USING CONVERSATIONAL SYNTAX FOR 3D MODELED ANIMATION

by

Glen Cameron Darling
B.Sc. University of Victoria, 1984

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

We accept this thesis as conforming
to the required standard

[redacted]

Dr. H.A. Müller, Supervisor (Department of Computer Science)

[redacted]

Dr. J. Muzio, Departmental Member (Department of Computer Science)

[redacted]

Dr. G.F. McLean, Outside Member (Department of Mechanical Engineering)

[redacted]

Dr. C. Peter Keller, External Examiner (Department of Geography)

University of Victoria

Supervisor: Dr. H. A. Müller

ABSTRACT

Commercial 3D modeled computer animation systems tend to fall into two categories: easy-to-use but less capable direct manipulation systems and difficult-to-use but highly capable programmable systems. The former allow animators to directly manipulate visual images of their actors in strictly limited ways. The latter have greater flexibility but require knowledge of computer programming and matrix algebra in order to manipulate mathematical descriptions of the actors.

This thesis presents a compromise system based on direct manipulation together with an English-like programming language and an integrated distributed rendering system. The *Animaker* system is suitable for use in any application area not requiring real-time response. Simple animations can be expressed concisely without using programming constructs such as decision or looping structures. Skilled programmers may also use the procedural and object-oriented *Animaker* language to encapsulate "clip animations" for later use by less skilled animators.

Examiners:

[redacted]

Dr. H.A. Müller, Supervisor (Department of Computer Science)

[redacted]

Dr. J. Muzio, Departmental Member (Department of Computer Science)

[redacted]

Dr. G.F. McLean, Outside Member (Department of Mechanical Engineering)

[redacted]

Dr. C. Peter Keller, External Examiner (Department of Geography)

Table of Contents

List of Figures

Acknowledgements

First of all I would like to thank my friends Brian Corrie and Jim McBride and my father, Peter Darling, for reading my early drafts and providing many constructive comments.

Secondly I want to thank Dr. Micaela Serra and Dr. Jon Muzio who are responsible for bringing me into the Master's program here at the University of Victoria. Their initial encouragement was the catalyst that enabled me to start down this road.

I would also like to express my appreciation to my thesis supervisor, Dr. Hausi Müller. I made several formal proposals of this thesis idea to Dr. Müller over 15 months before he reluctantly allowed me to proceed on this topic, despite his original objections that its scope was too large for a Master's Thesis. There were many times while programming or writing that I wished I had heeded his advice and taken on a simpler task. In retrospect I am now glad that I was persistent, and I am grateful that Dr. Müller had enough faith in me to let me proceed.

Finally I want to thank my wife, Shawna Darling, for tolerating the many nights I have come home from work an hour or two late only to spend the next six to eight hours planted in front of my computer, programming or writing. Thanks Shawna. I am looking forward to spending a lot more time with you now that this thesis is finished.

Chapter 1

Introduction

1.1 In the beginning...

Animation is a visual art form that first appeared about a century ago. Even early examples of animation in films such as Georges Méliès' *A Trip To The Moon* (1902) and Winsor McKay's *Little Nemo* (1912) showed the ability of this medium to express ideas that still remain difficult or impossible to express with conventional photographic movies.[1]

---

1 A Trip To The Moon illustrates a rocket flight to the moon landing in one eye of the moon's "face." Little Nemo 's main character is a mosquito.

For generations famous fictional characters have been created in this medium. Few North Americans are unaware of *Betty Boop*, *Mickey Mouse*, *Bugs Bunny*, or the *Flintstone* family (to name a few of the classics). Many of the industry's pioneers like Max Fleischer, Walt Disney, Tex Avery, Chuck Jones, Bob Clampett, Friz Freleng, William Hanna and Joseph Barbera have become household names. Recently, animation has enjoyed a resurgence in popularity. Movies such as *Roger Rabbit*, *The Little Mermaid*, *Beauty and the Beast*, and *Aladdin* ranked highly on the box office charts and television shows like *The Simpsons* have been very successful in the prime time ratings wars.

Animated motion pictures have received accolades from within the entertainment industry as well. Perhaps the most notable acclaim came when Walt Disney Studio's *Beauty and the Beast* (1991) won two Oscars and received an Academy Award nomination for best picture.

The special abilities of animated films are also used in many application areas outside of the entertainment industry. Since animation can produce convincing illusions of reality, it is often used when photography is too expensive or simply impossible. For example, visualization and simulation are common scientific applications of animation. Several excellent examples of this animation genre come from James Blinn's work at the NASA Jet Propulsion Laboratory. His films have been used to show both scientists and the public what was to happen on many NASA space missions. Real-time animations in airplane or space flight simulators and other virtual reality environments are also widely used today.

1.2 The illusion, and the associated costs

The motions or transformations we perceive in any movie (animated or not) are actually illusions created by displaying still pictures one after another in rapid succession. These pictures, or frames, must be presented at a rate of 10 or more per second in order for the human visual system to perceive an illusion of movement. In practice, rates any slower than 15 frames per second tend to have a visibly jerking, staccato appearance[2]. Films typically present 24 frames per second (i.e., 1440/minute) while video normally displays 30 frames per second (i.e., 1800/minute). At these frame rates a short film of only a few minutes requires several thousand individual frames to be created then photographed or placed onto video tape.

The traditional animation process[3] uses a completely manual process to prepare each frame to be photographed. In "cel" animation each frame is created as a simple line drawing on paper. Each one is then transferred onto a transparent plastic sheet called a *cel* (since they were originally made of cellophane). Cels are then painted and placed in layers on an opaque background painting. In "claymation" individual frames are prepared by manually repositioning flexible physical models made from modeling clay. In "pixelation" frames are prepared by repositioning collections of everyday objects. Other less usual techniques have also been used to create animated films. For example, the famous Canadian animator Norman McLaren

---

2 See [Foley 90] page 1058, or [Magnenat Thalman 90] page 15, for information regarding frame rates.
3 See Appendix A for a more detailed description of the traditional animation process.

received critical acclaim for an unusual animation he created by scratching directly onto the surface of an exposed film stock.

Regardless of the technique used, the creation of a traditional animated motion picture of any significant length is labour intensive. Feature length animated films are therefore very expensive to produce.

1.3 Computerization and 3D Modeled Animation

Over the last few decades some animation studios have begun using computers to assist in the traditional animation process to save labour and thereby reduce costs. This process is called, "computer-assisted animation."

In computer-assisted animation, the line drawings are scanned into the computer. Once the drawings are in electronic form, test animations can easily be made and automated painting programs can be used to colour the characters. Precise synchronization with the audio track can also be assisted by the computer.

Some animators have replaced the traditional process altogether with a completely computerized process involving no cels, no physical models and no traditional photography. This is called completely computerized animation, or simply, "computer animation."

Computer animation has expanded the horizons of animation in many new directions. Many of these advances rely upon mathematically describing the objects of the animation either as planar, two dimensional

(2D) objects or more realistically as three dimensional (3D) models. These types of animation are called "modeled animations."

For the purposes of this thesis, computer-assisted animation and the 2D forms of computer animation are ignored completely. The focus from this point onward is strictly on 3D modeled animation.

1.4 Programming versus Direct Manipulation

There are two distinct types of tools for creating and manipulating object models in 3D modeled animation. Animation programming environments were the first type of tools to appear. They require both computer programs and mathematical descriptions of the models to be coded (e.g., lists of numerical coordinates for their vertices). Figure 1.1 shows an example of a model description (a single planar square, perhaps one of the six faces of a cube model) from such a system.

```
#define SIDELENGTH (0.05)
Ppoint3 cubeface[] = {
   {0.0,          0.0,          0.0},
   {0.0,          SIDELENGTH,   0.0},
   {SIDELENGTH,   SIDELENGTH,   0.0},
   {SIDELENGTH,   0.0,          0.0},
   {0.0,          0.0,          0.0}
};
```

Figure 1.1: A simple 3D model description from a PHIGS program.

Some easier-to-use animation systems have recently been developed with user friendly approaches to model creation and manipulation. These

systems provide direct manipulation and visualization interfaces to allow the artist to visually create and test the animation of these models without any computer programming. Figure 1.2 gives a screen image showing an example of the modeling user interface in such a system. Only the top and front views are shown here. A side view and a camera view are also available, but are not shown in this example.

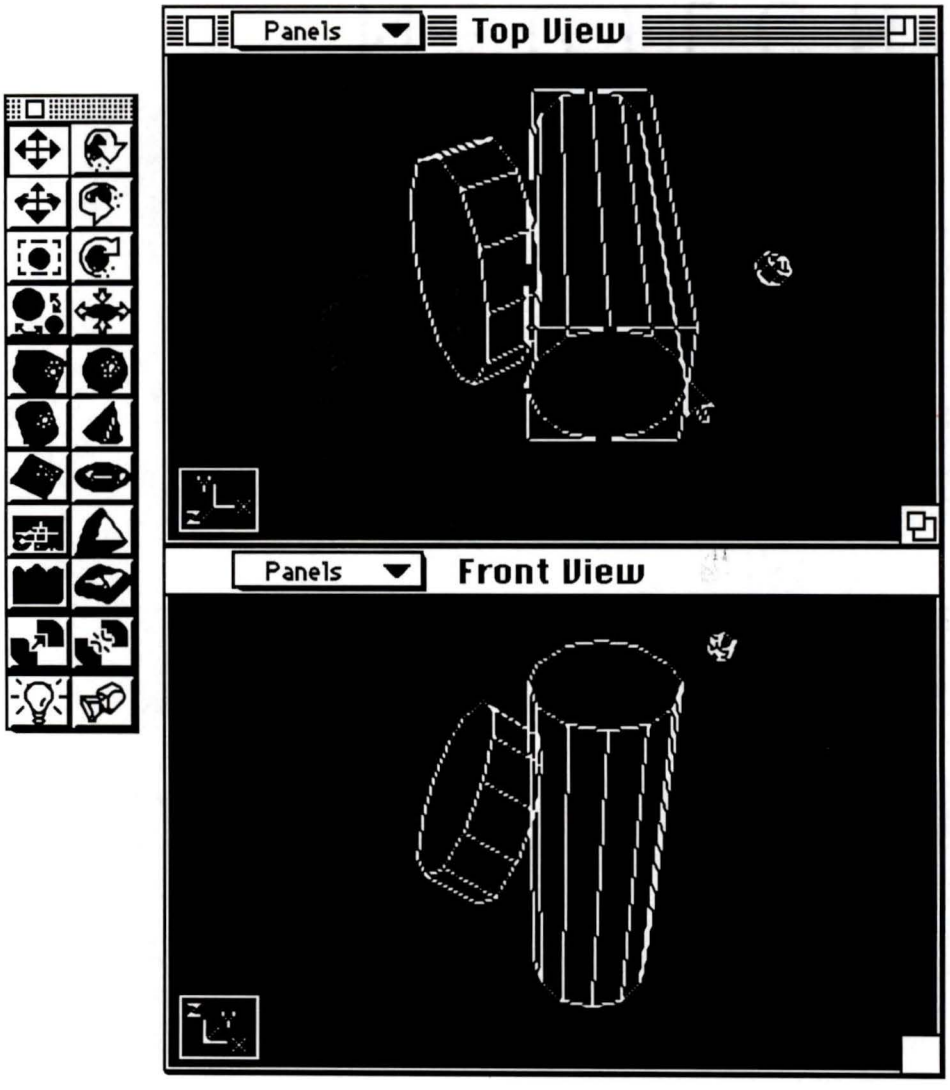

Figure 1.2: Part of the 3D modeling interface in the Infini-D program.

Notice that the top view shows the mouse pointer dragging the long cylinder to a new location. A bounding volume moves with the mouse until

it is released in the new location then the object is redrawn in all of the view windows.

Although direct manipulation systems make computer animation accessible to non-programmers, this is not accomplished without cost. Direct manipulation systems unfortunately sacrifice much of the expressive power available in the animation programming systems. Animators must therefore choose between convenience and ease of use on the one hand versus flexibility, control and complexity on the other hand.

1.5 Model manipulation and frame rendering

Though both of these types of computer animation systems eliminate all manual drawing of frames, a 3D model must be created for each frame (specifying the location, orientation and other characteristics of all objects participating in that frame). Once these models have been created and positioned, each individual frame of the animation can be imaged by having the computer mathematically render it, pixel[4] by pixel. The rendering will be computed using the vantage point of an imaginary camera and the relative locations of these object models and imaginary light sources. Model positions, orientations and other characteristics can be varied between frames to show movement and/or transformation. Typically the number of imaginary lights and their characteristics (e.g., type, location, orientation,

---

4 Pixel is short for "picture element." A pixel is the smallest element of a picture that can be displayed on a typical computer display. That is, a pixel is a single dot of coloured light on one of these screens.

luminosity, colour balance) may also be changed. One can sometimes change the imaginary camera's characteristics (e.g., its focal length, location and orientation) as well during the animation.

Whatever object, lighting and camera manipulation is possible, manually manipulating models for each frame without computer assistance would obviously still be a labour intensive process for the artist. Animation programming environments of course avoid this problem by using procedural control for movement and transformation. Direct manipulation systems typically minimize the number of models that must be individually specified by using "key frames."

1.6 Key frames and "tweening"

Key frame systems save labour by allowing the animator to fully specify only a small subset of the frames in the animation. The computer then moves the objects, light sources and camera along animator defined trajectories between these key frames to create each frame in-between. This mimics the traditional animation key-framing process whereby master animators create the most important frames and assistant animators create the remaining key frames. Less experienced animators then tediously fill in all of the rest of the frames in a process traditionally called "in-betweening" (or just "tweening").

In computerized key framing systems the animator can typically specify the starting point and ending point for an object to move between any two key frames. It is often possible to specify that the object should

ease into and out of this motion as well (to simulate the effects of inertia). Sometimes rates of acceleration and deceleration can be specified for objects to follow along these paths. Newer commercial animation systems provide a direct manipulation interface for describing parameters like these. Figure 1.3 shows an example of one such system. Animation programming environments simply avoid this user-interface issue by requiring complex procedural and mathematical specifications for object movements and transformations. On the other hand, their procedural specifications also allow a larger variety of motions and transformations to be animated.

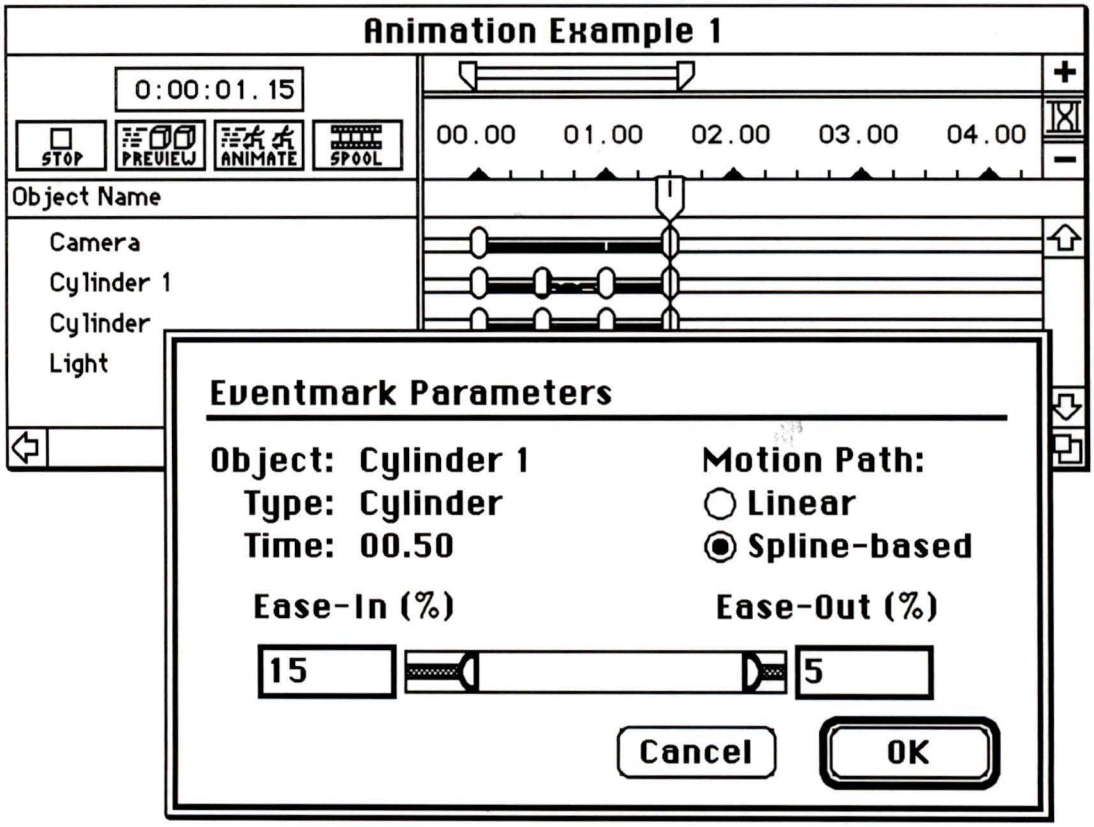

Figure 1.3: Easing in and out of a movement in the Infini-D program.

The tweening between the starting position and end position in key framing systems is mathematically calculated. In the example shown, exactly two choices are provided for easing in and easing out of the motion

(linear, or spline-based). Here, either a straight line or a spline curve may be used to model the acceleration and deceleration of the object over the initial 15 percent, and the last 5 percent of the total travel time but in the intervening 80 percent of the travel time, a constant speed is used. In competitive systems, other acceleration models (e.g., sine curves) may be used to define object acceleration for tweening purposes.

1.7 Object independence and object interaction

Recently, direct manipulation systems have begun to allow "key events" to be independently specified for individual objects (or groups of objects). This is a significant improvement over the specification of key frames for the animation as a whole. Key events allow the animator to describe an individual object's actions independent of the actions of any other objects in the animation.

It would be useful however if the animator could ask the computer to automatically detect object interactions like collisions and have it modify object behaviours based upon these interactions. Key event systems are currently not capable of representing such complexity through their direct manipulation interfaces. Animation programming systems on the other hand can implement detection procedures to check for any interesting object interactions and simply alter program flow based upon the results returned by those procedures. Animation programming systems are therefore capable of creating "intelligent" objects capable of reacting appropriately and automatically in a variety of situations. Direct manipulation systems are not.

1.8 The best of both worlds

Currently direct manipulation systems rely upon key frames and key events with automatic in-betweening and therefore constrain the animator's ability to express complex movements, transformations, object reactions and interactions. Current animation programming systems on the other hand provide greater flexibility in these areas but require the animator to express form and function using computer programming techniques for the maniplulation of mathematical models. This thesis presents a new animation language called "Animaker" as a compromise solution lying somewhere between these other types of system. *Animaker* retains all of the flexibility of current animation programming systems (such as "intelligent" actors) but allows direct manipulation of models, has an English-like syntax, and minimizes the need for traditional programming language control structures.

Although *Animaker* is a complete programming language,[5] it is designed so that no background in matrix mathematics or computer programming is required in order to be productive with this system. Once familiar with the basics of *Animaker,* an animator can begin acquiring knowledge of more complex programming constructs one at a time (e.g., when there is some need for this new complexity in the animation at hand).

---

5 Animaker is complete in the sense that it supports the standard programming language operations of "sequence", "decision", and "iteration". In fact, it is a procedural language much like any other. Animaker also provides variable and value parameters, and a wide assortment of flow control structures.

Animaker makes use of direct manipulation modeling tools to construct the physical characteristics of its actors. *Animaker* is a key event based system like the best direct manipulation systems. This language is more powerful than most of those systems though because it permits the scheduling of procedurally controlled actions for each object or group, upon the occurrence of any event. It also allows these object actions to generate new events that can trigger actions in other objects. This results in an animation scripting environment that has the greater expressive power of the programming language systems, while retaining some of the popular features previously found only in direct manipulation systems.

In order to appreciate the strengths of this system, some detailed background information is required. The next chapter outlines the current state of the art in 3D modeled animation and discusses the strengths and weaknesses of current solutions.

Chapter 2

Background

Existing animation programming languages and direct manipulation systems are many and their capabilities vary widely. The first section of this chapter provides background information on the most important animation programming systems. The second section focuses on selected examples of currently used direct manipulation systems. As each system is examined, its influence upon the design of the Animaker language is discussed. Each section concludes with a discussion of recent work in the area and a summary of the section.

2.1 Animation programming systems

Animation languages have existed for three decades and their capabilities have grown along with advances in computer languages, computer imaging, and computer animation. This section looks at some important languages already in use in the area, and examines their individual strengths and weaknesses. The languages discussed here include the GKS[1] and PHIGS[2] standards, SMALLTALK, RenderMan and a few others that are mentioned in recent papers.

2.1.1 GKS and PHIGS

This section begins with a look at two officially sanctioned worldwide standard languages used in the field. These languages are GKS and PHIGS. The oldest of these, GKS, was originally specified as a 2D system and then extended to include 3D capabilities.[3] GKS is unfortunately restricted in its ability to express complex structures, since it does not support multiple level hierarchical groupings of model primitives. PHIGS on the other hand does have this capability and supports a super set of the GKS capabilities. PHIGS

---

1 GKS is the Graphical Kernel System. It was sanctioned by both the American National Standards Institute (ANSI) and the International Standards Organization (ISO) in 1985. See [ANSI 85] for more details.

2 PHIGS is the Programmer's Hierarchical Interactive Graphics System. It was sanctioned by both the American National Standards Institute (ANSI) and the International Standards Organization (ISO) in 1988. See [ANSI 88] and [ANSI 89] for more details.

3 When its capabilities were extended to include 3D, it was renamed GKS-3D -- which was made an official graphics standard in 1988.

has also been extended (to PHIGS+) to support more powerful primitives and more realistic rendering processes.

PHIGS' support of arbitrarily complex hierarchical graphical structures is particularly useful. It allows the modeler to create objects that are made up of other objects (for example a human model might include two instances of an arm model, each of which might include several finger models). One advantage of this approach is that transformations (e.g., moving, rotating, scaling) applied to a parental structure in the hierarchy are also applied to all of its offspring structures. Unfortunately the strengths of this hierarchical mechanism are offset by an inherent limitation.

When multiple copies of an object are added to a hierarchy they may only differ from each other in transformation. That is, a transformation is applied to a master object to yield each instance of this object type. As a result, if two complete arm objects were added to a torso object they could only differ from each other in this one, limited way. For example, if the hand on one arm was formed into a fist then the hand on the other arm would necessarily also have this configuration. One way to avoid this would be to have two separately defined arm structures (with obvious similarities and resulting redundancies). This awkwardness arises out of the limited ability of a parent object in the PHIGS hierarchy to pass information on to its offspring. On the other hand, it benefits PHIGS' calculation mechanisms by helping to keep them simple and efficient -- an important concern since PHIGS was designed to be capable of real-time animation.

In addition to this representational limitation both GKS and PHIGS suffer from other weaknesses. They are embedded in host programming

languages (like C or FORTRAN) so their users must be able to program in the host language. Unfortunately users must also be able to create and specify 3D mathematical models at low level in the host language (numerically specifying coordinates of vertices, etc.). See Figure 1.1 for an example of a PHIGS model description. Model manipulation in these environments consists of creating and combining two-dimensional matrices -- a non-trivial task even for an experienced computer programmer.

Although GKS and PHIGS represent significant milestones in the standardization of 3D graphics programming, they require highly technical mathematical and programming skills. Unfortunately these skill requirements immediately exclude many animators. The Animaker system, on the other hand, is accessible to animators with limited mathematical or programming skills.

It is also worth noting that general purpose host languages like C and FORTRAN have no built-in animation-specific features. They require the programming animator to design and write many of these primitives before beginning to animate. Object interaction in particular can be difficult or time consuming to code without built-in language support.

