MIT/LCS/TR-358

PRIMITIVES FOR REAL-TIME ANIMATION IN THREE DIMENSIONS

Carol J. Chiang

Primitives for Real-Time Animation in Three Dimensions

by

Carol J. Chiang
B.S., Massachusetts Institute of Technology (1982)

**Submitted in partial fulfillment
of the requirements for the
degree of**

**Master Of Science
in Electrical Engineering and Computer Science**

at the

Massachusetts Institute of Technology

September 1985

Signature of Author ....Carol J. Chiang..........
Department of Electrical Engineering and Computer Science
September 6,1985

Certified by ....
Professor David K. Gifford
Thesis Supervisor

Accepted by ....
Chairman, Departmental Committee

Primitives for Real-Time Animation in Three Dimensions

by

Carol J. Chiang

Submitted to the
Department of Electrical Engineering and Computer Science
on September 6, 1985 in partial fulfillment of the requirements
for the Degree of Master of Science

Abstract

We present a general purpose imaging model which can efficiently produce computer-generated animated scenes. Displaying sophisticated graphics scenes is a computationally complex operation. Thus, an efficient imaging model is necessary for producing real-time motion. We provide a model, called the Animation Imaging Model, which lets an application programmer easily specify the transformation of objects over time. A clear separation between the application program and imaging system permits animation to be performed by special purpose processors. To demonstrate the feasibility of the proposed imaging model, a trial implementation has been devised using a Silicon Graphics IRIS Graphics System.

Thesis Supervisor: David K. Gifford
Title: Assistant Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to express my gratitude to all of the people who provided guidance and encouragement throughout my work on this project:

David Gifford, for advising the project and giving me guidance and deadlines insure that I would finish,

My family, especially my sister, Shirley, for providing advice and encouragement,

And to Carey Rappaport, John Wroclawski, and Steve Heller for the many patient hours they spent helping me with advice, encouragement, moral support, and understanding when I needed it the most.

Table of Contents

Table of Figures

Chapter One

Introduction

In the past few years, improvements in microprocessor and display terminal technology have caused an increased interest in 3-dimensional (3-D) color graphics. Much of the work in 3-D graphics has been devoted to hidden-surface elimination [24], shading [11], and anti-aliasing [7]. Recently, more research has been devoted to kinematics, or the addition of motion to graphic images. Although many people associate the term animation with cartoons and Walt Disney Productions, animation is actually the concept of adding motion to a still scene. Moreover, when dealing with 3-D images and scenes, animation adds another aspect of realism by providing the viewer with an enhanced understanding of the object's actual appearance. For example, the perception of depth is enhanced by motion, especially when conveyed by rotation about a vertical axis. The lines and surfaces which are closest to the viewer appear to move more rapidly than those at a distance [16].

Animation plays an important role in many situations. It has been applied to

- entertainment, for movie special effects, commercials, and cartoons,
- science and engineering, for motion analysis, computer-aided design, and computer-aided engineering, and
- education and training, for applications such as flight simulation.

For these applications, it is important to create realistic images in real-time. This is not a simple task; rendering a 3-D scene on a 2-D display involves perspective or orthographic projection and other complicated tasks.

1.1 Previous Research in Animation

Past research in kinematics includes the areas of film making and the generation of facial expressions. Such work has been done by Frederic Ira Parke [18], Richard Chuang and Glenn Entis [4], and Kenneth Kahn [14]. Parke's work concentrated on computer-generated animation of faces. The skin of the face was represented by 3-D polygons. This data was stored as the sets of points which defined the polygons. To allow a face to change expression, data files were created for a number of different facial expressions. Moving each point of skin a small distance in successive frames would cause a change in facial expression. Given a previous expression and a next expression, an animation program would generate the intermediate facial expressions using cosine interpolation on the positions of each point.

Although the application was different, Chuang and Entis [4] used similar methods to create their motion design system. The system was designed to help the director and animator choreograph the photographic image and graphic objects for commercial film use. In this system, the animator specifies the camera's corresponding view, focal length, and direction of view for each actor. Key frame positions for the camera and actor are then processed by computer to generate the intermediate frames.

Also related to film making was Kahn's work on the Ani computer system [14]. Ani uses high-level film descriptions to create an abstract animated film of Cinderella. The animation is accomplished solely by choosing the locations and movements of the characters. Decisions, such as the speed, direction, or distance a character is moved, are based on suggestions constructed from aspects of the film or from scene and character descriptions and relationships.

