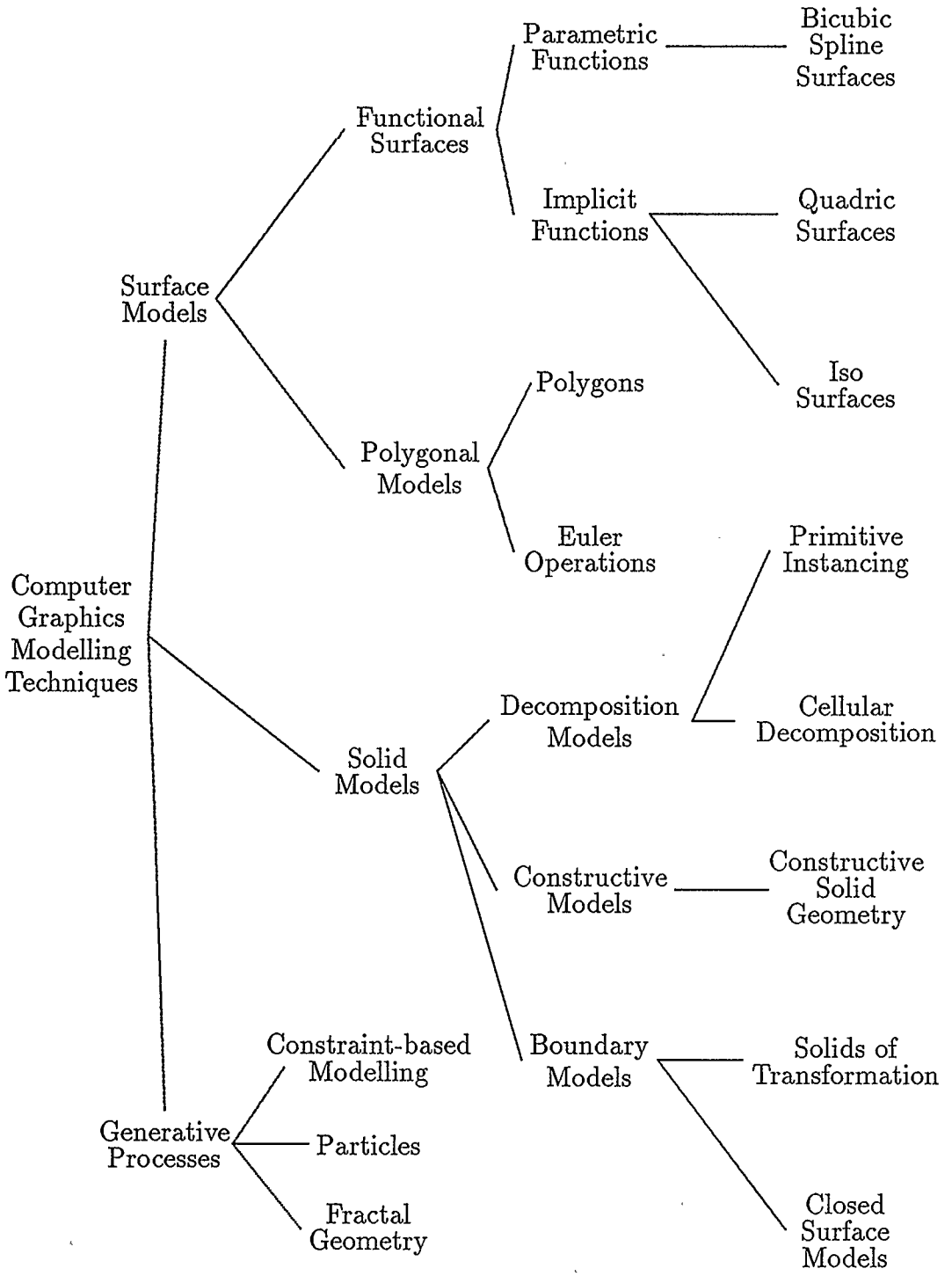


Figure 1.1: A taxonomy of computer graphics modelling techniques [Allan 89]

Allan defines a *modelling technique* to consist of three components: the *description* supplied by the animator, a number of *manipulation methods* or *operations* for altering the description, and the resulting model produced from the description. An example is the polygon mesh modelling technique. Polygons are bounded regions of a plane, and are useful for modelling flat surfaces; a polygon mesh is a collection of connected polygons which share common edges and vertices, and is useful for modelling polyhedral surfaces. Polygon meshes may be described by listing their vertices (points in 3D space) and the edges connecting the vertices. They may be manipulated by "grabbing" and "pulling" vertices into new positions.

A *modelling primitive* is an *instance* of a modelling technique; primitives may have *parameters* which determine their individual characteristics. (A "sphere" primitive has one parameter which defines its radius.)

Models also have *attributes* such as colour, surface texture, reflectivity, and transparency. The attributes just listed are used by rendering programs but, in a more general sense, an attribute may be any additional data that is specified for a model, such as temperature, mass, or age. These attributes may provide information for motion control programs or specialised renderers. The parameters of an attribute indicate its value. For instance, a colour attribute may have three parameters which indicate the intensities of the red, green, and blue components of the colour.

Finally, there is a class of 3D geometric transformations which are generally applicable to all modelling primitives [Foley 82]. These are commonly referred to as *transforms* or *coordinate transforms.* They can be used to translate (move), rotate, and scale primitives along the three coordinate axes (x, y, and z). A *transformation matrix* represents a transform. Matrices may be concatenated together so that one

matrix represents a series of transforms. The order of concatenation is important. The matrix is applied to 3D coordinate points contained in a model's data to transform it. The parameters of transforms indicate the degree to which the transform is to be applied: for translation, they indicate the number of coordinate units to move; for rotation, the number of degrees of rotation; and, for scaling, the scaling factor.

Attributes and transforms are generally independent of a particular modelling technique.

Modelling primitives, attributes, and transforms will be referred to collectively as *scene data.* They are the basic data types from which all scenes are created.

1.2.2 Animation

Motion can be defined as any change that occurs in a scene over time [Thalmann 85]. This applies not only to changes in the positions of models, but also to changes in attributes (ie. colour), or shape—to be precise, it applies to any change in a degree of freedom. This is not an intuitive definition of motion, but it serves to make the following discussions consistent. A *motion control technique* is a method by which the changes may be specified and controlled.

Animators specify motions in relation to *animation time.* This is the time, or the frame number, when the action will occur on the final film or videotape. When generating animation frames, animation time may proceed more quickly or slowly than real time.

Motion control techniques are typically categorised as low-level and high-level. Low-level techniques require explicit control of the motion of each degree of freedom. High-level techniques use algorithms to calculate the motions, based on knowledge

about models and the scene. High-level techniques can be seen as "surrogate" animators: they ultimately manipulate the same degrees of freedom that an animator would with low-level techniques.

1.2.3 Rendering

Rendering is the process of generating an image from a set of models [Hall 89]. In the case of 3D modelled animation, it is the process of creating a 2D image from a 3D scene. The scene is rendered from a viewpoint; this is akin to the use of a "synthetic camera." There are several algorithms used for rendering; the images they generate vary in complexity and realism from simple wireframe drawings to ones which could be mistaken for photographs of real objects.

A close relationship exists between rendering programs and models: the rendering program generates a 2D image from the 3D models; it must be able to ascertain where a model lies in 3D space, in order to determine where it will appear in the 2D image, whether it is occluded by other models, and how its surface will be shaded. The renderer also examines the attributes of models, to determine their colour, texture, etc.

1.2.4 Animation Specification Media

Two specification media for model and motion data have emerged:

Graphical, interactive interfaces allow animators to interact with the models in the scene; they may "select" models displayed on the screen, and manipulate them. Manipulation may include creating and deleting models, positioning them, and altering their shapes.

Language interfaces can be divided into two types: programming languages that are enhanced with data structures and function libraries useful for computer graphics; and *script* languages that are designed specifically for describing scenes or motions. With the first type, programs are written, compiled, and executed to render images or generate image data for a renderer. One drawback is that the program may have to be recompiled whenever changes are made. With the script approach, the animator is not encumbered with the more complex syntax of a programming language: all language statements are related to the specification of model and motion data. The difference between the two approaches can be seen by comparing the program written in the LISP-based ASAS language [Reynolds 82] on page 23 with the script on page 60. The model is shown in figure 2.2 (page 21). Scripts are read into a modelling program and built into a data structure to be processed.

The merits of a graphical interface are clear: computer graphics is a visual medium, so designers of computer graphics should be allowed to work in a visual fashion [Gomez 85]. Using an interactive interface for modelling and animation, it is easy to refine or "tweak" motions until they look correct [Hanrahan 85] [Ostby 89]; Van Baerle considers this to be especially important in character animation, where subtle differences in motion can make large differences in the emotional appeal of characters [van Baerle 86].

[Entis 86], [Sturman 86], [Reynolds 82], [Hanrahan 85], and [Ostby 89] all argue the merits of a language-based approach to model and animation specification:

- Models and motions may be generated according to algorithms.
- A same motion may be difficult to create by interactive techniques; it may also

be "incorrect." This is especially true of scientific applications of animation.

- A script is a precise notation of how a model is to be built, or a motion performed. It may be edited to progressively refine a model or a motion sequence until it is correct. The actions specified in a script are repeatable.

- Scripts may be written on machines which do not have graphics capabilities.

Reynolds states that language-based systems are flexible, and may be easier to extend into "unexpected realms" than interactive systems.

[Ostby 89] and [Hanrahan 85] describe systems which make use of both graphical and script interfaces. In these systems, models are specified in a script, and parameters are defined which may be used to control motions. A model may then be loaded into a graphical interface program, and interactively manipulated.

My personal experience with an interactive modelling interface and a scripting system has shown that it is easier and more efficient to define the general structure of models with a scripting system, but that interactive interfaces are superior for manipulating and "fine-tuning" the models. This is confirmed by Hanrahan.

1.2.5 Applications

The most visible application of computer animation is its use for commercial and entertainment purposes. It is impossible to watch television without seeing commercials and station identifications that use computer graphics (often in the form of "flying logos"). Some theatrical films, such as *Tron* (1982), *Star Trek II: The Wrath of Khan* (1982), and *The Last Starfighter* (1985) have featured sequences of computer animation used as "special effects." The film *Tin Toy* (1988), which features

character animation, won an Academy Award for Best Animated Short.

Character animation is perhaps the most refined form of animation. The goal is to "bring to life" characters so that they have an emotional impact on the viewer. This is done through subtle use of motion and timing. The thoughts and emotions of characters in a story are conveyed through "body language." Specific principles that are useful in creating character animation are listed in [van Baerle 86] and [Lasseter 87]; they are derived from [Thomas 81]. These are the ones that are relevant to animation system design, in regarding to providing the facilities to apply them:

Squash and stretch: Defining the rigidity and mass of an object by distorting its shape during an action.

Timing: Spacing actions to define the weight and size of objects and the personality of characters.

Slow in and slow out: The spacing of "inbetween" frames to achieve subtlety of timing and movement.

Exaggeration: Accentuating the essence of an idea via the design and the action.

Secondary Action: The action of an object resulting from another action.

It is possible to animate 3D character models, using the same techniques employed for flying logos and special effects. Lasseter gives examples of how this may be done. In order that an animator may adhere to the animation principles, an animation system must allow the shapes, attributes, and positions of models to be finely controlled; subtle control must be possible over the timing and speed of motions; and the cycle of modifying a motion or model and seeing the animated result must be short.

Computer graphics and animation also have technical application. Computer-aided design (CAD) is used in manufacturing. Prototypes of items that are to be

manufactured, from circuit boards to automobiles, may be modelled. In some cases, the computer will control the manufacturing of the modelled item as well. The military uses computer graphics for realtime flight simulation, and other forms of training simulation.

Computer animation is used in science and medicine to visualise data for studying fluid dynamics, molecular modelling, geophysics [Upson 89], biology [Leith 89], modelling natural phenomena [Prusinkiew 88], prosthetics [Allan 88], and many other data. The goal is to represent data in a meaningful manner. Representations may be 2D or 3D, abstract or realistic, animated or static; some may allow interactive manipulation of the data. Often, the graphical display is managed through low-level graphics libraries such as GKS and CORE, but it is sometimes possible to represent and animate the data using the modelled animation techniques described earlier in this section. The animation systems from Alias and Wavefront, and the MOVIE.BYU system, which all use these techniques, have been used for some visualisation applications [Upson 89]. The common process in data visualisation is that data is gathered from a source—either gathered from observation, or generated from a simulation—and mapped into a form which can be displayed. Each application of scientific visualisation may have a unique form of data, and unique display requirements. If the data can be mapped into the geometric modelling primitives that are used in modelled animation systems, then such a system can be used as the display medium. Otherwise, a more specialised system is required.

In these technical applications, the need for subtlety and interactive control is often very low, and the need for precision in modelling and motion control is very high.

1.3 Thesis Organisation

A critical review of previous work on animation systems is presented in chapter 2. The evolution of scene description and motion specification techniques is examined. This is followed by a review of animation systems. Critical case studies of three animation systems are made.

The general design of an animation kernel is given in chapter 3. The design satisfies the criteria proposed in chapter 1. This is done by making the kernel extensible and independent of specific applications, and by closely integrating the processes of scene description, motion specification, and rendering. A powerful scripting language, called Charli, is described; it allows integrated specification of scene description and motion control data.

The details of the kernel's design are presented in chapter 4. These include the mechanisms which allow the kernel to be extensible, and by which the data structures used to store model descriptions, motion control data, and rendering data are integrated.

In chapter 5, the kernel's design is examined in relation to the criteria for animation systems. Examples are given to show how it meets these criteria. Conclusions of this research, and suggestions for further research, are offered.

Chapter 2

Review of Previous Work

This chapter provides the following:

- A review of the basic techniques employed in most 3D modelled animation systems, and in the kernel presented here. The purpose is of this persentation is to demonstrate their utility and indicate their diversity. This serves to illustrate the need for an extensible system, and the motivation for integrating the techniques.
- The historical development of computer animation systems is presented.
- Case studies are made of three existing systems to illustrate flaws in them. These case studies influence the design criteria and the kernel's design.

2.1 Techniques

The basic Techniques used in modelling and animation are presented in this section.

2.1.1 Scene Description

Modelling techniques and scene description media have already been discussed. The third element of scene description is the method of structuring the data. The structure has an important impact on what degrees of freedom are available to be animated, and how easy it is to create complex objects that may be animated. The

kernel provides a complex structuring method for an extensible set of scene data types.

The methods of structuring scene description data range from simple enumeration of all modelling primitives, their attributes, and positions, through to complex, generative structures that employ recursive self-references and programming language constructs. Hierarchical structure is used in most systems.

Hierarchical Structure

Structuring models in a hierarchical fashion is logical and useful. Even the earliest modelled animation systems ([Catmull 72] and [DeFanti 76]) used a hierarchical model structure. It allows models that are constructed as separate entities to be bound together. An example is a cup sitting atop a table: they are separately defined models which together form a composite "table assembly" model. 'Table assembly" may be manipulated as a single entity; if it is moved, both of its components maintain their relative positions.

A more sophisticated example is a model of a human figure. The hierarchical structure of a simple human figure is given in figure 2.1. Each segment of the body is represented by a box made from a cube primitive that has been scaled to the correct proportions; a more realistic model would have accurately modelled limbs. Modelling primitives are at the leaf nodes of the tree; the other nodes represent the structural links. Model components must be placed correctly relative to each other, using transforms. A transformation matrix is associated with each node in the tree. Transforms are "inherited" by children in the hierarchy. The transforms which are applied to any node in the hierarchy are also applied to all of its children. This is

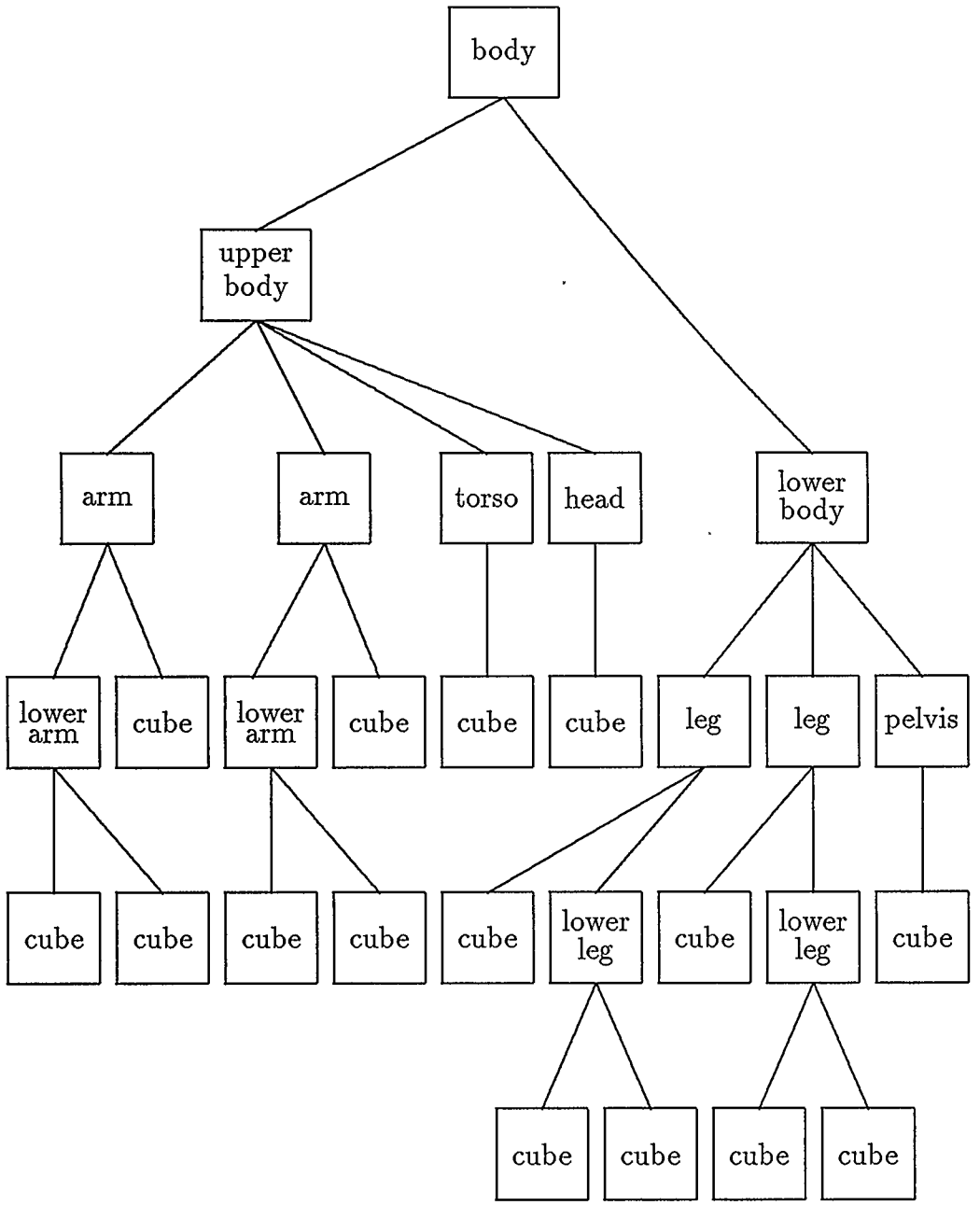


Figure 2.1: A hierarchical human body model

done by concatenating the transformation matrices at the nodes together as a branch is descended. The tree is traversed using a depth-first algorithm. When an edge is descended, the current transformation matrix and attribute list are pushed onto a stack; they are popped upon the return.

With a hierarchical structure, each model is defined in a "local" coordinate space. When it is attached to another model, it inherits the origin of its parent, and so it becomes part of its parent's local coordinate space. Eventually, all models are attached to the "world," and thus become defined in global coordinate space.

Transforms may be applied at any node in the hierarchy tree. This allows models like the human figure to be treated as *articulated bodies.* An articulated body is made from rigid segments that are connected at joints. In an articulated model, each node is a joint, and parameters are available at each joint to control the angle of rotation around the three axes. The parameters affect the transformation matrix of the joint.

[Gomez 86] suggests the possibility of propagating only a selection of transforms down the hierarchy to achieve different forms of attachment (ie. segments that do not inherit rotation transforms, and thus appear to "hang" from their points of attachment). For a "hanging" segment, the object to which it is attached may be moved freely, and the animator does not have to compensate by specifying counter-motions for the segment. An example where this is desirable is a weight attached to a string which is held in a figure's hand; the hand may move, but the weight will always hang down.