SMALLTALK on the other hand is a newer programming language that is completely object oriented. It facilitates object interaction through primitives that allow objects to send messages to each other.

2.1.2 SMALLTALK

SMALLTALK is a programming language and environment that has received a lot of attention since it was made public in the early 1980s. This language, and the concept of object orientation that it pioneered, have gained wide acceptance among computer scientists.

In addition to facilitating communication among objects, SMALLTALK enforces a programming style that is inherently modular and hierarchical. This facilitates complexity management by supporting the development of increasingly powerful composite entities from primitives that have been constructed separately.

Everything in SMALLTALK is an "object". An object is composed of a set of publicly known operations that it can perform as well as data private to the object. The private data owned by an object is stored in its "instance variables". A numeric object for example would contain a numeric value in an instance variable and might be able to perform addition and subtraction operations.

To invoke a particular operation in a SMALLTALK object, a program sends an appropriate "message" to the object. The "interface" to an object consists of the set of messages that it understands.

In SMALLTALK all parts of the system are object oriented. All computation is therefore managed by objects. Every object in turn knows how to perform the set of operations appropriate to its data type or "class". Individual objects of a particular class are called "instances" of that class. A class description outlines the form of all private variables needed by each of

its instances. Class descriptions also contain the shared "methods" which implement the operations that members (i.e., instances) of this class can perform.

When new classes are defined in SMALLTALK, they are defined in terms of existing classes. The new class is said to "inherit" the methods and instance variables of its "parent" class. To these it may add its own methods and instance variables. In addition, it is possible for the new class to "override" any of its parent's methods. That is, upon receiving the corresponding message, an instance of the new class may invoke its own method and therefore behave differently from an instance of its parent class. The tree of classes in SMALLTALK is called the "class hierarchy".

The SMALLTALK programming environment is graphically oriented but only in a 2D sense. It does not directly support any 3D graphical operations. It does support animation of 2D objects to some degree but an animator who wanted to work with 3D models then animate and render them in a photo-realistic manner would have a lot of programming to contend with.

Even so, SMALLTALK was worth mentioning here because the underlying structure of Animaker makes use of many of this language's object oriented concepts. Subsequent descriptions of the Animaker language also make use of the SMALLTALK terms: "object", "class", "class hierarchy", "inherit", "parent", "override", "instance", "instance variable", "message", "interface" and "method".

The implementation of Animaker was also influenced by another important language -- RenderMan.

2.1.3 RenderMan

The RenderMan Interface is the best known language now used for photo-realistic 3D modeled animation. RenderMan is primarily a scene description language. It is used to interface between 3D modeling programs and 3D rendering programs. RenderMan scene descriptions are embodied either in computer program statements that call routines in the RenderMan Interface library, or in simple text using the RenderMan Interface Bytestream (RIB) form. A programming language (normally C or FORTRAN) is required to host the programming library form. In contrast, the RIB form is normally used when a model is transmitted along a communication channel or stored in a file.

RenderMan's strength comes from its completeness as a model description tool. It provides a wide range of rendering primitives that can be produced by modeling programs and passed to rendering programs. Perhaps RenderMan's greatest strength is that it permits scenes and models to be constructed hierarchically. That is, RenderMan constructs permit the user to create different scope levels for the attributes, transformations and individual frames of an animation thereby subdividing the complexity into manageable components. RenderMan source code is also designed to be able to describe a scene with a suitable level of detail to permit photo-realistic rendering to occur.

Although RenderMan is a powerful tool for describing scenes and rendering processes, it would be inappropriate to call it an animation programming language. First of all, RenderMan is not a complete programming language at all because it is completely devoid of flow control

structures (such as decision or looping structures).[4] RenderMan also fails to provide basic support for model creation or animation.

It is possible to create model descriptions manually by mathematically specifying them in C language structures that are passed to the RenderMan Interface routines. Alternatively one could manually enter these structures in RIB format using a text editor. Using either technique the animator must work at a low level, explicitly defining the numerical coordinates of vertices in their models. Clearly RenderMan provides little to facilitate the model creation process.

Animation can also be handled manually in RenderMan. For example, a C programmer can create control structures that handle the desired flow. Required movements and transformations may be handled by passing appropriate matrices to the Interface routines. In RIB format the stream may contain a common background description followed by disjoint descriptions for each subsequent frame. Again movement and transformation are accomplished by means of transformation matrices. In either case the animator must be familiar with matrix algebra and be able to work with translation, rotation and other matrices. Although RenderMan provides many constructs for applying static transformations to objects, it provides no facilities for dynamically handling repeated or gradual transformations of objects over a sequence of frames.

---

4 The RenderMan Shading Language (a subset of the RenderMan Interface) does contain flow control structures but this language may only be used for procedural specification of surface characteristics.

In summary, RenderMan provides few tools to support model creation or animation -- it is designed to be used in conjunction with other tools. By itself RenderMan is simply not a complete programming system for 3D animation purposes. As Pixar (the corporation that created RenderMan) states in its own documentation:

> The RenderMan Interface is meant to be complete, but minimal, in its transfer of scene descriptions from modeling programs to rendering programs. ... The RenderMan Interface is not designed to be a complete three-dimensional interactive programming environment.[5]

Having said all of that, it is possible to build a more complete 3D animation system using RenderMan as a foundation. One can begin with a 3D modeling tool capable of producing RenderMan source code to physically describe the animation actors and backgrounds. To create animations of those models a computer program that either makes RenderMan library calls or generates RIB output is needed. Programmers can in this way create animations with any level of complexity by writing correspondingly complex programs. Unfortunately, computer programming skill still eludes many potential animators so they must look elsewhere for their animation tools.

Since RenderMan has become somewhat of an industry standard for scene description, the Animaker system makes use of this language both for 3D model input and for communicating with rendering programs. This allows the Animaker system to make use of any existing modeling and

---

5 [Pixar 89 RI] Page 4.

rendering tools that use the RIB format. The resulting complete 3D animation programming system has advantages (such as the ability to use a wide range of direct manipulation modeling tools, and photo-realistic rendering tools) over existing 3D animation systems such as GKS and PHIGS. It also has greater capabilities than the direct manipulation systems.

2.1.4 Recent animation language research

Recent research is integrating object oriented programming concepts into computer animation languages. For example, [vanDam et al. 91] describe a powerful object oriented framework where geometric (e.g., spheres, polygons, surfaces of revolution, etc.) and non-geometric (e.g., cameras, renderers, etc.) objects are able to send and receive messages and may even contain their own user-interface components. In this framework, objects are not instantiated from their parent classes in the SMALLTALK way. Instead, an *extension* (roughly like a standard object instance) is initially created by this system according to its prototype (roughly corresponding to its parent class). Each extension is also able to be independently changed. Also, by changing its prototype, all of the extensions descended from that prototype can be changed together. Extensions are more powerful than SMALLTALK objects because the behaviour of one extension may change independently of the others in its class.

These extension structures are beyond the scope of the current version of Animaker. If Animaker supported multiple inheritance (i.e., where a class could be descended from two or more parent classes, inheriting the union of

the parent classes instance variables and methods) then somewhat limited extension-like structures could be implemented. Unfortunately inheritance in Animaker is strictly hierarchical -- each class must have exactly one parent class.

[Gonzales 93] proposed another object-oriented framework for animations written in the C++ programming language. C++ is an object oriented language similar in some ways to SMALLTALK. Whereas all parts of the SMALLTALK system are object oriented, in C++ several parts of the underlying C language remain immutable, are not object oriented and may not be overridden. Nevertheless, objects in C++ (i.e., instances of C++ classes) are conceptually similar to objects in SMALLTALK.

Gonzales' framework outlines a small, efficient and elegant class hierarchy for real-time 3D graphics and animation. Here graphical objects consisting of 3D models and transformations can be hierarchically defined and manipulated. Each graphical object in this framework retains transformation information that applies both to itself and to all of its descendant graphical components. That is, before rendering a hierarchically defined graphical object, each component at the lowest level in the hierarchy is transformed by its own local transformation matrix as well as each of the transformation matrices of its predecessors as you move upward in the hierarchy. This overcomes the restriction placed on hierarchical objects in PHIGS that limits that system's ability to concisely describe bilaterally symmetrical objects.

This can be illustrated with the example given in Section 2.1.1 of two arm objects attached to a torso object. In PHIGS one would have to define

two complete but virtually identical structures. In Gonzales' framework, one would define an arm class, and attach two instances of this class (i.e., two arm objects) onto the torso object. As a result, each arm object can be independently manipulated. At the other end of this hierarchy, each finger could be an instance of a finger class. Each finger object would be transformed by appropriate matrices to position it relative to its parent hand in the arm hierarchy. The hand would subsequently be transformed to an appropriate position relative to its parent forearm object. As the hand was transformed in this way, so would all of its descendant finger objects also be transformed (effectively keeping them attached in the same relative locations on the parent hand). The process continues on up throughout the hierarchy. This ability to hierarchically specify relative transformations simplifies the animator's task.

Gonzales' method of distributing transformations is used by Animaker to apply local transformations to the RenderMan models that hierarchically combine to form the objects that "act" in Animaker scripts. Animaker defines a new transformation scope for each level in the hierarchy of an actor's structure. That is, as each new component object is encountered, a new transformation scope is entered; appropriate transformations are applied and the object's RenderMan description is output; when the component's description concludes, the transformation scope is removed. The parental scope is therefore restored before moving on to the next component object. These scope rules are analogous to the typical scope rules used in procedural programming languages when a subroutine is invoked. Using that analogy, each finger is a procedure that is invoked by the calling hand procedure.

The hand is in turn invoked by the forearm, which is invoked by the upper arm and so on.

2.1.5 Animation programming language summary

All of the language systems discussed in this chapter require their users to be proficient in computer programming before any sort of animation can be accomplished. In addition, the object oriented languages require the user to understand object oriented programming concepts. The next section of this chapter focuses on systems that do not require their users to have any programming skill. By their very nature they are less powerful than the programming systems described above but their ease of use and accessibility to non-programmers have made them very popular.

2.2 Direct manipulation systems

Since animators are not necessarily skilled computer programmers, many commercial animation packages have been created to eliminate the need for programming in computer animation. These systems allow the animator to directly manipulate the models of their actors and specify paths for them to follow from key frame to key frame. This section presents a variety of these new systems and highlights their individual strengths and weaknesses.

2.2.1 The Video Toaster system

The Video Toaster system runs on the Amiga computer and comes with a complete 3D modeling and key event animation system.

The Modeler program that comes with the Video Toaster allows its users to visually create complex 3D models using an interface similar to that shown in Figure 1.2. Once all of the appropriate models have been created the animator can switch to the Lightwave 3D program. In Lightwave 3D, objects are combined to construct scenes. Selected objects in the scene may be moved to new locations and the software can be instructed to perform these movements over any specified number of frames. Objects may also be set to perform cyclical behaviours that repeat every so many frames.

Lightwave 3D objects may also be configured to point at other objects. This is especially useful for camera and light objects. Objects may also be attached together to form complex composite structures which may be animated together as a unit.

Test animations may be performed using wire frame rendering inside the Lightwave 3D program. Final animations may be rendered using ray tracing or various other methods. The program may also be configured to interface directly with a programmable VCR capable of recording single frames. Since rendering can be slow (e.g., when using ray tracing) the rendering and recording process may be automated. As each frame is completed the VCR is automatically instructed to record it. Since this process is fully automated it is often left to run over night. Sometimes a

"farm" of several rendering machines is left working overnight when the animators go home.

The Lightwave 3D program is capable only of relatively simple animations using key events. This system is not programmable. The user can control objects only by selecting discrete points in time (frame numbers) where particular animated actions are to begin, cease or repeat.

Despite its simplicity, this system is in wide commercial use today. The Video Toaster system was used to create the special effects for the movie "Babylon 5" and is now being used on the television series it spawned (both are set in the future on a distant space station). This system is also currently being used to produce special effects for the Sea Quest television series (set in the future on a high-tech submarine).

2.2.2 Electric Image Animation System

Electric Image Animation System (EIAS) is a sophisticated modeling and key-event animation system available on the Macintosh computer system. EIAS is capable of all of the animation wizardry available in the Lightwave 3D program and has several additional features. EIAS is capable of animating almost any parameter of an object from its location and orientation through its size and surface characteristics. This includes such useful variables as the intensity of a light source and the focal length of the imaginary camera.

Most of the possible animations in EIAS can be configured using velocity controls based on spline curves. The rate of acceleration and

deceleration of an object along its flight path may therefore easily be adjusted using direct manipulation in this system. Splines may also be used to control the rate of change of other object attributes from one frame to another. Figure 2.1 shows an example of an animator-adjustable velocity graph in EIAS.

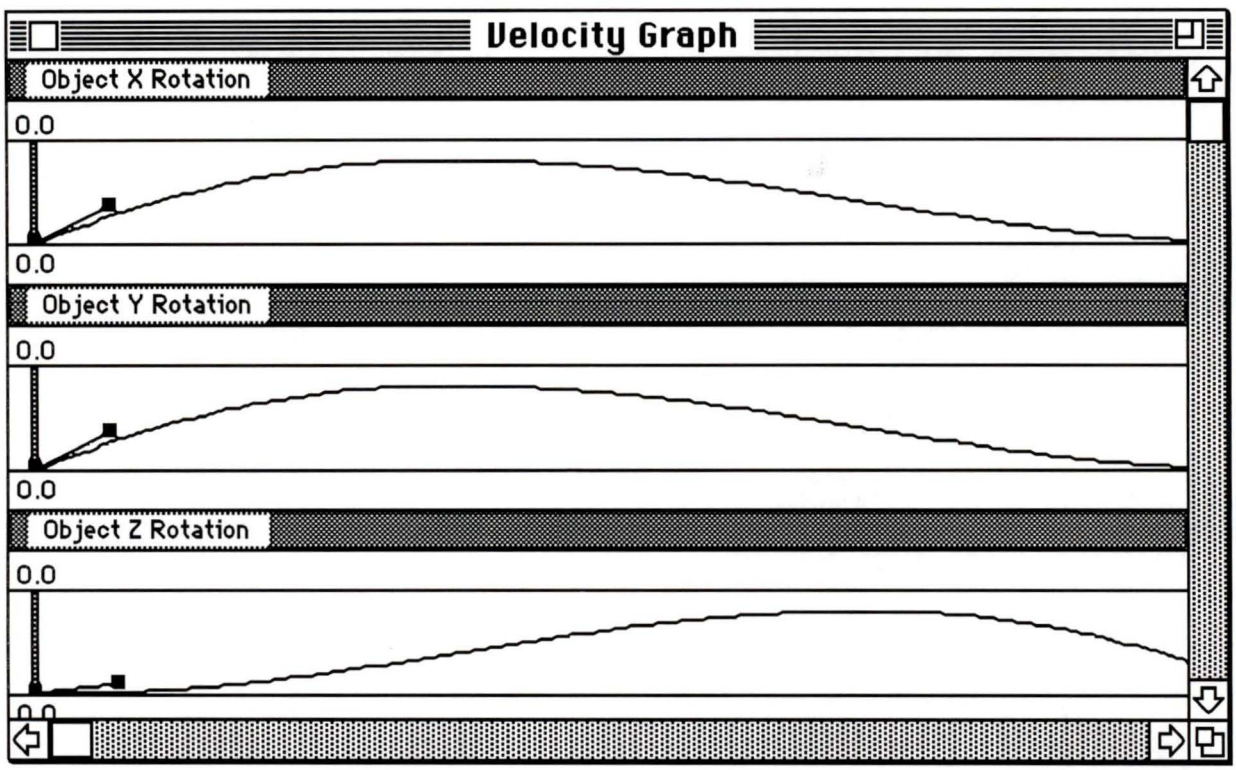

Figure 2.1: Spline based controls in the EIAS program on the Macintosh.

EIAS also supports model deformations. That is, an object may first be defined using the standard direct manipulation object modeling tools then it may be deformed in any of several different ways (e.g., bending, rippling, shearing, stretching, tapering and twisting).

Sound track synchronization is also facilitated by allowing the sound track to be placed beside the animation time line. An animator may easily line up key events with their corresponding sound track cues.

2.2.3 Advanced Visualizer

Wavefront Technologies' Advanced Visualizer is an example of a high end system designed to run on Silicon Graphics workstations. Advanced Visualizer is capable of all of the features that EIAS provides. Advanced Visualizer (unlike most direct manipulation systems) is also capable of both simulation and scripting.

The Dynamation module in Advanced Visualizer allows the user to create reality based simulations. These simulations use the physical characteristics of the 3D object models and preprogrammed formulae for friction, gravity and turbulence. In this system, an artificial set of circumstances can be created and unleashed. The system completely controls the behaviour of the objects over time from that point onward (using the physical characteristics of the objects and the built-in formulae that model physical laws). Scripting can also be used to provide greater user control over the movement and transformation of objects in this system.

Advanced Visualizer also supports the use of "clip animations". Clip animations are conceptually similar to clip art. A clip animation includes both a set of object models and associated animated abilities for these models. One animator may create clip animation collections and sell them to other animators. A purchaser may add one of these clip animations into any animation that is being built. The clip animation objects are already inherently capable of certain behaviours which the purchaser may use. Clip animations can make it easy to create complicated animations with little effort.

2.2.4 Recent work in direct manipulation systems

Since the lower end direct manipulation systems are consumer-oriented products, they are constantly evolving. The high end systems also tend to participate in the software industry's unavoidable "feature wars." Most direct manipulation systems now support key events instead of mere key frames. Most also provide some type of linear or spline-based control for easing in and out of motions (to more correctly simulate inertia). Higher end systems all also provide the ability to stretch and otherwise deform objects over time. Advanced features such as clip animations, particle systems and inverse kinematics are gradually being added to most of these systems too.

A few of the high end systems have recently added modules (such as the Dynamation module in Advanced Visualizer) that provide the ability to simulate physical systems under restricted conditions. These tools are useful in scientific visualization, accident reconstruction and many other fields. Entertainment industry animators also use these features to simplify the realistic simulation of special effects such as fires or explosions.

Since the Animaker system is completely programmable it is capable of handling any of these processes. In fact, clip animations were a prominent feature in this system's design. Since Animaker makes use of existing modeling tools it may also make use of direct manipulation techniques for model creation thereby achieving a significant leap in user friendliness above the other animation language systems presented in the first half of this chapter.

2.2.5 Direct manipulation system summary

Direct manipulation systems are easy to use. Despite the inherent limitations of most of these systems, they have become popular consumer products and have also achieved widespread commercial use. Their application in feature length movies, television programs and commercials has become commonplace. They are also used for many other visualization applications.

Rapidly increasing demand has created a large number of similarly powerful animation systems. Macromedia 3D, Infini-D, Strata Studio Pro and Will Vinton's Playmation are all consumer-oriented systems available on the Macintosh. They each provide key event animation capabilities similar to the other programs described in this section. Alias Research's PowerAnimator, SoftImage's Creative Environment, Thompson Digital Image's Explore, ElectroGIG's 3DGO, Vertigo Technology's Vertigo and Wavefront Technologies' Advanced Visualizer are all examples of high end systems available on Silicon Graphics workstations.

Intense competition in the industry (driven by the increasing demand) tends to keep the high end systems very similar in terms of the features they support. Any innovation that propels one system ahead of the others is usually adopted quickly by all of the competitors.

The nature of these direct manipulation systems restricts the abilities of most of them to express arbitrarily complex animations. Those few systems that provide scripting interfaces allow users with programming skills to transcend these limitations. Since Animaker provides only a

scripting interface, its users become familiar with the scripting environment even as they develop simple animations. Complexity such as flow control can be added gradually and incrementally. Systems which are primarily based upon direct manipulation may make the transition to scripting more distinct and more difficult for their non-programming users to accomplish.

The Animaker system allows its users to work in a middle ground somewhere between the complexity of other available animation programming environments and the features found in the popular direct manipulation systems. It provides all of the power of the programming systems while emphasizing simple beginnings and incrementality. That is, Animaker provides features to facilitate the production of relatively complex animations from simple programs. Novice programmers may also add to their simple programs as they learn the more advanced features of the language. Animaker is therefore suitable to a wide variety of users for a wide variety of uses.

Chapter 3

Applications

Animaker is a tool that is initially accessible to most animators. Novices can use the Animaker environment to create simple animations using their own simple actors or more complex actors provided to them in clip animation libraries. Once one has acquired a superficial understanding of events and flow control, one may move on to build more interesting Animaker animations. With still more skill and experience, Animaker can be used to create actors with arbitrarily complex, striking or subtle characteristics and abilities. Animaker is therefore suited to many animation applications.

Any computer algorithm, and therefore any computer animation, can be expressed in Animaker but this system does have limitations. The domain of real-time animation in particular is ignored by Animaker. It would be possible to arrange an animation pipeline to render relatively low quality images of individual frames in real time if Animaker could produce frame models at the corresponding rate. The latter may be possible for some relatively simple animations involving very few and very simple actors but if so it would be incidental to the goals of this system. Animaker is designed to be an element in an animation pipeline that feeds complex models to a high quality (e.g., photo realistic) rendering tool such as a ray-tracing program. Figure 3.1 shows an example of an image rendered by a ray-tracing program.

Figure 3.1 An image ray-traced from a 3D model.

Currently, photo realistic rendering programs (even when run on very fast computers) take so long to produce images from even moderately complex models that generation of high image quality in real time remains unlikely in the near future.

Given that Animaker is capable of describing any animation, in what application areas would this system excel? To answer this question one must understand the strengths of programming languages in general and those of Animaker in particular. Animaker gives the animator all of the advantages computer programmers have come to expect from language systems: automation, extensibility, incrementality and domain retargetability. Each of these is examined below and Animaker's particular strengths with respect to animation are identified. This chapter concludes by giving two detailed concrete examples of possible Animaker applications.

3.1 Automation

At the least, the Animaker language is a tool that animators can use to automate model transformations and manipulations. Actions may be described procedurally and hierarchically. That is, movement and other commands may be grouped into named procedures that can be invoked to perform those commands according to specified parameters. Objects may also be attached to other objects hierarchically so transformations applied to a parent object are subsequently applied to all descendant objects. These simple automations are augmented by standard programming features such as control structures, variables, etc.

Animaker control structures allow the animator to automatically make an actor repeat an action, or perform it under particular conditions or at particular times. Actions are conditionally performed using standard (if/then/else) decision structures. Actions are conditionally repeated using the language's standard (for/repeat/while) looping structures. Automatic and

timely execution or repetition of a particular performance may also be handled in more innovative ways using Animaker's event management mechanism.

Variables in Animaker are data containers similar to those used in other programming environments. In addition to globally accessible variables, Animaker allows object instances to contain private local variables which record the state of the object or any other data local to that object. This permits multiple instances of an object type to be created and manipulated independently of each other. That is, each instance automatically has the same abilities as all other objects of its type but the animator may choose either to have it act independently or in concert with others of its type.

In many application areas, a relatively small set of object types performs relatively few actions relatively frequently. Animaker can provide significant automation opportunities in these situations. By defining the structure of each object type, and procedurally (and parametrically) describing the most frequently performed actions one creates an Animaker class. A library can be built up from a collection of these automated actor classes, each with certain physical characteristics and behavioural abilities. Using this library, any number of animations can be constructed and automated by creating instances of these object types and invoking their procedures along specified time lines using specified parameters.

For example a model for a human actor could be created from modeling primitives. Procedures could be written to describe how it would walk. These procedures would consist partly of commands to manipulate

the polygons forming the model's legs. An animator could add one of these actors to their animation and make it walk. An animator could also make many instances of the model walk together in step as an army on parade, or randomly out of step like independent vacationers strolling along a beach.