While the above methods are adequate for some applications, they are not necessarily appropriate for uses which require moving realistic images in real-time. For each of these projects, the animation aspect is used in conjunction with a specific application:

either film making; or expression generation. Integrating the application with the creation of animated scenes *a priori* is impractical as a general scheme for generating real-time animation. Furthermore, the first two examples store a large number of images for intermediate scene generation, which may require an excessive amount of computer memory. Ani, on the other hand, provides only abstract images.

1.2 Thesis Goals

The purpose of this thesis is to produce a new imaging model which includes primitives for specifying the transformation of objects over time. This model allows the applications programmer to easily design a scene which includes animation. A clear separation between the application program and imaging system permits animation to be performed by special purpose processors. Once the scene has been designed, the graphics subsystem automatically brings about the animation. The imaging model lets individual objects be moved without the need to recompute the position of nonmoving objects and background objects for every frame, making the process of animating scenes much faster. Hardware is now becoming available which permits auxiliary processors to perform the tasks of computing the coordinates of objects for displaying a scene, thus freeing the host processor to do other jobs. In this thesis, the tasks in the system are distributed so that an auxiliary graphics processor performs the animation tasks.

1.3 Road Map of Chapters to follow

The following is a brief overview of the remaining chapters of the thesis. Chapter 2 describes existing imaging models and a new imaging model which includes animation. Chapter 3 describes a typical graphics system architecture and how it may be modified to incorporate animation. This is followed by observations regarding implementation, in Chapter 4, and in Chapter 5, the conclusions mention some possible extensions to the work done as well as some suggestions for future uses of the system.

Chapter Two

Imaging Models and Animation

2.1 Imaging Models

A graphics imaging model is a device independent set of primitives which allow a programmer to create a desired scene [25]. The model specifies how colors, lines, surfaces, geometric properties, etc are combined to form images. Complex scenes can then be created by repeatedly applying varied parameters to the primitives which specify the images. The two primary types of 3-D imaging models are solid modeling and polygon modeling.

Solid modeling can create very realistic images by combining primitive shapes such as spheres and cubes. A common method used for performing solid modeling is raytracing. Raytracing can realistically shade 3-D objects in the presence of multiple light sources, as well as produce reflections off shiny surfaces and refractions through transparent objects. By tracing the path of light rays, those rays which pass through the viewpoint can be determined [9]. An example of work employing this method of image generation is a 3-D visual simulation technique developed at MAGI[1] [10]. This technique simulates the photographic process by creating an image which closely resembles that of an object as it would appear to an imaginary photographer and camera. By using raytracing, the transmission and reflection of light can be simulated with a large degree of realism. The resulting images are of very high quality. Raytracing, however, is very computationally intensive and works most efficiently on larger computer systems. It has not yet generally been applied to real-time applications.

[1] MAGI stands for Mathematics Applications Group, Inc.

Unlike solid modeling, which works in the 3-D world directly, polygon modeling approximates 3-D surfaces by using 2-D shapes. Objects are described as collections of points, vectors, and polygons. These objects are assembled into a *display list* [16] by a program to create a desired scene. A display list consists of a sequence of graphics commands used to create and manipulate objects within scenes. A graphics interpreter scans the display list and transfers graphics commands to the display hardware for processing. Polygonal systems are less computationally intensive than raytracing systems, and are therefore more suitable for real-time applications.

2.2 A New Imaging Model - The Animation Imaging Model

Since real-time animation algorithms must produce scenes at the rate of 20 to 30 frames per second, applications involving real-time animation require an imaging model which specifies animation in such a way that the program does not have to recompute the position of all the components of the scene every frame time. Thus, polygon modeling, implemented with display lists, was chosen as the basis for developing a new imaging model, hereby referred to as the *animation imaging model.* This model provides primitives which allow the user to specify how objects should be transformed over time to create the desired animation effects.