The hierarchical structure has problems. They arise from the need to rearrange the tree structure. The classic problem is to make a figure pick up a cup from a

table assembly: initially, the cup is attached to the table assembly, but, at some point in time, it should become attached to the figure's hand. The problem has two parts: first, the tree must be rearranged so that the cup, which is a subnode of the table assembly, becomes a subnode of the hand; second, the absolute position of the cup at the moment of transferral must remain the same—this requires recalculation of its local transformation matrix. [Kroyer 86] proposes the rearrangement of the hierarchical structure, but provides no detail on how this may be best accomplished. [Ostby 89] proposes a simple and elegant solution to this problem: the cup is not structurally attached to either the table or the figure's hand. The global transformation matrix of both attachment points may be determined and stored, and the correct transformation matrix is applied to the cup according to the current animation time (ie. for the first second of animation, the table assembly's matrix is used, and after that, the matrix from the figure's hand is used).

[Kroyer 86] discusses the importance of using a hierarchical structure for animating models. His main point is that the grouping and articulation capabilities that a hierarchical structure can provide are very important in controlling the complexity of animating articulated models.

Recursive Structure

[Wyvill 75] (for 2D systems) and [Wyvill 84] for 3D (in the PG modelling language) enhances the hierarchical structure by adding *recursion.* In this structure, a model may be self-referential, as in figure 2.2. This creates a generative structure: the original data is *amplified* when the structure is traversed. Some classes of fractals [Mandelbrot 82] may be generated using this structure.

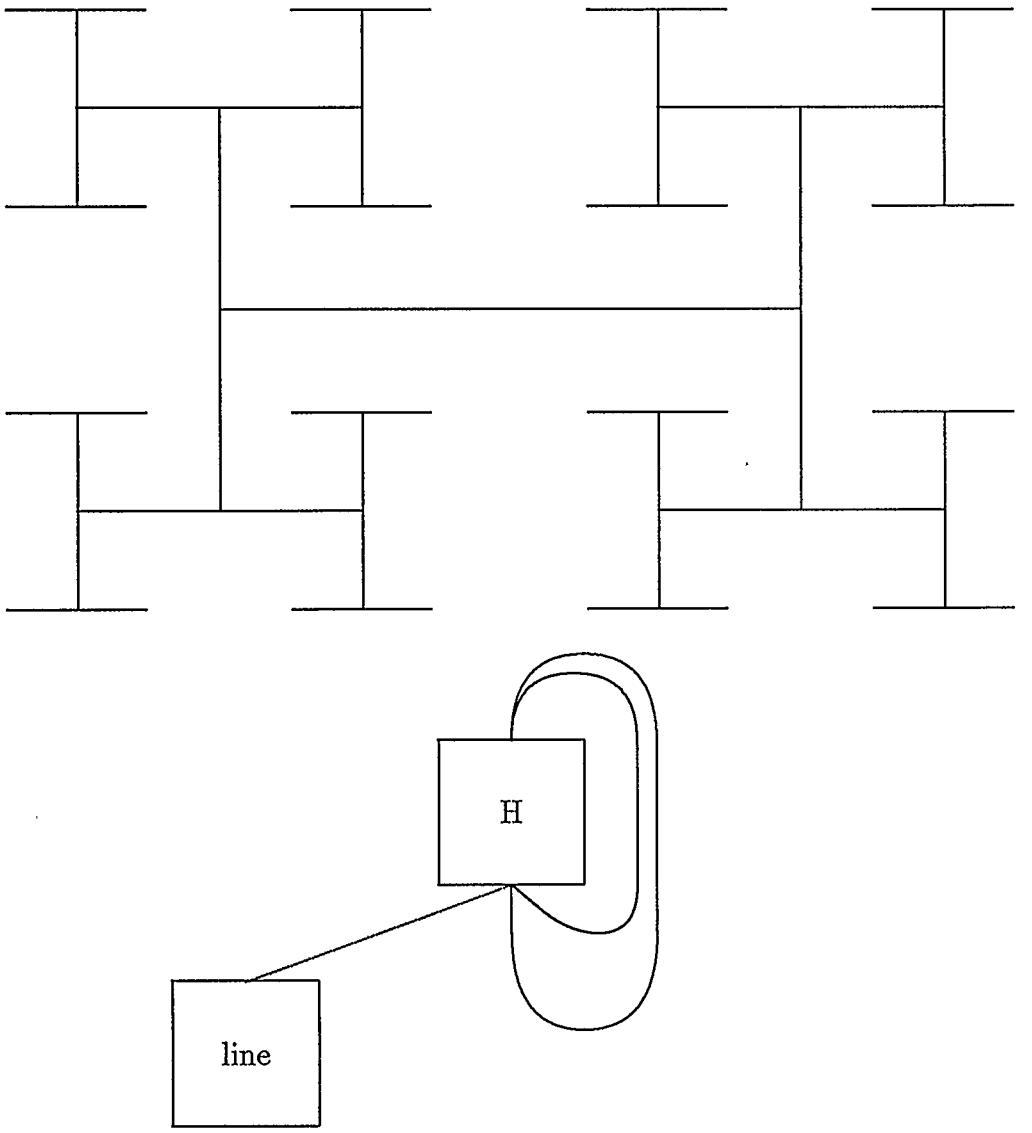


Figure 2.2: Recursive picture and data structure (recursion depth = 5)

A recursion counter must be added to each node in the tree; this counter is used to determine when the recursive traversal will end. The counter is initially set to the number of recursive passes through the model definition that are allowed. It is decremented an each pass, and incremented on each return. When it reaches zero, the next pass is refused, and the recursive descent stops. The depth-first algorithm described for hierarchy traversal is employed; it is modified to use the recursion counter.

Recursive structure is employed in the kernel.

Language Enhancements

Programming languages offer some control facilities that are useful in generating models. They can provide additional generative capabilities. Some of these facilities can be incorporated into script languages and interactive systems.

If an enhanced programming language is used to generate models, it may not use an explicit hierarchical data structure to store model data. However, the structure may be implicit in the flow of execution of the program as it generates the model. Figure 2.3 shows an ASAS [Reynolds 82] program that creates the recursive "H" model; the essential recursive structure of the model can be seen in the "H" definition. (The parameter "H-element" to "H-fractalizer" is a line primitive.) ASAS is an animation language that is an extension of LISP.

The flow control and data structuring constructs available in many programming languages suggest further enhancements to the basic hierarchical data structure:

- Variables may be used to link parameters of primitives, attributes, and transforms, so that a change made to the value of the variable updates a number of

```
(defop  H-fractalizer
        (param:   H-element levels)
        (local:   (total-levels      levels)
                  (offset-dist       0.5)))
                  (sub-H-offset-1    (vector offset-dist 0 0))
                  (sub-H-offset-2    (mirror x-axis
                                             sub-H-offset-1)))
        (H  levels))

(defop  H
        (param:  levels)
        (if (zerop levels)
            (then nothing)
            (else (add-H-level (H (dif levels 1))))))

(defop  add-H-level
        (param:   sub-H)
        (grasp    sub-H
                  (scale  0.7)
                  (move   (vector 0))
                  (rotate 0.25 z-axis))
        (grasp    H-element)
        (subworld (group  H-element
                          (move sub-H-offset-1 sub-H)
                          (move sub-H-offset-2 sub-H))))
```

Figure 2.3: Sample ASAS script that generates the recursive "H" model

elements of scene data.

- Multiple instances of one model may be made, and each may have unique parameters which control its behavior.
- Conditional statements may check the values of variables and parameters, and cause execution to proceed accordingly.
- Looping constructs such as "while" and "for" loops may be used to amplify model data.

Some of these enhancements are employed in the kernel's data structure and script language. These will be discussed in chapter 3.

2.1.2 Motion Control

Low-level Motion Control

The lowest level of motion control requires animators to "pose" each model for every frame in a sequence. Usually, however, the computer is harnessed to calculate a portion of the models' motions over a series of frames.

The BBOP system developed at the New York Institute of Technology [Stern 83] is a 3D *keyframing* system. The keyframe method, for 3D modelled animation, involves interactively "posing" all of the models at certain "key" frames in the sequence. At each key frame, the state of the models is saved. Articulated models are used in the system and parameters to the transforms at each joint are saved. The animation system interpolates the parameters to the transforms for the frames in between the key frames (called *inbetween frames*).

Fine control over motions requires a large number of key frames to be stored. Another problem is that, at most of the keyframes, only the parameter changes of a few transforms are really important. In the BBOP system, *all* parameters are saved, even though they have not changed, or do not need finer control.

NYIT's experience with BBOP led to development of EM [Hanrahan 85]. EM employs the motion control method called *parameter interpolation.* It is known by many other names: *key-parameter interpolation*, *track animation*, *parameter paths*, *direct kinematics.* Each parameter is controlled independently. The values of a parameter at various times in the animation are given, and the system interpolates between the values for intermediate times. Mathematical functions are used to vary the rate of change of the interpolant, to simulate the effects of steady motion, acceleration, and deceleration.

[Gomez 85] attempts to formalise this style of animation as *event driven* animation. He defines a *track* to be set of events that describe the activity of a parameter over the duration of animation time. A track is an abstract data type; it may be queried to find a value for a parameter at a given time. The implementation of a track is hidden, so the value of a parameter may be found by interpolation, or by some others means, such as random number generation or table look-up. This abstract data type is used in the kernel as the mechanism for integrating the model and animation data.

The minimum set of information that is required to animate one parameter is the initial and final values for the parameter, the animation time when the parameter should begin changing, and when it should reach its final value, and the interpolation function. At the start of the sequence, the parameter has its initial value; once the

animation time reaches the point where the parameter should start changing, the interpolation function is used to compute each intermediate value, until the end time for the parameter change is reached. The time/value pairs for parameters are called *cue points.*

A popular and versatile method for interpolating values uses cubic splines to specify an interpolation curve [Kochanek 84]. The spline, as an interpolation function, provides continuity of motion (first derivative) and continuity of acceleration (second derivative). Cue points become the control points of the spline. The animator may also exercise control over the "shape" of the spline curve (as it would be graphed) by altering the control points. When precise time and motion control is required for a parameter, the spline may be subdivided by inserting additional cue points. This is usually done with *B-splines*, as they guarantee second derivative continuity between spline sections. The user can then exercise "local" control at critical points in the action.

Spline interpolation has been criticised by [Voelpel 86] as being responsible for the "sameness" of computer animation, because the resultant motions are too "smooth." This is due to the first and second degree continuity for which splines have also been praised. Van Baerle points out that in character animation, idiosyncratic motion, which may include "jerks" and "limps" and other noncontinuous motions, are desirable. If Gomez's track formulation is used, this type of motion can be accommodated.

It is possible to generate any motion using the parameter interpolation technique, given a rich enough set of interpolation functions. The only limiting factor is the set of parameters which are available to be animated.

High-level Motion Control

The following brief survey of high-level motion control techniques shows their diversity and utility. The kernel provides an interface to incorporate them. A more complete discussion is available in [Wilhelms 86].

Motion trajectory involves moving models along paths in 3D space. The path may be constructed from a 3D spline. The orientation of the model may also be controlled. The rate of travel is controlled from another source.

Inverse kinematics deals with the movement of articulated bodies. With inverse kinematics, the animator may specify a final position for a substructure in a hierarchical model (eg. A figure's hand should be in position to grasp a cup). The system will then decide what movements are required to reach that position from the current one (ie. rotate the shoulder joint by 10 degrees around the x axis, and the elbow by 40 degrees around the z axis).

Stochastic or fractal methods are used in animation to generate random perturbations in motions, or to vary the lengths and starting times of motions. If sparks are to be animated, they could be modelled as a *particle system*, where each spark is given a life-length, initial velocity, and initial direction, based on stochastic sampling [Reeves 83].

Dynamic simulation involves the use of Newton's laws of mechanics to control animation. Realistic looking motion is achieved by simulating the effects of forces and torques on masses. This requires that the masses of models must be specified, as well as the forces and torques that act on them. Dynamic equations of motion are de-

veloped for each degree of freedom, describing the relationships between masses and the forces and torques acting on them. Once all the forces and torques in the system are known, the dynamic equations can be solved, and the motions of the segments are revealed. The use of dynamic control in computer animation has been explored by Wilhelms and Barsky[Wilhelms 85], and Armstrong and Green[Armstrong 85]. Some of the problems involved in controlling the results of dynamic simulation are discussed in [Wyvill 88b].

Goal-directed motion is a term which is applied to many different techniques. What is common among them is the ability for an animator to specify a goal in a high-level manner, and the system works out how to accomplish it. The ideal system would understand very high-level commands and generate the appropriate motions. Some goal-directed systems have been implemented ([Zeltzer 82] and [Calvert 89]), but they operate in highly constrained environments.

Metamorphosis is a type of animation that modelled animation systems do not handle well. Metamorphosis involves the drastically changing the shape of a model, such as changing a teapot into a Volkswagon. The two models may not even be topologically similar. It would be very difficult to animate such a change using the motion control techniques that have already been discussed. Some techniques that attempt to generate intermediate models between two defined models are given in [Hong 88] and [Wyvill 88a].

Algorithmic control involves the use of algorithms to generate very specific motions. The term is generally applied to motion control programs that are written to

generate one particular motion, and are not generally applied.

2.1.3 Scene Realisation

The scene realisation is the data that is produced when the scene description data structure is traversed. This is the complete list of modelling primitives, their absolute positions, and their attributes. If a modelling primitive has multiple instances, all instances are explicitly created. This data is used by rendering programs.

The difference between a scene realisation and a scene description is an important one. This is especially true if the scene description is generative. A scene description contains the instructions for constructing the scene. The scene realisation is the constructed scene.

Pixar has proposed the RenderMan interface [Pixar 88] as a standard interface for data exchange between 3D modellers and rendering programs. It defines a set of function calls which are used to pass data to a rendering program from modelling or animation programs. It contains functions which may be called to describe modelling primitives and their parameters, set attributes, specify transforms, and save and restore the graphics state.

Previously, most systems used *ad hoc* data exchange formats and interfaces.

If the sequence of calls made to the RenderMan interface is saved in a list, a scene realisation data structure is created.

The kernel explicitly stores the scene realisation data in order to avoid regenerating portions of it while rendering a series of frames.

2.2 Historical Background

2.2.1 Conventional Animation

The information sources of the following discussion are [Thomas 81], [Peary 80], and [Heraldson 75].

The best known form of "conventional" animation is the hand drawn type, which is most familiar from the "cartoons" of the Walt Disney, Warner Brothers, and Hanna-Barbera studios. In this type of animation, artists are required to draw an image for every frame in a sequence.

Credit for the creation of the first animation, like the creation of the motion picture camera, is a matter of dispute. It is primarily a matter of defining what is considered as animation. James Stuart Blackton performed a vaudeville act in which he drew faces on a blackboard, and changed them slowly by erasing and redrawing lines; in 1906, he filmed *Humorous Phases of Funny Faces*, by taking single exposures of his changing blackboard drawings. In 1907, Frenchman Emile Cohl created *Mr. Stop*, which used crude individual drawings for each frame.

Many uncredited animated films followed these early attempts; they were used as gags or filler between movie features. Most of these featured simple drawings, and usually forsook background images.

Winsor McCay's *Gertie the Dinosaur* (1909) is usually credited as the first real cartoon animation; it features the "character" of Gertie, drawn before a drawn background, and required 10,000 drawings. This was actually McCay's third animated film; the first two were *Little Nemo*, which required 4000 drawings, and *How a Mosquito Operates*, the first "scientific visualisation."

Early animators such as Winsor McKay drew every frame, including the background, on separate sheets of paper. In 1913, Earl Hurd developed the *cel animation* technique, where characters are drawn on transparent celluloid sheets, and overlaid upon a painted background. This not only reduced the artists' labour, but provided for a more stable background image.

In the early days, one animator would draw an entire cartoon, or a scene in a cartoon. Drawings were made in sequence, and the action was "improvised," with little regard for timing. With advent of sound movies, cartoons were often synchronised to musical soundtracks, creating the need for better control over timing. As well, the increasing length of cartoons, and the desire to tell better stories, caused more elaborate methods of controlling timing and action to be developed.

Eventually, the *keyframe* animation technique evolved. This involved drawing the scene at certain "key" times, when characters were in important positions, and then drawing the frames that fit in between.

Disney's studio released the first feature-length cartoon, *Snow White and the Seven Dwarfs*, in 1937. To produce this film, the Disney Studio developed a system using a hierarchy of animators. At the top is the *director*, who manages all of the animation processes. Next are the *chief animators*, who are responsible for drawing the key frames which define the action, timing, and characters; below them are *keyframe animators*, who draw additional key frames. The lowest level of animators are the *inbetweeners*, who fill in the gaps between the keyframes; this is mostly an automatic process, because the action has been well defined at this point. *Cel painting* is another low-level task; it involves colouring the drawings on the celluloid sheets. This system for animation is in common use today.

In some computer animation, various levels of the traditional animation hierarchy are automated. In some 2D computer assisted animation systems, the jobs of the inbetween animators and the cel painters are taken over by the computer. In 3D modelled animation, the interpolation process eliminates the inbetweeners, and the rendering programs replace the cel painters. There is also no need to "draw" images: each frame is generated automatically by the renderer from the models. Some high-level motion control techniques attempt to perform tasks at higher levels in the animation hierarchy.

Drawn animation is not the only conventional form of animation. The clay animation process involves models made from clay which are posed and photographed for each frame. The same technique may be employed with models made from other materials. This bears a closer resemblance to computer modelled animation than drawn animation.

2.2.2 Early Computer Animation

Computer generated animated films first began production at Bell Labs in 1961 with *Two-gyro gravity-gradient attitude control system* [Zajac 64], [Zajac 66]. It contains a number of sequences of satellite (represented by a box) orbiting a spherical earth; the drawings are made in perspective, and the sides of the box which face away from the viewer are removed. A dozen films were produced by Bell Labs during the 1960's; they consist primarily of two-dimensional mathematical animations and two-dimensional abstract animations. These films were mostly generated using custom programs written in conventional programming languages. [Knowlton 64] and [Knowlton 65] describe a language, BEFLIX, for 2D abstract animation. EX-

PLOR, a library of FORTRAN functions for 2D abstract animation is described in [Knowlton 70]. No attempt was made to emulate the conventional style of animation.

2.3 Computer Animation Systems

Animation systems use a selection of the techniques described above. The emphasis, in this section, will be on the techniques implemented in a system, the structure of the system, the uses of the system, and comments on the system.

2.3.1 Computer Assisted Animation

The phrase "computer assisted animation" refers to the automation of the conventional style of hand drawn animation. The computer is used as a tool to facilitate the conventional animation process: it is used to compute the in-between frames, or to color frames. Computer assisted systems are essentially 2D in nature.

A very early 2D interactive animation program is Sketchpad [Sutherland 63]. It allows 2D images to be drawn on a vector display, and manipulated with a light pen. The components of objects in the images may be structured hierarchically. Animation is created by moving the objects over time, or by making multiple drawings of an object, and cycling through them.

[Baecker 69] describes the GENESYS system, which allows the animator to create 2D shapes, and bind them together so that composite shapes may be manipulated as single entities. Motions are controlled by interpolating the positions of shapes along a path.

ANIMATOR [Talbot 71] is a 2D version of simple modelled animation. Points,

lines, and circles may be structured hierarchically, and the three types of coordinate transforms may be applied to them. Motion control is specified via a script language.

[Burtnyk 71a] and [Burtnyk 71b] introduce the principle of performing in-between calculation by computer, in the 2D keyframe animation system MSGEN. This system worked by interpolating lines between two line drawings, over a series of frames. MSGEN was used to created the Academy Award nominated film, *Hunger* [Foldes 74]. The technique of skeleton animation [Burtnyk 76] was developed to provide more control over the interpolation process.

ANTICS [Kitching 73] is a large 2D animation system which provides many forms of motion control. The principle method is keyframing, but others include wave vibration, path following, random motion, and tracking motion. Multiple "layers" of animation may be composited together, similar to cel animation.

CAAS is a more recent computer assisted animation system. It consists of three components: Tween [Catmull 79] is keyframe system that offers an excellent user interface, and sophisticated interpolation control; SoftCel [Stern 79] performs the cel painting process, and provides antialiasing of the images; Paint [Smith 78] is a paint system, used for painting backgrounds.