3.2 Extensibility and incrementality

Animaker provides three distinct types of extensibility. First, actor scripts may be completely independent from animation scripts. This makes it possible to incrementally modify one script or the other without having to modify both. Second, Animaker actors may be defined in terms of each other (i.e., actors may be constructed hierarchically using other actors of the same or different types as component parts); this opens the door to many types of extensibility. Finally, an Animaker actor may inherit characteristics and abilities from the object types which precede it in the class hierarchy. This allows existing actors to be adjusted in increments to suit new requirements.

By storing procedural descriptions of each object's behaviour together with its physical description (and away from the script describing the whole animation) Animaker allows its actors to become independent of the animations they participate in. This independence also allows Animaker actors and animations to be extended and enhanced independently after their initial creation.

An animator could, for example, create a crude model of a car that consisted of a single cube with two lights attached to the front but no moving

parts. This car object could be given an ability to drive in a particular direction, to turn, etc. An animation script that drove this car around some other objects could also be created. The animator could view this animation, and later return to enhance the car model. The car's physical characteristics could be enhanced by adding wheels; its behaviour could be enhanced by making its wheels turn appropriately as it moved, and as it turned corners. After updating the model, the animator could have Animaker recreate the animation from the original script. This time the new, enhanced car would be used. The car's appearance and behaviour could be enhanced incrementally and iteratively until the animator was satisfied.

Animaker actors can be composed of primitive, system defined objects and user defined objects of arbitrary complexity. Therefore Animaker actors can also be extended through the use of components in clip animation libraries (i.e., simply other Animaker objects). For example one might create (or find in an existing library) a wheel object capable of turning on an axis at a particular speed. Four instances of this object type could be added to the original cube car from the preceding paragraph in order to make the new, enhanced car object. This wheel class could also be used as a component of many other actor types from unicycles and roller blades through shopping carts and furniture dollies.

Each Animaker object class inherits the abilities and attributes of its superclass. This provides another mechanism for extensibility and incrementality within Animaker. An existing library of classes can be extended by building upon the existing classes. For example, a basic car class could be created from which other specialized car subclasses would be derived. A convertible class, a mini-van class and a pick-up truck class

could all descend from the original car superclass. Their physical characteristics (and perhaps their behavioural abilities) would differ, but they would all inherit certain basic attributes and abilities from the parent class.

3.3 Domain retargetability

As noted previously, Animaker class definitions may be collected into libraries of behaviourally intelligent actor classes. Instances of these object types may be used by any animator in any appropriate application domain. Derivative subclasses may also be used to retarget a generic class to the specific needs of a new application domain. Of course, as a derivative class is adapted to a new domain, its generality is reduced as is its potential applicability to other domains.

For example, the automobile classes described so superficially above could (with sufficient programming) be made suitable for multiple application domains. For a cartooning application it might be appropriate to give a car the ability to rear up on its hind wheels in anticipation before beginning to move forward. In a special effects application it could be important for the car to know how to dramatically but somewhat realistically disintegrate into many pieces to simulate the results of an explosion. In an accident reconstruction application a car may have to be able to act exactly as a real car would act in an impact with another car (using the laws of physics and according to each car's mass, structural characteristics, etc.).

Conceivably one could create a single car class capable of being used in both of the first two application areas. It could be made useful in many

entertainment application areas. One might alternatively create a single car class to handle both of the last two applications perhaps with parameters which controlled the level of exaggeration to be performed. But could one car class be used for all three application areas?

The answer is probably, "yes, but that may not be the wisest choice." It seems clear that the cartooning and accident reconstruction application areas have conflicting goals. The former relies upon exaggeration to emphasize and dramatize whereas the latter has physically correct behaviour as its paramount goal. In general, when common ground between application areas can be found, a base class can be created incorporating these common characteristics and abilities. Derivative subclasses can subsequently be created to target each individual application domain.

3.4 An example of an entertainment application

This section explores how Animaker could be used in an entertainment application -- one episode in an animated series. Like many animated series (from Roadrunner and Coyote through Beavis and Butthead) there are two main characters in this animation; let's call them Alpha and Beta. For now, Alpha and Beta are simply anthropomorphic robots. This is a common choice of character type in computer animation because it is still difficult to realistically model biological forms.

Construction of an entertainment application such as this normally begins with a synopsis of the animation plot:

> The story opens inside a futuristic looking house where Alpha is lying on a couch watching a television. Alpha says, "I hate TV; I need some adventure!" The phone rings and it is Beta suggesting they go out for a drive in Beta's new car. Alpha heartily agrees and gets ready while Beta comes over to Alpha's house. They both get in Beta's car and leave. Beta is a maniacal driver and they encounter several adventures during their drive. Eventually Beta returns Alpha home. Alpha goes in (looking somewhat the worse for wear) collapses onto the couch and starts watching television again. The phone rings. It is a another person asking if Alpha wants to go out for a drive. Alpha makes an excuse, says no, and politely ends the conversation. The phone rings. Alpha ignores it. In the background the answering machine answers the phone. Alpha says, "I love TV". Fade to black.

Given a plot synopsis like the one above, the next step is the development of a storyboard itemizing the sequences and scenes in the film. The sequences include at least: Alpha using the remote control while lying on the couch, Alpha answering the phone, Alpha walking and Alpha entering and exiting both Beta's car and Alpha's house. The film's scenes include at least two in a room in Alpha's house containing a couch and television set. Other scenes include the exterior of Alpha's house (to show Alpha going and coming), Beta's car exterior (for entry and exit) and Beta's car interior (for the middle scenes of the movie during their driving adventure).

The layout phase begins with the construction of a basic model to be used for both Alpha and Beta. Rough models of the backgrounds must also be constructed at this stage. The required backgrounds include Alpha's room and all of its fixtures, Alpha's house exterior and Beta's car (both interior and exterior). Example renderings are used to refine all of these models. The

director can also experiment with different camera positions at this stage using these models.

The sound track is recorded next. From the sound track, the animators can extract precise timing information for the animation.

When basic models and timings are in hand, Animaker scripts describing the required movement sequences may be constructed and test animated. All of the sequences described earlier would have to be described in detail here. Some of the test animations at this stage would include the background models as well to ensure consistency. On the other hand, backgrounds are normally omitted for most test animations to improve rendering speed.

The required Animaker scripts for this animation include both object scripts and animation scripts. The object scripts describe such things as how Alpha walks or operates the remote control and how Beta operates the car. The animation scripts direct Alpha's walk to the door, door opening and closing, etc. and synchronize all movements to the established sound track time line.

As movement sequences and models are refined, and test animations get better synchronized with the sound track, line tests can be put together. The director can use these line tests to review and refine what is wanted.

Eventually the models, and their actions reach a satisfactory level and a final test print is made. Additional sound effects can be dubbed in at that point. Finally the animation goes on to post production and eventual release.

In this application, Animaker stands out with several clear advantages over other computer animation tools. Animaker actors consist of both form and actions. As a result this animation is developed by giving the robots, the car, doors, etc. the various abilities they need to exhibit. These abilities are stored with each object's physical description. They can also be developed independently from the main animation. Once these abilities are written into an actor, they can be used and reused in different situations in the current animation and future animations. Since these animated abilities are manipulations of 3D models only, the camera may also be independently positioned anywhere to shoot or reshoot frame images without having to change the models or their scripts.

Animaker supports incremental refinement of models, actor scripts and animation scripts. In this application the physical characteristics of the robot actors' models could be incrementally refined to give them increasingly human appearance. Their scripts could be similarly incrementally enhanced to make their actions appear quite human.

Animaker helps in some other, more mechanical, ways as well. It makes test animations and line tests trivial to assemble. Animaker animation scripts also support precise synchronization of events to particular frame numbers or time marks within the sound track.

3.5 An example of a scientific application

This section looks at a hypothetical scientific application of Animaker. In this application, we visualize an accurate simulation of the

movement of an articulated object in a low gravity environment. As has been noted earlier, applications with goals rooted in realism can have very different requirements from entertainment applications but Animaker is still applicable.

Since there is no plot *per se* in this situation, the process begins with the development of 3D models for each component in the articulated structure. Mathematical formulae must be developed to govern the movements of these components according to our understanding of the laws of physics. These formulae must be programmed into the system, using Animaker animation scripts (possibly including external routines written in C or Pascal).[1] During the animation these coded formulae need to receive physical information about each component regarding its mass, centre of mass and the forces acting upon it (including parameters describing the low gravity environment). Each actor must be given instance variables to record all of the required information.

In summary, creating an animation in this application consists of the following steps. 3D models are constructed and given the desired visual attributes (shape, relative size, rotation, etc.) and physical attributes (mass, velocity, etc.). The appropriate physical laws are modeled in the animation scripts. Initial positions, speeds, etc. are assigned to each actor instance. At this point we have a more-or-less complete description of a physical system

---

1 Note that the formulae modeling gravity in this environment would probably be coded as part of a class that would appear as a descendant of the Actor class (but placed relatively high in the class hierarchy). All classes of objects using gravity in this simulation would be defined as descendants of this gravity class.

at a particular instant in time. All that remains is to allow time to begin running and observe what happens.

If the articulated object contains any motors, hydraulics or other sources of kinetic energy, their effects must also be programmed into the object and animation scripts. The object should probably provide methods for this that allow the animation script to control the object in some appropriate manner. The division of responsibility between the object script and the animation script is a design issue for the animator. Since this particular object is emulating a real-world mechanism, it may be useful for the animator to provide a similar interface to the object that its real-world counterpart provides to its users.

To facilitate observation of this simulation we need to specify the positions and movements of cameras and lights. It may also be aesthetically appropriate to add additional non-participating objects (e.g., background objects) into the scene. Of course, if any additional outside forces are injected into this system, or if the object's properties change over time, these things must also be added into the animation script.

When all of this is ready, test animations can be used to refine formulae and verify their correctness. Eventually final animations can be produced.

By handling applications such as this one Animaker shows it has the same utility as other animation programming systems. This gives Animaker a distinct advantage over most direct manipulation systems (which are incapable of modeling behavioural laws without manual intervention from an animator). The next chapter describes how the Animaker language is

designed to retain this expressive power while incorporating some of the strengths of the direct manipulation systems .

Chapter 4

Language Design Goals

First and foremost, Animaker is designed to require few mathematical or traditional programming skills from its animators. Whenever any other design goal conflicted with the original, primary goal, the primary goal took precedence. It is hoped that by keeping the system accessible is this way, novice programmers may grow into programming once they have become familiar with other aspects of the system.

The design process began by observing that computer animation implies 3D model creation, the specification of various manipulations of these models (in both spatial and temporal terms), and finally image

rendering. This suggests that any computer animation language must at least be able to:

- describe model structures,
- describe model manipulations, and
- describe the rendering process.

Since RenderMan provides a complete set of tools for describing both models and rendering, it was selected both for Animaker's model input and for its frame description output.[1] Once this decision was made, all that remained was the need for tools to describe model manipulations through space and time. The selection of RenderMan interfaces also means that existing modeling and rendering tools are seamlessly incorporated into this system.

The spatial rearrangement of objects is normally accomplished in animation programming environments using the standard operations of translation (i.e., movement), scaling (i.e., resizing), and rotation. All of these manipulations are handled at some level by means of matrix multiplications.[2] More complicated distortions such as skewing, stretching and squashing can also be accomplished using matrices. To satisfy

---

1 The Animaker system actually supports several different model input and frame description out languages including RIB, POV and NFF. See Chapter 6 for more details.

2 More precisely, 3D coordinate points are usually represented using *homogeneous coordinates*. That is, (x,y,z) is represented as [x,y,z,1]. The homogeneous coordinates defining a model can be multiplied by 4X4 translation, scaling, rotation, or other matrices in order to move, resize, rotate or otherwise manipulate the model. Since all of these transformations are performed by means of matrix multiplication, they can be combined easily. Note that translation, scaling, rotation and some other 4X4 matrices perform *affine* transformations. That is, they preserve parallelism of lines, but not line lengths or angles. See [Foley 90] page 207 and page 1106 for more details.

Animaker's accessibility goal, constructs have been provided which eliminate any need for the animator to use matrix algebra for the most commonly used manipulations. At the same time, other constructs are provided to assist the advanced animator in the use of any arbitrary matrices.

The management of model manipulations over time is normally handled in animation languages by means of traditional programming constructs. Animaker provides traditional programming constructs, but it has also been designed to incorporate a powerful event scheduling mechanism. This tool facilitates the management of complex scheduling problems by supporting:

- key frames and key events,
- precise synchronization of the animation with the sound track, and
- complex communication and interaction between objects.

The Animaker event scheduling mechanism also allows the animator to independently direct multiple, overlapping actions. This is something that can be difficult for novice programmers using traditional programming languages.

4.1 Fleshing out the skeleton

The skeleton of Animaker consists of the accessibility goal, RenderMan interfaces, and sets of tools for spatial and temporal manipulation to satisfy both novice and advanced animators. Given this framework, the remaining design issues are the usual ones dealt with by computer language designers. These issues include the language type, the

translation method, syntax and semantics. This section outlines each of these design concerns.

First of all, Animaker is designed as an imperative[3] language since the fundamental nature of animation implies iteration. An applicative or goal-oriented approach would be awkward in this context.

Like many computer languages, Animaker is amenable to translation either by compiler or interpreter. Traditionally, compilation offers improved run-time efficiency with the drawback that there is an initial delay for compilation. Unfortunately though, compiled programs tend to have less diagnostic information available at run-time than do their interpreted counterparts. In Animaker, neither the translator's performance, nor the script's execution speed are significant when compared with the long time required for rendering each frame. In addition, the translation process may be carried on in parallel with the rendering process. All of this suggests that compilation of Animaker scripts into machine language offers no significant speed advantages. Since an interpreter may be able to provide better run-time diagnostic information (and since there is little advantage to compilation) this thesis presents an Animaker interpreter. Chapter 6 discusses the design and implementation of this interpreter.

A conversational syntax model based on the English language was selected over other more traditional programming language models.

---

3 Imperative languages are characterized by the use of iteration and assignment of expressions to variables. Examples of imperative languages include C, Pascal, FORTRAN, PL-I and COBOL. They are contrasted with applicative or functional programming languages (such as LISP, Scheme and Prolog) which are characterized by recursive functions and recursively defined data in place of iteration and direct assignment.

Conversational syntax has been used with success in other computer languages[4] and although it is more difficult to translate, it supports Animaker's paramount design goal of accessibility to non-programmers. The issues considered while designing the syntax of Animaker are covered in detail in Chapter 5 as each language construct is examined. Some additional syntax issues are analyzed in Chapter 6 when the interpreter design is discussed.

Obviously the Animaker semantics must deal with model manipulation. There are many ways that this model manipulation could be handled and specified. Traditional programming constructs would be adequate, but a relatively recent advance in computer language design (evidenced by languages such as SMALLTALK -- discussed in Chapter 2 -- and C++) called object-orientation offers some significant advantages. The remainder of this chapter discusses the object oriented nature of Animaker in general and outlines the specific advantages this technology offers to animators. Chapter 5 lays out the precise semantic details of every Animaker construct. Chapter 5 also includes examples of the use of each of the most important constructs.

4.2 Object-orientation

Animaker is designed as an object-oriented language with conceptual ties to SMALLTALK and C++. Objects in Animaker are language

---

4 HyperTalk and AppleScript are popular examples of conversational programming languages.

components that have particular attributes and abilities. That is, in addition to containing information, objects are also capable of performing specific functions.

An object's attributes are normally private to the object itself. An object's abilities on the other hand are publicly accessible. They are the object's interface to the outside world. Object abilities are invoked by calling upon corresponding object methods (passing appropriate information in the process). Methods may return information to their callers. Selected object attributes can therefore be made publicly accessible through calls to the object's methods if desired.

Each object is said to be an instance of a particular class of objects. A class definition describes the objects of its class. It contains complete structural and other attribute details along with descriptions of methods. The physical structures are described using RenderMan's model description language. Attribute descriptions are similar to variable declarations in other imperative programming languages. Method descriptions are written procedurally like sub-programs in other imperative environments.

In Animaker, every class except the root class is described in terms of its parent, or superclass. By default, each descendant, or subclass, inherits the attributes and methods of its superclass. Each subclass typically adds its own additional attributes and methods to those it inherits; it may also change the behaviour of (i.e., override) any of the methods it inherits.

It is worth noting that an Animaker object's structure and other attributes may be unique for each object instance. Object method descriptions on the other hand are consistent across an entire class. They

therefore occur only once and are shared by all object instances which use that method. Note that a particular method description may either occur locally in an object's class description or closer to the top of the class hierarchy in one of its superclass' descriptions.

Although Animaker is not completely object-oriented, everything that an animator wishes to animate must be an object. This single requirement enforces an intuitively simple, but conceptually sound organization upon all of the items participating in Animaker animations. Modularity, information hiding and hierarchical design are all built in to this organization and they are difficult to avoid. By enforcing this rigour (an object-oriented rigour) many traditional programming pitfalls are avoided.

Most of the objects animated by Animaker are instances of user-defined classes that are created as subclasses of the system-defined "actor" class. User-defined actor descendants can be used to model real-world objects -- both structurally and kinetically. That is, user-defined classes can be imbued with methods that invoke the movement capabilities of the corresponding real-world object. Actor descendants may be composed of connected parts that are cameras, lights, geometric structures and other actors. An Animaker automobile, for example, might begin with an appropriate geometrical form including headlights and a camera at the location of the driver's head. One could create a simple animation by just moving this car; the animation would contain a view through the windshield of whatever the headlights illuminated along the car's path.

4.3 Animation is naturally object-oriented

Animated objects in the real-world include biological entities, wonders of nature and mechanical creations. Many similar components are found on different objects in each of these groups. Many animals have articulated limbs for example. Many machines contain wheels. Animaker facilitates modeling these real-world items by encouraging hierarchical design. One can design a limb class, and use a number of instances of that limb in the definition of an animal class. Similarly a wheel class could be created and used to construct various different machine classes.

Real-world objects also tend to have overall similarity to other objects. For example, male animals tend to be similar to female animals of the same species. Evolutionary theory also suggests that species change incrementally, so those near to each other on the evolutionary tree tend to have similar characteristics. Animaker facilitates the modeling of similar types of objects through inheritance. In each case a base class can be created embodying the common characteristics of the objects to be modeled. Descendant classes can be defined from this base class to describe each of the variants[5].

---

5 In the object-oriented context, the term "inherit" has approximately the same meaning as it does in discussions of biological heredity. A descendant class *inherits* all of the characteristics of its parent class and it adds to those any additional characteristics defined specifically for this descendant class. A descendant class with no programming code of its own, has exactly the same characteristics as its parent class. Note that the code for a descendant class may also override any of the characteristics of its parent class.

Hierarchical design, and inheritance through class definition are intuitively simple concepts that give Animaker significant organizational advantages over animation systems that do not support these features.

4.4 Real-time versus realism

It is worth emphasizing here that although many computer animation systems (particularly the animation programming environments) concentrate on rapid manipulation and rendering of images for real-time display, Animaker does not. The aim of Animaker is to provide a simple but powerful system for generating sequences of models for high quality rendering. Currently only relatively low quality images (e.g., shaded polygons) can be generated in real-time for applications such as flight simulators and other virtual reality environments. Even so, the complexity of the scenes being rendered is typically limited. It is intended that Animaker models be rendered by a higher quality process requiring significant rendering time thereby precluding real-time image generation.

Notice that a direct benefit from removing any concern for real-time response is that realism can be enhanced significantly in several ways. Photo-realistic image rendering techniques like ray tracing or radiosity,[6] for example, make it possible to use 3D computer models to generate images which are virtually indistinguishable from reality. Perspective, opacity,

---

6 Ray tracing and radiosity are the names of popular algorithms currently used in computer graphics to produce photo-realistic images from scene description files.

translucency, transparency, shadows, reflection, refraction, motion blur, depth of field, distortions due to different focal lengths and many other effects can be accurately rendered by these programs. Figure 3.1 illustrates a photo-realistic image rendered using the ray tracing technique.

It is also possible to enhance the realism of animations by accurately modeling the behaviour of objects over time under selected circumstances. As mentioned earlier, scientific applications can program mathematical models of the laws of physics and use the properties of the modeled objects to calculate the appropriate manipulations required from frame-to-frame. Systems capable of such behavioural modeling release the animator from the need to describe object behaviour at all. Instead the animator specifies the object's properties and circumstances and the software takes care of moving the object in a manner consistent with the given models of physical laws. Animaker has been designed to allow this type of programming to be included as clip animation.

Though realistic image rendering and physically correct motion are laudable goals, every animator involved in the entertainment industry has goals that go beyond realism. In fact, one of the acknowledged principles of animation is simple exaggeration to emphasize appearances and dramatize movement. Animaker supports both realism and exaggeration.

4.5 Design summary

Any animator who uses Animaker needs a basic understanding of animation concepts and at least a rudimentary ability to procedurally

describe the processes which occur in the script. That is, algorithms must be used to describe the repetitive processes of transforming and repositioning objects, lights and cameras. Animaker is designed to be accessible to most animators and simple sequential scripts require little if any traditional computer programming ability. On the other hand, Animaker remains extensible since the algorithms used may be as simple or complex as the animator chooses to make them. As a result, Animaker is able to handle complexities beyond the capabilities of most current direct manipulation systems.

Animaker is also simpler and more accessible than existing animation programming systems such as PHIGS. Despite its simplicity, the object-oriented structure of Animaker gives it a superior ability to organize and represent complex hierarchical structures containing similar substructures. Objects of the same class embedded within a hierarchical Animaker object may be individually configured without adding the redundancies required in PHIGS. The prevalence of bilateral symmetry in the real-world (e.g., in the animal kingdom) makes this ability a significant advantage over PHIGS in many animation application areas.

Chapter 5

The Animaker Language

In order to fully understand the Animaker language the reader must understand:

- the Animaker class hierarchy and where user-defined classes fit into this class hierarchy,
- the main features provided by Animaker's built-in classes,
- the basic syntax and semantics of the Animaker language,
- how Animaker simplifies flow control and event management,
- and other details regarding built-in constants, variables, pseudovariables and functions.

This chapter introduces each of these points, but complete details are contained in the Appendix D, the Animaker Language Reference.

5.1 Animaker's Class Hierarchy

Animaker provides several built-in class definitions and allows the animator to define new user classes as descendants of any of these classes. The inheritance hierarchy of these built-in classes is shown in Figure 5.1. Notice that some of the built-in classes are abstract and therefore no objects of these classes may be instantiated[1]. Others of the built-in classes (such as the Light and Camera classes) are useful to instantiate. In fact a default camera object is automatically instantiated by the system for each animation. In general though, it is intended that most of the participants in an animation script be instances of user defined classes descended from the built-in Actor, Heterogeneous or Homogeneous classes.

In order to foster a rudimentary understanding of each of the system defined classes a brief summary of each is provided here. A more detailed and formalized description of these classes appears in Appendix E.

---

1 That is, no instances (i.e., objects) of this class may be created. The abstract classes exist only to enhance the conceptual clarity of the class hierarchy.