2.2.1 Building a Scene

There are three building blocks for 3-D geometry – points, vectors, and polygons. As described above for conventional polygon schemes, objects are described by commands involving these building blocks, possibly in conjunction with other graphics commands, and are stored as a display list. An object stored in a secondary display list acts like a subroutine in a program; although the commands defining the object are stored only once, the object can be called multiple times for display. Likewise, objects can also contain calls to other objects. The main display list therefore consists of pointers to objects, interspersed with other graphics commands. This is illustrated in Figure 2-1. Note that `callobj` *door* instantiates an instance of the object, *door.*

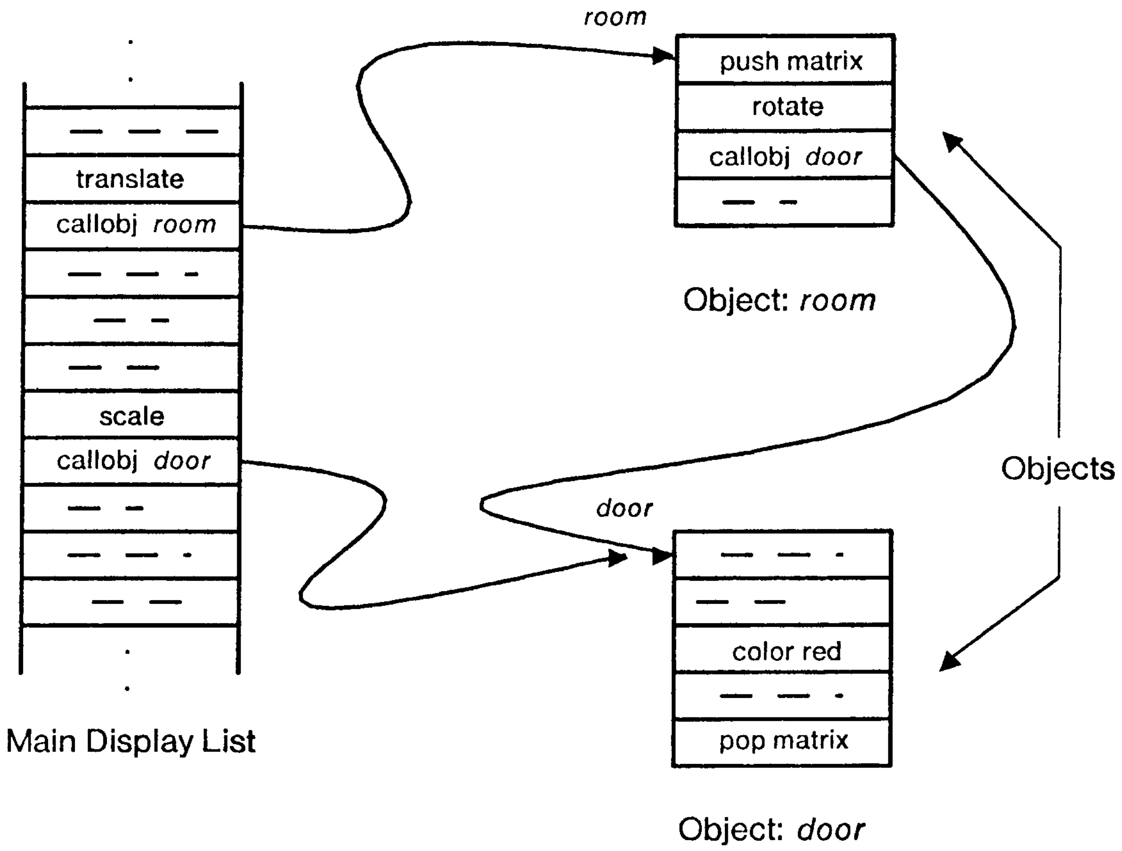


Figure 2-1: Display List with Pointers to Objects

There are many different types of graphics commands, including transformations, such as rotations, translations, and scaling, as well as attributes specifying color and texture, which can be applied to both the primitive building blocks and objects. Although multiple instantiations of an object can occur, the transformations and properties applied to one instance of the object are independent of those applied to another instance of the same object. The transformations allow objects to appear on the screen in various locations, sizes, and orientations, thus producing a scene.

In the case of animation, it is necessary to save the contents of the entire scene in a display list because the scene must be redrawn whenever a transformation is performed. The object or objects then move according to this transformation. To facilitate the repainting of the entire scene, an object, referred to as the *MainObject*, encompasses the entire scene. Since objects may be part of other objects, all objects and graphics commands are ultimately part of *MainObject*. Repeated instantiations of *MainObject* by the graphics subsystem reflects the movement in the animated objects. Use of this *MainObject* concept is a requirement for the animation imaging model, distinguishing it from previous schemes.