2.3.2 Modelled Animation Systems

Modelled systems are those in which models are constructed, and then motion is specified for them. These systems use the 3D modelled animation techniques described above, in varying forms, and with varying success.

Sketchpad III [Johnson 63] is 3D extension of Sketchpad. It allows 3D, hierarchically structured models to be constructed and displayed interactively.

MOP [Catmull 72] uses polygonal models and parameter interpolation of transforms. Hierarchical models and motion is specified with a script language. A realistic human hand animation was produced with this system.

GRASS [DeFanti 76] produced real-time animation on a vector display. Models are contructed from line segments; they may be structured hierarchically. Models and motions are specified with a script.

ANIMA [Csuri 75], ANIMA II [Hackathorn 77], and ANTTS [Csuri 79] are steps in the evolution of an animation system. ANIMA was a language-based system which provided realtime display of vector graphics. ANIMA II added an interactive polygon modeller, enhanced the animation language and rendering, but forfeited realtime display. ANTTS provided high quality, shaded images.

BBOP [Stern 83], developed at NYIT, is an interactive 3D keyframe system. Articulated models are "posed" interactively, and the positions are stored as a key frame. Joint angles are interpolated to produce the in-between frames. Experience with BBOP led to the development of EM [Hanrahan 85]. EM combines scripting with interaction. Models are defined in scripts, and parameters that may be manipulated are declared. The script is read by an interactive program, and the model is displayed. Valuators may be used to change the parameter values.

Twixt [Gomez 85] is an interactive animation system. It uses parameter interpolation techniques. Gomez's comments on the design of animation systems, which influenced his Twixt design, are given later in this chapter.

ASAS [Reynolds 82] is a script language that is an extension of LISP. An example of an ASAS script is given in figure 2.3. It is a very powerful language for scene and motion description. ASAS introduces the concept of "actors" to 3D graphics. An

actor is an object which can send and receive messages. In ASAS, an actor is a model in a scene; all actors can receive and act on messages to translate, rotate, and scale themselves. They may implement additional messages for more complex actions. ASAS also makes use of *newtons*, which are numbers whose values change over time; they are used for interpolation. A sample ASAS script is shown in figure 2.3. ASAS is a system aimed at computer programmers rather than animators; the sample script shows that describing models in ASAS is not a simple process.

CINEMIRA [Thalmann 85] is claimed to be an actor system by its developers. However, it does not contain the message passing facilities of ASAS. It does use animated data types which are similar to newtons; the idea is extended to animated vectors. CINEMIRA is implemented as an extension to Pascal. Like ASAS, it has a very complex syntax, and is very poorly suited for use by animators.

[Ostby 89] briefly discusses an animation system developed at Pixar. This system is notable because many of its design features are similar to those of the kernel described in this thesis. The designs were developed independently. Ostby's system, like EM, allows models to be described in a script, and then manipulated interactively. The script language has powerful data amplification facilities; however, changes made in the interactive system cannot be reconstituted into the script. The script is executed to generate all instances of models before they may be used in the interactive system.

Three other modelled animation systems, the Symbolics system, Graphicsland, and FRAMES will be examined more closely in the case studies, later in this chapter.

2.3.3 Commercial Systems

A number of commercial systems are now available; they provide the combination of modelling, animation, and rendering. These systems do not offer any capabilities that are not available in the previously mentioned systems, but they do tend to offer good user interfaces that are well suited to inexperienced computer users. They also offer very high quality rendering facilities, but they tend to have very simple animation facilities. The systems from ALIAS [ALIAS 87], Wavefront [Wavefront 89], and Vertigo [Vertigo] constrain their users to animating hierarchical models using geometric transforms, and animating attributes. They offer no high-level forms of motion control, and it is not possible to animate shape changes to modelling primitives.

2.3.4 Specialised Systems

Specialised systems use the same techniques as the more general systems described above, but they provide some combination of a fixed set of preconstructed models, high-level motion control, or a user interface that is tailored to a specific type of user.

SAS, the Skeleton Animation System [Zeltzer 82], is aimed at accurately reproducing the motion of human figures, based on skeletal joint motion. The system generates very realistic walking motion, using the dynamics of motion. Skeletons adjust their motion to take account of slightly uneven surfaces.

The COMPOSE system [Calvert 89] is a system for dance choreography. Figures may be set into predefined dance stances at various times, and in various positions on a stage. The system interpolates between the positions. In some cases, such as when a figure moves across the stage, high-level motion control takes effect (in

this case, to make the figure walk, rather than slide, across the stage). New stances may be created, and then used in the system. The user interface to the system is oriented specifically towards the requirements of choreography. The system provides an interactive display of the sequences, but the figures are crudely drawn, and the motion appears jerky. The Cubicomp/Vertigo 2000 system is used for high quality rendering.

2.4 Case Studies

This section examines the problems that are found in systems that implement the techniques discussed earlier. The source of the problems is traced to the use of a reductionist design philosophy. Some motivation is also given for the design criteria listed in section 1.1.

2.4.1 Symbolics

Three modules comprise the Symbolics animation system [Symbolics]:

S-Geometry: A modelling system. It allows interactive manipulation of polyhedral models.

S-Dynamics: A motion controller. It implements the parameter interpolation technique.

S-Render: A rendering system. It renders polyhedral models with a variety of attributes.

The system is implemented using Flavors, an object-oriented variety of Lisp. Because of the language and operating system environment used on the Symbolics computers, the three programs may be used separately, or together with a combined user interface. The interface allows graphical interaction, and is extremely powerful and comfortable to use.

The S-Geometry modeller gives a comprehensive implementation of operations on polyhedra: faces may be extruded, edges and corners bevelled, models twisted, etc. The user interface is well matched to the polyhedron modelling technique. For instance, a face on a model may be selected with the mouse, and extruded interactively, using mouse movements to control the extrusion motion. New modelling techniques may be added to the system; however, models must be converted into polyhedral form before they will appear in the graphical window of the user interface, or before they are rendered by S-Render. Integrating new techniques into S-Geometry would require major reprogramming of the user interface. S-Geometry allows models to be structured in a hierarchical fashion. Bounding boxes may be substituted for complex models in the graphical display, in order to increase interactive speed.

S-Dynamics is used to animate models created in S-Geometry. Its user interface is extremely well suited to the purpose of the program. It understands the hierarchical structure of S-Geometry models, so articulated bodies may be animated. Motions are specified by setting cue points for parameter values. Cues may be set at absolute times, or at times relative to other cues. This allows a cue to be moved in time, and all relatively cued motions will maintain their temporal relationships. Relative cues generate a hierarchy of cues in time. Some extensibility is possible—[Reynolds 87] describes a high-level motion controller that simulates the motion of flocks.

A major problem with this system is that modelling operations cannot be animated. For instance, a face on a polyhedron may be extruded in S-Geometry, but the extrusion cannot be animated by S-Dynamics. This points out a fundamental flaw in the system design: model shapes are "baked" before they are passed to the motion controller. The only way that the motion controller may animate modelling operations is if it contains a re-implementation of the modeller. This problem makes it very difficult to animate shape deformation.

The strong points of the system are:

- It has a very habitable user interface.
- The one modelling technique it uses is powerful and fully implemented.
- The hierarchical cue structure makes it very easy to fine tune motion timing.
- The system can be accessed and extended by writing programs in Flavours.

The weak points of the system are:

- Modelling operations may not be animated.
- Only one type of modelling technique is implemented in the system.
- The system may only be used interactively, on a Symbolics machine.
- The graphical display of objects tends to be rather sluggish.

2.4.2 Graphicsland

Graphicsland is a collection of programs for 3D computer modelling and animation. It was developed at the University of Calgary, and is used at a few other sites. It has

been used to create six short animated films, as well as for numerous other projects. I have been involved in the production of two films: *The Great Train Rubbery* (1987) and *Lumpy's Quest for Sev* (1989). Because Graphicsland is the system which I am most familiar with, it will receive the largest share of criticisms, both mine those of other users. The kernel design presented in this thesis is influenced to a large degree by the stronger aspects of Graphicsland.

Graphicsland is a loose collection of programs. The primary components of the system are a polygon modeller (PG), a simple motion control program (Ani), and a number of renderers (Z-buffer, A-buffer, ray tracing). Secondary components include interactive viewing programs and interfaces, high-level motion controllers (dynamic simulation), and programs which support modelling techniques other than polygons (these are linked into PG so that they are accessible through its interface).

Both PG and Ani have interactive, textual command interfaces. When using PG, it is possible to describe models interactively, or load script files that contain the commands to build models. Various rendering programs and interactive viewing programs may be selected, and model data sent to them. PG stores a log file detailing all of the commands that were issued during a session; it is not able to write its scene description data structure to a file. Ani works in a similar fashion to PG.

PG sends data to rendering programs via the Unix pipe facilities. It traverses its scene description data structure, and sends the model data to the renderer in a binary scene realisation format. The renderers expect to receive polygon modelling primitives, so any other type of modelling primitive used in PG must be converted into polygons before it can be rendered. The scene realisation is not always sent to a renderer; often it is stored in a file, and other programs are used to manipulate the

polygons. PG is able to read scene realisation data files, but all of the data in the file is regarded as one model which cannot be manipulated via the command interface.

Ani operates by starting up a copy of PG, and communicating to it in the same manner as an interactive PG user, by issuing commands to PG through PG's command interface. The process is this:

- PG is asked to load a file containing the model descriptions.
- For each frame, Ani computes the transformation matrices for models that have changed position, and sends them to PG.
- Ani also computes colour changes and sends these to PG.
- PG is asked to send the scene data to the renderer. For each frame, it traverses the scene description and regenerates the scene realisation.

Ani cannot properly access the subcomponents of hierarchical models, so animating articulated bodies is a tortuous task. Only geometric transforms and colours may be animated using Ani. This is mostly due to the interface between Ani and PG. Other attributes could be animated, but this would require major modifications to PG.

Ani provides only simple motion interpolation functions. Any more complicated style of motion must be generated from another source.

For each frame in an animated sequence, non-polygonal models must be converted into polygons. This is done for every frame, regardless of whether the model has changed. This conversion may be a computationally expensive process, especially in the case of the *soft object* modelling technique [Wyvill 86].

For each frame in an animated sequence, the entire scene realisation is sent to the rendering program. For some scenes this involves tens of thousands of polygons. Communicating this volume of data through Unix pipes is not an efficient process.

High-level motion control programs are not linked well into the rest of the system. They often produce their results by writing out a separate PG script for every frame in an animation. The script contains all of the information about the model that is being animated, so the information is often redundant, and time-consuming to read. The common occurrence of this process has resulted in features being added into Ani to manage the separate PG scripts needed for a sequence of frames.

Creating models often involves a roundabout process: an initial version of the model is made in PG, and the model data is written to a data file; the data file is read into a model manipulation program, such as Delta [Allan 88]—a polygon mesh manipulation system—where it is changed; the new model is written into a data file; the model is included in a new PG script, as part of a scene. Figure 2.4 illustrates this process. In order to create one model two PG scripts are required, as well as two data files, and one (or more) model manipulation programs. This is an organisational nightmare, especially when changes are made to the model.

There is no single, unified user interface from which all aspects of the system may be accessed.

Lights and the viewpoint are not part of the modelling hierarchy in PG, so they cannot be attached to other models.

When an animation project is small, and an animator knows the system well (and is an experienced computer user), Graphicsland proves to be a very flexible system. This is due, in part, to the loose organisation that causes many of the other

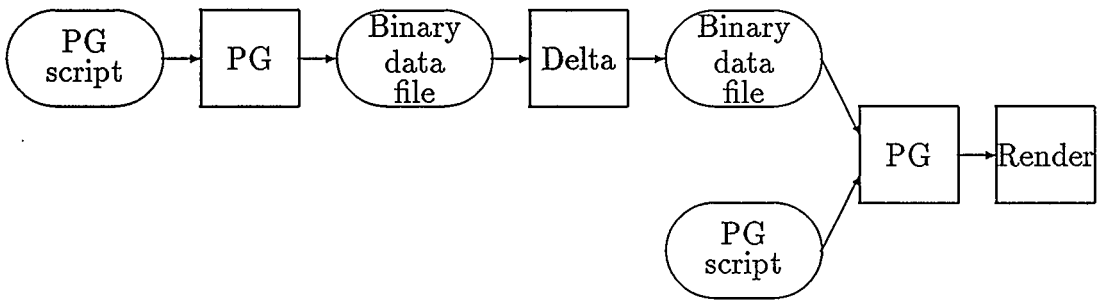


Figure 2.4: Using Delta with PG

problems. It is possible to write small, custom programs to perform any required function.

The strong points of Graphicsland are:

- It contains a very powerful modelling program.
- Many different types of modelling primitives are available for use.
- The system is flexible, because new programs can be added to perform new or specialised tasks.
- It can be used on many different types of hardware, including non-graphical terminals.

The weak points of Graphicsland are:

- Its tools are poorly integrated.
- Its motion controller cannot animate articulated bodies very easily, has limited types of motion interpolation, and cannot animate all possible degrees of freedom in a scene.

- No interactive animation capability exists to allow animators to fine tune their work.

- The system is inherently polygon-based.

Graphicsland contains a large number of tools, and is capable of producing state of the art animation, but the organisation of the tools, and the interfaces between them, make it necessary for users to be programmers in order to fully use the system.

2.4.3 FRAMES

FRAMES [Potmesil 87] uses a set of "graphics tools" analogous to the "software tools" described in [Kernighan 81] to perform 3D modelling, animation, and rendering. The process of creating graphics, and especially of rendering, can be seen as a pipeline of tasks to be performed on a stream of data [Foley 82]; each FRAMES "tool" is a program which implements one of these tasks. The programs read an input stream which is in a standard format, modify or add to it, and output it to the next tool. Tools are provided to create geometric bodies, animate them, shade them, texture them, and so on. Instructions for individual tools are mixed into the data stream. Each tool recognises the instructions meant for it, and passes the rest along. An initial script contains all of the instructions.

This approach has advantages:

- The order of operations (programs) can be changed around for different results.

- New modules may be coded independently and added to the system.

- Modules may be substituted for others to do a similar task at a different quality level or speed.
- Bugs can be isolated.

However, disadvantages also exist:

- The authors developed a "binary" data format for intertool communication because too much system time was spent passing data.
- The communication is unidirectional—intertool feedback is prohibited.
- It would be difficult to develop an interactive user interface to sit atop this system.

In the chain of modelling, animation, and rendering, the order of operations are not interchangeable, so the first advantage listed above is not valid; it is really only of value for a rendering testbed.

The basic advantage to such a modular system is flexibility and the encapsulation of data manipulation algorithms. The primary disadvantage is the assumption that bi-directional communication between model generators and motion controllers is not needed.

FRAMES is an extreme case of modularisation, but most systems do isolate the processes of modelling, animation, and rendering. The S-Dynamics system provides an integrated user interface to modelling and animation, yet treats them as separate processes.

2.4.4 Critical Discussion

A trend that is apparent in the designs of these systems is a tendency towards "reductionism" (cf. section 1.1). In the computer animation systems studied above, it is best illustrated by FRAMES. However, Graphicsland and the Symbolics system also subscribe to a reductionist philosophy, to a lesser degree—the modelling and animation processes remain segregated.

The disadvantages of the "reductionist" approach are these:

- Because the models' geometry and structure has been given before motion control is specified, animation is limited to using the three basic transformations (*translation*, *rotation*, and *scaling*), and changing attributes. It is difficult to animate changes in geometry, because the geometry of models has been committed at a previous stage, and there is no feedback from the animation program to the modelling program.

- It is inefficient to pass large amounts of scene description data between programs. This is especially true if the entire scene description, with small changes, is retransmitted to the renderer for each frame of animation.

- The user must specify three or four types of data, in three or four contexts. If modelling and animation are done with separate programs, there may be two or more interactive interfaces, as it is difficult to provide one unified interface.

- Integration of external high-level motion control programs, such as for *dynamic simulation*, cannot be done unless they can access information about model geometry.

- It is impossible, in a system like FRAMES, to provide both interactive and script interfaces, and allow interchangeable use of them.

Chapter 3

Kernel Design

An overview of the animation kernel will be given in this chapter. The details of the data structures and algorithms used in the kernel will be presented in the following chapter and appendix B.

3.1 Design Decisions

The primary decision made in the kernel's design is to integrate the data structures for models and motion control. The reasons are listed in section 1.1. Many of the other design decisions follow from this.

The other important decision is to provide a kernel for animation systems, rather than a "complete" animation system. This is because a kernel, if it is carefully designed, forms a more flexible basis for complete animation systems. A number of decisions were made regarding the structure of the kernel, and the facilities it should provide.

Other decisions are influenced by the desire to incorporate, or have the mechanisms to incorporate, all of the modelling and animation techniques described in chapter 2, while avoiding the problems discussed in the case studies—especially those of Graphicsland. The intention has been to retain the good elements of the Graphicsland system and fix the problems. The kernel is a framework that can tie diverse components, like those of Graphicsland, together into a more coherent system, and

add considerable power to them. Here is an outline of the major design decisions:

- Both scripting and interactive user interfaces are available. Animators are free to use either type of interface, and they may move between one and the other.
- The kernel does not include an interactive interface, a renderer, or high-level motion controllers as part of its facilities. These are considered to be too application-dependent. Programming interfaces are available to add these components to the kernel.
- The set of scene data types may be extended. It is possible to add new modelling techniques, attributes, and transforms to the kernel with superficial modification. These new resources are available for immediate use by animators, high-level motion controllers, and renderers.
- Parameter interpolation is provided. The set of interpolation functions may be extended. Access to the parameter interpolation facilities is available through the script language, and through interactive interfaces. In scripts, the motion control information is embedded into model descriptions.
- The scene realisation is stored explicitly. The scene description and scene realisation data structures are linked so that, between two frames in an animation, only the parts of the scene realisation that change will be updated, avoiding unnecessary recalculation of model data.
- The script language contains features from programming languages. These include variables and conditional statements. These features are also available through interactive interfaces.

- Interaction occurs at the level of the scene description, not the scene realisation. This is opposite to Ostby's approach [Ostby 89]. It is necessary in order to allow an animator's interactive changes to be stored in a script.

- Modelling primitives may be explicitly converted from one type to another. This is an important and original feature of the kernel. It allows a model to be built using one modelling technique, and then converted into the form of another modelling technique. The operations that are available for the second technique may then be performed on the model, to further refine it.

Systems intended for specific applications may be made by configuring the kernel with the apropriate components. New tools may have to be implemented and interfaced to the kernel.

A complete animation environment may include a number of differently configured kernels. One may be configured with an interactive user interface and a wireframe renderer, to allow animators to design animation sequences; the data is then stored. Another configuration may include a high-quality renderer; this version is used to read completed scripts, and generate the final frames of the animation.

3.2 Kernel Approach

Animation systems become tied to a particular application or type of user according to their user interface, rendering style, modelling technique, and motion control method. What all applications have in common, however, is a need for these resources. A parallel can be drawn to computer operating systems, which can become tied to a particular machine or machine architecture due to poor design—this may

make it perform optimally on one machine, but a price is paid in portability and extensibility An operating system kernel contains the elements of an operating system that are independent of machine type; an animation kernel contains the elements of an animation system that are independent of application. An attempt is made to endow a kernel with very powerful but flexible facilities, in order to satisfy as many potential applications as possible.

3.2.1 Operating System Background

The concept of a kernel is familiar in computer operating system design [Comer 84]. It is used to allow complex operating systems to be portable between machines and configurations of machines. An operating system is primarily a resource manager; it allows users of the system to access the resources of the computer (memory, CPU cycles, file system, printers, etc.). Users issue commands from programs or command interfaces which access these resources and use them to perform a task.

The resources managed by an animation kernel are modelling techniques, attributes, transforms, motion controllers, and renderers. Animators use these resources to create and render animated scenes. They write a script or build a data structure that is "executed" by the kernel to generate the animation.

The resources of a machine may change according to the application of the machine. However, resources may be classified according to general type (ie. disc drive vs. Fujitsu M2392D), and the kernel may treat them generically.

Similarly, the resources of an animation system may change according to its application.

An operating system kernel embodies some design decisions that limit the range

of user environments and applications that it may be used for. The decisions are fundamental ones, such as whether the system should be single or multitasking, what method should be used for interprocess communication, or what measures should be taken to handle deadlock. The proliferation of operating systems testifies that no one kernel has been developed which satisfies the needs of all users and applications.

The same can be said of an animation kernel: it embodies design decisions regarding the data structures it uses to store information about models and motion control, the method of animation it supports, and the interfaces it provides for adding resources and extending the system.