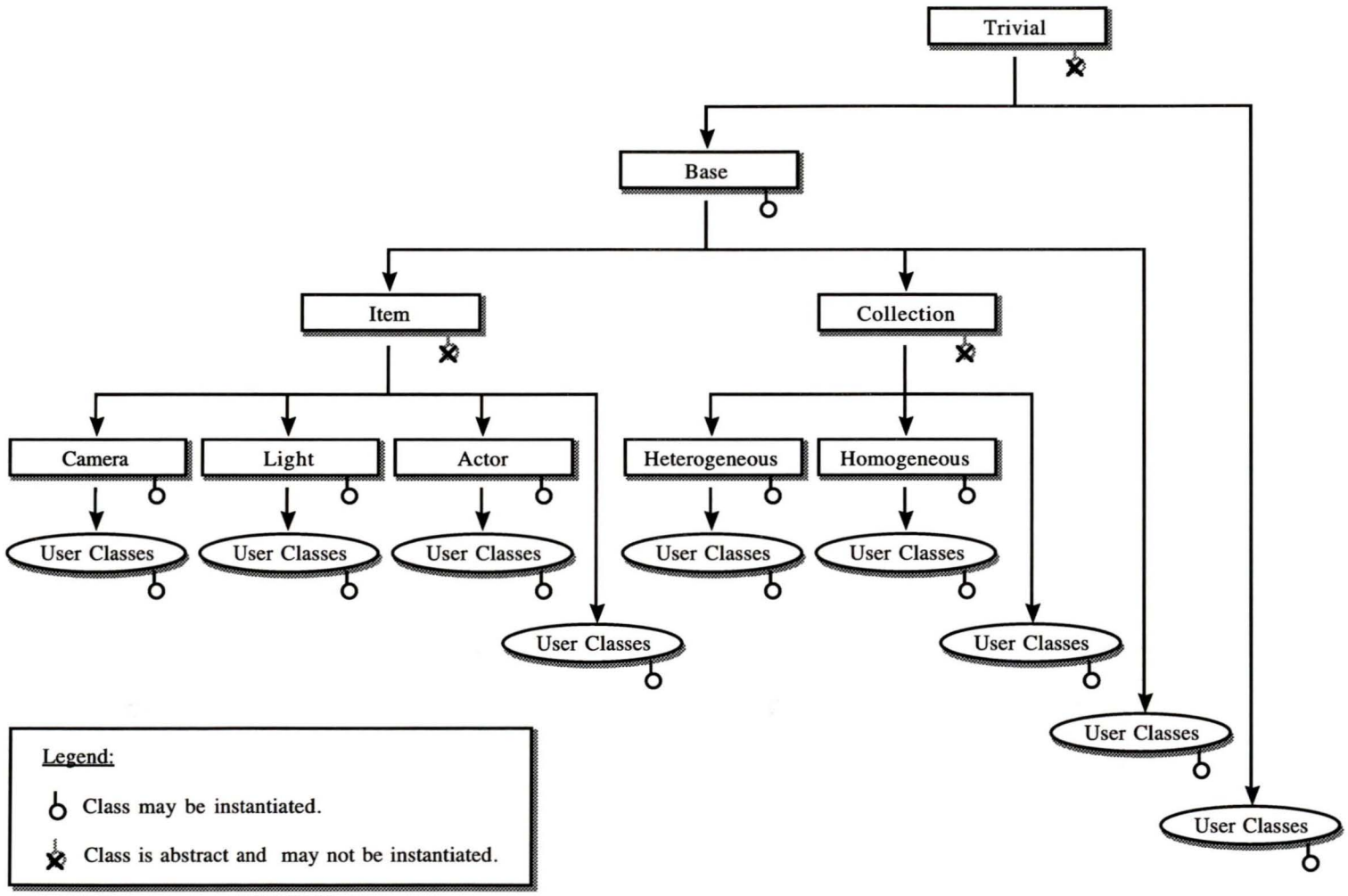

Figure 5.1 Animaker's built-in class hierarchy.

The Trivial class is the parent of all classes. It provides no instance variables and its methods take no action whatsoever. It may be used to spawn any general purpose class that does not require any Animaker features. Since any object of this class would be completely useless, this class is abstract and instantiation is prohibited.

The only system-defined descendant of the Trivial class is the Base class. The Base class is therefore the parent of all other built-in Animaker classes. The services provided by this class include support for scheduling (which is discussed in detail later in this chapter) and the ability to **cue** an object or **cut**[2] it from the animation. Cued objects are included in subsequent frame descriptions output by the interpreter; cut objects are not included. Note that objects that have been cut from the animation remain instantiated and may continue to be instructed and used as usual -- only their output is suppressed. Objects of the Base class may be instantiated but they may not be very useful[3].

The Base class has two system defined descendants: the Item and Collection classes. The Item class provides support for spacial manipulation of objects. The Collection class provides support for large groups of objects. They are intended only as conceptual parents which group their descendant classes according to these common features. Both the Item class and the

---

2 Cue and cut are methods defined for the Base class. When specific methods are named in the body of this paper they will be emboldened for emphasis.

3 It is unlikely they would be useful since there is almost nothing that objects of the Base class can do.

Collection class are abstract and may not be instantiated, however, they may be used as parents for user defined classes.

The Collection class has not been fully developed at the time of writing. Neither have its descendant classes the Heterogeneous and Homogeneous classes. It was intended that these classes would provide support for large collections of similar (Homogeneous) or dissimilar (Heterogeneous) objects. There are many applications of large collections of objects in animation, including marching armies, flocks of birds, cars on the freeway, snow flurries, bunches of leaves caught in the wind, etc. These classes are designed to support the simultaneous manipulation of many objects while also providing the ability to add some randomness into the movement of individual members of each collection. For now, the Item class, and its descendants, are the only subclasses of the Base class that are implemented.

The spacial manipulation features of the Item class are somewhat limited. One may tell any descendant of the Item class to **move** a specific distance in a specific direction or **moveto** a particular location. Item heirs may also be told to **rotate** some number of degrees along a specified axis or axes, or to **rotateto** some specific angle. An animator may also query attributes of any Item descendant such as its **location**, the **direction** it is pointing, and the direction it considers to be **up**. Note that more general manipulations are supported for descendants of the Actor class described later in this section.

All manipulations of Item offspring are performed within an infinite 3D Cartesian space defined by perpendicular X, Y and Z axes which meet at

a common origin[4]. The manipulations effectively redefine the coordinate system of the object relative to that of its parent object (i.e., the object in whose scope it was created). For example, assume that "A", "B" and "C" are objects instantiated from some Item descendant, and that "A" has created objects "B" and "C" as components (i.e., "A" is the parent object of "B" and "C"). "A" may manipulate its component objects "B" and "C" relative to its own coordinate system. When object "A" is subsequently manipulated a side effect is that "B" and "C" are similarly affected (because the entire coordinate system in which "B" and "C" are defined has been manipulated). This side effect has profound implications. It means that component objects are effectively attached to points on their parent object. They move with, rotate with and even scale with their parent objects. This is usually a very convenient behaviour for component parts to exhibit.

The Item class has parented three other system defined classes. These are the Camera, Light and Actor classes. As the reader has undoubtedly guessed, these classes are the Animaker analogues to the real world objects of the same names. It is expected that most of the objects instanced in Animaker scripts will be objects of one of these classes, or their descendant classes. The Camera, Light and Actor classes are briefly described below.

The Camera class is the only class to have a default object instanced at start up. Instances of this class may be instructed to **shoot** a frame. This results in a frame description being created (from which an image may eventually be rendered). The animator may cut the default camera from the

---

4 A brief explanation of 3D Cartesian coordinate spaces is included as an appendix.

animation at any time and substitute another, or an object of some other Camera descendant class. The animator may also have more than one camera cued at a time, or have all cameras cut from the scene. The latter implies no output of frame descriptions. The former implies that two or more complete frame descriptions are generated for each frame that is shot.

It is possible to set many camera attributes such as focal length (e.g., from wide angle through normal to telephoto), depth of field, motion blur, fading in or out, etc. but all of these are dependent upon the capabilities of the rendering program that is used to produce the images from the frame descriptions.

At least one object of the Light class (or a descendant user class) must be instantiated for any given scene, or the resulting frame image is completely black. Nevertheless, no default light is instantiated. You may tell Light objects to **turnon** or **turnoff**. You may also select the type of light source, its colour and various other attributes. However, as with camera objects, the variety of attributes you may use for lights is dependent upon the capabilities of the renderer used. Note that in many rendering programs, light sources are conceptually infinitely small point sources of light. That is, their light must strike an object in order to be seen. Pointing the camera directly at a light source may not produce any visual representation of the light source itself.

The other system defined Item descendant is the Actor class. The Actor class is intended to be the parent class for most visible participants in an animation. Effectively, Actor objects are illuminated by Light objects and photographed by Camera objects. Of course, this is all just simulated

mathematically by the Animaker interpreter in concert with the rendering program being used.

Objects of the Actor class also respond to instructions telling them to **scale** themselves up or down (i.e., in size) by some factor, or to **scaleto** some fixed size. Moving and rotation (of which all Item progeny are capable) and scaling are the three main types of manipulation used in computer animation, but the Actor class also provides the ability to do various other manipulations including the application of any arbitrary transformation matrix.[5]

Note that although the Actor class is not abstract (i.e., it may be instantiated), it has no visible structure whatsoever. As a result, the animator should normally create an Actor subclass with some visual components and instantiate that subclass rather than making an instance of the Actor class itself. Typically Actor descendants are combined together (possibly including Camera and Light objects) to create complex entities.

In order to make use of these built-in and user-defined classes, the animator must write programs in the Animaker language. That requires at least a superficial understanding of its syntax and semantics. An overview of both of these topics is given in the next section.

---

5 More precisely, the homogeneous coordinates that define the components of an Actor may be multiplied by any user-specified 4X4 matrix.

5.2 Basic Syntax and Semantics

The terms syntax and semantics have the same meanings when used to describe computer programming languages as they do when used in discussions of human languages. The syntax of a language refers to the rules that govern the construction of valid statements in that language -- that is, how can the various language elements be legally juxtaposed? The semantics of the language refers to the meaning embodied in particular language constructions such as expressions and statements. This section begins by examining the basic syntax of an Animaker program and its component parts. Semantic issues are discussed as they come up during this exposition.

The basic structure of an Animaker program is shown below:

script:
...
the end.

The executable program statements which direct the action in the animation script appear between the keywords **script** and **end** (indicated by the ellipsis above). Since most Animaker programs have user-defined class definitions, the more typical program structure is as follows

actors:
...
script:
...
the end.

User-defined classes are defined between the keywords **actors** and **script**. These class definitions may either be coded therein or read from a file when the script is translated by the interpreter (i.e., when the script is

run). Given any system or user defined class descriptions, objects may be created as instances of these classes by executing the **create** statement. For example, the following Animaker program reads in an Actor descendant class called Beagle from a file named "BeagleFile". Inside the program script, it creates two new Beagle objects called Snoopy and Spot. Once an object has been created, the animator may **tell** the object to invoke any of its methods (here the Snoopy object is told to **moveto** a new location[6]). It is also possible for an object's method to return a value (here the Spot object's **location** [7] method returns a value which is subsequently passed to the Snoopy object's **moveto** method).[8]

```
the actors:
        include "BeagleFile";
the script:
        ...
        create a Beagle named Snoopy;
        create a Beagle named Spot;
        ...
        tell Snoopy to moveto( Spot's location ).
        ...
end.
```

The observant reader may have noticed that the word **the** has appeared in a few new places in this program and it has disappeared from its location in the original. The Animaker interpreter discards articles ("**a**", "**an**", or

---

6 Recall that **moveto** is one of the methods defined for the system-defined Item class. The Actor class is a descendant of the Item class, so it inherits all of the Item class methods. Similarly, the Beagle class is an Actor descendant so it inherits all of the Actor methods (including those inherited from the Item class). The **location** method (used later in this example) is also defined in the Item class.

7 Animaker can use apostrophe-s (i.e., '**s**) or the keyword "**of**" to invoke object methods that return values. That is, the syntax: Lassie'**s location**, is semantically identical to the alternate syntax: **the location of** Lassie. Both invoke object Lassie's **location** method and return the appropriate value.

8 This would cause the local origin of the Snoopy Beagle to occupy the same point as the local origin of the Spot Beagle. Depending upon the positions of their local origins relative to the rest of their forms, and the rotation and scaling applied to each of them, the resulting image could either result in a single imbedded double Beagle or two completely distinct Beagle forms.

"**the**") so they may be freely used for clarity or readability anywhere in the program or omitted at the user's discretion. Animaker is also quite tolerant with respect to punctuation. For example, none of the colons, semicolons or periods used in the preceding program are required and all could be omitted without affecting the correctness of the program. However, some punctuation must be retained when it is required to avoid ambiguity. For example, the parentheses in the preceding program are required, as are the quotation marks around the file name.

The next example program is equivalent to the preceding example, except that the definition of the Beagle class is imbedded in the program instead of being read from a file.

```
actors
        define Beagle parent Actor
        constructor
                ...
        destructor
                ...
        end
script
        ...
        create a Beagle named Snoopy;
        create a Beagle named Spot;
        ...
        tell Snoopy to moveto( Spot's location )
        ...
        discard Snoopy
        discard Spot
end
```

Any statements that appear between the keywords **constructor** and **destructor** are executed when each new Beagle object is created. To be more precise, when each "**create**..." statement is executed, a new Beagle object is created, and these constructor statements are executed (in that object's scope). Similarly, when the object is later **discard**ed, any statements between the keywords **destructor** and **end** are executed (again, in the

object's scope). Class constructors can be used to **create** component objects and manipulate them appropriately. Destructors can subsequently **discard** the component objects. The main script in this last example illustrates the use of the **discard** command to dispose of the two Beagle objects prior to exiting.

Now let's look at a simple Animaker program that produces a single frame of output. In order to create any frame output an object of the Camera class is required. One such object, named "defaultcamera" is created (instantiated) by the system before your program begins running. You may directly **tell** this object to do anything that an Animaker Camera is capable of doing. On the other hand, you may avoid any direct interaction with the camera at all and let the system take care of this for you as has been done in the next example.

Another requirement for most Animaker scripts is an object of the Light class. Otherwise the resulting frame images may be completely black. The system does not create a default light object so the animator must normally take care of this as shown below:

```
actors
        include "BeagleFile";
script
        // The only Light
        create a Light named L1
        tell L1 to MoveTo( 0, 30, 0 )

        // The only Actor descendant
        create a Beagle named Barney
        tell Barney to MoveTo( 0, 0, 30 )

        shoot
end
```

This example introduces four new things. First of all, the lines that begin with "//" are called comments[9], and they are ignored by the interpreter (i.e., they are included for the benefit of the reader and contribute nothing to the animation itself).

The second addition is the instantiation of a Light object. Light objects are frequently instantiated directly from the system defined class. More interesting Light types can be created but their capabilities are completely dependent upon the renderer being used and it is often desirable to minimize such dependencies.

Third, this script includes some concrete examples of coordinate specification in the Animaker 3D Cartesian space. Light "L1" is moved to an initial location of (0, 30, 0). That is, 0 along the X axis; 30 along the Y axis, and 0 along the Z axis. Since the Y axis (by default) increases as you move upward, "L1" is directly above the "defaultcamera" object (which is always initially located at the origin, looking toward the positive Z axis). The Beagle Barney is initially placed at (0, 0, 30) which is directly in front of the camera, 30 points away.

Finally, this script introduces the **shoot** system function. Each time the system function **shoot** is called, it **tell**s all of the currently active Camera descendants to shoot one frame. This results in the creation of a single scene description from each camera for that frame (from its own point of view).

---

9 Animaker allows comments to be delimited according to C or C++ rules, and also permits Pascal-style comments using "(*" and "*)" delimiters.

With the basic syntax and semantic issues covered, we can now look at some of the more powerful features of the Animaker system.

5.3 Simplified Flow Control and Event Management

The Animaker language provides the standard flow control structures that are available in most programming languages:

- decision structures (**if/then** and **if/then/else**),
- counted looping structures (**for** and **repeat** n **times**),
- top and bottom tested general purpose looping structures (**while**, **repeat**, **until**, etc.), and
- procedures and functions (with full support for local variables and recursion).

In addition to these features, Animaker contains an unusual scheduling mechanism which enables the animator to simply specify different action threads within a single animation. This is analogous to a motion picture director providing independent direction to each of several actors prior to shooting a scene. Once this has been done, the director need only begin shooting and the actors take responsibility for their own actions.

The next example animation is an extension of the last example. Assume it is being shot for standard video.

```
actors
        include "BeagleFile";
script
        set the framerate to 30

        // The only Light
        create a Light named L1
        tell L1 to MoveTo( 0, 30, 0 )

        // The only Actor descendant
        create a Beagle named Barney
        tell Barney to MoveTo( 0, 0, 30 )

        // The next two lines describe all the action
        always tell Barney to Rotate( 0, 1, 0 )
        for 12 seconds shoot
end
```

This animation script produces a 12 second animation (360 frames) of the Barney object rotating very slowly (1 degree per frame) around its own vertical axis. This example introduces the Animaker scheduler.

First of all, notice the command that sets the framerate to 30. This was included because the default framerate is calculated by the interpreter based upon the capabilities of the computer being used. Since this animation is destined for videotape it is a good idea to force the interpreter to output the appropriate 30 frames per second regardless of the speed of the host computer . It was important to mention this first because the interpreter always uses the current framerate in order to interpret the meaning of expressions involving time measurement (e.g., "**for** 12 **seconds**" in this example). Ultimately all time measurement must be related back to a certain number of frames since Animaker produces frames as its output.

Now direct your attention to the last two statements in this animation. The **always** command instructs the interpreter to repeatedly issue the command that follows it, once per frame for the rest of the animation. Here, the interpreter **tell**s Barney to rotate a little bit more around the Y axis each

frame. Note that this rotation takes place at Barney's own location. That is, Barney remains at location (0, 0, 30) instead of moving in an orbit around the camera using a circular path of radius 30 as one might expect. The latter is possible too, but a different command is used for this other type of rotation.

Notice that once this **always** command is issued, the interpreter accepts this responsibility for the remainder of the animation. Multiple objects can be given their directions independently in this manner. Animaker can schedule actions:

- to occur every frame,
- to occur some fixed number of times, frames or seconds,
- to begin after a selected event takes place and continue,
- to begin immediately but stop when a selected event takes place,
- only the first time a selected event takes place,
- only when a selected event is taking place, or
- only when a selected event is not taking place.

The last statement of the current example script illustrates one of the simplest types of scheduling control -- basic repetition for a particular duration. This statement is the one which actually causes all of the frames to be generated. That is, the **for** statement here calls the **shoot** statement 360 times. Behind the scenes, the interpreter manages the Barney object, making sure it is instructed to rotate appropriately for each one of these frames.

The next example adds another Beagle and illustrates how it can be directed independently of the first Beagle. This example also illustrates a different way that the scheduler can manage change over time.

```
actors
        include "BeagleFile";
script
        set the framerate to 30

        // The only Light
        create a Light named L1
        tell L1 to MoveTo( 0, 30, 0 )

        // Create our cast...
        create a Beagle named Barney
        tell Barney to MoveTo( 0, 0, 30 )
        create a Beagle named Fred
        tell Barney to MoveTo( -10, 0, 30 )

        // The next three lines describe all the action
        always tell Barney to Rotate( 0, 1, 0 )
        tell Fred to MoveTo( -10, 0, Z_Coordinate )
                varying the Z_Coordinate smoothly
                from 30 to 1000 over 20 seconds
        for 12 seconds shoot
end
```

This animation script produces another 12 second animation. Barney rotates slowly as before. The Fred object starts out at location (-10, 0, 30) beside the Barney object, but accelerates rapidly away from the camera as the animation continues.

Here the interpreter is instructed to repeatedly **tell** Fred to **MoveTo** a new location each frame over a span of 20 seconds. However, the location that Fred is given changes each frame according to the value of the "Z_Coordinate" variable. "**varying the** Z_Coordinate **smoothly**" tells the interpreter to use a sine function to accelerate at the start of the time span and decelerate as it approaches the other end of the span. If the word **evenly** had been coded instead of **smoothly** a constant rate of change would have been used instead.

Notice that the Fred object is only be part way to its final destination when shooting stops after 12 seconds. At this point Barney could possibly

be given new instructions and shooting could resume. Of course, if the script were to end here without shooting any more frames, the animation would contain only 12 seconds (or 360 frames). Notice also that the frame rate can be easily changed. Regardless of the frame rate, Fred always moves through the same distance in a given amount of animation time. The Barney object however is rotating a fixed amount per frame (e.g., a lower frame rate implies fewer frames in the animation which implies less rotation by the end of the animation). It is a good idea to avoid frame rate dependencies since different frame rates are useful. For example, low frame rates can be used to save rendering time in test animations.

The preceding two example programs illustrate some key Animaker strengths.

First, each object may be given directions independently, before the camera begins to roll. In contrast, using traditional flow control structures portions of the animation would be wrapped up in a large loop which would execute once per frame. Within that loop would be instructions directing each participating object, and a clear understanding of flow control would be required. To avoid this, Animaker animations use smaller, separate sets of directions for each participant. This makes complex animations more accessible to novice programmers. Each participant's directions may be reworked independently without concern for the others.

Secondly, Animaker scheduler structures usually eliminate the need for looping structures altogether (by creating implied looping managed by the interpreter). The result is that Animaker animations tend to contain

simpler instructions that accomplish more than traditional programming language statements.

Finally, Animaker's object orientation enables programmers to create libraries of intelligent Actor class descendants that are capable of responding to a wide range of commands. Any non-programmer can create instances of objects from these intelligent classes, direct them according to their own ideas, and produce complex animations with little effort. Even relatively novice programmers can create descendent classes from these classes and add to, or modify their behaviours as desired.

5.4 Miscellaneous Details

Wherever possible, Animaker statements have been made syntactically similar to common English language command sentences. This was done in order to make Animaker more accessible to non-programmers. Several simple language devices are used to support this natural language style and many natural language constructions are possible.

For example, an authoritative English person can give commands like: "walk", "move the table", "squash the bug", or "put the cookie into the jar". The implied subject in these command sentences is the listener, so the command typically begins with a verb. This is often followed by an object and other information. There are a few exceptions but Animaker commands usually begin with a verb. They generally take the form "verb object" (possibly followed by other information as well). The implied listener here

is the Animaker interpreter itself (or an object instance, as you have already seen).

A simple pronoun device is also used for clarity in Animaker. The object pronouns "**him**", "**her**", "**them**", "**it**", "**this**" and "**that**" are all synonyms that refer to the most recently returned value.

Data such as numbers, character strings, logical values, object instances, etcetera may be stored in generic containers called variables. Animaker variables are normally dynamically created and their types are set at assignment time. Facilities are also provided for formal declaration of local variables, global variables, and both variable and value parameters. The fact that variables need not be formally declared, and that their types may change dynamically, helps to keep Animaker programs short and simple[10].

Variables may either contain a single data item or a collection of data items called an array. Each item in an array of items may itself be a single data item or an array, and so on. The various items in any array may be the same or completely different. For example, the first element of an array might be a singular data item while the second was an array. Animaker variables may dynamically change in type or quantity whenever desired at runtime. That is, the third element of an array might initially contain a single number; later it could be assigned a character string, and still later it could hold an array with hundreds of elements. This permits Animaker to

---

10 The obvious trade-off here is that this lack of strong typing also prevents the interpreter from catching many types of logic errors at compile time.

handle dynamic memory allocation without the use of complex, recursively defined data structures such as linked lists.

Expressions are constructed in Animaker as they are in most other programming languages and the precedence rules are modeled after those of the C language. See Appendix D, the Animaker Language Reference, for details.

The Animaker commands that store data into variables come in two flavours. The first is a set of English-like commands:

```
put Herbie's EngineSpeed into  h
put myEngineSpeed + 2 into myEngineSpeed
set theAngle to 6
get
```

For the benefit of experienced programmers, the complete set of C language assignment operators is also provided with the exception of the "=" operator. For simple assignment you must use the Pascal (i.e., ":=") assignment operator. Here are some examples[11]:

```
h := Herbie's EngineSpeed
myEngineSpeed += 2
angle2 %= 6
i++
--j
```

---

11 Experienced C programmers should note that Animaker prohibits side effects. Any C expression which results in assignment is a *statement* in Animaker, and has no value (hence it may not be used as an expression). This means, for example, that the C preincrement and postincrement operators are identical in function within Animaker.