2.2.2 Transformation Mechanics

Several different types of transformations affect the displaying of a scene on a screen [16]. Modeling transformations directly affect the object by changing its position, size, and orientation. Viewing transformations determine the viewpoint from which the scene is observed. Projection transformations affect the method by which the 3-D scene is projected onto the 2-D screen for observation. The composition of these operations provide the total transformation which maps the object's 3-dimensional coordinates to 2-dimensional screen coordinates.

For the animation imaging model, a convenient representation for 3-D transformations is a 4×4 matrix. All 3-dimensional transformations can be represented as such. Furthermore, any composite transformation can be represented as a concatenation (multiplication) of individual simple transformation matrices [1]. Thus, a single matrix can represent a composite transformation. For example, a scaling transformation, represented by a matrix S, followed by a translation, represented by the matrix T, will have an equivalent matrix associated with it which equals the product of the matrices, ST, assuming that a resultant matrix is postmultiplied by any subsequent transformation matrices. Most modeling, viewing, and projection transformations are compositions of the three basic types of matrices given in Appendix A.

The 4×4 matrix representation for transformations requires that points be represented by the first 3 elements of a 1×4 matrix, [x y z 1], with the fourth element, always set to 1, which is necessary for translation. The transformed point is the result of a postmultiplication with a 4×4 transformation matrix.

2.2.3 Points of View

Scenes can be designed from either the point of view of the observer or the point of view of the object. With the latter, each object in the scene supports its own fixed coordinate system, centered at the object's origin. When a transformation is applied to an object, the object moves relative to its own coordinate system. The animation imaging model assumes that scenes are generated from the point of view of the object.

2.2.4 Types of Motion

Several types of motion are often observed in animated scenes. These consist of movement with constant velocity, growth and shrinking, spinning, rolling, spiraling, and acceleration. Each of these motions, except acceleration, can be represented by a matrix, M, which is concatenated with itself at constant time intervals of Δt. In other words, from time = 0 until Δt, M is applied to the object. From time = Δt to $2\Delta t$, M^2 is applied, from time = $2\Delta t$ to $3\Delta t$, M^3 is applied and in general, from time = $\mathbf{n}\Delta t$ to $(\mathbf{n}+1)\Delta t$, where $\mathbf{n}$ is the number of elapsed Δt's, $M^{(\mathrm{n}+1)}$ is applied. Moreover, at any time just after $\mathbf{n}\Delta t$, $p = p_0 M^{\mathrm{n}+1}$ where p_0 is the original point and p is the transformed point.

Once the method of motion representation has been chosen, other details such as speed and smoothness, should be examined. The speed at which the object moves is determined by both M and Δt; by changing either parameter, the object can easily be sped up or slowed down. Using the case of translation as an example, smoother motion can be achieved through the use of a smaller Δt and translating a smaller distance each

Δt, thus increasing resolution. It will, however, involve more work on the part of the processor than the converse implementation because the motion occurs more often. Unfortunately, in the case of a fast moving object, if the object is not moving quickly enough with $\Delta t = 1$ (the smallest time increment), the user may have to resort to translating at larger increments.

Use of the basic modeling transformations - translate, rotate, or scale - to represent *M*, produces the types of motion previously mentioned. Translation produces linear motion in any direction and is independent of the point of view and coordinate system origin but dependent upon coordinate orientation. Rotation is circular motion about an axis and thus is strongly dependent on the coordinate system and point of view. Scaling is uniform growth or shrinking in all directions and is also independent of the coordinate system. Since rotation does depend on the coordinate system, rotation followed by a translation is very different from translation followed by rotation. The former yields rolling motion while the latter produces spiraling motion.

Straight line acceleration, on the other hand, can not be specified in such simple terms. The displacement, s, of the object from its position, s_0, at time $= 0$, is described by the formula $s = s_0 + v_0 t + ½at^2$ where v_0 is the initial velocity, t is the elapsed time and a is the desired acceleration. The transformation matrix which exhibits this behavior is a translation matrix which requires the translation parameters to be computed by the given formula after each time interval. The animation imaging model will not support motion represented in this manner.