3.2.2 Animation System Structure

Figure 3.1 depicts the basic structure of the kernel. It shows the major components and their relationships. It also shows the external components that make up a complete system.

Within the kernel, the major components are the hierarchical scene description data structure, the interpolation functions and specifications, and the track mechanism that integrates the two. The track mechanism is also used to interface with high-level motion controllers and the user interface, in order to retrieve values from them.

The solid arrows represent strong connections between data structures. The directions of the arrows indicate that one component accesses data from another. The hierarchical scene description data structure accesses data stored in parameter lists and expressions. These may, in turn, be linked to tracks to receive time-changing values. Traversing the hierarchy yields the scene realisation.

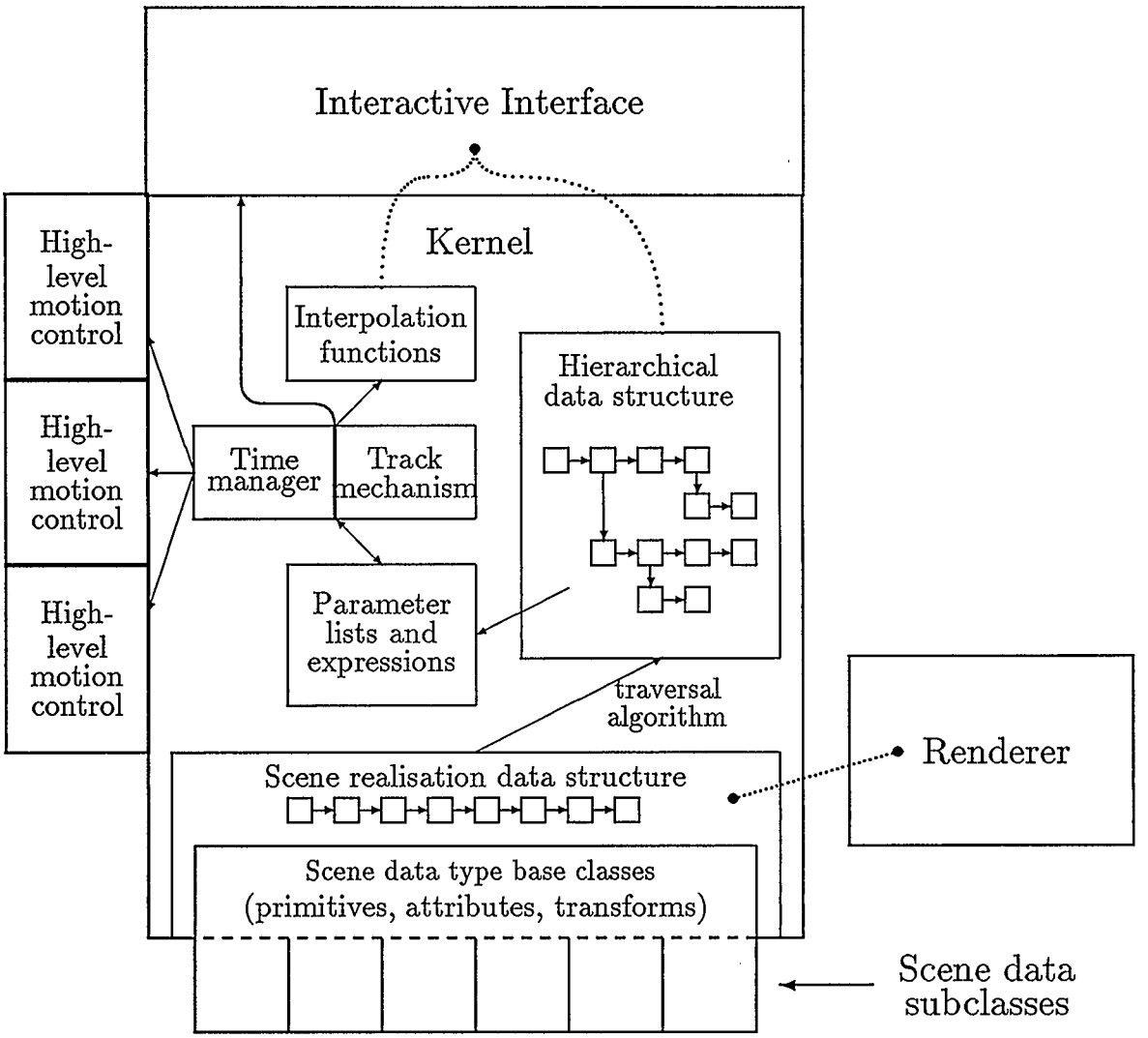


Figure 3.1: Kernel structure

The dotted lines indicate weaker connections. They show that an interface is available to an external system component to examine and modify a data structure. The user interface is permitted to edit the scene description data structure and the the interpolation specifications. The same access is granted to high-level motion controllers (not marked in the diagram). The renderer is allowed to read the scene realisation.

The scene realisation makes use of the object-oriented class derivation mechanism and virtual functions to make the set of scene data types extensible. The dashed line is used to indicate that these extensions, while not part of the kernel, are very closely tied to it. (Virtual functions are explained in appendix C.)

3.2.3 Kernel Facilities

The kernel manages the following data structures:

- The hierarchical, recursive scene description data structure holds the definitions of models, which are made from specifications of modelling primitives, attributes, and transforms. It provides an interface for external manipulation of the data structure, and performs the traversal to generate the scene realisation.

- The parameter lists for the scene data types are particularly important. Parameters are used to communicate with instances of the data types.

- The interpolation data structure stores the cues and interpolation functions for parameters.

- An event list is used to manage animation time and synchronise motion controllers.

- The scene realisation contains data for the renderer. The traversal of the scene description hierarchy generates the scene realisation data structure.

3.3 The Script Language

The Charli script language is important to the kernel's design. It serves as an application independent user interface to the kernel's facilities, as a data storage medium for storing model and motion data structures that have been made with interactive user interfaces, and as a powerful scene description and motion specification language. Charli embodies the integration of model and motion data, as later examples will illustrate. The language grammar is given in appendix A.

Charli examples are presented in advance of the kernel's data structures because they provide a means of illustrating the capabilities of the kernel. As well, the structure of Charli scripts is identical to that of the scene description data structure, so the examples will provide some knowledge of the data structures. Because Charli mirrors the kernel's data structure, it is possible to access all of the kernel's data creation and structuring facilities through it.

It is possible to convert the scene description data structure into a Charli script. This means that a Charli script may be read and converted into a data structure which is then, modified by an interactive interface, and saved as a script. Modifying the kernel's scene description data structure from an interactive interface is equivalent to editing a Charli script, with the benefit that immediate feedback is

provided.

Model and motion specifications are made together in one Charli script. This helps to organise the data for an animation sequence, and it also encourages animators to think of models as entities which change over time. It also permits all accessible degrees of freedom in a scene to be animated.

Some aspects of the kernel's data structures were designed to permit features in the Charli language to operate. These are features modelled after those found in conventional programming languages, such as variables, expressions, and conditional statements. In some respects, the kernel has been designed as an engine to execute Charli scripts.

Charli is independent of any particular animation application. More precisely, it is as independent of any particular application as the kernel.

Statements

Statements in Charli are used to create primitives, attributes, and transforms. Here are some sample statements:

```
polygon( <0,0,0>, <1,0,0>, <1,1,0>, <0,1,0> );

colour( 0.2, 0.5, 0.5 );

rotatex( 45 );

mesh( ... ) extrude( 4, 2 );

polygon( <0,0,0>, <1,0,0>, <0.5,0.66,0.66> ) scale( 0.4, 2, 1 )
                                             rotatex( 45 )
                                             translatex( 10 );
```

The first causes an instance of a polygon primitive to be made. The parameters are points in 3D space that make up the vertices of the polygon. The second causes a change in the colour attribute. The parameters indicate the intensity of red, green, and blue in the interval from 0 to 1. The third causes a rotation matrix to be concatenated onto the current transformation matrix. The fourth creates a polygon mesh (the parameters have been omitted) and indicates that the fourth face of the mesh is to be extruded a distance of two units. The fifth creates a polygon, and applies a list of transforms to it.

Statements are terminated by a semicolon.

Definitions

A model definition is a group of statements that make up a model. The model is given a name so that instances of it may be made. It may have parameters.

```
def square( size )
    colour( 0.3, 0.5, 0.02 );
    polygon( <0,0,0>, <size,0,0>, <size,size,0>, <0,size,0> );
end;

square( 2 );
```

The statements in the body of the definition describe what modelling primitives, attributes, and transforms make up a model.

This example illustrates the use of parameters, and how named values may be substituted for numbers in statements. Parameters are passed by value. Parameters may be numbers, points (three numbers enclosed between < and >), character strings, variables, or references to other models.

The model definition does not cause the model to appear in the scene, it only defines how the model is made. The final statement in the example specifies that an instance of the square model should be made, with the parameter value of 2.

Instances

An instance of the square model is specified like this:

```
square( 2 ) translate( 0, 10, -5 );
```

As the example shows, a parameter is passed to the instance, and a transform is applied to it.

Structure

Charli supports hierarchically structured models. The model definition for a simple body model, given in figure 3.2, generates the model structure shown in figure 2.1. The "cube" statements create an instance of a cube primitive, which is then scaled to produce rectangular prisms for the body components.

This example also demonstrates that models may be defined within a scope. A definition that is given inside the body of another definition is said to be defined within the scope of the latter. This allows scoping rules similar to those used for function names in Pascal [Jensen 74] to apply to model definition names.

Recursion

The technique of recursion is used to amplify model data, as in PG [Wyvill 84]. The following definition produces the model pictured in figure 2.2.

```
def H
    line( <-0.5,0,0>, <0.5,0,0> );
    H scalexyz( 0.7 ) rotatey(  90 ) translatex( -0.5 );
    H scalexyz( 0.7 ) rotatey( -90 ) translatex(  0.5 );
    limit 5;
end;
```

Variables and Expressions

Named values have already been seen in the form of parameters for model definitions. It is also possible to declare variables which can be assigned values, used in mathematical expressions, and passed as parameters.

Here is a simple example of using a variable:

```
def artic_segment
    var seg_length;

    seg_length = 1;

    line( <0,0,0>, <seg_length,0,0> );
    line( <0,0,0>, <2,0,0> ) rotatex( 10 )
                             translatex( seg_length );
end;
```

The model is made from two line segments that are joined in an articulated fashion (the `translatex` transform moves the second line to the second endpoint of the first line). The second line can be made to swing by changing the value of the rotation. The way the model is constructed, using the variable `seg_length`, the length of the first segment may be changed, and the second segment will remain attached to it.

References to variables are resolved using scoping rules similar to those found in Pascal.

```
def body
    def upper_body
        def torso  cube( 1 ) scale( 1, 2, 0.5 );  end;
        def head   cube( 0.5 );  end;
        def arm
            def lower_arm
                cube( 1 ) scale( 0.3, 1, 0.3 )
                          translatey( -0.5 );
                cube( 1 ) scale( 0.3, 0.5, 0.1 )
                          translatey( -1.25 );
            end;
            cube( 1 ) scale( 0.3, 1, 0.3 ) translatey( -0.5 );
            lower_arm;
        end;

        torso translatey( 1 );
        head translatey( 2.25 );
        arm translate( -0.65, 2, 0 );
        arm translate(  0.65, 2, 0 );
    end;

    def lower_body
        def pelvis  cube( 1 ) scale( 1, 0.8, 0.5 );  end;
        def leg
            def lower_leg
                cube( 1 ) scale( 0.4, 1, 0.4 )
                          translatey( -0.5 );
                cube( 1 ) scale( 0.4, 0.2, 0.6 )
                          translate( 0, -1.1, -0.2 );
            end;
            cube( 1 ) scale( 0.4, 1, 0.4 ) translatey( -0.5 );
            lower_leg;
        end;
        pelvis translatey( -0.4 );
        leg translate( -0.25, -0.8, 0 );
        leg translate(  0.25, -0.8, 0 );
    end;

    upper_body;
    lower_body;
end;
```

Figure 3.2: A Charli script for a hierarchical body model

Flow Control

Flow control is provided through the use of an "if" statement:

```
def one_or_two( how_many )
    cube( 1 );
    if( how_many > 1 )
        cube( 1 ) translatex( 1 );
    end;
end;
```

The statements between the "if" statement and the matching end are evaluated if the conditional expression is "true," otherwise they are skipped.

The following example shows how the conditional statement may be used to put leaves at the ends of the branches of a recursively generated tree. The pseudo-variable `limit` returns the current recursion limit value for the definition being traversed.

```
def tree
    def leaves  ...  end;
    def trunk  ...  end;

    if( limit = 0 )
        leaves;
    end;
    if( limit > 0 )
        trunk;
        tree scale( 0.5, 0.7, 0.5 ) rotatez( 30 )
             rotatey( 60 ) translatey( 5 );
        tree scale( 0.5, 0.7, 0.5 ) rotatez( 45 )
             rotatey( -60 ) translatey( 5 );
        tree scale( 0.5, 0.7, 0.5 ) rotatez( 45 )
             rotatey( 180 ) translatey( 5 );
        tree scale( 0.4, 0.5, 0.4 ) rotatez( 60 )
             translatey( 3.5 );
    end;

    limit 6;
end;
```

Casting

Casting is the conversion of a model from one representation, or modelling technique, to another. As mentioned earlier, this is an important and original part of the kernel. An example of the usefulness of this technique is to build a grid of triangles with polygons and recursion, and then convert the grid into a polygon mesh. A hill may then be formed from the grid:

```
def grid
    def square
        polygon( <0,0,0>, <0,0,1>, <1,0,0> );
        polygon( <0,0,1>, <1,0,1>, <1,0,0> );
    end;
    def row
        square;
        row translatex( 1 );
        limit 40;
    end;

    row;
    grid translatez( 1 );
    limit 40;
end;

grid cast to mesh select( <13.4,0,23.87> )
                  range( 5 )
                  decay( "bell" )
                  pull( 3 );
```

Casting between two types of modelling techniques may only be performed if an algorithm exists to perform the conversion.

Animation

The kernel provides facilities for low-level motion control. These facilities may be accessed through Charli scripts. It is very simple to animate a degree of freedom in a scene—all that is required is to substitute values that change over time for numeric values in scripts:

```
polygon( <0,0,0>, <{1 at 0 sec linear 2 at 1 sec},0,0>,
         <1,1,0>, <0,1,0> );

colour( 0.2, {0.5 at 0.2 sec slowinout 0.7 at 3 sec}, 0.5 );

rotatex({ 45 at 3 sec spline 55 at 4 sec spline 70 at 10 sec });

mesh( ... ) extrude( 4, {0 at 0 sec slowinout 2 at 4 sec} );

polygon( <0,0,0>, <1,0,0>, <0.5,0.66,0.66> )
                scale( 0.4, 2, {1 at 4 sec linear 1.2 at 12 sec} )
                rotatex( {30 at 0 sec slowout 45 at 9 sec} )
                translatex({ 10 at 3 sec slowin 15 at 13 sec });
```

The motion specifications are substituted for ordinary numbers in the script; they are enclosed in braces ({}). A specification is made from two or more *cues*. Between any two cues, the name of a type of interpolation function is given. A cue is the combination of a value and a time specification.

These examples are based on the original statement examples given on page 57, but some of their parameters have been substituted with values that change over time: the shape of the polygon changes because one of its vertices moves; the green component of the colour attribute grows stronger; the rotation angle changes over time; the fourth face of the polygon mesh is extruded; and the polygon performs a complex motion.

Animation specifications may also be used in expressions that assign values to variables:

```
def artic_segment
    var seg_length;

    seg_length = { 1 at 0 sec slowinout 3 at 10 sec };

    line( <0,0,0>, <seg_length,0,0> );
    line( <0,0,0>, <2,0,0> ) rotatex( 10 )
                             translatex( seg_length );
end;
```

The current animation time, and the current frame are available through the pseudo-variables `time`, and `frame`. These may be used in expressions, and passed as parameters.

Special Primitives

There are certain modelling primitives that are considered to be special. These are cameras and lights. To animators, they are treated like any other primitive: they may be specified in Charli scripts, and they may be attached to other models in the hierarchy. This permits lights to move with objects, and the camera to track along with models.

3.4 Data Structures

3.4.1 The Scene Description Data Structure

Structure

As the Charli examples show, the scene data is structured using a a recursive hierarchy. The scene description data structure mirrors the structure of Charli scripts.

Scripts are read and converted into a data structure where a node in the hierarchy corresponds to each model definition, and each node contains a list of records which represent statements. In general, one record is made for each statement. The kernel is able to write its scene description data structure as a Charli script. In order to do this, information about the names of models, variables, parameters, etc. are maintained in the data structure.

The important point to note here is that the Charli script is not evaluated as it is read in. It is merely converted into a new form which represents the script. To actually generate the scene realisation, the data structure must be traversed. The specification of a scene data item is nothing more than a specification. It contains the name of the data type and its unevaluated parameter list. The instances of the specification (more than one, if the enclosing model is multiply instanced) actually reside in the scene realisation data structure.

In the hierarchy, a node exists for every model definition. The node contains information about the model, such as its name, recursion limit, and the types and names of its local variables. It also contains a pointer to a list of all of the statements in the body of the model definition. Each statement is converted into a record which represents the statement:

- Statements which specify modelling primitives, attributes, and transforms are converted into records which contain the type of scene data that was specified, and the list of parameters that were given.

- Reference statements are converted into records which contain a pointer to the node for the model definition that is being referenced, and a list of the

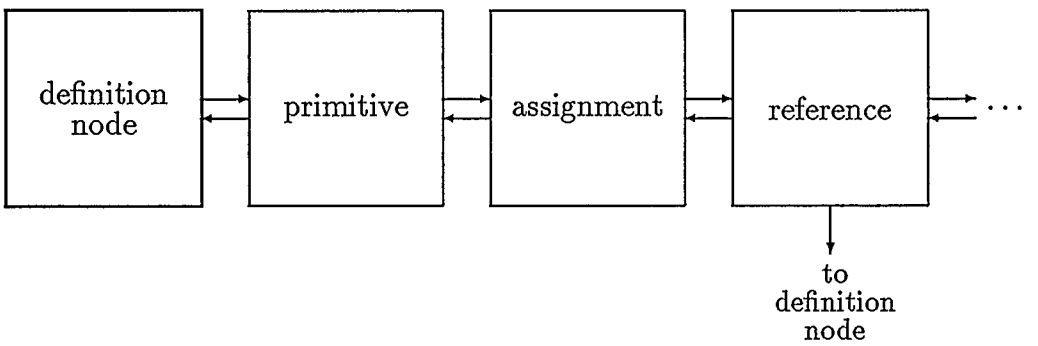
parameters that were given.

- Assignment statements are represented by records which have a reference to the variable to which the assignment is being made, and an expression tree that stores the expression. Expression trees may contain additional references to variables.

- Conditional statements are represented by records which contain the conditional expression that is evaluated, and a pointer to a node that heads the list of statements in the body of the conditional statement.

- Declaration and limit statements alter values in the definition node. They are not represented in the statement list.

This is the basic form of a model definition:



Modelling primitive specifications may be followed by a series of operation, cast, and transform specifications; these are added to new lists, which are headed by the modelling primitive record. Similarly, reference statements may be followed by transform and cast specifications; casts may be followed by operations. These are also stored into lists headed by the reference record.

An abstract data structure for the following script is shown in figure 3.3.

```
def cube
    def square
        polygon( <0,0,0>, <1,0,0>, <1,1,0>, <0,1,0> );
    end;

    def tube
        square;
        tube rotatex( 90 ) translatey( 1 );
        limit 4;
    end;

    colour( 0.7, 0.1, 0 );
    tube;
    square rotatey( 90 );
    square rotatey( -90 ) translatex( 1 );
end;

cube;
```

The script specifies a cube model, and it uses a recursive reference in the tube definition. In the diagram, transforms appear along reference pointers rather than in lists attached to references.

When operations are performed interactively on a modelling primitive, the sequence of operations is stored. This is equivalent to saving a "log" of the operations.