As you have seen in the example programs, Animaker provides several mechanisms to facilitate the conditional and timely execution of commands. Each of these mechanisms uses English conditional or temporal terminology (e.g., "**at**", "**every**", "**when**", "**once**", "**if**", "**otherwise**", "**until**", "**while**", "**always**"). Timely execution in particular is often based upon the occurrence of a particular event. The most basic Animaker events are temporal events such as the arrival of the animation at a particular frame number or time, or the passage of some specified number of frames or seconds from a given reference point. Preceding examples have illustrated some of these. However, an Animaker event may also be a character string that has been emitted by an object. That is, any object may emit an event string by means of the **signal** or **announce** commands. Once emitted an event string is made available to every object that is interested. When waiting for an event, an object may also indicate that it is only interested in events originating from a particular other object.

Although this chapter has covered the most important features of the Animaker language, one chapter cannot fully describe any large programming language. With well over 150 keywords and almost as many system defined constants, variables, functions, etcetera, Animaker has become a large language.

The next chapter covers the design and implementation of the Animaker Interpreter itself. An understanding of the interpreter is useful in understanding the Animaker language and in the design of Animaker scripts.

If you want more information about the Animaker language itself, there are three additional sources of information. First of all there are

several sample Animaker programs available. Some of these are included in Appendix F. Many additional language details may also be found in Appendix D, the Animaker Language Reference. Some detailed reference information is also available in the online Animaker Language Reference (a HyperCard stack).

Chapter 6

The Interpreter System

This chapter introduces the Animaker interpreter. It begins by showing the interpreter in the context of a complete animation system. This is followed by descriptions of the main interpreter components and their design goals. Whenever they are important to these discussions, the key implementation issues are identified.

The Animaker interpreter is a tool that forms part of an animation system. The Animaker program contains this interpreter plus other tools that complete a suite of programs suitable for the production of 3D modeled animations. Figure 6.1 shows a simplified block diagram of a complete

system indicating where each of the Animaker tools fits in the animation production pipeline and illustrating the flow of data throughout the system.

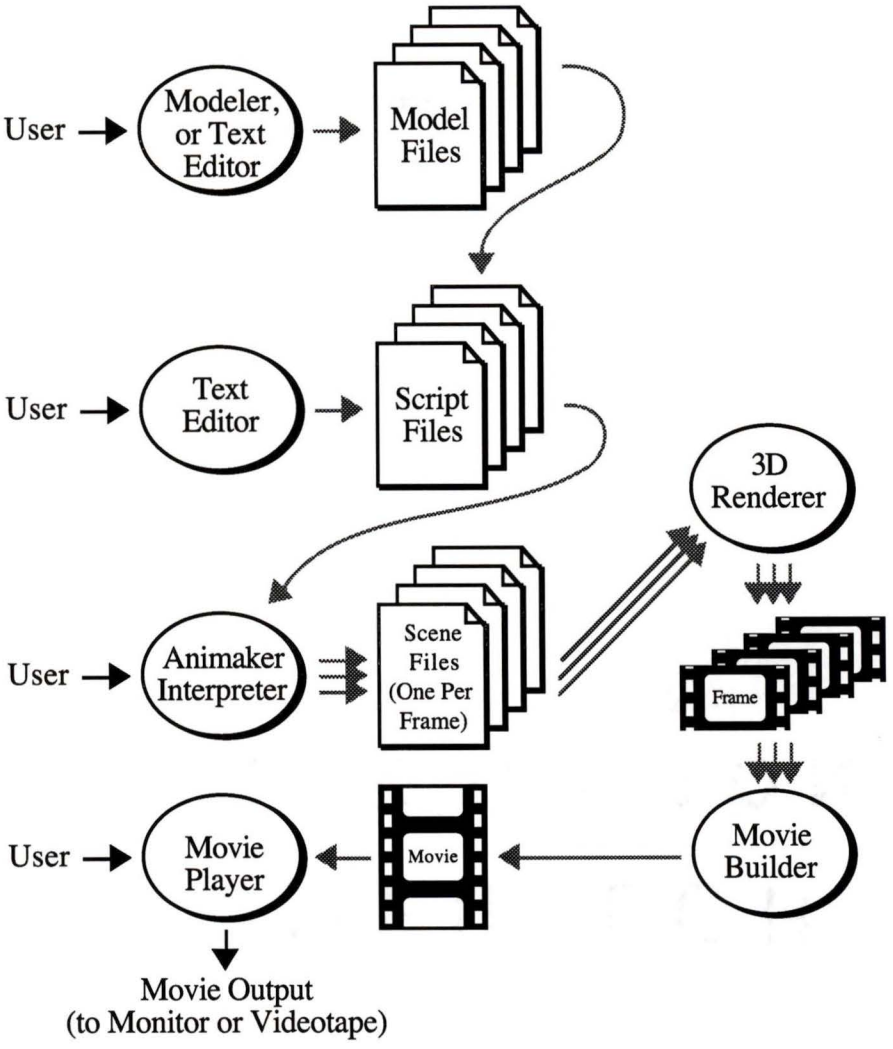

Figure 6.1 The Animaker interpreter in context with other system components.

A user of this system begins by creating some basic component model files (either visually with a 3D Modeler program or manually using a text editor). Animaker scripts must also be written (using a text editor) to describe the animation. The component model files may be referenced within these Animaker script files. These script files may also reference

each other. Once these models and scripts are ready, they may be processed by the Animaker interpreter to create the resulting animation.

When the Animaker interpreter reads the main animation script, all referenced script and model files are included as appropriate. The interpreter executes the scripts and produces models (i.e., scene description files) for each frame in the animation. Each model is delivered to a rendering program which produces a frame image corresponding to that model. The resulting frames are subsequently collected and built into a movie that may be viewed on the computer monitor or recorded to video tape by means of a movie player program.

Notice that the complete Animaker-based animation system is designed to make use of as many existing tools and data formats as possible. 3D models are stored as text using standard scene description languages.[1] Animaker scripts are also stored as simple text files. Individual movie frames are stored in the standard Apple Macintosh "PICT" format.[2] Movies are stored in the Apple QuickTime format.[3] As a result, a wide variety of existing text editors, 3D modeling programs, 3D rendering programs, and QuickTime movie player/editor programs may be used as components of this system. In fact, the only program that required custom design and implementation to complete this system was the Animaker Interpreter itself.

---

1 At the time of writing, the Animaker Interpreter supports Pixar's RIB (i.e., RenderMan Interface Bytestream) format as well as NFF (Neutral File Format), and POV (Persistence of Vision) scene description languages. All of these formats are supported for both input, and output models.

2 Readily available tools can convert this format to GIF, JPEG or other platform independent graphics formats when desired.

3 The QuickTime movie format is used on both Apple Macintosh and MicroSoft Windows systems.

6.1 The Basic Design of the Interpreter

The basic function of the Animaker interpreter is to read Animaker script files, parse (i.e., syntactically analyze) them, and follow the instructions therein (i.e. actually execute the script). The process of execution generates scene descriptions for each frame as a by-product. Of course, wherever possible, syntax and semantic errors must be detected by the interpreter and brought to the attention of the user.

The design of the Animaker interpreter itself began with a grammatical analysis to determine what type of parsing algorithm would be appropriate. This process culminated in the creation of a complete, formal grammatical definition of the Animaker language. This formal definition is given in Backus Naur Form (BNF) in Appendix C.

The formal grammar was designed to be amenable to a simple right (i.e., bottom-up) parsing algorithm. To verify this, the Animaker grammar was submitted to the standard UNIX "yacc" (Yet Another Compiler Compiler) parser generator.[4] When given an appropriate grammar, yacc automatically detects any shift-reduce or reduce-reduce conflicts and produces the state tables for an LR (i.e., Left-to-right scanning and deterministic Right parsing) parser.[5] The Animaker grammar contains no

---

4 When a language has been described in BNF the process of submitting it to yacc is straightforward since the yacc input language is very similar to BNF.

5 The yacc source code for this grammar can be easily constructed from the BNF grammar given in Appendix C.

conflicts so yacc automatically produces the appropriate state tables without generating any error messages.

Since the Animaker grammar causes yacc to produce appropriate output, the construction of both lower levels of the interpreter can be fully automated. That is, the scanner, or lexical analyzer for Animaker can be automatically built with the standard UNIX "lex" utility and yacc can produce the parser. Given appropriate input files describing the language tokens (e.g., the punctuation and keywords used in the language) and the grammar of the language, these two utilities produce as their output, a C language module that is capable of parsing Animaker scripts and detecting syntactic errors.

To enable the best possible error detection, the Animaker interpreter incorporates this parsing module in a standard two pass process. In the first pass, the interpreter first reads the entire script, and analyzes it syntactically and semantically. Because this step parses all portions of the script, it gives the Animaker interpreter the ability to detect some types of problems even in portions of the code which have never been executed.[6]

As the interpreter makes its first pass it constructs a "parse tree" for use in the second pass. A parse tree is an internal data structure which completely and unambiguously describes an entire script. The second pass,

---

6 A weakness typical of interpreters (versus compilers) is that they often parse as they execute. As a result, rarely used portions of a program may harbour dormant syntax errors that are difficult to detect without exercising every statement in the program. Like a compiler, the Animaker interpreter parses all parts of a program prior to execution so it does not have this weakness.

or execution phase of the interpretation only needs to traverse this parse tree (i.e., the script itself does not need to be read again).

The portion of the Animaker interpreter that implements the execution phase is called the Execution Engine. The Execution Engine is the most complex portion of the interpreter. It consists of several large modules including:

- the statement processor,
- the expression evaluator,
- the symbol table,
- the runtime memory manager,
- a module that contains the code for built-in constants, variables, classes, and functions,
- the scheduler, and
- the frame output module

The statement processor is the portion which traverses the upper levels of the parse tree and takes appropriate actions down to and including the level of individual statement nodes. The expression evaluator traverses and evaluates expression[7] subtrees. The symbol table manages the database of names created at runtime (e.g., the names for procedures, constants,

---

7 Some examples of expressions are: 3 + x - 0.72 or "Hi " + "mom" or a < b and c > d
Expressions consist mainly of combinations of literals (like 3, 0.72, "Hi", and "mom"), variables (like x, a, b, c, and d) and operators (like +, -, <, >, and 'and') but also include parentheses, function calls, etc.

variables, classes, etc.). It also controls the scope[8] of these names. The symbol table makes use of the runtime memory manager to allocate space to variables (including object instances) as needed during execution.

The module that contains the code for all built-in support is really external to the interpreter . It could have easily been written in the Animaker language and included as some sort of library at runtime. For efficiency this code is actually written in C and forms part of the interpreter. Among other things, this module contains all of the matrix manipulation utilities that allow models to be rotated, scaled and otherwise manipulated in the virtual 3D universe of Animaker. Although this module may be the most complex portion of the interpreter, it is relatively uninteresting since it consists mainly of well known implementations of these mathematical functions.

All of the modules described so far are relatively standard components of an interpreter for any language with features similar to those that Animaker provides. The last two component modules (the scheduler, and the frame output module) are unique to the Animaker interpreter. Each of these modules is described in a separate section below.

---

8 Scope is a technical term that refers to the accessibility and duration of names in a programming language. For example, in Animaker (and many other languages such as Pascal and C) local variables within procedures are created on the runtime stack when the procedure is invoked. These local variables are accessible only within the scope of that procedure, and are destroyed when the procedure returns. In this interpreter, the symbol table module manages this.

6.2 The Scheduler

The scheduler module provides mechanisms for controlling the execution of a script over time. When an animator wants to have some process take place starting at a particular time and continuing over some span of time, the scheduler can facilitate this. As discussed in Section 5.3, the scheduler is capable of many different types of scheduling control. All of these forms of scheduling are handled in a similar manner internally.

Syntactically, when the statement processor encounters a scheduling request it enters an appointment into the object's timetable. All of the relevant information about this appointment is saved for later use (e.g., what to do, when to start, when to stop, duration, whether smoothing is to be performed, etc.). No other actions are taken at this time. Semantically, the object has been directed to do something according to schedule and conceptually an appointment for this has been registered; execution continues and at the appropriate time the scheduled action is taken.

This independence of direction and action may seem awkward, but it allows the animator to describe the part that each actor is to play, independently of the other actors in the animation. The part of any one actor may be easily modified later without affecting the others. This is analogous to the widely accepted *information hiding* principle[9] in software engineering.

---

9 The information hiding principle advocates program module independence as a vehicle to facilitate change in software systems. By keeping private any information which is not required externally, the internal workings of a software module may be modified without affecting any of the other modules of the program. Conversely, when the internal design of a module is accessible to the programmers who work on other parts of the program, they may write their modules in ways which depend upon that particular internal structure. As a result, if the first module is redesigned, the other modules may no

Simply put, independence facilitates changes that may be required in the future.

Of course, when animating, one often wants the actions of two or more actors to be coordinated. For this reason, the scheduler also provides synchronization mechanisms based upon events.[10] The shooting of each frame, and the passage of time from one second to the next are events which can be used for the synchronization of any number of actions by any number of actors. For example, a moving target and a projectile could be scheduled to arrive at the same coordinates at the same time and at that same instant an explosion could be scheduled to appear at the same location.

The Animaker scheduler also allows the animator to **signal** customized events. Individual objects may announce events as well. The built-in **signal** procedure allows the animator to announce that something[11] has occurred. Actions may be scheduled to occur when such an event occurs. In addition, any object may **say** that something[12] has occurred. Again, actions may be scheduled to occur when that event occurs, but they may also be restricted to occur only when a particular object **says** that the event has occurred.

---

longer work correctly. By hiding private information, module independence is enhanced, and it becomes easier to make changes to each module.

10 You will recall that the better direct manipulation computer animation systems provide the ability to direct movements from one key event to another. The event mechanism provided by the Animaker scheduler module is analogous to using key events in a direct manipulation system, but it is more powerful as you will see.

11 An arbitrary character string is used for signaling. That is, the animator may describe an event with a character string, then signal that event with that character string. Actions could then be scheduled to occur whenever an event named by that character string occurred.

12 Again, any character string may be used to indicate what has occurred.

Event announcement abilities give the Animaker system significant strengths over direct manipulation systems. In effect, they allow Animaker objects to be somewhat intelligent. For example, one Animaker object can be told to do something, and to announce an event when that action is finished. Another Animaker object can be told to watch for this event to happen, and spring into action when it occurs. When the animator decides to make the first object's action take more or less time, the second object's actions independently remain synchronized with the completion of the first action. In most existing animation systems the animator would have to go back and change the second object's behaviour to keep it synchronized with the completion of the first object's actions.

The implementation of the scheduler relies upon the implementation of the Camera class. Each time a camera is instructed to **shoot** a frame, the interpreter invokes the **smile** method of each active object. If the object has any appointments scheduled the **smile** method asks the scheduler to check to see if any of its actions need to be taken prior to the shooting of this frame. Once every active object indicates that it is ready for the frame to be shot (technically, once every object's **smile** method returns **true** -- and it is called repeatedly until this occurs) frame output begins.

6.3 Frame Output

The frame output module is responsible for implementing the exposure of each frame using our virtual camera(s). As mentioned earlier, each active camera produces one scene description as output for each frame that is shot (i.e., by invoking the **shoot** procedure). That is, the frame output

is a scene description incorporating all of the active objects (actors, lights, etc.) moved, rotated, scaled and otherwise manipulated as appropriate for the current frame.

The scene description language that is used for input and output is controlled by the setting of the **_renderer** variable in the Animaker script. Note that the frame output module is the only portion of the interpreter that actually needs to understand the scene description language that is being used to communicate with the 3D renderer. The most important ramification of this is that incoming model descriptions need not be parsed. In fact, in the current implementation, they are simply read as text inclusions. This dramatically simplifies the portion of the interpreter that handles model input. It also necessitates some manual intervention after using a modeling program on an input model file. Any superfluous objects added by the modeler (typically a camera and one or more light sources) must be manually edited from the text of the model file prior to its inclusion in an Animaker script. To avoid this manual editing step, it would be desirable for the model input portion of the interpreter to understand the scene description language -- but this is not required.

The frame output module is necessarily complex. Depending upon which scene description language is being used for output, this module must construct a scene description template appropriate for that language. The template is filled in with the objects that are active in the current frame. Each language has different syntactic requirements with respect to camera, light source and other object descriptions and where they must be placed in the scene description file. Each language also has different ways of describing model transformations such as scaling, rotation and movement.

Some use right handed coordinate systems, some use left handed systems, and some use both.[13] All of these considerations must be taken into account, in order to present a consistent Animaker programming environment regardless of the output language being used. Once all of these concerns are addressed, appropriate output can be prepared for each frame, and written into a uniquely named file.[14]

In order to automate the rendering process, the frame output module of the current implementation does more than simply output scene description files. Once a frame is completed, and written to its output file, the frame output module makes contact with another program called the Conductor, and asks it to manage the rendering of this frame.

6.4 The Conductor, Bureaucrats, and Renderers

The Conductor takes the place of the 3D renderer in the pipeline diagram of Figure 6.1. It does the work of a 3D renderer, but it is capable of using parallel distributed processing to achieve dramatic improvements in

---

13 A left handed 3D coordinate system is one where the Y axis increases upward, the X axis increases to the right of the observer, and the Z axis increases away from the observer. It is called left handed because if you hold your left hand in front of you with the thumb up, the index finger pointing forward, and the middle finger pointing right you have a model of this system. A right handed coordinate system is the same, except the X axis increases to the left. As an example, consider a left handed coordinate system mapped onto this page. The origin is at the bottom left and the X axis runs along the bottom of the page, increasing to the right. The positive Y axis runs up the left edge of the page, and the positive Z axis increases as it grows away from you on the other side of the page.

14 Unique names are formed by appending the current frame number to the selected output file prefix (by default, "Frame"), then the appropriate file type suffix for this renderer is appended. For example, RenderMan renderers usually expect their input files to have the ".RIB" suffix.

rendering time using a network of computers to produce animations containing many complex frames.

The Conductor program can either locally render a frame image, or it can distribute the rendering of the image to other computers. To be more precise, the Conductor can transfer the scene description file to another program called a Bureaucrat, either locally on the same computer or remotely on another computer over a network. When the Bureaucrat program receives the scene description file, it instructs a local rendering program on its computer (i.e., perhaps not the one where the Animaker interpreter is running) to render the image. When the image has been rendered, the Bureaucrat program transfers the image file back to the original Conductor program, and signals that its computer is idle and ready for another scene description file.

In a networked computer environment, it is possible to have many copies of the Bureaucrat program running on many different computers at the same time. This effectively creates a parallel processing computer with a processing capability that increases directly with the number of computers available. In this environment, the Conductor remains in contact with each Bureaucrat program, and with the Animaker interpreter. When the interpreter gives the Conductor a scene description file, it immediately dispatches it to an idle Bureaucrat (if one is available). When the next scene description file is received, it is sent to another idle Bureaucrat, and so on. Figure 6.2 illustrates the flow of information in this environment.

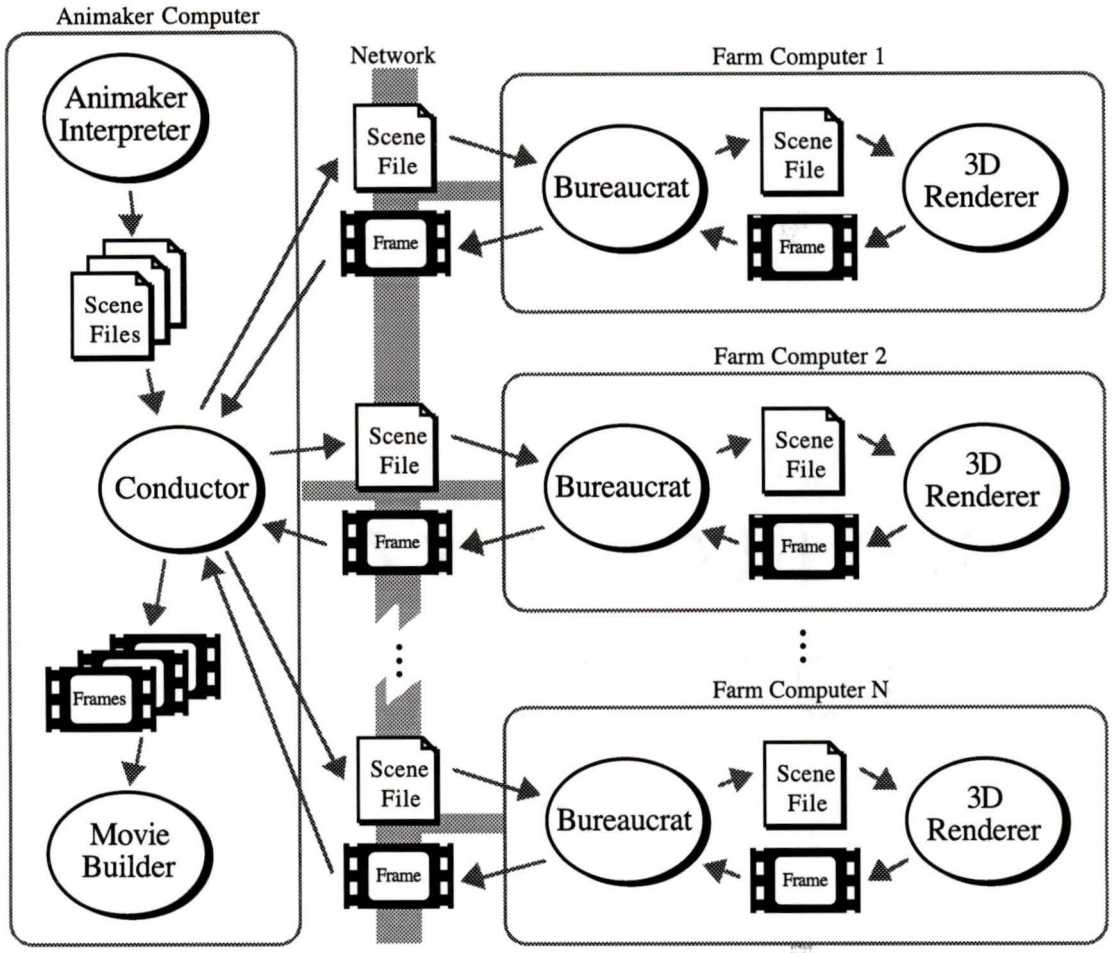

Figure 6.2 Data flow in an Animaker renderer farm.

Scene description files take very little time for the Animaker interpreter to produce (typically a few seconds to a few minutes). Both scene description files and image files take very little time to transfer over a network (typically only a few seconds). The rendering process on the other hand is extremely slow (often taking several hours to complete a single frame image) so there is an excellent opportunity to exploit parallel processing here. Ideally in an Animaker system, a farm of rendering computers would be connected over a network to work in parallel on the rendering of many frame images at once. Consider for example a typical animation whose scene description files each take only a few minutes to

write out, but which require about an hour each to render. In a single computer environment, a one-second (30 frame) animation would take over 30 hours to produce. Add a rendering farm with 10 computers of the same processing speed and the same animation could be produced in about 3 hours.

With large numbers of small networked computers sitting idle overnight in private offices on many sites, there is ample opportunity to create late night rendering farms. Note also that it is not important that the computers have similar processing capabilities. The Conductor uses a simple algorithm to dispatch models to the first idle Bureaucrat. In the long run faster machines tend to get more work, and slower ones get less work. The current implementation has been successfully used on a small rendering farm with 6 Macintosh 68030 based computers connected over a LocalTalk network.

Chapter 7

Conclusion

This thesis presents an animation programming language and an interpreter for that language. Together they constitute a complete computerized 3D modeled animation system. The Animaker system is designed as a compromise between the existing, powerful, but difficult to use, animation language systems and the less powerful, but easy-to-use, direct manipulation systems.