2.2.5 Animation Environment

As indicated in the previous section, several common types of animation can be described by two user-specified pieces of information – a single *animation matrix*, *M*, and a time interval, Δt. A third parameter, the matrix, *A*, initially equals *M* but is later updated to contain the most recent composition of M with A. A is initially M, then it becomes M^2, followed by M^3, etc. A is updated automatically at Δt intervals by the graphics subsystem. Applying A to an object will animate that object. Thus, a feasible method of animating a scene, or objects within a scene, is to create an *animation environment* such that all objects within the environment are transformed by A, now called the *animation environment matrix*, which is the most recently updated version of the A (M^2, M^3, etc) associated with that animation environment. The display list is annotated with information pertinent to the kinematic behavior specified by the user using animation primitives, which are described in a later section. These primitives allow the user to define M, the transformation matrix corresponding to the specified type of motion, Δt, time interval between the updates of the transformation matrix, and the name of the animation environment. The display list only contains an application of the newest A. The M and Δt associated with each environment is stored in the graphics subsystems in an appropriate data structure.

Two commands, a **BeginAniEnv** command and an **EndAniEnv** command, define the the boundaries of a scene which will be animated by an animation environment. Insuring that the **EndAniEnv** command ends only the scene which was begun most recently allows nested animation environments.

2.2.5.1 Nested Animation Environments

The concept of nested environments implies that one animation environment can be defined within another animation environment. These environments will be referred to as the child and parent environments respectively. Three possible configurations depicting parent/child environment relationships are shown in Figure 2-2.

Figures 2-2a and 2-2b are examples in which the child environments are explicitly defined within the same parent environment. If the time intervals of the parent and