Parameters and Expressions

Some Charli statement types are followed by a parameter list. In the data structure, they are stored in arrays in the statements' records. If a parameter is an expression, the expression is not evaluated; it is made into an expression tree and stored. This is depicted in figure 3.4. The figure shows the data structure built from an

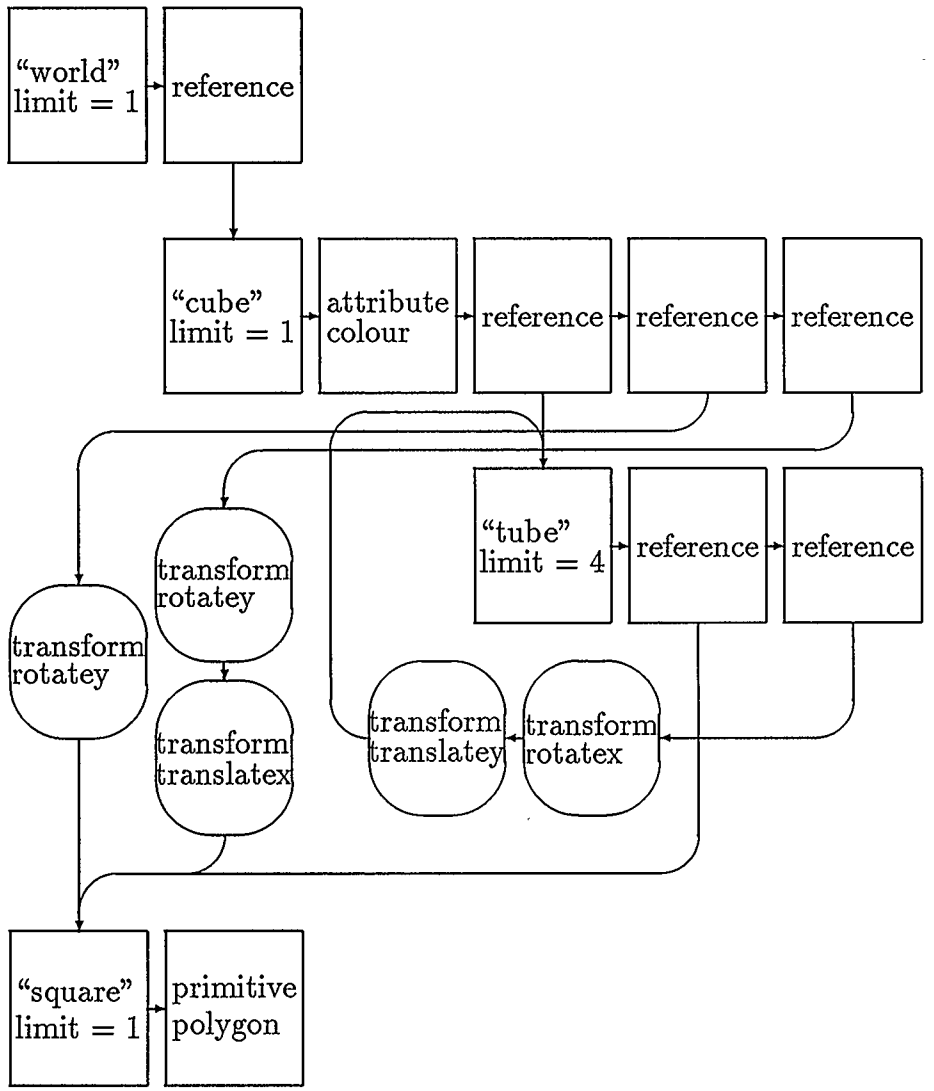
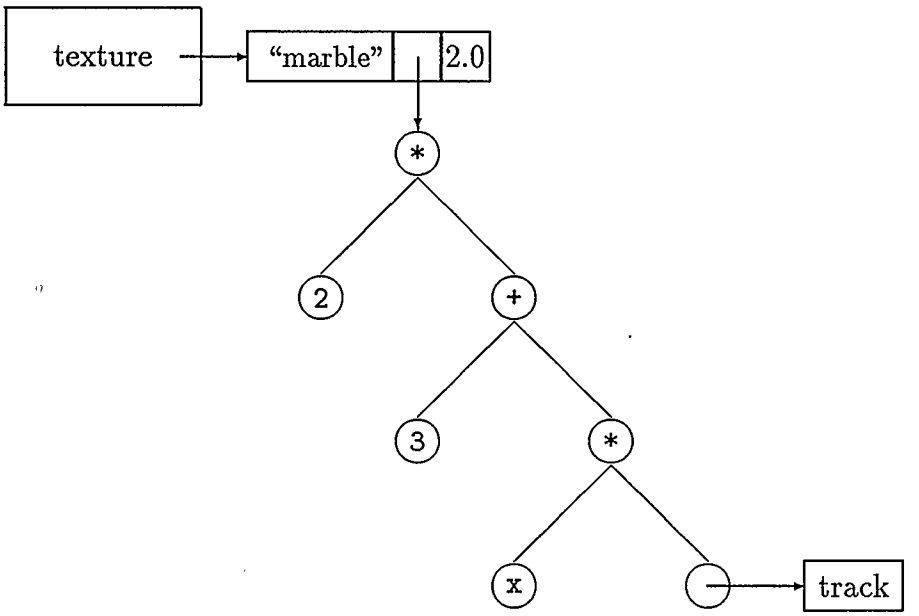


Figure 3.3: Data structure for the cube script.




```
texture( "marble", 2*(3+x*{0 at 0 sec linear 2 at 5 sec}, 2 );
```

Figure 3.4: Parameter list example.

attribute specification that has a complex parameter list including an expression (which includes an interpolation specification) and a character string.

Expressions may also appear in conditional and assignment statements, and are treated similarly.

Traversal

The traversal of the scene description data structure can be seen as equivalent to "executing" a Charli script, as if it was a program written in a procedural programming language like Pascal. The basic traversal uses the algorithm mentioned in section 2.1.1. The statements in the body of a model definition are executed se-

quentially. Executing the script generates the scene realisation. This data is stored in a separate data structure. The result of executing each statement depends on its type:

- Statements which specify modelling primitives, attributes, and transforms generate data into the scene realisation data structure. The parameter lists of the statements are evaluated, and actual instances of the data types are created. Because of the structure of Charli scripts, each specification statement may be executed many times during one traversal, and its parameter values may be different each time.

- A reference to a model definition is similar to a procedure call. The definition of the referenced model is executed. A stack is maintained during the traversal of the data structure for local variable storage.

- The expressions of assignment statements are evaluated, and the resulting value is assigned to the variable.

- The statements in the body of a conditional statement are executed if its condition evaluates to "true."

Expressions and parameter lists are evaluated when the statement record containing them is processed.

If a modelling primitive has a list of operations, the actual instance of the primitive is generated for the scene realisation, and the operations are applied to it. If a modelling primitive is cast, the instance is made, but it is not placed into the scene

realisation; instead, the conversion is performed, and the new data is added to the scene realisation.

It is more complicated when a model definition is cast: the model may contain a number of modelling primitives. This is handled by traversing the model, the realisation of the model is added into a secondary scene realisation data structure, rather than the primary one. This is described fully in chapter 4. The conversion algorithm must then examine the secondary data structure to find all of the primitives, and work from there. It is therefore necessary for conversion algorithms to understand the format of the scene realisation data structure.

Extension

The set of scene data types—modelling primitives, attributes, and transforms—may be extended. When new types are added, they are accessible through Charli scripts and interactive user interfaces.

To add a new type, the following steps are taken:

- The details of the new type are implemented in a class which is derived from a base class provided by the kernel. The subclass defines the internal representation of the new type, as it will be constructed from its parameters, and implements the virtual functions. (The base classes are actually part of the scene realisation data structure, as the scene description merely contains placeholders.)

- The name of the new class is entered into a table, along with a pointer to a function that will create an instance of the class. Separate tables are used for primitives, attributes, and transforms.

- Instances of the new type may be made by calling functions in the kernel interface.

If a new primitive is added, these additional steps may be performed:

- For each type of operation that may be performed, a class is derived from class `operation`. This class will implement a function that reads and interprets a list of parameters, and operates on an instance of a primitive.

- The name of the operation, and a pointer to a function that creates an instance of the class is added into a table of operations. Each primitive class has its own table of operations.

- Algorithms may be implemented to convert one type of modelling primitive to another. Pointers to the functions are entered into a conversion table.

Prototyping

In order to maintain reasonable performance with interactive interfaces, it may be desirable to replace a complex model with a simpler representation, or *prototype*. This is accommodated by adding a field to the model definition node which points to the prototype, and a flag indicating whether the prototype is to be activated.

When the prototype is activated, the model is not traversed. The prototype representation is rendered instead. This also has the side effect that any animation that is specified within the model is not generated.

The prototype may be stored to a Charli script, using a prototype statement:

```
prototype modelling_primitive;
```

Once a model or a piece of animation has been completed, it may be "turned off" with a prototype, so that other aspects of the scene may be worked on. During final production, prototypes are ignored.

3.4.2 Motion Control

The basic method of producing animation is to traverse the scene description data structure once for every frame that is rendered. At each traversal, some of the parameters passed to scene data types will be changed from the previous traversal.

Tracks

All forms of motion control in the system are realised through a track mechanism similar to the type that Gomez formulates. Essentially, any numeric value that appears in an expression in the scene description data structure (or a Charli script) can be replaced with a track. A track is an abstract data type that may be queried to return a value for a particular instant in animation time. When a numeric value is replaced with a track, then its value can change as animation time changes. The implementation of a particular instance of a track is not important; the only important thing is that it returns a value.

To be practical, the implementation of a track is important. The value it returns may result from an interpolation process, it may be computed by a high-level motion control algorithm, or it may come from user interaction, as shown in figure 3.1.

The track is the mechanism that integrates the scene description and motion control data structures.

The implementation of the track data structure is presented in the next chapter.

Low-level Motion Control

The low-level motion control component of the kernel uses interpolation techniques. Interpolated values are available through tracks. The Charli examples show how the interpolation tracks are specified: the minimum information required for an interpolation track is two cues, and an interpolation function. A cue consists of a time value and a numeric value. Additional cues and interpolation functions may be added to an interpolation track. This information is read and stored in an abstract data structure called an *interpolator*. Interpolators may also be specified from interactive user interfaces, or by high-level motion controllers.

An interpolator is queried with a time value, and it returns a numeric value.

The set of interpolation functions is extensible. An interpolator is also not strictly required to perform interpolation. It may actually return a random value, or a value calculated through some other means.

High-level Motion Control

High-level motion control tracks are used to interface high-level motion controllers to numeric values in the scene description data structure. For every parameter that will be controlled, a track is made. The high-level controller is responsible for updating the values that are read through the track.

Some forms of high-level motion control are computationally expensive, so the calculations they perform should not be repeated unnecessarily. For this reason, high-level motion control tracks capture the values that are passed through them at each frame, and store them. The saved values constitute a type of interpolator, and may be saved in a Charli script. More importantly, an animator is now free to

modify the values received from the high-level controller, if he wishes to.

User Interaction

User interaction with models in a scene is implemented via tracks. In this case, a track is used to read values from a valuator that an interactive user is controlling, such as a mouse or a slider. This type of track differs from the others because its value changes in real time, rather than animation time. Animation time may stay static while the user interacts with the scene.

Valuator tracks allow animators to alter degrees of freedom in a scene interactively. Models may be posed by an animator at various points in time. When the animator signals that he has finished interacting with a particular track, the track is deleted, and its final value is substituted for the number or cue value.

Synchronising Motion Control

An event list similar to the type used in discrete event simulation [Birtwistle 79] is used to synchronise high-level motion controllers with animation time. An event list is a sequence of *event notices* sorted according to time. Each event notice contains a time value, and a pointer to a function that should be executed at that time. The kernel steps through the event list, and executes the function belonging to each event notice. The time value in the event notice that is currently being processed is the *current animation time.*

The kernel places a notice in the event list for each frame that is rendered. The function that belongs to these frame event notices sets up the kernel to render a frame, and tells the kernel to traverse the scene description.

High-level motion controllers also place event notices in the event list. They

require an event for each point in animation time that they will want to perform some processing function to update the tracks they are controlling.

Event notices are sorted according to a priority value that they contain, as well as by time. This is to allow events that occur at the same animation time as a frame event to be processed before or after the frame event.

While an event is being processed, the processing function may add new event notices into the event list. The only restriction is that new event notices must have time values later than the current animation time.

The low-level motion controller in the kernel does not use the event list. The values of tracks that are influenced by interpolation are computed during traversal.

3.4.3 Scene Realisation

Scene realisation data is generated when the scene description data structure is traversed. It is stored. Rendering algorithms access the data. The format of the data structure used is similar to the implicit one of the RenderMan interface.

The data in the scene realisation includes modelling primitives, attributes, and transforms. Their parameters have been evaluated (ie. expressions have been evaluated) and their internal representations have been built. All specified operations have been performed on modelling primitives.

The basic format of the scene realisation data structure is a linear list of data. Each item in the list provides information to a renderer. The information includes primitives, attributes, and transforms, and instructions to the renderer to push and pop the current graphics state. The graphics state is the current transformation matrix, and the current attributes.

The scene realisation for the "H" script on page 60 would contain the data in figure 3.5. Indentation has been used to accentuate the hierarchical structure create by the "push" and "pop" instructions. The data structure is highly repetitive because every instance of scene data types is made explicitly. This is necessary because each instance may have different parameter values.

Certain modelling primitives—those that represent the camera and lights—are treated specially. Their transformation matrices are stored, and they are put into a special list in the scene realisation. This is because a renderer must know where the camera and lights are in the scene before rendering can commence.

Traversal of the data structure is straightforward. The list is scanned by the renderer. At each item in the list, a task must be performed:

- If the item is push or pop instruction, the graphics state is pushed or popped.
- If it is an attribute, the value of the attribute is changed in the attribute list.
- If it is a transform, the transform is concatenated onto the current transformation matrix.
- If it is model, the internal representation is read and rendered. This involves applying the current transformation matrix to the data, and consulting the attribute list.

Modelling primitives in the scene realisation data structure must be in a form which the rendering algorithm can handle. However, primitives which cannot be directly rendered may be specified in the scene description. This problem is solved by casting modelling primitives into a form that the renderer can cope with, before

```
push
  line( <-0.5,0,0>, <0.5,0,0> )
  push
    translatex( -0.5 )
    rotatey( 90 )
    scalexyz( 0.7 )
    line( <-0.5,0,0>, <0.5,0,0> )
    push
      translatex( -0.5 )
      rotatey( 90 )
      scalexyz( 0.7 )
      line( <-0.5,0,0>, <0.5,0,0> )
    pop
    push
      translatex( 0.5 )
      rotatey( -90 )
      scalexyz( 0.7 )
      line( <-0.5,0,0>, <0.5,0,0> )
    pop
  pop
  push
    translatex( 0.5 )
    rotatey( -90 )
    scalexyz( 0.7 )
    line( <-0.5,0,0>, <0.5,0,0> )
    push
      translatex( -0.5 )
      rotatey( 90 )
      scalexyz( 0.7 )
      line( <-0.5,0,0>, <0.5,0,0> )
    pop
    push
      translatex( 0.5 )
      rotatey( -90 )
      scalexyz( 0.7 )
      line( <-0.5,0,0>, <0.5,0,0> )
    pop
  pop
pop
```

Figure 3.5: The scene realisation for the "H" script

they are added to the scene realisation. This conversion is not specified in the Charli script or the scene description data structure, it is performed by the kernel. The rendering algorithm must provide a list of the types of primitives that it can use to the kernel. The kernel then attempts to cast the unlisted primitives to types that are on the list.

Extension

When new scene data types are being added to the kernel, the hardest task is implementing them for the scene realisation data structure.

Modelling primitives must read their parameter list, and build an internal representation from them. Then they must examine the list of operations, and apply them to the internal representation. If a cast is applied, it must call the appropriate conversion function to create a new primitive.

Transforms read their parameters, and construct an appropriate matrix, which they store internally.

Attributes read their parameters, and store them. They may be required to do some processing. For instance, if the colours are represented as red, green, and blue intensities, but specified as hue, saturation, and value, the conversion between the two must be computed.

The base classes in the scene realisation, and the use of virtual functions, is described in the next chapter.

Linking to the Scene Description

There is one reason for saving the scene realisation data: between frames in an animated sequence, or while an interactive user is modifying the scene description

data structure, very little of the scene data actually changes. Therefore, there is no need to regenerate the entire scene realisation. It is more efficient to merely update the parts of the scene realisation that have changed. This is especially important if modelling primitives are used in a scene which must be cast to another representation before they can be rendered, and which are computationally expensive to cast.

In order to update only those parts of the scene realisation data structure that have changed, the correspondence between scene data specifications in the scene description data structure and their instances in the scene realisation data structure must be stored. A change is signaled by a change in the parameter values that are passed to scene data types. The problem is compounded because one specification in the scene description may create many instances in the scene realisation. The following example illustrates the problem:

```
def thing( p )
    colour( p, p, p );
end;

thing( 0.4 );
thing( { 0 at 0 sec linear 0.9 at 10 sec } );
```

Two instances of the `colour` attribute will appear in the scene realisation, but the parameters to one of them never change, and it does not need to be updated, while the other does have parameters which will change over time.

Each modelling primitive, attribute, and transform specification in the scene description data structure could maintain a list of its instances in the scene realisation data structure, but this is not necessary. On each traversal of the scene description, the scene realisation data will be generated in the same order, so it is easy to step through the existing scene realisation in synchronisation with the traversal. The

parameters to the instances in the scene realisation may be compared to the evaluated parameters of the specification record.

3.4.4 Lazy Traversal of the Scene Description

The notion of lazy traversal is to only update the parts of the scene realisation data structure that have changed. This is done to avoid recomputing the internal representations of the primitives in the scene realisation, and to avoid recomputing casts.

The approach is as follows:

- The scene description is traversed once. All instances of modelling primitives, attributes, and transforms are created; all operations are performed; and all casts are performed. All final data is retained in the scene realisation data structure.

- The scene description is traversed again for each frame. Those parts of the description which have altered parameter values need to re-evaluate their parameters. All subsequent operations and casts must be performed again.

Problems arise in lazy traversal because casts and "if" statements ruin the correspondence between the scene description and scene realisation data structures.

With casts, the following problems arise:

- For models that are cast, the model may change between frames, making it necessary to perform the cast operation again. In order to discover if this must be done, the secondary realisation of the model that was originally made must be saved, so that it can be checked against the model as it is traversed again.

The secondary traversal is stored by the cast statement record. There may be more than one instance of the secondary realisation, so they are stored in a list.

- The data generated into the scene realisation for a model that is cast will not correspond to the records in the scene description. Different primitives are created, and there may be more of them. For instance, a polygon mesh that is cast into discrete polygons will be represented by a number of instances of the polygon primitive in the scene realisation. This situation can be accommodated by storing a pointer in the first primitive that references the last primitive. The entire set of primitives is now grouped, and may be treated as one entity.

"If" statements are the only statements that will cause the order of data in the scene realisation to change over time. When the value of the conditional expression is true, the statements in the body of the "if" statement will generate data, but when the condition is false, the data is not generated. This causes difficulty when the conditional expression contains time-varying values; at certain times in the animation, it will generate data, and not at others. This means that data must be excised from and added to the scene realisation. To remedy this situation, the data from the statements in the body of the "if" statement is stored in a secondary scene realisation data structure. When the condition evaluates as "true," the data is spliced into the primary scene realisation; when it is "false," the data is removed from the scene realisation.

3.5 Interfaces

The interfaces into the kernel allow programmers to implement extensions to the kernel. They define the basic functionality of the kernel. Details of these interfaces are given in appendix B. It lists the public members of the kernel's classes.

3.5.1 User Interface

The kernel provides facilities so that interactive interfaces may perform useful tasks. It provides controlled access to its data structures, or implements functions to accommodate these tasks. However, specific details of the user interface are not implemented.

- The scene description data structure may be edited. Functions are provided to create and delete model definitions, and elements in the bodies of model definitions. This is equivalent to editing model definitions in Charli scripts, except that it is done interactively, and immediate visual feedback may be provided.

- Access is given to edit parameter lists.

- Expressions that appear in parameter lists or in assignment and conditional statements may be edited.

- Numerical values that appear in parameter lists or expressions may be specified through a valuator, such as a slider, a knob, or through mouse motions. The kernel provides the mechanism to link the valuator to the numeric value. The

user interface must handle the valuator. Immediate visual feedback may be provided.

- Scene description data to be edited may be selected with a picking device, or by moving around in the definition hierarchy. The kernel maintains a "current item" which can be set and manipulated. Functions are provided to move to the parent of the current item, its siblings, or its children.