This chapter begins with a summary of results. The summary is followed by a critical analysis of these results. Finally, the chapter

concludes with some suggestions for improvements and some recommended directions for future research.

7.1 Summary

The author has implemented a demonstration version of the Animaker system described in this thesis. To facilitate the implementation, some minor compromises and omissions were required. On the other hand, most of the system, including all of the special control structures and other language features, are fully implemented. The completed implementation functions as a convincing "proof of concept."

As described in Chapter 6, the Animaker interpreter program communicates with the Conductor program which in turn communicates with Bureaucrat programs that manage the rendering. Screen snapshots of each of these programs are shown in Figures 7.1 through 7.3.

The user interface in a Bureaucrat program is underwhelming since users do not normally interact with it directly at all. That is, a Bureaucrat is simply a drone that accepts orders dispatched to it by the Conductor program over the network. Each Bureaucrat supports a nearly minimal set of menu commands, and provides a single status window. As the Bureaucrat operates, progress or problem reports are displayed in this window. Figure 7.1 shows an example of a Bureaucrat program running.

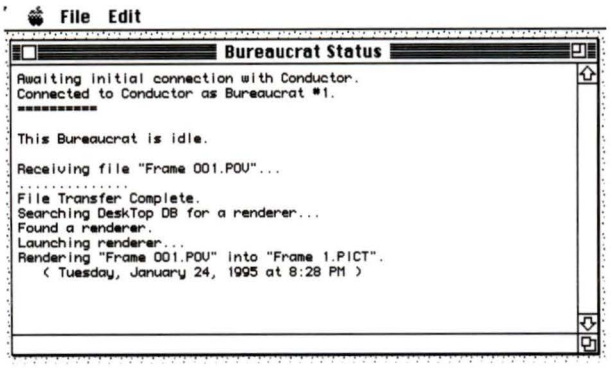

Figure 7.1 An example of a Bureaucrat program in use.

The Conductor program also has a minimal user interface since it is primarily driven by the Animaker interpreter program rather than by direct user interaction. The Conductor may either automatically search the network for all Bureaucrats it can find, or the user may manually select a set of Bureaucrats using a dialogue box.. Like the Bureaucrats, the Conductor provides a single status window. As the Conductor operates, progress or problems reports are shown in this window. Figure 7.2 shows what the Conductor program looks like when it is running.

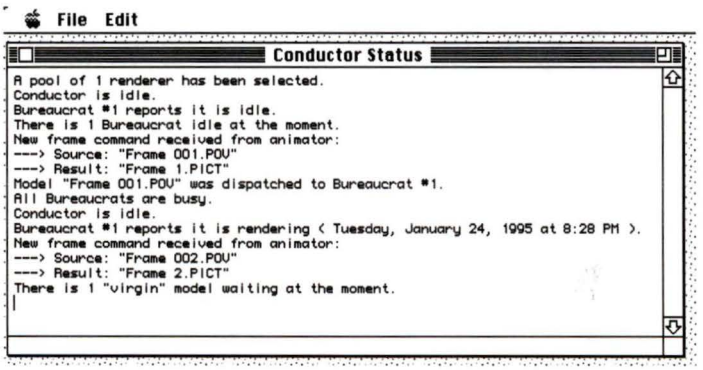

Figure 7.2 The Conductor program in use.

The program hosting the Animaker interpreter provides a much richer interface since it is the focus of most of the user interaction in this system. The features of the Animaker interpreter program include:

- a program editor that colour codes the different syntax elements (e.g., keywords, variable names, etc.);
- separate windows for program output, runtime status, etc.;
- detailed compile-time and run-time error reporting, and automatic error location in the source window;
- an image window with the ability to display single frame images,
- a virtual VCR with the ability to play, pause, and single frame the resulting movies forward or in reverse;
- the virtual VCR is also capable of digitally editing movies (using standard cut and paste operations).

Figure 7.3 shows the Animaker interpreter program running with several of its windows open.

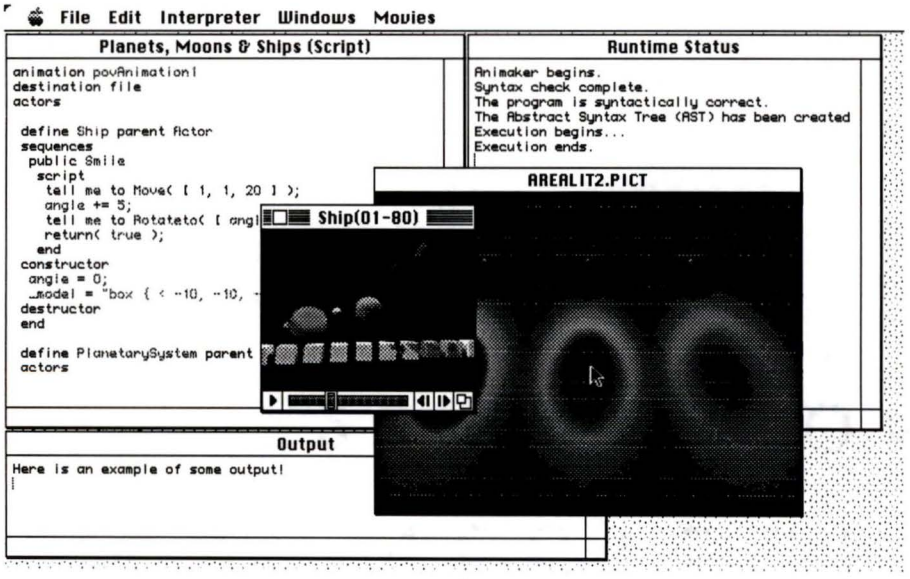

Figure 7.3 The Animaker interpreter program.

The most important omissions to the general design are three of the system defined classes. The Collection class, and its two descendants: the Homogeneous and Heterogeneous classes. These classes are designed to support large collections of similar and dissimilar objects (respectively). That is, these support routines were to introduce certain efficiencies when dealing with large numbers of objects. Note though that large collections of objects may still be used in this system as it is implemented. For example, once a bird class is created, one can easily create an array of a few thousand bird objects if desired. Many of the rendering programs now available have built-in features to improve rendering efficiency under these conditions and these three Animaker classes were designed to allow the interpreter to take advantage of these renderer features.

The other omission worthy of note is the incoming model parser. As noted in Chapter 6, models are read in purely as text. If the interpreter parsed the models it would be able to detect problems. It would also be able to do more interesting transformations of the models (such as changing the base form or texture of the object).

One of the compromises made during implementation was the decision to target the implementation to the Apple Macintosh. Most of the interpreter code is completely platform independent but the rest of the system has inherent dependencies upon this platform. The universal file formats, the built-in high level support for networking, and the collection of existing utilities available on the Macintosh made it the ideal first host for the Animaker system. The interpreter could be quickly and easily ported to any system with a C compiler. Porting the entire system (especially the networking portions) would require much more coding.

Another compromise that was required during implementation was imposed externally. The author was unable to acquire a rendering program that supported either the RIB or NFF scene description languages. As a result, output in those two languages has only been manually verified. The only renderer available during development was the POV-Ray system. Unfortunately the Macintosh implementation of POV-Ray has a severe limitation on the size of its input files which makes it impossible to render complex images in this system.

Several relatively simple animations were created within this system. These test animations, and over one hundred small test programs were used to test the implementation of the interpreter and to refine the design of the language.

7.2 Critique

There are some problems with the Animaker system both in terms of its design and in terms of the way it has been implemented. The design problems are all directly related to the emphasis placed upon accommodating novice programmers. The implementation problems have to do with the periphery of the system, not with the language or the interpreter.

When designing a programming language there are many places where binary decisions must be made. Languages which are designed in one way have certain strengths and languages designed with the opposite philosophy have other strengths. The main problems with the Animaker language are attributable to design decisions which garnered certain

advantages at the cost of others. For example, Animaker is virtually a type-free language in the sense that variables may be assigned values of any type. Type-free languages liberate the programmer from having to comprehend the notion of type with respect to variables. On the other hand, it is generally acknowledged in computer science that the use of strongly typed languages tends to help reduce programming errors.

Some other problems with the Animaker language include the fact that it provides no true record or structure variables. Instead, records are treated as arrays with named subscripts. This is awkward both syntactically and semantically. Similarly pointer variables, dynamic memory allocation, and other scope options would improve the utility of the language -- especially for experienced programmers.

The problems with the Animaker system implementation center around omissions and limitations built into the system. The most significant of these is the hard limit on file sizes in the renderer. The Animaker program editor also has such a limitation. Another problem is that the Animaker interpreter provides no support for separate compilation, although the language itself was designed to facilitate this through the **include** mechanism.

When creating test animations it became apparent that a much faster, perhaps lower quality, rendering process would be very useful for test animations. This is exactly analogous to the "line tests" of the traditional animation process (see Appendix A). For example, even if the final work would have to be ray-traced, a lot of the preliminary work could be tested using only low quality wire-frame renderings. This would save significant

rendering time. Most of the test animations the author created had to be left running overnight on a rendering farm of two or more machines. In some cases, the results that were discovered the next morning were poor because of a minor error in the script. With a faster test animation tool, these bugs could be fixed right away, and the final animation could be left for overnight.

Finally, note that current hardware has difficulty in retrieving full colour, full screen movies from a hard disk in real-time without dropping frames. As a result, some image size and/or frame rate restrictions do apply when playing movies in this system unless you are using a very high end system.

7.3 Suggestions For Future Work

There are many ways that the Animaker language and the Animaker system could be improved. First of all, artificial limits should be removed, particularly the highly restrictive renderer file size limit. This would enable the system to produce commercial quality animations (although recording them onto videotape may still be difficult). Secondly, the interpreter should be able to parse incoming models as discussed in Chapter 6. Third, a faster rendering tool is needed for test animations. Each of these changes would require significant programming effort, but each of these is a relatively small task in comparison with the other suggestions that follow.

The Animaker interpreter could also be enhanced to accept more than one type of language (while retaining the underlying utility). That is, a

strongly typed language with other features aimed at experienced programmers could be designed around the Animaker features. The interpreter would handle the parsing phase differently for each language, but the runtime engine would be able to execute either parse tree with little modification. The main motivation for this suggestion is the fact that the author (an experienced programmer) found the novice-oriented features of the existing Animaker language to be cumbersome and obtrusive when programming test animations.

Of course, full support for the system-defined Collection class and its descendants could also be added. In order to do this well, a detailed study of recent research into particle animation should be conducted before implementing these classes. Particle animation and the variants which include flocks of birds (etc.) are popular research topics so it should be easy to find information about this.

System defined functions should be added to support collision detection in all primitive types. Collision detection is another popular research topic. This is a complex problem especially when the objects are concave, convex, or flexible. This addition to the language would undoubtedly be very challenging.

It would also be desirable to somehow exploit the significant amount of image coherence that exists from one frame to the next in most animations. This is a difficult problem when the rendering algorithm is photo-realistic. A slight change in the position or orientation of one element of the scene can significantly alter the entire frame image. The Animaker system addresses this issue after rendering is complete. That is, when the

individual frame images are composed into a movie, compression techniques can be applied. Many of these compression algorithms take advantage of the similarities between frames in order to reduce movie size (which surprisingly can also reduce the amount of processor time required to redraw the frames).

7.4 Animaker as a Vehicle

The Animaker language and system provide a powerful platform upon which these last two research ideas (i.e., particle animation and collision detection) can be explored. It has significant advantages over traditional animation programming environments in this regard. The competing direct manipulation systems are not intended to be programming environments so they are not really suitable for experimentation in these areas.

References

[ANSI 85] American National Standards Institute, "Computer Graphics -- Graphical Kernel System (GKS) Functional Description," X3.124-1985, ANSI, New York, 1985.

[ANSI 88] American National Standards Institute, "Computer Graphics -- Programmer's Hierarchical Interactive Graphics System (PHIGS) Functional Description," X3H3/89-54, September 26, 1988.

[ANSI 89] American National Standards Institute, "Computer Graphics -- Programmer's Hierarchical Interactive Graphics System (PHIGS) Part 4 -- Plus Lumiere und Surfaces (PHIGS+)," X3H31-89-05, July 25, 1989.

[Barrett 79] S.Barrett, R.Bates, D.Gustafson and J.Couch, *Compiler Construction*, Scientific Research Associates, Toronto, 1979.

[Foley 90] J. Foley, A. van Dam, S. Feiner and J. Hughes, *Second Edition Computer Graphics Principles and Practice*, Addison-Wesley, Reading, 1990.

[Forcade 93I] T. Forcade, "Evaluating 3D on the High End (Part 1)," *Computer Graphics World*, Volume 16, Number 10, October 1993, pg 44-66.

[Forcade 93II] T. Forcade, "Evaluating 3D on the High End (Part 2)," *Computer Graphics World*, Volume 16, Number 11, November 1993, pg 57-70.

[Goldberg 83] A. Goldberg and D. Robson, *SMALLTALK-80 The Language and Its Implementation*, Addison-Wesley, Reading, 1983.

[Goldberg 84] A. Goldberg, *SMALLTALK-80 The Interactive Programming Environment*, Addison-Wesley, Reading, 1984.

[Gonzales 93] R. Gonzales, "An Object-Oriented Library for Hierarchical Animation Sequences," *SIGGRAPH Computer Graphics*, Volume 27, Number 2, September 1993, pg 82-85.

[Halas 71] J. Halas and R. Manvell, *The Technique of Film Animation*, Focal Press, London, 1971.

[Kamin 90] S. Kamin, *Programming Languages*, Addison Wesley, Don Mills, 1990.

[MacNicol 93] G. MacNicol, "3D Animation: Inexpensive & Effective," *Computer Graphics World*, Volume 16, Number 4, April 1993, pg 37-44.

[Magnenat Thalmann 90] N. Magnenat Thalmann and D. Thalmann, *Computer Animation Theory and Practice Second Revised Addition*, Springer-Verlag, Tokyo, 1990.

[Magnenat Thalmann 91] N. Magnenat Thalmann and D. Thalmann, *New Trends in Animation and Visualization*, John Wiley and Sons, Chichester, 1991.

[Hill 92] H.Hill, K.Brown, "ADDMotion II User Manual," Motion Works International Inc., Vancouver, 1992.

[Pixar 89RI] Pixar, *The RenderMan Interface*, September 1989.

[Pixar 89RC] Pixar, *The RenderMan Companion*, September 1989.

[Robertson 1/93] B. Robertson, "Prime-Time Proving Ground for 3D Graphics," *Computer Graphics World*, Volume 16, Number 1, January 1993, pg 35-44.

[Robertson 12/93] B. Robertson, "Easy Motion," *Computer Graphics World*, Volume 16, Number 12, December 1993, pg 33-38.

[Rubin 84] S. Rubin, *Animation the Art and the Industry*, Prentice-Hall, Englewood Cliffs, 1984.

[Salt 77] B. Salt, *Basic Animation Stand Techniques*, Pergamon Press, Toronto, 1977.

[Sung 90] H. Sung, G. Rogers and W. Kubitz, "A Critical Evaluation of PEX," *IEEE Computer Graphics and Applications*, Volume 10, Number 6, November 1990.

[Upstill 90] S. Upstill, *The RenderMan Interface*, Addison-Wesley, Reading, 1990.

[van Dam et al. 91] R. Zeleznik, D. Brookshire Conner, M. Wloka, D. Aliaga, N. Huang, P. Hubbard, B. Knep, H. Kaufman, J. Hughes and A. van Dam, "An Object-Oriented Library for the Integration of Interactive Animation Techniques," *SIGGRAPH Computer Graphics*, Volume 25, Number 4, July 1991, pg 105-112.

[Wagstaff 93] S. Wagstaff, *Macintosh 3-D Workshop*, Hayden Books, Indianapolis, 1993.

Appendix A

The Traditional Animation Process

When looking at the ways computers have been applied to animation, a basic understanding of conventional animation procedures and terminology is useful. This appendix describes these procedures and defines some popular animation terms.

The production of an animated motion picture traditionally follows the process that is described here. Note that although the process described here

focuses specifically on "cel" animation,[1] similar processes are used for animations involving a wide variety of other techniques (e.g., "claymation"[2], "pixelation"[3], etc.).

The first step in the process is the creation of a brief "synopsis" and its refinement into a "scenario" detailing the complete story. Note that the visual action in the plot is typically more important than the dialog.

The story begins to take visual form when the "storyboard" is developed and adapted to the animation medium. The storyboard is similar to a roughly drawn cartoon strip with boxed drawings and captions outlining the film's key moments. The storyboard also sketches out the main "sequences" (plot segments) in the order they will appear in the film. Each sequence is divided into scenes (each normally involving a single location). Scenes are sometimes composed of multiple "shots" (each normally involving a single camera activity).

The "layout" stage comes next. Here the visual appearance of each character is designed and preliminary sketches are made of the backgrounds for each scene. Prototype drawings of the characters are drawn from different angles and used as references by the animators later in the process. Sometimes physical models of the main characters are also made.

---

1 cel is an abbreviation of cellophane. In cel animation drawings are painted onto cellophane sheets which are placed in layers above an opaque background then photographed.

2 claymation refers to the form of animation that uses flexible physical models typically made from modeling clay.

3 pixelation refers to the form of animation that uses moveable physical models.

Before the process can go any farther the dialogue sound track must be recorded (along with any significant music). The animated movements can then be synchronized precisely with the corresponding sounds.

The animation begins with master animators drawing the "key frames" of the film. At this point these frames are filmed and projected in loose synchronization with the sound track. This filmed version of the storyboard is called a "Leica reel". It allows the director to do early checking and modification of the basic visual imagery of the film.

From the key frames, assistant animators begin the "in-betweening" by drawing the most important or artistically demanding frames between the key frames. "In-betweeners" draw the remaining frames.

Trial animations of these line drawings (called "line tests") are then filmed and tested against the precise timings of the sound track. As each line test is completed it is cut into the Leica reel allowing the director to view successively improved approximations of the final film.

After cleanup and revisions have been completed the director approves the line drawings. They are then photocopied onto cels, the lines are first inked then appropriate colours are painted on. At this point the backgrounds must also be painted.

Final checks are performed, then the cels are layered onto the appropriate backgrounds and each frame is individually photographed. Once the first test print has been approved by the director additional sound effects, etc. are selected and dubbed in. The film then goes into post production for final editing and eventually the final print is made.

Appendix B

Principles of Animation

In the earliest days of the art form, animators struggled to achieve life-like motion in the characters they used. By the late 1930's certain principles of animation were developed that have guided animators ever since. Today the basic principles of animation are universally acknowledged and we can see them reflected in most of the animations produced by the entertainment industry.

This appendix begins with a list of these fundamental principles. Following the list, each of the principles is described briefly. Note that the principles are presented here in no particular order.

1. squashing and stretching (with conservation of mass)
2. pauses and anticipatory movements
3. follow-through and overlapping action
4. easing-in and easing-out
5. sequential action and pose-to-pose action
6. exaggeration
7. visual artistry:

    in form: (simplicity, consistency, definition, clarity)

    in movement: (fluidity, non-linearity -- e.g., arcs)
8. characters with personality
9. staging principles
10. embellishment with secondary actions
11. timing
12. audio artistry

B.1 Squashing and Stretching

The principle of squashing and stretching is especially important in modeling anything biological. Consider your own behaviour if you wanted to jump over a tall obstacle. First you would squash yourself down toward

the ground, and stretch out the leg muscles you use for jumping. When you were ready you would rapidly stretch upward, contracting your leg muscles to give you the thrust necessary to make the jump. To clear the obstacle you would probably scrunch up a little, then stretch out again in anticipation of your landing. When you make contact with the ground on the other side of the object, your natural tendency is to again squash down toward the ground to absorb and cancel the thrust of your impact. In fact, even non-biological objects (such as a simple bouncing ball) exhibit this sort of squashing and stretching. Only the most rigid objects are immune to these distortions[1].

In animation, it is important to be able to model squashing and stretching to achieve realistic motion in non-rigid objects. The rigidity of an object tends to be emphasized if it squashes or stretches insignificantly during motion. Conversely, the fluidity or pliability of an object is emphasized if it squashes or stretches markedly during motion.

It is important to note that stretching and squashing need to be combined with the notion of conservation of mass. That is, when an object is elongated vertically, it must shrink horizontally by a corresponding factor or it appears to increase in volume. Also, when an object is articulated (as with a jumping human) the squashing and stretching may be accomplished without using distortion simply by folding at the joints.

There is another reason to use stretching in animation. When a rapidly moving object is rigid, its image tends to appear in very distinct

---

1 Of course, almost any real object (even if highly rigid) will distort under sufficient force.

locations on successive frames. This can make the object take on a strobing effect, or worse, it can cause the illusion of movement to be lost altogether. In a regular movie, this effect is minimized by the blur that is naturally introduced by the photographic process[2]. Blur effectively elongates images along the line of motion, helping to retain the fluidity of motion[3]. Animators can achieve this fluidity during rapid[4] movement by elongating their moving objects in the same way.

B.2 Pauses and Anticipatory Movements

The human visual system has a broad field of view which approaches 180 degrees wide, although our direct attention is focused only on a narrow segment near the center of this field. In the medium of film, the field of view is much more restrictive (usually less than 90 degrees wide) but it is still possible to be looking at one portion of the screen and miss what is going on elsewhere. An author in the medium of film therefore needs to deal with the limitations of the screen, and with human perceptual limitations.

---

2 This is dependent upon the shutter speed used. A rapid shutter speed will eliminate motion blur, and can introduce problems in regular movies too. You may remember an old western or two where a wagon wheel appears to be rotating in a direction opposite to that which you know it must be turning. When a particular spoke on the wheel makes a large move forward between frames, its new location becomes far away from its old location. At the instant of that next frame, if a second spoke is closer to the old location, but on the opposite side of the old location then this fools our visual system. Our persistence of vision suggests to us that the spoke has moved backward to the location of the second spoke so we think we see the wheel rotating backward.

3 Note that some Ray Tracing programs support motion blur to add this measure of realism to their animations. The RenderMan Interface, for example, contains programming constructs to specify motion blur.

4 Note that when the movement is slow enough, object images in successive frames will overlap and the fluidity of motion will be retained without any extra effort, and without requiring blur.

When a fleeting action is about to occur, the viewer should be alerted to this action in advance so they won't miss it. Animated cartoon characters, for example, always lean to one direction briefly (often spinning their legs frantically without gaining traction) before running off in another direction. When a character is thrown off balance in one direction, it signals to us that movement in the other direction must be coming next. By taking a relatively long time to make their anticipatory movement, they can zip off the screen in only two or three frames (i.e., an eighth of a second or less) and the viewer still understands what has happened, even if they are distracted or blinking at that instant.

Similarly, when an important action is about to occur on one portion of the screen, the viewer's attention should be drawn there. This can also be done by means of pauses and anticipatory motions. When all of the characters on the screen suddenly turn to look at one location, and stop to hold their poses, our attention is naturally drawn to the place they are looking.

To some extent, anticipatory movements are also required for balance and mechanical correctness. For example, it would be unnatural for someone to stand up from sitting in a chair without leaning forward first, and possibly pushing down on the arms of the chair. As another example, consider the animated figure that did not move its foot backward prior to kicking a ball. The kick would seem awkward without this preamble.

Anticipatory movements can also be used to add tension by suggesting to the audience that something is about to happen.

B.3 Follow-Through and Overlapping Action

Newton's law of inertia (oversimplified) states that bodies in motion tend to remain in motion. Inertia is intuitively obvious to us since we see it in all of the moving objects we encounter every day. In order to look natural therefore, animated figures need to obey this law. When an object lands on the ground from a height, it has to stop moving, but there should be some visual evidence of the inertia that had to be overcome as it came to a sudden stop. The object could squash, maybe kick up some dust and knock over a few bystanders. Perhaps some of its component parts could continue to move toward the ground, stopping only shortly after the main structure stops.