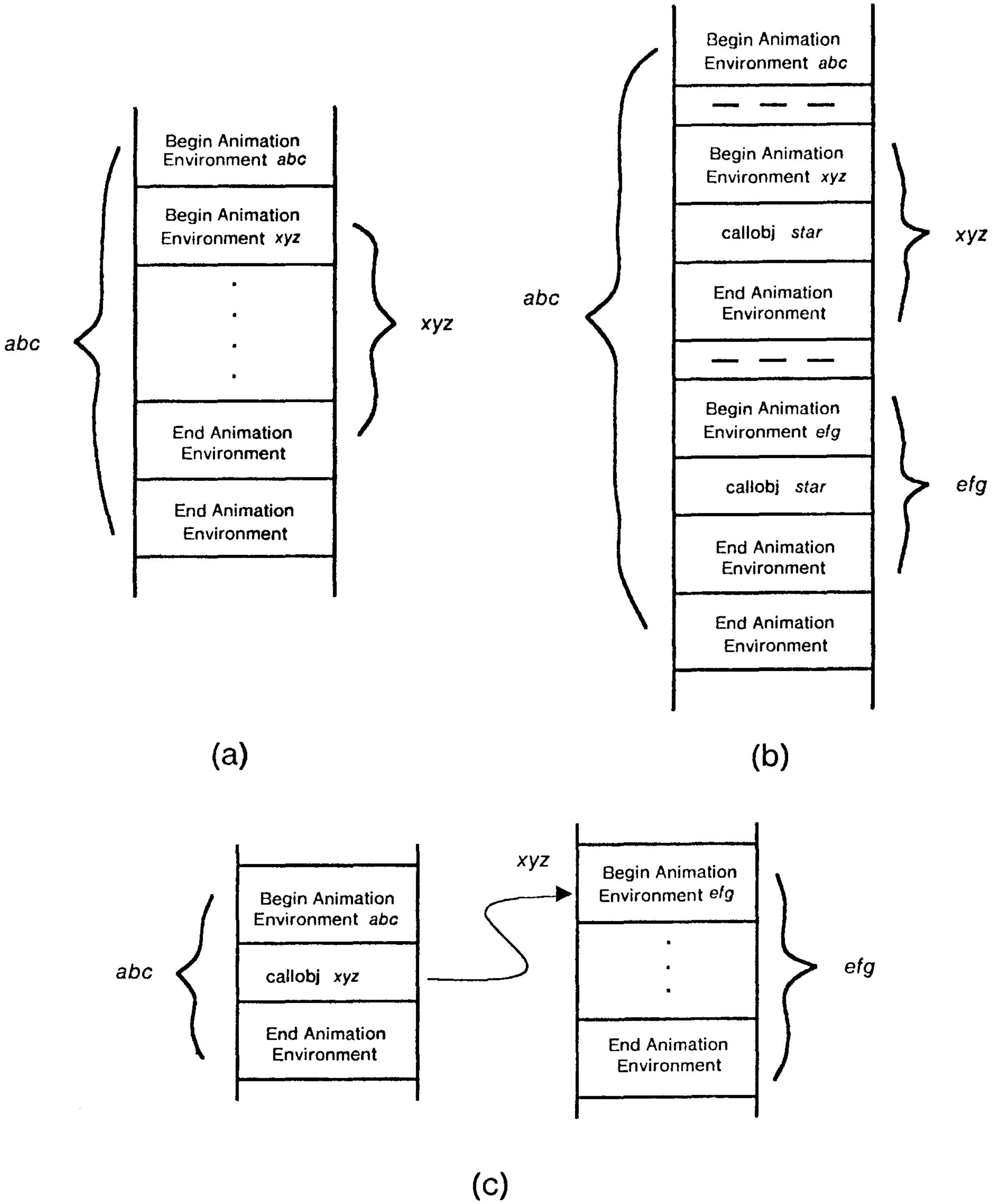

Begin Animation Environment abc
Begin Animation Environment xyz
End Animation Environment
End Animation Environment
abc
xyz
(a)
Begin Animation Environment abc
Begin Animation Environment xyz
callobj star
End Animation Environment
Begin Animation Environment efg
callobj star
End Animation Environment
End Animation Environment
abc
xyz
efg
(b)
Begin Animation Environment abc
callobj xyz
End Animation Environment
abc
xyz
Begin Animation Environment efg
End Animation Environment
efg
(c)

Figure 2-2: Examples of Nested Animation Environments

child environments are equal, 2-2a can actually be thought of as one equivalent animation environment which has a composite animation environment matrix, A_{comp}, represented by $M^{n}_{xyz}M^{n}_{abc}$ at time $n\Delta t$ (where M_{xyz} and M_{abc} are the animation matrices for the two environments. The advantage of using multiple environments in this configuration is apparent if separate modification to the environments is desired; the Δt or M of one environment can be changed while the other environments are left untouched.

In example 2-2b, the two child environments, *xyz* and *efg* are used to animate separate objects within the parent environment, *abc*. This idea illustrates that objects can exist simultaneously within the same animation environment and also be animated independently. The child environments have no effect on each other because one environment ends before the next begins; nevertheless, both are affected by the parent environment. At any given time, the effective animation environment matrix is equal to the composition of all the children and parent environments which have not ended. Thus, the following scenario is possible:

> Two stars are rotating in opposite directions while
> the entire scene is moving at a constant velocity across
> the screen.

The third example of nested environments, shown in 2-2c, depicts an animation environment, *abc*, which contains a call to the object, *xyz*. This object, *xyz*, contains an animation environment, *efg*, in its definition. Thus, the parts of *xyz* that are within *efg* are affected by both.animation environments. Nestings of this type let the applications programmer easily animate hierarchical objects.

2.2.6 Basic Primitives

Basic primitives are used to create the display list. These primitives correspond very closely to those found in traditional implementations of polygon modeling. A short list describing some relevant primitives is given.

- PUSHMATRIX/POPMATRIX – This implementation maintains a stack of transformation matrices such that the top of the stack matrix, **TOS**, is directly applied to the objects. All transformations affect the **TOS**. PUSHMATRIX and POPMATRIX are used to preserve the state of **TOS** so that transformations occurring within the PUSHMATRIX/POPMATRIX do not affect the objects drawn after the POPMATRIX.

- MAKEOBJ(*obj*)/CLOSEOBJ – MAKEOBJ and CLOSEOBJ define the starting and ending points of the object, *obj*, which is represented by a unique integer, generated by the system. The object definition consists of all graphics commands enclosed by these two commands.

- CALLOBJ(*obj*) – CALLOBJ instantiates an instance of the object, *obj*.

- MULTMATRIX(*m*) – MULTMATRIX premultiplies **TOS** by *m* and stores the resulting product matrix in **TOS**. The previous contents of **TOS** are destroyed.

- EDITOBJ(*obj*) – EDITOBJ opens the object, *obj*, for editing. It should subsequently be closed by CLOSEOBJ.

- TAG(*obj-tag*) – TAG labels the next command as *obj-tag*, which is also a unique system-generated integer. TAG provides a reference point within an object definition from which any subsequent graphics commands can be accessed.