- Interactive picking chores are shared between the kernel, the user interface, and modelling techniques. The interface must determine the 3D *pick vector*. The kernel scans the hierarchy, and queries each primitive to see if it intersects the vector; it maintains a list of *pick paths*. The interface is responsible for disambiguating between picked items, and notifying the kernel about which pick path is chosen.

- Low-level motion control may be specified. This involves selecting a numeric value that appears in a parameter list or expression, and substituting an interpolation specification for it. The interpolation specification consists of two or more cues, and the names of interpolation functions that will be used to compute intermediate values between the cues.

- A request can be made to show the animated sequence. This will cause the kernel to render each frame of the animated sequence.

- The animator may specify that the scene be displayed as it would appear at a certain frame, or point in animation time. A kernel function is called to set the time.

- A Charli script may be read in, and added to the scene description and motion control data.

- The current scene may be stored as a Charli script. The kernel handles this process.

3.5.2 High-level Motion Control

A high-level motion controller is treated much like an interactive user interface. It is given the ability to access the model definitions, and numeric values that appear as parameters and in expressions.

Many of the high-level motion control algorithms require a model to be built especially for high-level control. Such a model can be specified in a Charli script. The high-level controller "attaches" to the parameters in the model that it wishes to read or whose values it will control.

High-level controllers are allowed to place event notices into the event list. An event notice specifies a point in animation time when the high-level controller wishes to do some processing. When this time arrives, the high-level controller is "awakened."

For example, a high-level motion controller might be calculating the motion of a whip that is held in a figure's hand. The motion of the arm is specified using low-level interpolation techniques. The motion controller reads the changes in the joint angles of the figure's arm at the wrist, elbow, and shoulder, to determine the forces that will be acting on the whip and computes the motion of the whip. To do this, it places event notices into the event list for each instant in time when it wishes

to read the joint angles, and for each animation frame time, when it will update the parameters to the whip model. It is allowed to read the joint angles at closer time intervals than the interval between frames.

3.5.3 Rendering

Renderers are given access to the scene realisation data structure. They may examine the data structure, and they are free to process it.

3.6 Chapter Summary

The major decisions that influence the kernel's design have been presented in this chapter. The decisions are motivated by the design criteria, and the desire to avoid the problems noted in the case studies.

The chief decisions are to closely integrate the data structures for model and motion control data through the track mechanism, and the use of a kernel approach.

Examples of the Charli script language are given. They illustrate the kernel's main features, and the power of integrated model and motion specification.

Discussions of the kernel's data structures and interfaces demonstrate the utility of the data structure integration mechanism in providing the following capabilities:

- The ability to animate any degree of freedom.
- An interface for high-level motion controllers and interactive manipulators.
- The ability to animate operations on primitives.
- The ability to read and write Charli scripts.

Chapter 4

Kernel Implementation

The details that are crucial to the implementation of the kernel are:

- The implementation of the track mechanism.
- The classes used in the scene description and scene realisation data structures, and the way in which subclasses may be derived from them to add new scene data types to the kernel.
- The traversal algorithm that generates the scene realisation from the scene description.

4.1 Object-oriented Programming

A prototype of the kernel has been implemented using the C++ programming language [Stroustrup 86]. The design makes extensive use of the object-oriented programming mechanisms of *class derivation* and *virtual functions*; [Stroustrup 87] states that these are the mechanisms that differentiate object-oriented languages from others. They are described in appendix C.

Although C++ was used for implementation, the kernel could be implemented in any other object-oriented programming language, such as Simula [Birtwistle 73], Smalltalk-80 [Goldberg 85], or Eiffel.

4.2 Tracks

The track mechanism provides the interface between the scene description data structure, and motion controllers. A track may be associated with a numeric value in a parameter list or an expression. This is possible because the data structures which store parameter lists and expression trees allow a track reference to be put in place of a number. This is done using the following class:

```
class trackdouble
{
    boolean     is_track;
    union       {
                double  d;
                track*  t;
                };
public:
    operator    double()
                {
                    if( !is_track ) return d;
                    else return t->get_value( current_time );
                }
};
```

This class is used in parameter lists and expression trees for storing numbers. The use of the C++ operator mechanism allows objects of class `trackdouble` to be treated as if they are ordinary C++ doubles.

The next important class is `track`. There are three types of tracks: interpolator tracks, high-level motion control tracks, and valuator tracks. These are classes derived from `track`. Class `track` itself is used as a base class for these others; its definition is simple:

```
class track
{
public:
    virtual double get_value();
};
```

The interpolator track class stores the data that specifies cues and interpolation functions. When it is queried via the `get_value` function, it computes the interpolated value for the requested time.

The high-level motion controller track class has a pointer to a memory location from which it will read values. The high-level motion controller is responsible for updating the value. When the track is queried for a value, it reads the memory location, and returns the value. The value is also stored in an array of values; there is one array location for each frame of animation. On subsequent display of a frame, the kernel may not invoke the high-level controller; the value for the track is read from the array.

The valuator tracks also read a value from a memory location. The user interface is responsible for updating the value of the memory location.

In the scene descriptions data structure, `trackdoubles` may appear in parameter lists, and expressions.

Parameter Lists

Parameter lists are stored in arrays. The parameter items may be of many different types: numbers, tracks, expressions, strings, vectors, and model references. The parameter array is made of elements of this data structure:

```
struct param
{
    int     data_type;
    union   {
            trackdouble   t;    // double or track
            char*         s;    // string
            expr*         e;    // expression tree
            vector*       vec;  // vector (three trackdoubles)
            reference*    r;    // reference to model def
            variable*     v;    // variable (stack frame & offset)
            };
};
```

The data_type field is accessed to discover the type of the parameter; the appropriate value may then be read from the union. If the parameter is a track, the trackdouble class will procure the parameter value from the correct source.

Expression Trees

Numeric and conditional expressions are converted into expression trees. A sample expression tree is shown in figure 3.4.

The value of the expression is found by performing an inorder traversal of the tree:

```
evaluate( tree_node )
begin
    if tree_node is a leaf
        return the value
    else
        value = apply_operator( op, evaluate( left ),
                                    evaluate( right ) )
end
```

The data elements at the leaf nodes can be numbers, variables, or tracks. This data structure is used to represent them:

```
struct expr_value
{
    int     data_type;
    union   {
            trackdouble   t;    // double or track
            variable*     v;    // variable (stack frame & offset)
            };
};
```

4.3 The Scene Description Data Structure

The scene description data structure is constructed from two basic classes: `node` and `item`. A node corresponds to a model definition, and items correspond to statement records. These classes are derived from `listhead` and `listitem`, respectively. The list classes are part of a package that is used for manipulating doubly linked lists; classes derived from `listitem` may be freely manipulated in lists headed by classes derived from `listhead`.

A `node` contains data about a model definition. This includes the name of the model, the recursion limit, and the prototype information. It also includes the list of variables and other models that are defined within its scope.

The `item` class is a base class from which other classes will be derived; these classes store the data for specific statement types. Figure 4.1 shows the class derivation tree. The intermediate class `param_item` is used by classes that represent statements with parameters; it stores the parameter list, as shown in figure 4.2. The `expr_item` class is similar, except that it stores a pointer to the expression tree for statements that have expressions.

The classes for `primitive`, `attribute`, and `transform` are quite simple. The

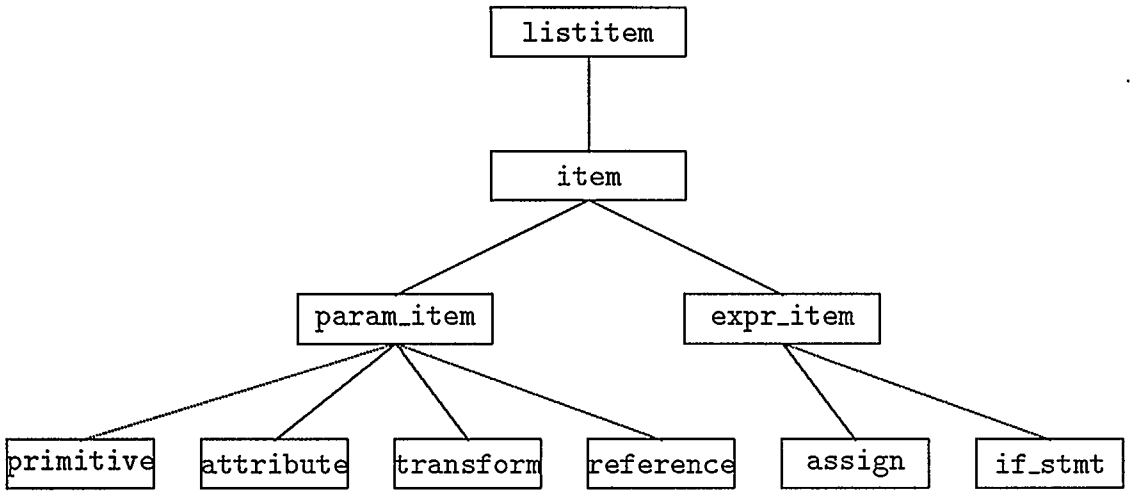


Figure 4.1: Hierarchy classes

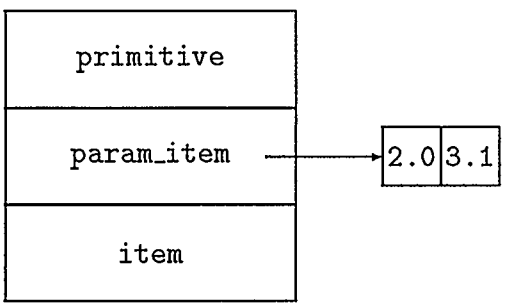


Figure 4.2: An item with parameters

kernel maintains tables which list the types of primitives, attributes, and transforms that are available. The tables contain the names of items, and pointers to the routines that are used to create instances of them. The classes for primitives, attributes, and transforms in the scene description data structure merely contain a pointer to the appropriate table entry. When the scene description is traversed, the objects of these classes evaluate their parameter lists, and the instance creation function is called with the evaluated parameters.

Class `reference` contains a pointer to the model definition that is referenced.

Class `assign` contains information about the variable to which it will assign a value. This information is the level of the stack frame in the execution stack where the storage for the variable will be found, and the offset into the frame.

Class `if_stmt` contains a pointer to a `node` which holds all of the statements in the body of the "if" statement. This is an "anonymous" node; it does not have a name or recursion limit. `if_stmt` also has a pointer to a scene realisation data structure. This is the realisation of the statements in the body of the "if" statement, when its condition is "true."

Finally, records are needed to store operation specifications, and cast specifications. These are put into special lists that are headed by objects of the classes `reference` and `primitive`. They are derived from `listitem`.

Items of class `operation` contain a pointer to a parameter list, and a pointer to an entry in an operation table that the kernel maintains. It also contains a pointer to its last evaluated parameter list; this is to allow the newly evaluated parameter list during a traversal to be compared to the previous values, to see if they have changed.

Items of class cast contain a pointer to the conversion function that will be called perform the conversion. They also contain a pointer to a list of secondary scene realisations. Each scene realisation is for an instance of the model that is being cast. The scene realisations are stored so that the model may be compared with its form from the previous traversal, to see if it has changed.

4.4 The Scene Realisation Data Structure

The scene realisation data structure is generated when the scene description data structure is traversed. The structure of the scene realisation is simple: it is a linked list of modelling primitives, attributes, and transforms, and instructions used to push and pop the graphics state.

The items that appear in the scene realisation are fully evaluated. Primitives have built their internal representations, and all operations have been performed on the them; attributes contain their internal representations; and transforms contain their matrices.

The traversal of the scene realisation is also simple: the list is scanned, and the appropriate action is taken for each item:

- For transforms, the matrix is concatenated onto the current transformation matrix.
- For attributes, the values in the attribute list are updated.
- For primitives, the internal representation is accessed, and dealt with by the renderer.

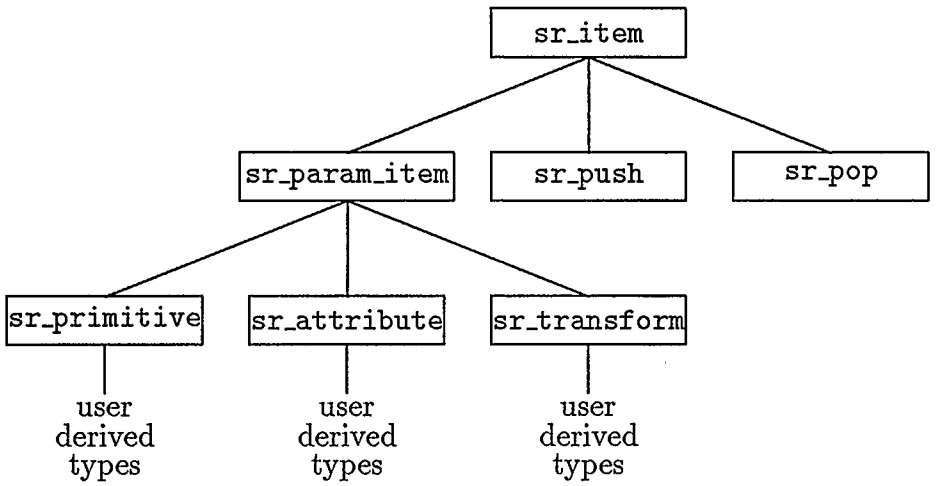


Figure 4.3: Scene realisation classes

- For push and pop instructions, the current graphics state is saved or restored.

When the set of scene data types is extended, the new types must be added to the scene realisation. This is made easier by the use of some classes: the classes `sr_primitive`, `sr_attribute`, and `sr_transform` are derived from `sr_item`, which is derived from `listitem`.

New primitive classes are derived from `sr_primitive`. They implement the function that generates the internal representation from the parameter list and the data structure needed to store it. The operation functions are implemented separately; they access the internal representation, and modify it.

Class `sr_transform` contains storage for a matrix. Specific transforms are derived from this class; they construct a matrix according to their parameters.

Specific attribute types are derived from the `sr_attribute` class. They contain the specific data for the attributes.

The renderer must understand the format of the data for the primitives and attributes that it is rendering.

4.5 The Traversal Algorithm

The basic hierarchy traversal algorithm is simple. However, it is complicated by the use of casting, the ability to pass models as parameters, and the desire to only recalculate the portions of the scene realisation that have changed.

4.5.1 Basic Algorithm

The basic traversal algorithm is this:

```
traverse_node( current_node, transform_matrix )
begin
    if recursion counter != 0
    begin
        decrement recursion counter
        concatenate the transformation matrix for this node
         onto current_matrix
        for current_item = each item in the node's list
        begin
            perform the appropriate action for the item
            (if reference item, call
              traverse_node( ref_node, current_matrix ))
        end
        increment recursion counter
    end
end

traverse_node( world_node, identity_matrix )
```

For each type of item in a node's list, a different action is performed:

Primitive: The parameter list is evaluated, and an instance of the primitive type is created. The list of operations and casts is performed on the primitive. The final data is added to the scene realisation. The special requirements of casting will be described later.

Attribute: The parameter list is evaluated, and an instance of the attribute is created and added to the scene realisation.

Transform: The parameter list is evaluated, and an instance of the transform is created and added to the scene realisation.

Reference: The parameters that are to be passed to the model are evaluated. Stack space on the execution stack is allocated for the parameters, and for the local variables of the model; an activation record is also added to the stack. The traversal function is called for the referenced node. If transforms follow the reference, they are evaluated first. The method of handling casts will be described later.

Assignment: The expression is evaluated, and the value is assigned to the variable.

If statement: The condition is evaluated. If it is true, then the node containing the statement records for the body of the "if" statement is traversed.

4.5.2 Handling Casting

If a primitive or model is cast into a new representation, this is handled as a special case of the traversal algorithm. The basic idea is that the scene realisation data from the primitives or model that is being converted should not be added into the

scene realisation. To accomplish this, the current scene realisation data structure is "pushed," and a new one is started. The current transformation matrix is also saved, and replaced by an identity matrix. The model is then traversed in the normal fashion; its data will be put into the new scene realisation data structure. When the traversal is finished, the new scene realisation is given to the conversion algorithm. The conversion may cause one or more new primitives to be generated; they are generated into a new "dummy" node, which is, in effect, a new submodel. The scene realisation data is "popped," as well as the old transformation matrix. The submodel is then traversed.

The same special traversal, up to the point where the conversion algorithm is called, is performed for model instances that are passed as parameters. The operation, primitive, attribute, or transform that receives the new scene realisation data structure must deal with it.

4.5.3 Updating the Scene Realisation

The traversal algorithm is called once to generate the initial scene realisation. After this, the traversal is done to discover which elements of the scene realisation have changed, and to update them.

The update traversal algorithm is essentially the same as the initial traversal algorithm. The only difference is that the parameters to modelling primitives, attributes, and transforms are checked to see if they have changed. If they have, then their data is regenerated; if they have not changed, then no work needs to be performed.

If a model that has been cast has changed, then the conversion must be recomputed. This is checked by traversing the model, and setting a flag if any changes

occur to it. When the traversal returns to the point where the cast is specified, the flag is checked. If it is true, the conversion is computed again.

If the parameters to an operation have changed, the model is regenerated from its original data, and all operations are performed on it again.

Some types of modelling primitive may change even if their parameters have not changed. An example of this is two soft objects [Wyvill 86] that are moving in proximity to each other; a property of soft objects is that their surfaces blend together, so moving them might cause their shapes to change; this happens even though their parameters have not changed. These types of primitives are considered as "volatile," and are re-evaluated on every traversal. It is the responsibility of the primitive to decide if it has changed.

Chapter 5

Conclusions

A kernel for 3D modelled animation systems has been presented in this thesis. It has been shown that many existing animation systems suffer from design flaws which hinder animators from producing the results that they desire. The chief design flaw is the lack of integration of the data structures and processes that are employed in the tasks of building models and specifying motions. Secondary design problems involve the medium or interface through which scenes and motions are specified, and the lack of appropriate tools, in the form of techniques for modelling, motion control, interaction, and rendering, that are available to animators.

The kernel design presented here solves some of these problems. The principal accomplishments of this work are:

- The techniques and data structures that are common to many applications of 3D modelled animation have been isolated and incorporated into a kernel.
- Extension interfaces have been developed to allow the kernel to be built into complete systems that are aimed at specific applications.
- The integration of the data structures for scene description and motion control has been accomplished through the use of an abstract data structure, the *track*.
- A language for scene and motion description, Charli, has been developed. It permits integrated specification of model and animation data. The script lan-

guage and interactive interfaces may be used interchangeably; this is a unique feature of the kernel.

- The notion of casting, or converting between different modelling techniques to represent an object, has been introduced. This is another unique and important feature of the kernel design. It cannot be found in any other system.

5.1 Addressing the Design Criteria

The purpose of this section is to demonstrate how the kernel's design satisfies the general design criteria listed in section 1.1.

5.1.1 Access to All Degrees of Freedom

The ability to access all degrees of freedom in a scene, and control their changes, is at the heart of the kernel's design. This ability is possible primarily because of the close integration of the data structures that store the scene description and motion control data: the parameters of any item of scene data may be animated.

However, this does not mean that *any* type of motion may be generated; nor does it mean that motions are easy to specify. The only types of motions that may be generated are those that are possible through parameter interpolation. Consider the metamorphosis of a model from one shape to another; there are three ways that this may be done:

- The parameters to the primitives in the model, and the use of transforms, may provide sufficient control over the shape of the model to perform the task. The parameter values are changed, causing the model's shape to change.

- Operations (eg. extrusion) may be applied to the primitive to change its shape.
- A mapping must be established between the data of the models such that intermediate shapes may be generated.

The first two methods may be accomplished using the kernel's native facilities. The third requires a high-level motion controller, or a special type of modelling technique, to be added to the kernel.

The problem encountered with the Symbolics system, where the extrusion of a face on a polyhedron cannot be animated, is solved by the low-level integration of the scene description and motion control data structures in the kernel. The parameters to the extrusion operation may be animated.

5.1.2 Extensibility

Four types of extensibility are provided.

Scene Data Types

The set of basic scene data types may be extended. This includes the addition of modelling techniques, attributes, and transforms. Modifications to the kernel entail the addition of information to tables, and relinking. These new resources are immediately available for use via Charli scripts and interactive interfaces. They may be used in model definitions, and animated by controlling their parameter values.