Overlapping actions can also add realism. Activity in the real world does not consist of discrete events which have to end before the next one can begin. Animation should therefore use actions that flow smoothly into one another. For example, while an animated character is walking up to a door it can be taking a key out of a pocket. When an animated phone rings, the character answering can be made to exert a coordinated effort to answer the phone. The eyes would normally begin to move first, leading the movement of the head, which in turn would lead the movement of the hand toward the receiver. Although a set of actions may need to occur in a particular logical order, overlapping them can often add a natural-looking smoothness to the animation.

B.4 Easing-in and Easing-out

The law of inertia also says that bodies at rest tend to remain at rest. For this reason, it appears unnatural if an object suddenly starts moving at a high velocity. In nature objects generally accelerate smoothly to some speed, and decelerate smoothly until they stop. To achieve natural movement in animation you must also ease-in to motion and ease-out of them prior to stopping.

B.5 Sequential Action and Pose-to-Pose Action

This principle refers to an artistic choice which must be made by the animator (or a constraint which may be imposed by the computer animation system being used). When animation proceeds sequentially from frame to frame it tends to promote a fresher, more spontaneous result. Animation may also be created by developing key frames where characters hold particular poses and expressions at particular frames. The latter technique is important when timing and particular postures are required, but the former can encourage more creativity.

B.6 Exaggeration

Exaggeration is often used as a vehicle in animation to emphasize physical features, emotions and actions. Size, position, colour, brightness and distortion can all be used to emphasize physical features and emotions. Squash, stretch, pauses, anticipatory movements, follow-through, easing-in

and easing-out can all be exaggerated to emphasize movement. Exaggerated sound effects can also be used to emphasize an action.

B.7 Visual Artistry

Many acknowledged principles of visual art are applicable in animation. First of all, simplicity of form is important since objects in motion must be recognized quickly. For clarity, simplicity must be balanced with definition of detail. You may have noticed that almost all anthropomorphic animated characters have only three fingers (and a thumb). The detail of the extra finger makes the smaller fingers less distinct and hand gestures therefore become more difficult to distinguish. The balance between simplicity and detail can be affected by the amount of movement in the object. Quickly moving objects do not need as much detail as slowly moving or static objects do. Having said that, it is also important to maintain consistency of presentation from frame to frame.

Some natural asymmetry can be desirable, especially to add emphasis to a feature in an essentially bilaterally symmetrical form such as the human figure.

Character movement also benefits from visual artistry. That is, when designing character movement the animator needs to think like a choreographer. Movements are often more pleasing when they are fluid and non-linear. Move your objects along flight paths that follow arcs rather than straight lines. Vary their velocity along their path..

B.8 Characters with Personality

This is more than just a principle. It is the Holy Grail of character animation. If animated characters have personalities, the audience can identify with them and find them believable. The animator needs to treat each character as an actor might consider a part, asking, "What is the character feeling at this moment?" and, "How can this feeling be communicated?". The same character can perform the same task in different ways to express different states of mind to the audience. Different characters can also perform the same task in different ways to express their different personalities.

B.9 Staging

Within the limited size of the screen, staging can be critical. When it is important that the audience see a character's expression, the character's face must be staged appropriately -- usually close to the camera. When an important action is taking place it needs to be staged where it is the focus of the audience's attention. Minimize the movement of other characters in order to attract attention to a moving character (or conversely, maximize the movements of others to attract attention to a stationary character).

Staging is one way to guide the audience's attention from one key idea to the next. For this reason, only one idea should normally be staged at a time. Staging should first direct the audience's attention and then try to make its idea clear.

B.10 Secondary Actions

Secondary actions may be caused by the main action or may be a reaction to the main action.. They are used to embellish the main action, and to add interest and intricacy. For example, a character might show deep concentration on the main action by partly sticking out its tongue, or the roadway behind a fast moving character might smoke, or curl up to emphasize the character's speed.

B.11 Timing

Timing is extremely important in animation for several reasons. First, since all action is artificially created, it must take place at the appropriate rate of speed or it appears unnatural. Also, since inertia is related to mass, an object that takes a long time to start or stop moving appears to be heavy. One that responds quickly appears to be light.

The mood of a character can also be conveyed by timing. Slow movements can convey lethargy, discomfort or intoxication, while fast movements may suggest excitement, nervousness or fear. Note that staging becomes especially important when an action is fast.

B.12 Audio Artistry

Finally, the importance of the audio track needs to be acknowledged. Although a good animation is entertaining and understandable without its audio track, audio can dramatically enhance any animation. Sound effects and music can be used to emphasize actions, to set moods and to direct the attention of the audience. The right sound track can make even a weak animation seem exciting and interesting.

Appendix C

A BNF Grammar For Animaker

This appendix contains the grammar for the Animaker language. The grammar is given in Backus Naur Form (BNF). This grammar is complete with the exception of those portions that are handled by the lexical analyzer. The lexical analyzer parses names (e.g., variable, constant, procedure and function names), numbers (i.e., integer or floating point literals) and strings (i.e., character string literals enclosed in double quote characters as in the C language). In this grammar the symbols <name>, <number> and <string> (respectively) are used for these three language components.

Note also that the lexical analyzer strips comments (either Pascal style, C style or C++ style) and selected words (specifically: "a", "an", "the" and "do") from the input stream before delivering anything to the parser.

C.1 The Program Grammar

```
<program> ::=       [ animation <names> ] [:] [ <output> ] <script> [.]
<names> ::=         { <name> | <names> , <name> }
<output> ::=        destination [:] { none | window | file }
<script> ::=        [ <actpart> ] [ <seqpart> ] script [ <stmts> ] end
<actpart> ::=       actors [:] [ <defs> ]
<seqpart> ::=       sequences [:] [ <seqs> ]
<defs> ::=          { <definition> | <defs>  [{;|.}] <definition> }
<seqs> ::=          { <sequence>  | <seqs>  [{;|.}] <sequence> }
<stmts> ::=         { <statement>  | <stmts> [{;|.}] <statement> }
<definition> ::=    { define <names> parent <names> <classscript> |
                    model <names> = <string> |
                    model <names> include <string> |
                    include <string> }
<classscript> ::=   [ <actpart> ] [ <seqpart> ]
                        constructor [ <stmts> ]
                        destructor [ <stmts> ] end
<sequence> ::=      { [ { static | private | public } ] <procdecl> |
                    include <string> }
<exprs> ::=         { <expression> | <exprs> , <expression> }
<procdecl> ::=      <names> [:] [ ( params ) ] <script>
<params> ::=        { <param> | <params> [,] <param> }
<param> ::=         [ var ] <name>
```

C.2 The Statement Grammar

<statement> ::= { { [<stmts>] } |
dimension <lvalue> **:** [exprlist] |
record <lvalue> **:** { namelist } |
get <expression> |
set <lvalue> **to** <expression> |
put <expression> **into** <lvalue> |
<lvalue> { **=** | **:=** | **+=** | **-=** | ***=** | **/=** | **%=** }
<expression> |
<lvalue> { **--** | **++** } |
{ **new** | **create** } <name> [**named**] <lvalue> |
{ **dispose** | **discard** } <lvalue> |
{ **when** | **wait** | **once** | **every** }
<expression> <statement> |
{ **always** | **forever** } <statement> |
start [<lvalue>] <statement> |
stop { <lvalue> | **all** } |
return [(<expression>)] |
{ **const** | **constant** } <name> **=** <expression> |
{ **local** | **private** | **extern** | **external** |
public } (<names>) |
if <expression> [**then**] <stmts>
[{ **else** | **otherwise** } <stmts>] **end** |
repeat <statement> { **while** | **until** | **for** }
{ <timespan> | <timeaddr> | <expression> } |
repeat <statement> {;|.} <timespan>
while <expression> <statement> |
until { <timeaddr> | <expression> } <statement> |
for <name> **:=** <expression> **to** <expression> <statement> |
for (<lvalue> **:=** <expression> {;|.} <expression> {;|.}
<lvalue> { **--** | **++** }) <statement> |
for <timespan> <statement> |
<name> (<exprs>) |

tell { <lvalue> | **parent** } **to** <name>
[(<exprs>)] [<modifier>]

<timeaddr> ::= { **frame** | **second** } <expression>

<timespan> ::= <expression> { **frames** | **seconds** | **times** }

<modifier> ::= { **now** | **varying** <lvalue>
[**from** <expression> **to** <expression>]
[**in** <expression> **steps**]
[**over** <expression> { **frames** | **seconds** }] }

C.3 The Expression Grammar

<expression> ::= { [exprs] | <factor> |
<expression> { **or** | **||** | **and** | **&&** | **=** | **==** | **<** | **<=** |
> | **>=** | **+** | **-** | ***** | **/** | **%** | **mod** } <expresion> }

<factor> ::= { { **+** | **-** | **!** | **not** } <primary> |
{ **++** | **--** } <lvalue> |
<lvalue> { **++** | **--** } }

<primary> ::= { <lvalue> | <number> | <string > |
(<expression>) |
<lvalue> { **says** | **said** } (<expression>) |
<name> **of** { <lvalue> | **parent** } [(<exprs>)] |
{ <lvalue> | **parent** } **'s** <name> [(<exprs>)] |
my <name> [(<exprs>)]
<name> (<exprs>)

<lvalue> ::= { <name> | <lvalue> [<exprs>] }

Appendix D

Animaker Language Reference

The Animaker language is block structured (like C and Pascal). The main sections of an Animaker program are shown in the example shell below. Each of these sections is expanded and explained in this appendix.

```
animation ...
destination ...
actors
        ...
sequences
        ...
script
        ...
end.
```

Following the **animation** keyword you may give one or more names to your animation. This clause is optional.

Following the **destination** keyword you may state where the output frame descriptions should be written. By default (i.e., if the **destination** clause is omitted altogether) frames are output to files, which are subsequently sent to the Conductor program for processing. You may code either **file**, **window**, or **none** here to have the frames (respectively) written to files only, to the output window only, or to have them simply discarded.

Following the **destination** clause you may optionally code an **actors** section. The name of this section may be a little misleading. Although it may suggest that it has something to do with the system defined actor class, it is much more general. Here any number of class or model definitions may appear and files containing appropriate Animaker code may also be included. They are coded as shown below:

```
actors
        ...
        model modelname = "model string";
        model includemodelname include "modelfilename";
        define newclassname parent oldclassname
                actors ...
                sequences ...
                constructor ...
                destructor ...
                end;
        include "includefilename";
        ...
```

Next (after the **actors** section, if any) you may optionally code a **sequences** section. In the sequences section any number of procedure definitions may appear and files containing appropriate Animaker code may also be included. They are coded as shown in the following segment:

```
actors
    ...
    private privateprocedurename ( ... )
        actors ...
        sequences ...
        script ...
        end;
    public publicprocedurename ( ... )
        actors ...
        sequences ...
        script ...
        end;
    include "includefilename";
    ...
```

It is also possible to code the keyword **static** in place of the keyword **private**. These keywords are synonyms so either may be used to convey the same semantic. Both are supported by the language to afford choice to the programmer. This is a common theme in the language -- allowing several different constructions to express the same semantic.

At the bottom of an Animaker program (after the **actors** and **sequences** sections, if any) you must code a **script** section. Note that this is the only required section of the main program. That is, the shortest possible, complete, syntactically legal Animaker program is:

```
script
end.
```

In the script section any number of statements may appear. This section is analogous to the portion of a Pascal main program between the "begin" and corresponding "end.". Each of the statement types that may be coded in a **script** section are described separately, later in this appendix.

D.1 Class Definitions

Class definitions must always be coded within an **actors** section. They consist of the keyword **define** followed by one or more names (typically a plural and singular name are given in a class definition). After the new class name there must be a **parent** clause to define the ancestry of this class (i.e., a list of parent class names). Multiple inheritance is supported, but there must not be any common method or instance variable names across the set of all parent classes. Following this is essentially a script block containing an optional **actors** section, and an optional **sequences** section. The sequences section is where the methods of this class are defined. Instead of following this with a standard **script** section, a class definition contains two new sections called **constructor** and **destructor**. These sections are essentially required methods for this class.

The **constructor** section contains statements which are executed within the scope of a new object immediately after the object is created. This code is generally used to define variables, and to initialize the object. This may include the creation of subordinate component objects.

The **destructor** section contains statements which are executed within the scope of an object immediately prior to the object's disposal. This code is generally used to dispose of subordinate objects, if any.

D.2 Procedure Definitions

Procedure definitions must always be coded within a **sequences** section. They consist of a keyword that defines the procedure's scope. A **public** procedure may be called from any code in the animation script. A procedure defined **private** or **static** is only callable within the scope where it was defined. These scope rules are similar to those used by the Pascal language.

After the **public**, **private** or **static** keyword a name, or names is given to the procedure. Typically, procedures are named as verbs and more than one name is given to allow different conjugations for person or plurality.

Following the procedure name the procedure parameters (if any) are named and declared. If there are no parameters, this section is omitted. If there are parameters they are listed here, separated by commas, and the entire list is enclosed within parentheses. Procedure parameters (as with Animaker variables) are not declared with respect to data type. They are declared with respect to function, using the Pascal syntax. Parameters that are passed by value (i.e., those which deliver information to the procedure but which are not capable of returning any information to the caller) are listed simply by stating their names. Parameters passed by reference (i.e., those which are capable both of delivering information to the procedure and returning information to the caller) are listed by preceding each name with the keyword **var**.

After any parameter declarations comes a standard script block containing an optional **actors** section, an optional **sequences** section and a

required **script** section. Each of these sections has the same syntax and semantics as those discussed earlier regarding the main animation script with one exception.

That expectation is that within a procedure **script** section, you may code a **return** statement. When executed this statement causes the procedure to return immediately to its caller (like the C language statement of the same name). Any procedure which contains a **return** statement followed by an expression in parentheses may be called within an expression (i.e., it may be called as a function). The expression following the **return** statement will be returned as the function result.

D.3 Simple Statements

Animaker is rich in terms of the number of statement types provided. There are 23 different types of statements and many of these have several variants. Lets begin by looking at those statements that are concerned with variable declaration and definition.

In Animaker, variables are dynamically typed. That is, at one moment a variable may have string type, but later the same variable may have number type. Still later the same variable may hold an array of object instances. Several built-in functions support type coercion should this become necessary[1]. A variable may be created by simply using it as the

---

1 Type coercion is computer programming language jargon for the process of converting data to a specific type. This is useful in many programming contexts. For example, it is often desirable to be able to treat a character string as a number or vice versa.

target of an assignment statement, or by passing it as a reference parameter to a procedure. Variables may also be more formally declared using any of the statements shown below:

record recordvariablename **:** { fieldname, anotherfieldname, ... };
dimension arrayvariablename **:** [...];

Animaker variables may be declared to be record type with the **record** statement. This actually makes the variable into a single dimensional array with named subscripts. The syntax for accessing record fields in identical to that for accessing array elements.

Variables may also be declared as arrays with any number of dimensions by means of the **dimension** statement. The size of each dimension is listed within the square brackets at the end of the statement.

Note that each element in an array may have different type. One array element may be a record, another an array, another an array of records, another a simple number type.

Variable scope may also be explicitly declared using one of the statements below. By default newly referenced variables have private scope. However, if a public variable with the same name exists outside the local scope then the use of this name is not new, so a new private variable is not declared. For these two reasons, tools are required for the creation of both private and public variables.

private (variablename, anothervariablename, ...);
public (variablename, anothervariablename, ...);

Note that **local** is a synonym for **private** here. Similarly, **extern** and **external** are synonyms for **public** .

The tools provided for variable assignment are many. Many elements of the C language's rich assignment set are supported, along with a few others. For example,

set variable **to** expression;
put expression **into** variable;
variable **=** expression;
variable **:=** expression;

These statements all have the same effects as their equivalents in other languages. All of these simply store an expression into the variable (which may actually be any valid "lvalue"[2] expression itself -- e.g., a subscripted array for example). The simple statement: **get** expression; may also be used to store an expression into the special variable "it".

Each of the statements below has been taken from the C language. They all consist of an operator symbol followed by an equal sign symbol. The effect of each is to compute the result of the variable-operator-expression combination, then assign the result to the variable. For example, the statement "a += b" computes "a + b" and stores the result in variable "a".

variable **+=** expression;
variable **-=** expression;
variable ***=** expression;
variable **/=** expression;
variable **%=** expression;

The next two assignment statements also come from C. The first simply increments its variable. The second decrements it.

variable **++**;
variable **--**;

---

2 The term "lvalue" is used in describing the syntax of the C language. In simple terms, it refers to anything which may legally appear on the left side of an assignment operator.

Note that in the C language, assignment statements are not simply statements. They are expressions as well, and their value is simply the value that is being assigned. This makes it possible to construct complex C language statements which result in many assignments. Since Animaker is designed to be used by novice programmers, these side effects were deemed to be dangerous. In Animaker, most assignment statements are therefore not valid within expressions. However, as you will see in the expression section of this appendix, Animaker allows pre-increment, post-increment, pre-decrement and post-decrement expressions. As statements though, both pre- and post- versions would be equivalent.

Animaker also supports the definition of constant names. The **record** statement we have already looked at defines constants for its field names, but there is also a more general tool for this purpose:

constant constantname = expression;

The **constant** keyword may be abbreviated **const**.

As an object oriented language, Animaker also supports the dynamic creation and disposal of objects of any class. In fact, there is no other (e.g., static) way to create an Animaker object. To create an object named "myobject" of class "myclass" you would use:

new myclass **named** myobject;

Note that the **create** keyword is a synonym for **new**.

To dispose of this object you would use:

dispose myobject;

The **discard** keyword is a synonym for **dispose**.

D.4 Scheduling Statements And Control Structures

The Animaker scheduler is supported by several statements imbedded in the language. The simplest scheduling statements simply tell the scheduler to start doing something every single frame. They are:

always *statement* ;
forever *statement* ;

The keywords **always** and **forever** are Animaker synonyms so they both express the same semantic.

One can also tell the scheduler to start doing something every frame, but retain a name for this action so you may later stop doing it. These statements support this:

start nameforthisaction *statement* ;
stop nameforthisaction;
stop all;

The latter simply removes all appointments from the current schedule.

The more interesting scheduling tools use a logical or temporal expression to control when the registered appointment is acted upon (i.e., when the corresponding statement is executed). There are four statements of this type:

wait *expression statement* ;
when *expression statement* ;
once *expression statement* ;

The **wait** statement asks the scheduler to wait until its condition becomes true, then start executing its statement every frame from then on (even if the condition becomes false again). The **once** statement causes the scheduler to execute its statement exactly once when its condition becomes

true, then remove this appointment. The **when** statement makes the scheduler execute its statement only on frames when its condition is true. Note that **every** is a synonym for **when**.

Animaker also contains many other control structures. A complete listing is shown below:

if expression **then**
    statement ;
end;

if expression **then**
    statement ;
else
    statement ;
end;

for variable **:=** *expression* **to** *expression*
    statement ;

for variable **:=** *expression* **to** *expression*
    statement ;

for (var **:=** expr; expression ; var**++**)
    statement ;

for timespan
    statement ;

repeat
    statement ;
while *expression* ;

repeat
    statement ;
until *expression* ;

repeat
    statement ;
for *temporalexpression* ;

repeat
    statement ;
timespan ;

while *expression*
    statement ;

until *expression*
statement ;

Of course, anywhere in an Animaker program where a single statement may appear, a set of statements may also be used, enclosed in "{" and "}" characters.

D.5 Procedure And Method Calls

It must first be noted that Animaker procedures and methods may be invoked by means of a procedure call (statement) or by means of a function call (within an expression). This section refers only to procedure and method calls that occur outside of the context of expressions. Expressions are covered later in this appendix.

To invoke an Animaker procedure (either built-in or programmer defined) you use the standard syntax:

procedurename (expression, anotherexpression, ...);

To avoid a syntactic ambiguity, the parentheses are required here. The number of expressions listed must correspond to the number of parameters in the called procedure (possibly zero).

To invoke a method defined for an Animaker object you use this syntax:

tell myobject **to** methodname (expression, anotherexpression, ...) *modifier* :

Since it introduces no syntactic ambiguity here (unlike the case of a procedure call), the parentheses are not required unless arguments are being

passed to the method. When this statement is executed, the named object's scope is entered, as the expression list (if any) is passed to the named method. The optional modifier clause is used for scheduling purposes. Modifier clauses will be discussed in detail shortly.

Note that within the code of a method you may invoke any method defined in the immediate parent class of this class with the syntax shown below:

tell parent to methodname (expression, anotherexpression, ...) *modifier* :

Although it has other uses, this feature is particularly helpful in an overridden method because it can be used to invoke the corresponding method in the parent class. An overridden method may therefore easily add to, or reduce, the functionality of the parent method.

The modifier clause that appears at the end of these method calls is optional. If it is omitted, it is equivalent to coding the keyword **now**. As the word suggests, the method is invoked immediately in these cases. In all other cases, the scheduler is used to invoke the method according to the parameters of the appointment that is set up in the modifier clause. Modifier clauses are made up of one required portion, and three optional portions.

The required portion is a **varying** clause which specifies the variable to be altered by the scheduler. The first optional clause specifies what values the variable is to take on (i.e., **from** *expression* **to** *expression*). If the **from** clause is omitted, the variable will range from 0.0 to 1.0 as it is modified. Next, the optional **in** clause specifies the number of increments that are to be made to the variable (i.e., **in** *expression* **steps**). If the **in** clause is omitted, the number of increments will be the number of frames. Finally,

the (optional) **over** clause specifies a time span over which this variable is to be modified and the statement is to be executed. If it is omitted, the statement is executed the number of steps coded in the **in** clause, or simply once (if the **in** clause was also omitted).

The **over** clause is coded in one of the forms shown below, and it is the last part of the statement:

over *expression* **frames**
over *expression* **seconds**

It simply specifies that the statement is to be executed repeatedly for the duration specified in **frames** or **seconds**. The number of frames over which the statement will be executed if **seconds** is coded will depend upon the current frame rate.

D.6 Expressions

Expressions in Animaker are very similar in syntax and semantics to those used in other programming languages, but there are several extensions. Since Animaker is designed for novice programmers, a simple precedence hierarchy was used. The simplest, atomic expressions and expressions enclosed in round parentheses are evaluated first. The standard unary operators (i.e., like those found in most languages, e.g., C or Pascal) are evaluated next. Finally the standard binary operators are evaluated. All operators are associated from left to right, and the following precedence table is used:

Unary operators:	1.	**+, -, not, !, ++, --**
Binary operators:	2.	***, /, mod, %**
	3.	**+, -**
	4.	**=, ==, <, <=, >, >=, <>, !=**
	5.	**and, &&**
	6.	**or, ||**

Simple expressions in Animaker start with the usual numeric literals, string literals and logical literals you often find in other languages. Animaker also supports complex literals. For example:

```
[123, "wow", 4.5, [ "a", 67 ] ]
```

is a valid literal in Animaker. It defines a one dimensional array containing 4 elements. The last element is an array itself, containing two elements.

Of course variable names are also simple expressions in Animaker. Array variable names followed by subscript expressions (enclosed in square brackets as usual) are also relatively simple expressions. Their subscript expressions are independent of the other operations in the expression and are therefore evaluated first so a subscripted array may be considered a simple expression.

To support inter-object communication, the special logical expressions shown below are also evaluated as simple expressions:

someobject **says** (*expression*);
someobject **said** (*expression*);

The expression must be a character string. These expressions are true, or become true, when the named object utters the specified string. Strings are spoken by objects using the built-in procedure called "Say".

Procedures may be invoked within expressions (i.e., as functions) using the syntax shown below:

procedurename (expression, anotherexpression, ...)