- REPLACE(*obj-tag,offset*) – REPLACE allows commands within an object definition to be replaced. The object must already have been opened for editing by an EDITOBJ(obj). Then all commands within *obj* starting from the *obj-tag* position + *offset* are replaced by the commands following the REPLACE(*obj-tag,offset*) until another editing command or a CLOSEOBJ occurs.

2.2.7 Animation Primitives

The usefulness of the animation imaging model is determined by the flexibility it provides for programmers to specify animation. The animation primitives are used in conjunction with the basic primitives to describe the trajectories which will provide the desired animation effects. The animation primitives provided by the animation imaging model can be divided into two groups, primitives which directly affect the display list and primitives which simply access and manipulate the data structure storing the animation environment information. **BeginAniEnv**, **EndAniEnv**, and **StartAniProc** are part of the former. The remaining primitives, which are also defined below, fall into the latter grouping.

2.2.7.1 BeginAniEnv(*name*)/EndAniEnv()

An animation environment, *name*, is created by an **Animate** command which must be issued before the animation environment will cause any motion. **BeginAniEnv** and **EndAniEnv** define the boundaries for a part of the total scene which will be animated by the animation environment *name*. All graphics commands which appear after a **BeginAniEnv** and before the **EndAniEnv** are animated by *name*. Several portions of the total scene may refer to the same animation environment. If *name* is not a defined animation environment, the processor simply saves the state of the transformation matrix stack upon entering the environment and restores the state upon leaving it. A sample use of the animation environment as well as the resulting display list is shown in Figure 2-3.

2.2.7.2 StartAniProc(*MainObject*)

This primitive starts the process which animates the *MainObject*.

2.2.7.3 Animate(*name*,Δt,*M*)

Animate creates an animation environment. Its three arguments are the name of the environment, *name*, which is a unique system-generated integer, the time interval, Δt, and the animation matrix, *M*. **Animate** can also be used to modify the parameters of an existing animation environment. The initial matrix, *M*, is also used to initialize the

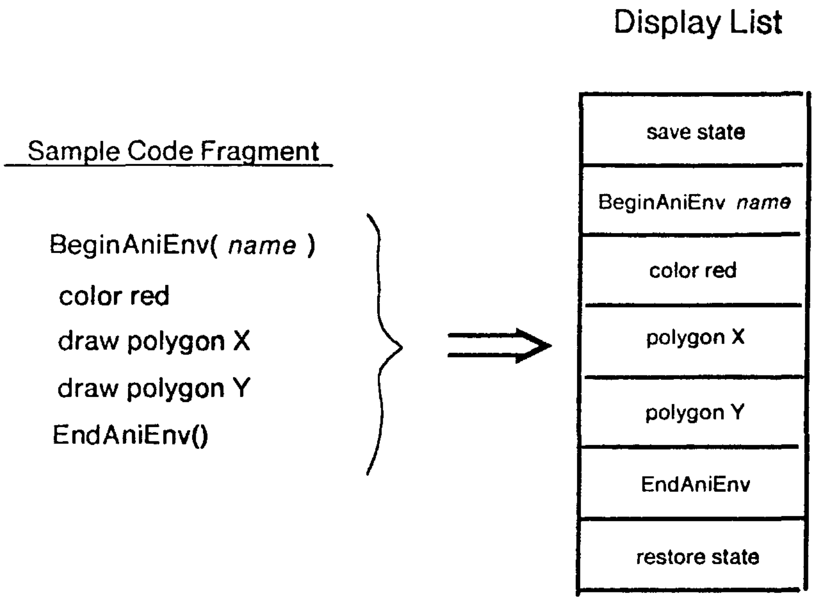


Figure 2-3: Sample Code Fragment and Resulting Display List

animation environment matrix, A, and therefore contributes to the initial location of the objects within the environment. When the specified time interval, Δt has elapsed, A is updated by multiplying it by M. This is followed by a call to the *MainObject* which will reflect the matrix change. Thus, the subsequent concatenation of matrices accompanied by calls to the modified *MainObject* will cause the observer to perceive motion. Note that **Animate** creates the animation environment, but does not apply it to any part of the scene. This is accomplished by **BeginAniEnv** and **EndAniEnv.**

Requiring that the animation be described by a matrix rather than a description of the motion, such as *rotate 90° "X"*, can be considered less user friendly. However, this method is more powerful because it allows the use of user-defined transformations and composite transformations such as the transformation matrix which describes rolling.