High-level Motion Control

High-level motion controllers may be interfaced to the kernel. They are permitted to interact with the scene description data structure. They may examine the structure

of models, and access the parameters that are passed to scene data items. Through the track mechanism, they may supply values that change over time to parameters. The changes in parameters may be "captured" so that they can be stored in a Charli script, and so that the calculations made by the high-level controller do not need to be performed unnecessarily. This is a very "thin" interface: it requires the high-level motion controllers to have specific knowledge about models and their structure, and it requires models to be constructed in a special manner so that the high-level controllers may find the parameters they wish to control. However, the high-level controllers are able to "react" to other motions in the scene that are generated through other means, and the "captured" values may may be "tweaked."

This type of interface is preferable to generating a new Charli script for every frame in an animated sequence. It solves the problem encountered in the Graphicsland system where high-level motion controllers generate complete PG scripts for every frame of animation.

Graphical User Interfaces

An interface is provided so that graphical user interfaces may be added to the kernel. The interface allows access to the scene description data structure and interpolation data structure, so that they may be modified. Valuator tracks are provided so that parameter values may be interactively controlled.

Renderers

Renderers are given access to the scene realisation data structure. Any type of renderer may be added to the kernel. A specialised renderer, such as a volumetric renderer for scientific data, may require special modelling primitives and attributes to

be implemented. This is accommodated by the first type of extensibility mentioned above.

Because of the highly extensible nature of the kernel, it is easy to imagine it at the core of very specialised systems like COMPOSE [Calvert 89] or SAS [Zeltzer 82] This would require a user interface to be built which is linked to a high-level controller. The kernel would used to store the data required to represent the models, and its rendering facilities are used to generate images.

The kernel is unsuitable for a specific application only if its basic hierarchical data structuring method is inappropriate.

5.1.3 High and Low-level Motion Control

The kernel incorporates low-level motion control, in the form of parameter interpolation, as part of its basic facilities. The set of interpolation functions may be extended.

As already discussed, an interface is provided for high-level controllers.

5.1.4 Script and Interactive Interfaces

The kernel provides a script language, and it provides an interface that allows interactive user interfaces to be added to it. Section 3 discusses how Charli scripts are converted into the kernel's scene description data structure, and how the data structure may be converted back into a script. This allows scripts to be read, modified via an interactive interface, and saved.

The potential is available to implement a variety of interactive interfaces. Each

may be tailor to a particular task or user type. Because of Charli, it will be possible to alternate between kernels that are configured with different interfaces.

Charli and each interface may be used where it is most appropriate. For instance, a complex model (ie. a human figure) may be designed in a Charli script, and an interactive interface can be used to determine that it appears correct, and to fix any problems (ie. limbs that do not meet). The interactive interface may also be used to pose the figure for animation.

5.1.5 Efficiency

There are many aspects of the kernel that have been designed with efficiency in mind. Efficiency, in the kernel, is primarily sought through avoiding unnecessary and redundant calculations.

There are two mechanisms that are used to avoid making redundant calculations:

- The scene realisation is updated between frames rather than recalculated. This allows expensive computations like generating instances of primitives from their specification and performing casts to be avoided.

- The streams of parameter values that are generated by high-level motion controllers are captured for each frame. High-level motion control techniques like dynamic simulation typically operate very slowly, so running them should be avoided when possible. Capturing the values allows the animation to be replayed, and allows other portions of the animation to be changed, without the expense of running the high-level motion system.

As well, the prototype mechanism may be used to "turn off" parts of the scene

description which are not of immediate interest to the animator, and which are causing a system to operate too slowly.

Whether these gestures towards efficiency are successful depends largely on the type of animation that is being done. For instance, if an "expensive" modelling primitive is used in a scene, and its parameters change for every frame, then it must be regenerated for every frame, and the first point above is invalidated. The second is invalidated if there is no interest in replaying the animation that has been generated by a high-level motion controller. This might be the case if the system incorporating the kernel is a flight simulator.

5.2 Additional Advantages to the Kernel Design

5.2.1 Recursive Animation

Because motion control data may be embedded into recursive model definitions, recursive animation may be generated. This is demonstrated by the Charli script in figure 5.1. The series of frames that accompany the script illustrate the recursive "H" model being animated. This complex motion is the result of animating two parameters in the scene description.

The recursive animation capability can be used to animate fractal images, and also to create animated sequences that feature complex, organised motions.

This capability has not been claimed for any other system, although it is likely that ASAS could perform the same function.

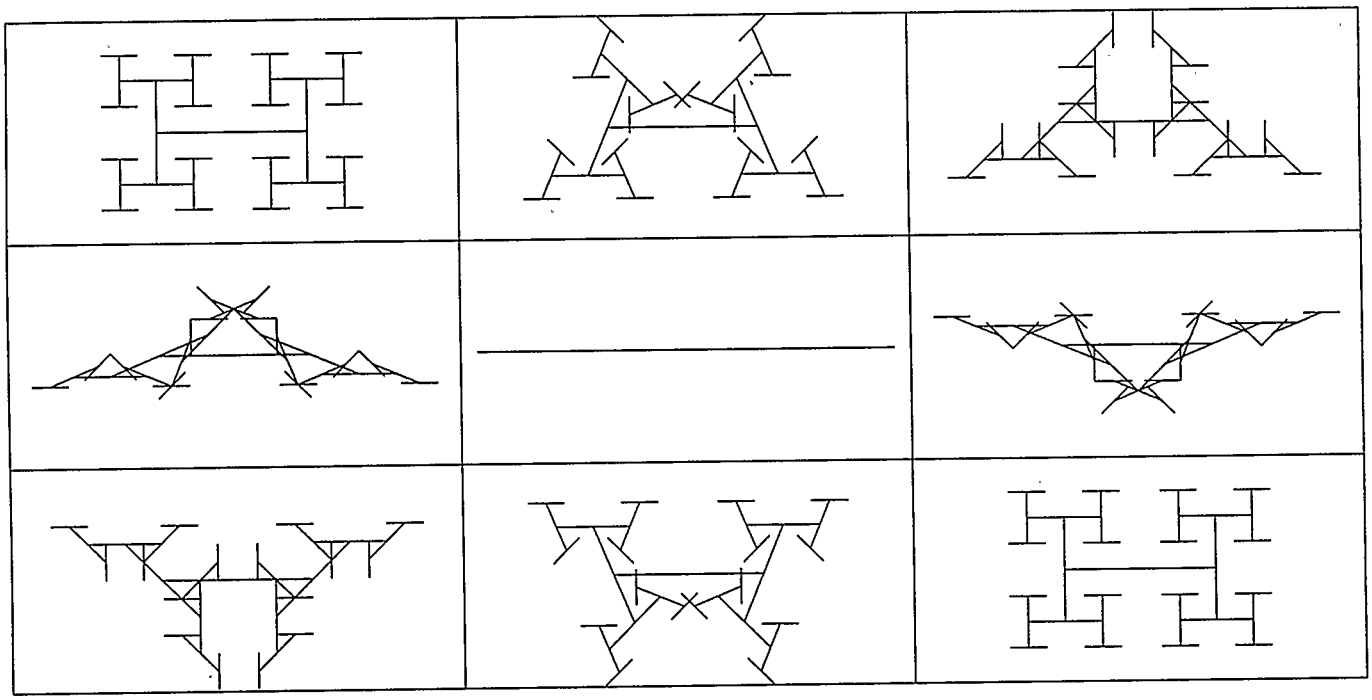


Figure 5.1: Recursive animation

5.2.2 High-level Motion Control Using Variables

The ability to use variables and expressions in a model description makes it possible to implement a form of high-level motion control.

```
def thing( same_rot, opposite_rot )
    line( <0,0,0>, <1,0,0> ) rotatex( same_rot + opposite_rot );
    line( <0,0,0>, <-1,0,0> ) rotatex( same_rot - opposite_rot );
end;
```

Two parameters are available to control the angle between the two line segments. One causes them to rotate around their joined ends in the same direction and to the same degree, and the other causes them to rotate to the same degree at opposite angles.

This simple example illustrates the potential of this technique. It is conceivable that a character model for character animation could be defined in this manner, and motions like squash and stretch could be controlled through a set of parameters. The model, and the set of parameters, would likely be quite complex.

A large, complex model that uses this capability has not been created, due to time constraints.

5.3 Discussion

The kernel design satisfies the design criteria, solves many of the problems encountered in other animation systems, and provides the new capabilities of modelling primitive conversion and recursive animation.

It forms a basis on which powerful systems for many applications of 3D modelled animation may be constructed.

5.4 Future Work

The kernel design provides the potential to develop a very powerful animation system. All that is required is the implementation of new scene data types, motion controllers, user interfaces, and renderers. Of course, the development of a large system is a daunting task.

On a more specific basis, some additions and improvements can be made to the kernel design:

- Dependency tables may be used to determine which parameter values have changed [Hanrahan 85] [Ostby 89]. This would increase the speed of the system, as all parameter values are checked for each animation frame during the update traversals.

- The mechanism used by [Ostby 89] to allow models to have different attachment points in the hierarchical structure should be added. This would require a new data type for matrices to be added, as well as the functions to read and apply the current transformation matrix at points in the hierarchy.

Another graduate student at the University of Calgary, Charles Herr, is currently developing a system for dynamic simulation [Herr 90]. The system is being designed to integrate into the kernel, and the Charli language, at a low level.

Appendix A

Charli Grammar

The formal grammar for the Charli language is presented in this appendix. The grammer has been used in conjunction with the Yacc parser generator [Johnson 75] to implement the Charli parser that is used in the kernel.

Non-terminal symbols are italicised. Terminal symbols are given in Roman typeface—these are either key words in the Charli language or special symbols (ie. commas, parentheses and other single-character tokens). These terminal symbols have special meaning:

- "name" tokens mean that an undefined name has appeared in the script. It is not in the symbol table in the current scope.
- "object" tokens mean that the name of a defined model has appeared in the script.
- "variable" tokens mean that the name of a declared variable has appeared.
- "primitive", "transform", "attribute", and "operation" tokens mean that the name of one of these types of data (ie. "polygon" as a primitive) has appeared. These names are entered into the symbol table depending on the configuration of the kernel.
- "interpolation" means that the name of an interpolation function has appeared.
- "string" represents a string, enclosed in quotation marks ("), has appeared.

- "number" represents a number that has appeared in the script.

The lexical analyser is responsible for returning the correct non-terminal symbol in these cases.

The grammar shows that interpolation specifications *interpspec* may appear in any expression (*expr*). However, in the case where an expression occurs in an interpolation specification, this will result in undefined behavior by the system.

Yacc allows precedence rules to be defined for expressions. The following is the precedence definition. The order is lowest to highest precedence, with symbols that appear on the same line being equal, and having either left or right associativity.

```
%right    '='
%left     '|'
%left     '&'
%left     EQ NEQ
%left     '<' '>' LEQ GEQ
%left     '+' '-'
%left     '*' '/' '%'
%left     UNARYMINUS '!'
%right    '^'
```

Conditional expressions are handled as special arithmetic expressions. "True" expressions return the numeric value of 1, and "false" returns 0. Any non-zero value from an arithmetic expression is considered "true."

---

script ⇒ *decllist deflist stmtlist*

deflist ⇒ ϵ | *definitions*

definitions ⇒ *objectdef* ; | *definitions objectdef* ;

objectdef ⇒ def name *paramdef script limitstmt* end

paramdef ⇒ ϵ | (*namelist*)

namelist ⇒ name | *namelist* , name

decllist ⇒ ϵ | *declarations*

declarations ⇒ *declaration* ; | *declarations declaration* ;

declaration ⇒ var name

limitstmt ⇒ ϵ | limit number ;

stmtlist ⇒ ϵ | *stmts*

stmts ⇒ *stmt* ; | *stmts stmt* ;

stmt ⇒ *refstmt* | *primstmt* | *transtmt* | *attrstmt* | *assign* | *ifstmt* | prototype *primstmt*

refstmt ⇒ object *paramlist taclist*

primstmt ⇒ primitive *paramlist taoclist*

taclist ⇒ ϵ | *tacs*

taoclist ⇒ ϵ | *taocs*

tacs ⇒ *ta* | *tacs ta* | *tacs caststmt taoclist*

ta	⇒	*transtmt* | *attrstmt*
taocs	⇒	*taoc* | *taocs taoc*
taoc	⇒	*transtmt* | *attrstmt* | *operstmt* | *caststmt*
transtmt	⇒	transform *paramlist*
attrstmt	⇒	attribute *paramlist*
operstmt	⇒	operation *paramlist*
caststmt	⇒	cast to primitive
assign	⇒	variable = *expr*
ifstmt	⇒	if *expr stmtlist* end
paramlist	⇒	ϵ | (*params*)
params	⇒	*param* | *params* , *param*
param	⇒	*expr* | string | object *paramlist* | *triplet*
triplet	⇒	< *expr* , *expr* , *expr* >
interpspec	⇒	{ *interp* }
interp	⇒	*expr* at *timespec interpfunc expr* at *timespec* | *interp interpfunc expr* at *timespec*
timespec	⇒	*expr* sec | frame *expr*
interpfunc	⇒	interpolation *paramlist*
expr	⇒	number | variable | *interpspec* | (*expr*) | ! *expr* | - *expr* | *expr* [+ | - | * | / | % | ^ | & | < | > | eq | neq | leq | geq] *expr*

Appendix B

Kernel Interfaces

The kernel provides interfaces for adding user interfaces, high-level motion controllers, renderers, and new scene data types. The interfaces are minimal and allow access to the public members of the classes that comprise the kernel's primary data structures. The approach is to allow user code to access and manipulate data values directly, except where it crucial to maintain the integrity of a data structure. Such a case is the hierarchy data structure, where functions are provided to add and delete items from it.

The following examples describe the public members of various classes. The true implementations of the classes contain other private members that are used internally.

Base classes are not described. The public members of the base classes are listed in the set of public members for their subclasses.

B.1 Class Kernel

Class kernel is the primary interface into the kernel. An instance of class kernel is an instance of the animation kernel. The class is the gateway to deeper access into the kernel data structures.

The following are details of the public elements of class kernel. They may be divided into a number of categories.

B.1.1 Hierarchy Access and Manipulation

Current Node

The kernel maintains a current node in the hierarchy. The current node pointer may be manipulated.

```
void     init_current_node();       // set to "world"
node*    get_current_node();        // pointer to current
node*    next_node_in_scope();      // set current to next node
                                    //   in this textual scope
node*    parent_node_in_scope();    // set to parent in scope
node*    first_child_in_scope();    // set to first node defined
                                    //   in current's scope
node*    new_node( char* name );    // make a new node and place
                                    //   it in list of nodes
                                    //   defined in current
                                    //   scope; current = new node
```

Current Item

A current item is maintained within the current node. If the current node is changed, the current item is set to null.

```
item*   get_current_item();          // NULL if none current
item*   next_item();                 // set to next item in node
item*   previous_item();
item*   null_item();                 // set to NULL
item*   first_in_ref( reference* ); // set to first item in a
                                     //   reference item's list of
                                     //   transforms
item*   first_in_co( reference* );  // set to first item in a
                                     //   reference item's list of
                                     //   casts and opers
item*   first_in_co( primitive* );  // ditto, for primitive item
item*   first_in_if( primitive* );  // ditto, for items in body
                                     //   of if statement
```

The next group of functions are used to create data elements that are placed into nodes. The are inserted after the current item, and current item is set to the new item. If there is no current item, they are inserted into the head of the node's list.

```
item*    new_primitive( char* name );
item*    new_attribute( char* name );
item*    new_transform( char* name );
item*    new_operation( char* primname, char* opername );
item*    new_assign();
item*    new_reference( node* ref_to );
item*    if_stmt();
void     remove_item( item* );
```

Picking

A direct method of choosing a current node and item is via interactive picking. This is a complex operation shared between the kernel and the user interface.

The kernel maintains a pick buffer, which may contain multiple pick paths. A pick path is a string of numbers. Each node and primitive has a unique number for its pick name. The interface must determine a pick vector, which is given to the kernel. The kernel polls the primitives in the scene realisation to discover which are picked, and fills the pick buffer. The user interface is responsible for disambiguating picked items, and informing the kernel of which is correct.

```
int      number_picked();        // how many pick paths
void     next_pick_path();       // use next path; set current
                                 //   item and node to it
void     parent_in_pick();       // shorten path by one name and
                                 //   set item and node
void     perform_pick( line );   // pick items that intersect line
```

B.1.2 Time

The animation time can be manipulated through these functions:

```
double   current_time();           // returns current time
void     goto_time( double time ); // set the current time
int      current_frame();
void     goto_frame( int frame );
int      fps();                    // returns frames per second
void     set_fps( int rate );      // sets frame rate

void     schedule( event_notice* ev );  // place an event in the
                                        //   event list
```

As an aside, here is the event notice class:

```
class event_notice
{
public:
             event_notice(double time, int prio, void (*func)());
    double   ev_time;
    int      ev_prio;
};
```

Rendering

The rendering functions cause frames to be rendered. The kernel performs any necessary updating of the scene realisation, and calls the renderer.

```
void     render();  // render current frame
void     animate( double stime, double etime );
                    // render frames from start to end time
```

Tables

The kernel maintains tables of its current configuration. These list the primitives, attributes, transforms, operations, and interpolation functions that are available.

```
desc     the_primitives[];  // each has operation table
desc     the_attributes[];
desc     the_transforms[];
desc     the_interps[];
desc     the_systems[];     // other item types
```

The public portion of the descriptor class is this:

```
class desc
{
    char*   name;
    table*  subtable;
};
```

Miscellaneous

The kernel maintains a number of other odds and ends.

```
matrix  global_matrix;  // current matrix value during
                        //   traversal

    // read and write Charli scripts
void    read_script( char* filename );
void    write_script( char* filename );

int     use_prototypes; // global prototype switch

scene_realisation sr;   // the scene realisation
```

B.2 Hierarchy Classes

Through the kernel, access can be gained to the hierarchy data structure. This allows hierarchy elements to be manipulated by user interfaces and high-level motion controllers.

B.2.1 Class Node

Once a pointer to a node is acquired, the following functions and data are accessible.

Data Elements

These data items can be read and modified:

```
char*    name;            // name of model
int      pick_name;       // unique integer
int      recursion_limit;
```

Definitions

New nodes can be defined in the scope of another node. The list of nodes can be scanned with these functions:

```
node*    first_def();
node*    next_def( node* );
void     del_def( node* );   // delete
```

These do not change the kernel's current node pointer. New nodes are created through kernel functions.

Variables

The list of variables that are declared within a node can be manipulated. Parameter specifications are treated like other variable declarations.

```
variable*  first_var();
variable*  next_var( variable* );
variable*  new_var( char* name ); // makes new and inserts into
                                  //   current node's list
void       del_var( variable* );  // delete
```

Miscellaneous

Other functions are:

```
void bounding_sphere(point*, double*); // returns a bounding
                                       //   sphere for the
                                       //   model
```

B.2.2 Item Classes

Class item is a base class from which others are derived. It is not used directly, but it defines some data and virtual functions that are used by its subclasses. Further base classes, param_item and expr_item are derived from item. The subclasses of these are actually used.

Param Items

These classes are derived from class param_item. The first is used to represent statements which set an attribute.

```
class attribute
{
    param     parameters[];
    desc*     descriptor;   // entry in descriptor table
    int       get_type();   // returns type of this item
};
```

The rest are essentially similar, but may add some additional fields. The following definitions indicate the addition information:

```
class reference
{
    node*   ref_node;
};

class primitive
{
    void    bounding_sphere( point*, double* );
};

class transform
{
    matrix  xform;
};
```

```
class operation {};  // no change
```

Expr Items

These items contain an expression:

```
class assign
{
    expr      root;        // root node of expression tree
    desc*     descriptor;
    int       get_type();
    double    evaluate();  // evaluate the expression tree
    var_info  var;
};

class if_stmt
{
    expr      root;
    desc*     descriptor;
    int       get_type();
    double    evaluate();
    node*     stmt_list;
};
```

B.3 Scene Realisation

The scene realisation data structure is accessed by the renderer. Scene realisations are also used by the casting algorithms. The extension of types for primitives, attributes, and transforms is done by deriving new types from base classes defined here.

The scene realisation class heads the list of items in the scene realisation. Picking is done by scanning through the scene realisation to see what is hit. Functions are provided to do this.

```
class scene_realisation
{
    sr_head* sr;              // heads the scene realisation list
    void     traverse();      // traverse for rendering
    void     pick_traverse(); // traverse for picking; fills
                              //   the pick buffer in the kernel
};

class sr_head
{
    sr_item* first();         // pointer to first item
};
```

The class sr_item is the base class for all scene realisation data elements. From it is further derived class sr_param_item which is a base class for items with parameters.