Methods may be invoked within expressions (like functions) using five different syntactic forms. The first two are shown below:

methodname **of** myobject (expression, anotherexpression, ...)
myobject**'s** methodname (expression, anotherexpression, ...)

Both of these forms have the same semantics. They each install the scope of "myobject", and then run the given "methodname" method, passing the arguments supplied (if any). The called method must return an expression to be used as the value of this expression.

As with the method calls described earlier when we discussed statements, there is a special syntax to allow parental methods to be invoked within any method of a descendant class. This is the syntax to use in such cases:

methodname **of parent** (expression, anotherexpression, ...)
parent's methodname (expression, anotherexpression, ...)

The last syntactic construct provided for calling methods was added to allow the programmer to emphasize that an object is calling a method of its own. This syntax often allows more natural language to be used in a script:

my methodname (expression, anotherexpression, ...)

Note that in all five of these syntactic forms, the parentheses must be omitted if the method being called accepts no parameters.

Appendix E

System Defined Classes

This appendix contains detailed descriptions of each of the Animaker system defined classes: Trivial, Base, Item, Collection, Camera, Light, Actor, Homogeneous and Heterogeneous. A brief description of each class is followed by a list of its ancestors and descendants. Additional detailed information such as what instance variables exist, and descriptions of each of the methods is also provided.

E.1 Trivial Class

Description:

The Trivial Class is just that. It is a virtually empty class definition that is intended to be used as the parent of any general purpose class. It is the parent class that provides the least services in this system; therefore it burdens its descendants with the least overhead. It has no instance variables, and all of its methods are null (i.e., they are intended to be overridden by descendant classes when necessary).

Ancestry:

None; this is the root of the class hierarchy.

Descendants:

All classes descend from this class. This class exists as an abstract class to allow classes not requiring any of Animaker's scheduling services to be created without that overhead. Direct descendants may still get timing information by overriding the null methods provided here. This class has only one system-defined direct descendant, the Base Class.

Instantiation:

You may not instantiate objects of this class; it is abstract.

Instance Information:

None.

Public Methods:

Action	Does nothing.
NewFrame	Does nothing.
Output	Does nothing.
Smile	Does nothing.

E.2 Base Class

Description:

The Base Class is the root class for all things making use of Animaker's scheduling facilities. Support for scheduling adds significant overhead in terms of space and processing time so classes not requiring these services may be better suited to direct descent from the Trivial Class.

Many of Animaker's language features are designed to provide Base Class scheduling services. Language features allow appointments to be registered for any objects instantiated at this level in the hierarchy or from any descendant class. Once registered, these appointments will be met at the appropriate times without further involvement from the animator.

Ancestry:

This is the only system-defined direct descendant of the Trivial (root) Class.

Descendants:

All other system-defined classes except for the root class descend from the Base Class. This class has two system-defined direct descendants: the Item Class and the Collection Class (both of which are abstract). Altogether seven specialized system-defined classes have been created as Base Class descendants with the intent that few user-defined classes will need to descend directly from this class.

Instantiation:

You may instantiate objects of this class, though there may be little motivation to do so because a Base object can do very little.

Instance Information:

<none>

Public Methods:

Action	Each appointment registered in the local schedule is reviewed. Any appointment that requires action at this time is met.
Cue	Adds this object to the list of active objects for this scene. When instantiated, objects are automatically **Cue**d.
Cut	Removes this object from the list of active objects for this scene.
NewFrame	Does nothing.
Output	Does nothing.
Smile	Does nothing except return the Logical value *true*. All active objects in the scene are sent the **Smile** message repeatedly until they respond with *true*. The system will not advance to the next frame until all active objects return *true*.

E.3 Item Class

Description:

The Item Class is the root class for all things making use of Animaker's spatial manipulation facilities. Support for these manipulations adds significant overhead in terms of space and processing time so classes not requiring these services may be better suited to direct descent from the Base Class, or Trivial Class instead.

Many of Animaker's language features are designed to provide Item Class spacial manipulation services. Any object instantiated at this level in the hierarchy or below has a spacial existence in its own virtual universe, and in the encompassing Animaker Universe.

Each object universe is an infinite 3D Cartesian space based on x, y, and z axes (i.e., where every point has x, y, and z coordinates). The object universe's coordinate system is left handed (see Chapter 6). The object is oriented within its universe relative to the coordinate origin and two of the coordinate axes. The positive y axis indicates which direction is up, and the positive z axis indicates the direction the object is facing.

The Animaker Universe is a similar 3D Cartesian space. An object instantiated at this level in the hierarchy or below has its own universe mapped onto the Animaker Universe; this mapping may include any number of translations, scalings, or rotations in any order. Advanced animators may also perform this mapping by means of arbitrary homogeneous transformation matrices. Simply put, the position, scale and orientation of any object's coordinate system within the Animaker Universe is completely flexible. The methods defined for this class illustrate this flexibility.

Ancestry:

This is one of two system-defined direct descendants of the Base Class. As such, it inherits the full scheduling facilities offered by the Base Class.

Descendants:

This class has three system-defined direct descendants: the Camera Class, the Light Class and the Actor Class. These three descendant

classes represent the three categories of real-world objects that this system is designed to mimic: cameras, light sources and "other things" (respectively). It is intended that these three descendant classes will minimize the need for user-defined classes to descend directly from this class.

Instantiation:

You may not instantiate objects of this class; it is abstract.

Instance Information:

<none>

Public Methods:

Location Returns the coordinates, in the Animaker Universe, of the origin of the object's universe.

Direction Returns a vector, in the Animaker Universe, emanating from the origin of the object's universe, representing the positive x-axis of the object's coordinate system.

Up Returns a vector, in the Animaker Universe, emanating from the origin of the object's universe, representing the positive y-axis of the object's coordinate system.

Move(p) Takes a single **p**oint parameter. The object's origin is moved in the Animaker Universe by the displacements specified in the three coordinates of this point.

MoveTo(p) Takes a single **p**oint parameter. The object's origin is moved in the Animaker Universe to the coordinates specified in the three coordinates of this point.

Rotate(p) Takes a **p**oint. The object's coordinate system is rotated within its own coordinate system by the three angles specified by the three coordinates of point **p**. The angles are measured in degrees, left-handed (see Chapter 6), along each respective axis.

RotateTo(p) Takes a single **p**oint parameter. First, all rotation transformations are removed from the object, then the object's coordinate system is rotated within the Animaker Universe by the three angles specified by the three coordinates of point **p**. The angles are measured in degrees, left-handed, along each respective axis.

RotateAtOrigin(p) Takes a single **p**oint parameter. The object's coordinate system is rotated within the Animaker Universe by the three angles specified by the three coordinates of point **p**. The angles are measured in degrees, left-handed, along each respective axis.

Transformation Takes no parameters and returns the current cumulative transformation matrix. That is, as transformations are applied to the object, this homogeneous coordinate (4X4) transform is updated to reflect the current mapping of object space onto the Animaker Universe. Normally animators will not need to use this function.

E.4 Camera Class

Description:

The Camera Class provides facilities for controlling view orientation and other view parameters. Objects instantiated in this subtree are capable of producing frame output and they control the portion of the output content that is related to view.

Ancestry:

This class is a direct descendant of the Item Class (which descends from the Base Class). It therefore has access to the full scheduling and spatial manipulation facilities (offered by the Base Class and the Item Class, respectively).

Descendants:

There are no system-defined descendants of this class. Advanced animators may wish to define more interesting image collection devices than are possible using the Camera Class without modification. All such image collectors should descend from this class.

Instantiation:

You may instantiate objects of this class. To remove some of the motivation to do so, the system defines a single object named "camera" which is an instance of this class. This system camera is the default object that is used to create frame descriptions (i.e., shoot pictures, take snapshots). If you wish to use more than one camera, simply instantiate more, and configure them as desired. Beware that multiple *active* cameras result in multiple renderings of each frame. Be careful to **Cue** and **Cut** cameras when appropriate or you may end up with more frame images than you intended. The default camera may also be **Cut** if desired.

Instance Information:

_attributes Elements of this class or its descendants may store string descriptions of camera attributes in this instance variable which is output in the camera/view description area of the frame description. It should contain valid renderer

commands and/or legal comments (but this is not checked).

_pointat This is the point that the camera is currently pointing at.

Public Methods:

PointAt(p) Takes a single **p**oint parameter, and orients the camera to point at this point.

Shoot Creates a picture (frame image) of the scene as it is composed at this frame in time. All cameras in the scene must receive **Shoot** messages before the system will send **Smile** messages to all active objects, and advance to the next frame.

E.5 Light Class

Description:

The Light Class supports the creation and adjustment of light sources. Objects instantiated in this subtree are capable of adding light to the scene.

Ancestry:

This class is a direct descendant of the Item Class (which descends from the Base Class). It therefore has access to the full scheduling and spatial manipulation facilities (offered by the Base Class and the Item Class, respectively).

Descendants:

There are no system-defined descendants of this class. Advanced animators may wish to define more interesting light sources than are possible using the Light Class without modification. All such light sources should descend from this class.

Instantiation:

You may instantiate objects of this class. To remove some of the motivation to do so, the system defines a single object named "ambient" which is an instance of this class. This ambient light is the simplest possible light source. If you wish to add other light sources to the scene, simply instantiate more, and configure them as desired. The ambient light source may also be reconfigured to meet your needs; when it is not wanted, simply tell it to **TurnOff** or just **Cut** it from the scene.

Instance Information:

_attributes Elements of this class or its descendants may store string descriptions of light attributes in this instance variable which is output in the attribute area of this object description. It should contain valid renderer commands and/or legal comments (but this is not checked).

_colour This is a point variable whose x, y, and z coordinates represent the red, green and blue

components (respectively) of the colour of this light source.

_pointat This is the point that the light is currently pointing at.

Public Methods:

PointAt(p) Takes a single **p**oint parameter, and orients the light to point at this point.

TurnOn Turns this light *on*. This light will begin emitting light onto the scene according to the settings of its attributes.

TurnOff Turns this light *off*. This light will stop emitting light onto the scene regardless of the settings of its attributes. This has the same effect as **Cut**ting the light from the scene.

E.6 Actor Class

Description:

The Actor Class supports the creation and manipulation of any individual scene components other than image collectors and light sources. Objects instantiated in this subtree are usually illuminated by the scene's lights, and photographed by the scene's cameras (virtually, of course).

Ancestry:

This class is a direct descendant of the Item Class (which descends from the Base Class). It therefore has access to the full scheduling and spatial manipulation facilities (offered by the Base Class and the Item Class, respectively).

Descendants:

There are no system-defined descendants of this class. Advanced animators may wish to define more interesting (non-light, and non-camera) *things* than are possible using the Actor Class without modification. All such *things* should descend from this class.

Instantiation:

You may instantiate objects of this class.

Instance Information:

_colour This is a point variable whose x, y, and z coordinates represent the red, green and blue components (respectively) of the colour of this object.

_model This variable contains either the text of this object's model, or the name of the file containing this included model.

_diffusereflectance This is a numeric variable representing the diffuse reflectance of this object.

_specularreflectance This is a numeric variable representing the specular reflectance of this object.

_transparency This is a numeric variable representing the transparency of this object.

Public Methods:

Scale(p) Takes a **p**oint. The object's coordinate system is scaled within its own coordinate system by the three factors specified by the three coordinates of point **p**. Factors greater than 1 enlarge; those smaller than 1 shrink.

ScaleTo(p) Takes a single **p**oint parameter. First, all scaling transformations are removed from the object, then the object's coordinate system is scaled within the Animaker Universe by the three factors specified by the three coordinates of point **p**. Factors greater than 1 enlarge; those smaller than 1 shrink.

ScaleAtOrigin(p) Takes a single **p**oint parameter. The object's coordinate system is scaled within the Animaker Universe three factors specified by the three coordinates of point **p**. Factors greater than 1 enlarge; those smaller than 1 shrink.

Transform(t) Takes a single **t**ransformation matrix parameter. The object's coordinate system is first remapped onto the Animaker Universe origin and axes (i.e., any previous transformations are discarded) then the object's coordinate system is transformed using the specified homogeneous (4X4) transformation matrix **t**. This method is provided for advanced animators who need to go beyond simple translation, scaling and rotation. Most animators will never need to use this tool.

E.7 Collection Class

Description:

The Collection Class supports the creation and manipulation of collections of objects.

Ancestry:

This is one of two system-defined direct descendants of the Base Class. As such, it inherits the full scheduling facilities offered by the Base Class.

Descendants:

This class has two system-defined direct descendants: the Heterogeneous Class and the Homogeneous Class. These two descendant classes are designed to support the groups of objects that are diverse in their capabilities and behaviours (the Heterogeneous Class) and groups of objects which are very similar in capabilities and behaviours (the Homogeneous Class). It is intended that these two descendant classes will minimize the need for user-defined classes to descend directly from this class.

Instantiation:

You may not instantiate objects of this class; it is abstract.

Instance Information:

<none>

Public Methods:

<none>

Note:

This class was not implemented.

E.8 Heterogeneous Class

Description:

The Heterogeneous Class supports the creation and manipulation of collections of objects that are diverse in their capabilities and behaviours.

Ancestry:

This class is a direct descendant of the Collection Class (which descends from the Base Class). It therefore has access to the full scheduling facilities (offered by the Base Class).

Descendants:

There are no system-defined descendants of this class.

Instantiation:

You may instantiate objects of this class.

Instance Information:

<none>

Public Methods:

<none>

Note:

This class was not implemented.

E.9 Homogeneous Class

Description:

The Homogeneous Class supports the creation and manipulation of collections of objects that are very similar in their capabilities and behaviours.

Ancestry:

This class is a direct descendant of the Collection Class (which descends from the Base Class). It therefore has access to the full scheduling facilities (offered by the Base Class).

Descendants:

There are no system-defined descendants of this class.

Instantiation:

You may instantiate objects of this class.

Instance Information:

<none>

Public Methods:

<none>

Note:

This class was not implemented.

Appendix F

Example Animaker Programs

This appendix contains two fairly large Animaker source code files. The first, called "Planets.Ani," is a main animation program. The second, called "PlanetarySystem.Inc" is an included clip-animation file. The clip-animation file contains a complete class description for a planetary system class (a subclass of the system defined Actor class). Objects of this planetary system class consist of a planet and zero or more moons which automatically orbit their host planets. The main program includes the clip-animation near the top of the program, then creates three planetary systems for the animation. The rest of the main animation program consists entirely of commands to manipulate the camera (i.e., moving it and pointing it in

different directions). This program produces an animation of 24 seconds duration at 30 frames per second (or 720 frames).

F.1 An Animaker Main Program

```
animation Planets

destination file

actors
        include "...:Planetary System.Inc";

script

        _renderer = "POV";

        constant x = 0;
        constant y = 1;
        constant z = 2;

        framesPerSegment = 2 * framerate;

        new Light L1;
        new Light L2;
        new Light L3;

        tell L1 to MoveTo( [ 100,100,-200 ] );
        tell L2 to MoveTo( [ 0,100, 0 ] );
        tell L3 to MoveTo( [ 0,100, 40 ] );

        new PlanetarySystem Venus;
        new PlanetarySystem Earth;
        new PlanetarySystem Mars;

        tell Mars to SetPlanetSurface( "Mars.pic" );
        tell Mars to SetPlanetSize( 10 );
        tell Mars to AddMoon( "MarsMoon1.pic", 3 );
        tell Mars to AddMoon( "MarsMoon2.pic", 2 );
        tell Mars to MoveTo( [ 0, 0, -100 ] );

        tell Earth to SetPlanetSurface( "Earth.pic" );
        tell Earth to SetPlanetSize( 13 );
        tell Earth to AddMoon( "EarthMoon.pic", 4 );

        tell Venus to SetPlanetSurface( "Venus.pic" );
        tell Venus to SetPlanetSize( 24 );
        tell Venus to AddMoon( "VenusMoon.pic", 6 );
        tell Venus to MoveTo( [ 0, 0, 100 ] );
```

```
InitialFocus = Mars's Location;
SecondFocus = Earth's Location;
ThirdFocus = Venus's Location;

InitialLocation = [ 100, 100, -300 ];
SecondLocation = [ 100, 100, InitialFocus[ z ] ]; // On Focus 1
ThirdLocation = [ -100, 100, InitialFocus[ z ] ]; // To Focus 2
FourthLocation = [ -100, 100, SecondFocus[ z ] ]; // On Focus 2
FifthLocation = [ 100, 100, SecondFocus[ z ] ];   // To Focus 3
SixthLocation = [ 100, 100, ThirdFocus[ z ] ];    // On Focus 3
SeventhLocation = [ 100, 100, SecondFocus[ z ] ]; // Back to Focus 2
EighthLocation = [ 100, 100, InitialFocus[ z ] ]; // Back to Focus 1

CurrentLocation = InitialLocation;
CurrentFocus = InitialFocus;

tell defaultcamera to PointAt( CurrentFocus );
tell defaultcamera to MoveTo( CurrentLocation ) varying CurrentLocation[ z ]
      smoothly from InitialLocation[ z ] to SecondLocation[ z ]
      over framesPerSegment frames;

repeat shoot() for 2 * framesPerSegment frames;

tell defaultcamera to MoveTo( CurrentLocation ) varying CurrentLocation[ x ]
      smoothly from SecondLocation[ x ] to ThirdLocation[ x ]
      over framesPerSegment frames;
tell defaultcamera to PointAt( CurrentFocus ) varying CurrentFocus[ z ]
      smoothly from CurrentFocus[ z ] to SecondFocus[ z ]
      over framesPerSegment frames;

repeat shoot() for 2 * framesPerSegment frames;

tell defaultcamera to MoveTo( CurrentLocation ) varying CurrentLocation[ z ]
      smoothly from ThirdLocation[ z ] to FourthLocation[ z ]
      over framesPerSegment frames;

repeat shoot() for 2 * framesPerSegment frames;

tell defaultcamera to MoveTo( CurrentLocation ) varying CurrentLocation[ x ]
      smoothly from FourthLocation[ x ] to FifthLocation[ x ]
      over framesPerSegment frames;
tell defaultcamera to PointAt( CurrentFocus ) varying CurrentFocus[ z ]
      smoothly from CurrentFocus[ z ] to ThirdFocus[ z ]
      over framesPerSegment frames;

repeat shoot() for 2 * framesPerSegment frames;

tell defaultcamera to MoveTo( CurrentLocation ) varying CurrentLocation[ z ]
      smoothly from FifthLocation[ z ] to SixthLocation[ z ]
      over framesPerSegment frames;

repeat shoot() for 2 * framesPerSegment frames;
```

```
        tell defaultcamera to MoveTo( CurrentLocation ) varying CurrentLocation[ z ]
                smoothly from SixthLocation[ z ] to SeventhLocation[ z ]
                over framesPerSegment frames;
        tell defaultcamera to PointAt( CurrentFocus ) varying CurrentFocus[ z ]
                smoothly from CurrentFocus[ z ] to SecondFocus[ z ]
                over framesPerSegment frames;

        repeat shoot() for 2 * framesPerSegment frames;

        tell defaultcamera to MoveTo( CurrentLocation ) varying CurrentLocation[ z ]
                smoothly from SeventhLocation[ z ] to EighthLocation[ z ]
                over framesPerSegment frames;
        tell defaultcamera to PointAt( CurrentFocus ) varying CurrentFocus[ z ]
                smoothly from CurrentFocus[ z ] to FirstFocus[ z ]
                over framesPerSegment frames;

        repeat shoot() for 2 * framesPerSegment frames;

        tell defaultcamera to MoveTo( CurrentLocation ) varying CurrentLocation[ z ]
                smoothly from EighthLocation[ z ] to InitialLocation[ z ]
                over framesPerSegment frames;

        repeat shoot() for 2 * framesPerSegment frames;

end
```

F.2 An Animaker Clip-Animation

```
define PlanetarySystem parent Actor
actors

        define SpaceBody parent Actor
        sequences

                private setUpModel
                script
                        external( _model );
                        _model :=
                                "sphere {" + CR +
                                "       <0, 0, 0>, 1" + CR +
                                "       pigment {" + _Surface + "}" + CR +
                                "       scale " + string( _Size ) + CR +
                                "}" + CR;
                end

                public SetBodySurface( filename )
                script
                        external ( _Surface );
                        _Surface := CR +
                                "               image_map {" + CR +
```

```
                "                   pict " + quote + filename + quote + CR +
                "                   map_type Sphere_Map" + CR +
                "              }" + CR +
                "              translate -(x+y)/2" + CR +
                "      ";
        setUpModel();
    end

    public SetBodySize( s )
    script
        external ( _Size );
        _Size = s;
        setUpModel();
    end

    public SetOrbitRadius( s )
    script
        external ( _OrbitRadius, _Size );
        if( _OrbitRadius > 4 * _Size )
            _OrbitRadius = s;
        else
            _OrbitRadius = 4 * _Size;
        end
    end

    public SetOrbitAngle( s )
    script
        external ( _OrbitAngle );
        _OrbitAngle = s;
    end

    public SetOrbitsPerSecond( s )
    script
        external ( _OrbitsPerSecond );
        _OrbitsPerSecond = s;
    end

    public Smile
    script
        extern( _Theta, _OrbitsPerSecond, _OrbitAngle );
        theX = _OrbitRadius * sin( _Theta ) * 1.2;
        theY = _OrbitRadius * cos( _Theta ) / 1.2;
        tell me to Rotate( [ 0, 0, _OrbitAngle ] );
        tell me to MoveTo( [ theX, theY, 0 ] );
        _Theta += _OrbitsPerSecond * 2 * pi / framerate;
        return( true );
    end

constructor
    _Size = 10;
    _Surface = "White";
    _OrbitRadius = 50;
    _OrbitAngle = -45;
    _OrbitsPerSecond = 1;
```

```
            _Theta = 0;
            setUpModel();

        destructor

        end

sequences

        public SetPlanetSurface( surface )
        script
            tell _Planet to SetBodySurface( surface );
        end

        public SetPlanetSize( size )
        script
            tell _Planet to SetBodySize( size );
        end

        public AddMoon( surface, size )
        script
            external( _MaxMoons, _NumberOfMoons, _Moon );
            if _NumberOfMoons <= _MaxMoons then
                create a SpaceBody named _Moon[ _NumberOfMoons ];
                tell _Moon[ _NumberOfMoons ] to SetBodySurface( surface );
                tell _Moon[ _NumberOfMoons ] to SetBodySize( size );
                tell _Moon[ _NumberOfMoons ] to
                    SetOrbitRadius( 4 * size + random * 10 );
                tell _Moon[ _NumberOfMoons ] to
                    SetOrbitAngle( random * 90 - 45 );
                tell _Moon[ _NumberOfMoons ] to
                    SetOrbitsPerSecond( random * 3 );
                _NumberOfMoons++;
            end;
        end

constructor
        create a SpaceBody named _Planet;
        tell _Planet to SetOrbitRadius( 0 );
        _MaxMoons = 10;
        _NumberOfMoons = 0;
        dimension _Moon: [ _MaxMoons ];

destructor
        discard _Planet;
        for i := 0 to _NumberOfMoons - 1 do
            discard _Moon[ i ];
end
```

VITA

Surname: **Darling** Given Names: **Glen Cameron**

Place of Birth: **Montréal, P.Q.**

Educational Institutions Attended:

University of Victoria 1975 to 1978
University of Victoria 1980 to 1984
University of Victoria 1990 to 1995

Degrees Awarded:

B.Sc. (Computer Science Major) University of Victoria, 1984

Honours and Awards:

Publications:

PARTIAL COPYRIGHT LICENSE

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis:

Using Conversational Syntax for 3D Modeled Animation

Author: [redacted]

GLEN CAMERON DARLING

Date: SEPT. 29, 1995