These are derived directly from sr_item:

```
class sr_push
{   // a record that indicates stacks should be pushed
    int    get_type();
};

class sr_push
{   // a record that indicates stacks should be popped
    int    get_type();
};
```

These are derived from sr_param_item:

```
class sr_primitive
{
    sr_params    parameters[];
    int          get_type();
    virtual void draw();       // perform rendering action
    virtual void touch();      // touch is called if params
                               //   have been changed
    virtual void is_picked();  // test if picked
};
```

```
class sr_attribute
{
    sr_params    parameters[];
    int          get_type();
    virtual void draw();
    virtual void touch();
};

class sr_transform
{
    sr_params    parameters[];
    int          get_type();
    virtual void draw();
    virtual void touch();
    matrix       xform;
};
```

The last three classes are used as base classes by implementors of new scene data types. They must implement suitable functions for the virtuals. These are dependent on the internal representation of the data type, and the renderer that is used.

B.4 Parameters and Expressions

Parameters and expressions play a key role in the integration of model and motion control data. There are two types of parameters: those used by items in the hierarchy, and those used in the scene realisation.

B.4.1 Hierarchy Parameters

Parameter lists in param_items can contain many data types. Parameter lists are an array of this structure:

```
struct param
{
    int         data_type;
    union       {
                trackdouble     t;
                char*           s;
                expr*           e;
                vector*         vec;
                reference*      r;
                struct          { int lev, off; } v;
                };
};
```

Class trackdouble is explained in section 4.2.

B.4.2 Scene Realisation Parameters

The parameters to sr_param_items are much simpler, because they contain only evaluated items:

```
struct sr_param
{
    int       data_type;
    union     {
              double          d;
              char*           s;
              vector*         vec;
              };
};
```

B.4.3 Expressions

Expressions are stored as expression trees. They are constructed from this class:

```
class expr
{
    int     is_leaf;
    union   {
            struct  {
                    int is_literal;
                    union   {
                            trackdouble t;
                            var_info v;
                    };
            } leaf;
            struct  {
                    int   operator;
                    expr* left;
                    expr* right;
            } node;

    };
    double  evaluate();  // evaluate this expr
};
```

B.5 Tracks

Classes derived from track are used to provide values that change over time.

```
class interp_track
{
    double get_value( double time );
    int    get_type();
    cue*   first_cue();      // get first cue point
    cue*   next_cue( cue* );
    cue*   new_cue();
    cue*   del_cue();
};
```

```
class cue
{
    expr*  time;
    expr*  value;
    char*  interp_name;
    double (*interp_func)();
};
```

Appendix C

Virtual Functions

The virtual function mechanism is used extensively in the implementation of the kernel. It, combined with the complementary mechanism of class derivation, are central to the extensible nature of the kernel. Because of the importance of these mechanisms in the kernel's design, a description of their operation in the C++ programming language is presented in this appendix.

Classes

A *class* defines an abstract data type. It describes the data elements and member functions that make up the data type. Here is a sample class that contains an integer and one member function:

```
class X
{
public:
    int     i;
    int     f();
};
```

Instances of classes, called *objects*, may be created. The values of an object's data elements may be accessed, and its member functions may be called. For class X, the following statements are valid:

```
X  x1;          // declare an instance of X
X* x2;          // a pointer to an instance of X

x2 = new X;     // make an instance of X
```

```
int i = x1.i;  // access the data element
i = x2->i

i = x1.f();    // call the member function
i = x2->f();
```

Some portions of a class may be defined as *private* and cannot be accessed except by member functions. In class X, "i" could be defined as private and the access statement examples would be illegal.

Class Derivation

The class derivation mechanism allows new classes to be created which are extensions of existing ones; they inherit all of the data elements and member functions from the parent class. Derived classes are called *subclasses*. The original class is a *superclass*. Superclasses are also called *base* classes, because others are constructed on them.

Class Y can be derived from class X:

```
class Y : public X
{
public:
    double d;
    int    g();
};
```

Class Y has an additional data element and member function. Here are some sample statements:

```
X x;
Y y;

y.d = 2.0;
y.i = 1;
```

```
x.i = 2;
x.d = 2.0;   // this is illegal: x is an X, not a Y

y.g();
y.f();
```

All public portions of X and Y are accessible in an instance of Y, but an instance of X can be treated only as an X.

An instance of Y can be treated as an X. However, an X cannot be treated as a Y.

```
X* x;
Y  y;

x = &y;        // pointer to X actually points to instance
               //   of Y
x->i = 1;
x->f();

x->d = 3.0;    // illegal
y.d = 3.0;
x->g();        // illegal
y.g();
```

Virtual Functions

Virtual functions permit a base class to "know" about classes derived from it. In the example above, the members of "y" cannot be accessed through the pointer "x", even though they exist. This is because class X does not "know" that classes are derived from it.

With virtual functions, this knowledge can be defined in a base class. Here is an example:

```
class A
{
public:
    A*          next;
    virtual char f()  { return 'a'; }
};

class B : public A
{
public:
    char f()  { return 'b'; }
};


A  a;
B  b;
A* ap;

ap = &b;       // pointer to A actually points to B

a.f();         // returns 'a'
b.f();         // returns 'b'
ap->f();       // returns 'b' !!!!
```

The final statement in the example magically returns the character "b" rather than "a", even though the pointer is declared to reference an instance of A. This is due to the "virtual" declaration; it specifies that classes may be derived from A, and may redefine the function "f".' When an instance of B is created, its underlying A component is informed that the function has been redefined, and given the address of the new one. Even if the instance is accessed as an A, the correct function is executed. This allows derived types to be manipulated generically.

A new class, C, can be derived from A and redefine "f":

```
class C : public A
{
public:
    char f()  { return 'c'; }
};
```

Consider that a list of instances of A, B, and C has been created, using the "next" pointer to link them together. The list must be treated as a list of A's. The following code will iterate through the list:

```
A* ap;        // assume this points to the first
              //   item in the list

while( ap != NULL )
{
    ap->f();
    ap = ap->next;
}
```

The function "f" is called for each item in the list, and the correct value of "a", "b", or "c" will be returned for each item.

This is a trivial example, but it serves to illustrate the virtual function mechanism.

Extensibility

The virtual mechanism promotes extensibility. In the example above, new classes (eg. D) can be derived from A and the list iteration code does not have to be changed.

A less trivial example is a simple 2D graphics system. It allows instances various shapes to be created and displayed. The shapes are stored in a list. Each type of shape stores a position on the screen, but otherwise their internal data is different: a square stores two corner points; a circle stores a centre and a radius.

A base class, "shape", can contain the position data and a virtual function called "draw". Each subclass will then define its unique data elements, and implement an appropriate drawing function. Drawing is accomplished by iterating through the shape list, and calling the virtual draw function for each shape.

Again, new shapes may be added without modifying the iteration code.

Use in the Kernel

Virtual functions are class derivation are used in the kernel to permit the set of modelling techniques, attributes, and transforms to be extended. Figure 4.3 shows the class derivation tree for the base classes. There are three levels of derivation.

While `sr_item` is the primary base class, some more specialised classes are derived from it. Class `sr_item` defines these virtual functions:

`draw`: The action to be performed for drawing the item. For primitives, this is the function to draw the item, or pass its data on to the renderer. For attributes, the function will change the value in the attribute list. Transforms will apply their matrix to the global one.

`touch`: This informs an instance of a scene data type that its parameters have been changed, and it should re-evaluate them and construct a new internal representation.

`get_type`: This returns the specific type of the class derived from `sr_item`.

Specialised base classes are `sr_primitive`, `sr_attribute`, and `sr_transform`. They implement `get_type` to return the specialised type. They do not implement the other two virtual functions; this is the responsibility of user derived subclasses.

Class `sr_primitive` defines an additional virtual function, `is_picked`, which is called to determine if a primitive has been picked. Class `sr_transform` declares a matrix.

The kernel "understands" `sr_item` and the specialised base classes, and contains the routines necessary for processing them. The main processing task the kernel performs is to iterate through the scene realisation which contains `sr_items` and call the `draw` function of each one. System implementors derive specific scene data types from the specialised base classes, and implement the virtual functions. The kernel's processing routines do not have to be modified to handle any of the new types.

Bibliography

[ALIAS 87] Alias Research Inc. *ALIAS/1 User Guide*, 1987.

[Allan 88] Jeffrey B. Allan. Polygon Mesh Modelling for Computer Graphics. Master's thesis, University of Calgary, Dept. of Computer Science, September 1988.

[Armstrong 85] William W. Armstrong and Mark W. Green. The Dynamics of Articulated Rigid Bodies for Purposes of Animation. *The Visual Computer*, 1985.

[Baecker 69] R. M. Baecker. Picture-driven Animation. In *Proc. Spring Joint Computer Conference*, volume 34, pages 273–288. AFIPS, 1969.

[Birtwistle 73] G. M. Birtwistle, Ole-Johan Dahl, Bjorn Myhrhaug, and Kristen Nygaard. *Simula Begin.* Petrocelli/Charter, New York, 1973.

[Birtwistle 79] G.M. Birtwistle. *Discrete event modelling on Simula.* Macmillan, London, England, 1979.

[Burtnyk 71a] N. Burtnyk and M. Wein. A Computer Animation System for the Animator. *Proc. UAIDE 10th Annual Meeting*, 1971.

[Burtnyk 71b] N. Burtnyk and M. Wein. Computer-generated Key-frame Animation. *Journal of Society for Motion Picture and Television Engineers*, 80:149–153, 1971.

[Burtnyk 76] N. Burtnyk and M. Wein. Interactive Skeleton Techniques for Enhancing Motion Dynamics in Keyframe Animation. *Comm. ACM*, 19(10):564–569, 1976.

[Calvert 89] T. Calvert, C. Welman, S. Gaudet, and C. Lee. Composition of Multiple Figure Sequences for Dance and Animation. In *New Advances in Computer Graphics (Proc. CGI '89)*, pages 245–254, 1989.

[Catmull 72] E. Catmull. A System for Computer-generated Movies. *Proc. ACM Annual Conference*, pages 422–431, 1972.

[Catmull 79] E. Catmull. New Frontiers in Computer Animation. *American Cinematographer*, October 1979.

[Chmilar 89] M. Chmilar and B. Wyvill. A Software Architecture for Integrated Modelling and Animation. In *New Advances in Computer Graphics (Proc. CGI '89)*, pages 245–254, 1989.

[Comer 84] Douglas Comer. *Operating System Design: The XINU Approach.* Prentice-Hall, 1984.

[Csuri 75] C. Csuri. Computer Animation. In *Computer Animation*, volume 9, pages 92–101. ACM SIGGRAPH, 1975.

[Csuri 79] C. Csuri, R. Hackathorn, R. Parent, W. Carlson, and M. Howard. Towards an Interactive High Visual Complexity Animation System. In *Computer Animation*, volume 13, pages 289–299. ACM SIGGRAPH, 1979.

[DeFanti 76] T. DeFanti. The Digital Component of the Circle Graphics Habitat. In *Proc. National Computer Conference '76*, pages 195–203, 1976.

[Entis 86] Glenn Entis. Computer animation—3D motion specification and control. *SIGGRAPH '86, Course #23, Computer Animation: 3-D Motion Specification and Control*, 1986.

[Foldes 74] Peter Foldes. Hunger, 1974. Computer animated film. 12 min.

[Foley 82] James D. Foley and Andries van Dam. *Fundamentals of Interactive Computer Graphics.* Addison-Wesley, 1982.

[Goldberg 85] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1985.

[Gomez 85] Julian E. Gomez. Twixt: A 3D Animation System. *Computers and Graphics*, 9(3):291–298, 1985.

[Gomez 86] Julian E. Gomez. Comments on event driven computer animation. *SIGGRAPH '86, Course #23, Computer Animation: 3-D Motion Specification and Control*, 1986.

[Hackathorn 77] R. Hackathorn. ANIMA II: A 3D Color Animation System. In *Computer Graphics*, volume 11, pages 54–64. ACM SIGGRAPH, 1977.

[Hall 89] Roy Hall. *Illumination and Color in Computer Generated Imagery.* Springer-Verlag, 1989.

[Hanrahan 85] Pat Hanrahan and David Sturman. Interactive animation of parametric models. *The Visual Computer*, 1985.

[Heraldson 75] Donald Heraldson. *Creators of Life: A History of Animation.* Drake Publishers Inc., 1975.

[Herr 90] Charles Herr and Brian Wyvill. Towards Generalised Motion Dynamics for Animation. In *Proceedings Graphics Interface '88.* CIPS, 1990.

[Hong 88] T. M. Hong, N. Magnenat-Thalmann, and D. Thalmann. A General Algorithm for 3-D Shape Interpolation in a Facet-Based Representation. In *Proceedings Graphics Interface '88*, pages 229–235. CIPS, 1988.

[Jensen 74] K. Jensen and Niklaus Wirth. *Pascal User Manual and Report.* Springer-Verlag, 1974.

[Johnson 63] Timothy E. Johnson. Sketchpad III: A Computer Program for Drawing in Three Dimensions. In *Proceedings SJCC*, volume 23, pages 347–353. AFIPS, 1963.

[Johnson 75] S. C. Johnson. Yacc—Yet Another Compiler-Compiler. Computer Science Technical Report 32, Bell Laboratories, Murray Hill, NJ, 1975.

[Kernighan 81] Brian W. Kernighan and P. J. Plauger. *Software Tools in Pascal.* Addison-Wesley, 1981.

[Kitching 73] A. Kitching. Computer Animation—Some New ANTICS. *British Kinematography Sound Television Journal*, 55(12):372–386, 1973.

[Knowlton 64] Ken C. Knowlton. A Computer Technique for Producing Animated Movies. *Proc. SJCC AFIPS Conference*, 25:67–87, 1964.

[Knowlton 65] K. Knowlton. Computer-produced Movies. *Science*, 150:1116–1120, 1965.

[Knowlton 70] Ken C. Knowlton. EXPLOR—A Generator of Images. *Proc. 9th UAIDE Annual Meeting*, pages 543–583, 1970.

[Kochanek 84] D. Kochanek. Interpolating Splines with Local Tension, Continuity and Bias Control. In *Computer Graphics*, volume 18, pages 33–41. ACM SIGGRAPH, 1984.

[Kroyer 86] Bill Kroyer. Animating with a hierarchy. *SIGGRAPH '86, Course #22, Advanced Computer Animation*, 1986.

[Lasseter 87] John Lasseter. Principles of Traditional Animation Applied to 3D Computer Animation. In *Computer Graphics*, volume 21, pages 35–44. ACM SIGGRAPH, July 1987.

[Leith 89] A. Leith, M. Marko, and d. Parsons. Computer Graphics for Cellular Reconstruction. *IEEE CG&A*, 9(5):16–23, September 1989.

[Mandelbrot 82] Benoit B. Mandelbrot. *The Fractal Geometry of Nature.* W. H. Freeman, 1982.

[Mantyla 88] Martti Mantyla. *An Introduction to Solid Modeling.* Computer Science Press, Rockville, Maryland 20850, 1988.

[Ostby 89] Eben F. Ostby. Simplified Control of Complex Animation. In *State-of-the-art in Computer Animation (Proc. Computer Animation '89).* Springer-Verlag, 1989.

[Peary 80] Gerald Peary and Danny Peary, editors. *The American Animated Cartoon.* E. P. Dutton, 1980.

[Pixar 88] Pixar. *The RenderMan Interface, Version 3.0*, May 1988.

[Potmesil 87] Michael Potmesil and Eric M. Hoffert. FRAMES: Software Tools for Modeling, Rendering and Animation of 3D Scenes. In *Computer Graphics*, volume 21, pages 85–93. ACM SIGGRAPH, July 1987.

[Prusinkiew 88] P. Prusinkiewicz, A. Lindenmayer, and J. Hanan. Developmental Models of Herbaceous Plants for Computer Imagery Purposes. In *Computer Graphics*, volume 22, pages 141–150. ACM SIGGRAPH, August 1988.

[Reeves 83] William Reeves. A Technique for Modelling a Class of Fuzzy Objects. *ACM Transactions on Graphics*, 2(2):91–108, April 1983.

[Reynolds 82] Craig W. Reynolds. Computer animation with scripts and actors. In *Computer Graphics*, volume 16. ACM SIGGRAPH, July 1982.

[Reynolds 87] Craig W. Reynolds. Flocks, herds, and schools: A distributed behavioral model. In *Computer Graphics*, volume 21, pages 25–34. ACM SIGGRAPH, July 1987.

[Smith 78] Alvy Ray Smith. Paint, 1978.

[Stern 79] G. Stern. SoftCel: An Application of Raster Scan Graphics to Conventional Cel Animation. In *Computer Graphics*, volume 13, pages 284–288. ACM SIGGRAPH, 1979.

[Stern 83] G. Stern. BBOP: A System for 3D Key Frame Figure Animation. *SIGGRAPH '83 Course Notes #7: Introduction to Computer Animation*, pages 240–243, 1983.

[Stroustrup 86] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986.

[Stroustrup 87] Bjarne Stroustrup. What is Object-Oriented Programming? In *C++ Workshop Proceedings*, pages 417–439. USENIX Association, 1987.

[Sturman 86] David Sturman. A Discussion on the Development of Motion Control Systems. *SIGGRAPH '86, Course #23, Computer Animation: 3-D Motion Specification and Control*, 1986.

[Sutherland 63] Ivan E. Sutherland. Sketchpad: A Man-Machine Graphical Communication System. In *Proceedings SJCC*, volume 23, pages 329–346. AFIPS, 1963.

[Symbolics] Symbolics Corp. *User Manuals for S-Geometry, S-Dynamics, and S-Render.*

[Talbot 71] P. Talbot, J. Carr, R. Coulter, and R. Hwang. Animator: An Online Two-dimensional Film Animation System. *CACM*, 14(4):251–259, 1971.

[Thalmann 85] Nadia Magnenat-Thalmann and Daniel Thalmann. *Computer Animation: Theory and Practice.* Springer-Verlag, 1985.

[Thomas 81] Frank Thomas and Ollie Johnston. *Disney Animation: The Illusion of Life.* Abbeville Press, 1981.

[Upson 89] C. Upson, T. Faulhaber, D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. van Dam. The Application Visualisation System: A Computational Environment for Scientific Visualization. *IEEE CG&A*, 9(4), July 1989.

[van Baerle 86] Susan van Baerle. Character Animation: Combining Computer Graphics and Traditional Animation. *SIGGRAPH '86, Course #23, Computer Animation: 3-D Motion Specification and Control*, 1986.

[Vertigo] Vertigo Systems International Inc. *Scene Composition.*

[Voelpel 86] Mark Voelpel. Parameter Paths: A General Approach to Computer Animation. *SIGGRAPH '86, Course #22, Advanced Computer Animation*, 1986.

[Wavefront 89] Wavefront Technologies. *The Personal Visualizer*, 1989.

[Wilhelms 85] Jane Wilhelms and Brian A. Barsky. Using Dynamic Analysis to Animate Articulated Bodies Such as Humans and Robots. *Graphics Interface '85*, 1985.

[Wilhelms 86] Jane Wilhelms. Towards automatic motion control. *SIGGRAPH '86, Course #23, Computer Animation: 3-D Motion Specification and Control*, 1986.

[Wyvill 75] Brian Wyvill. *An Interactive Graphics Language.* PhD thesis, University of Bradford, December 1975.

[Wyvill 84] Brian Wyvill, Breen Liblong, and Norman Hutchinson. Using Recursion to Describe Polygonal Surfaces. In *Proceedings Graphics Interface '84*, pages 167–171. CIPS, 1984.

[Wyvill 86] Geoff Wyvill, Craig McPheeters, and Brian Wyvill. Data Structure for Soft Objects. *The Visual Computer*, 2(4), February 1986.

[Wyvill 88a] Brian Wyvill. Soft Objects for Character Animation. *SIGGRAPH '88, Course #4, Synthetic Actors*, 1988.

[Wyvill 88b] Brian Wyvill, Mike Chmilar, and Charles Herr. A Simple Model of Human Animation. *SIGGRAPH '88, Course #4, Synthetic Actors*, 1988.

[Zajac 64] E. Zajac. Computer-made Perspective Movies as a Scientific and Communication Tool. *CACM*, 7(3), 1964.

[Zajac 66] E. Zajac. Film Animation by Computer. *New Scientist*, 29:271–280, 1966.

[Zeltzer 82] David Zeltzer. Motor Control Techniques for Figure Animation. *IEEE Computer Graphics and Applications*, 2(9), November 1982.