2.2.7.4 GetAniTime(*name*)/GetAniMatrix(*name*)

These routines allow the user to obtain the parameters of an animation list entry. For an animation environment, *name*, **GetAniTime** and **GetAniMatrix** return the values of Δt and M respectively. This allows the user to modify the animation environment relative to its current state. If, for example, the user wants to double the speed at which an object was moving, he could get the structure and either divide the Δt by two or concatenate M onto itself so that $M \leftarrow M^2$. These parameters could then be used to issue a new **Animate** command for *name.*

2.2.7.5 HaltAnimation(*name*)

HaltAnimation halts the motion of the animation environment, *name*, leaving the parts of the scene manipulated by *name* in the position last specified by A_{name}, which is the animation environment matrix associated with *name.* The animation can be restarted by issuing a new **Animate** command for that environment.

2.2.7.6 KillAnimation(*name*)

KillAnimation stops the motion for the animation environment, *name.* The system no longer realizes that *name* exists, but since the display list has been altered, it places the portion of the scene within *name* in the same position as it would be in if a **HaltAnimation** command had been issued. Restarting animation for an environment which has been killed requires the issuing of new **Animate**, **BeginAniEnv**, and **EndAniEnv** primitives.

Chapter Three

Implementation of the Animation Imaging Model

3.1 Graphics System Architecture

A typical architecture which represents most 3-D graphics systems [20] can be described in terms of the block diagram in Figure 3-1. The main components of such systems are the program, display list, geometric processor, display processor, frame buffer, and display. The program provides the means by which the user accesses the other components of the graphics system. It uses the operating system, file maintenance utilities, graphics library, and a network or bus to communicate with any external components. The applications programmer uses the program to create a display list. The display list stores the graphics scene the user wishes to display. It stores the graphics objects as well as the commands responsible for their manipulation. This collection of graphics commands is then fed to the geometric processor, which performs the point, vector, and polygon transformations as well as the clipping operations [22]. Sophisticated geometric processors contain special purpose hardware for mathematical calculations. The display processor then receives the processed frame of data. The objects are rendered and converted to screen coordinates. Techniques for scene enhancement, such as hidden surface elimination, shading, and anti-aliasing, can be applied by the display processor to aid in the rendering of realistic objects. The resulting pixels are stored in the frame buffer where it is accessed by the display. The display can actually be considered a controller and a video monitor or Cathode Ray Tube (CRT). The controller reads the data from the frame buffer and dumps it to the CRT screen. The controller is also responsible for refreshing the screen.

The components of the system are often divided into two physical boxes, a host

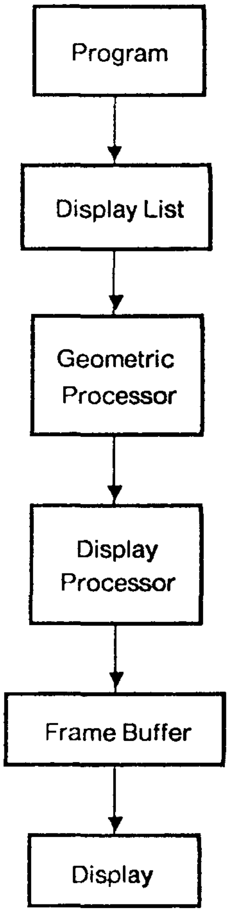


Figure 3-1: Architecture of a Typical 3-D Graphics System

processor and a graphics box, connected by a communication channel. The selection of components for each box depends on the implementation of the system. One common division is between the frame buffer and the display, as shown in Figure 3-2. The graphics box contains the display while the host processor handles everything else. This configuration is a typical graphics system for personal computers [15]. The remaining graphics hardware and software is completely integrated into one box. The system uses a memory mapped display; part of the microprocessor's memory is used to store the pixels and thus emulates a frame buffer. The memory is dual ported so that both the display and the microprocessor can access it. The microprocessor does all the work, including the storage of the pixels. The display is a CRT with a controller chip which reads the pixels out of memory, converts them to raster signals, and refreshes the screen. For graphics applications, many of the microcomputer displays have inherent limitations of display resolution and color graphics capabilities [2].

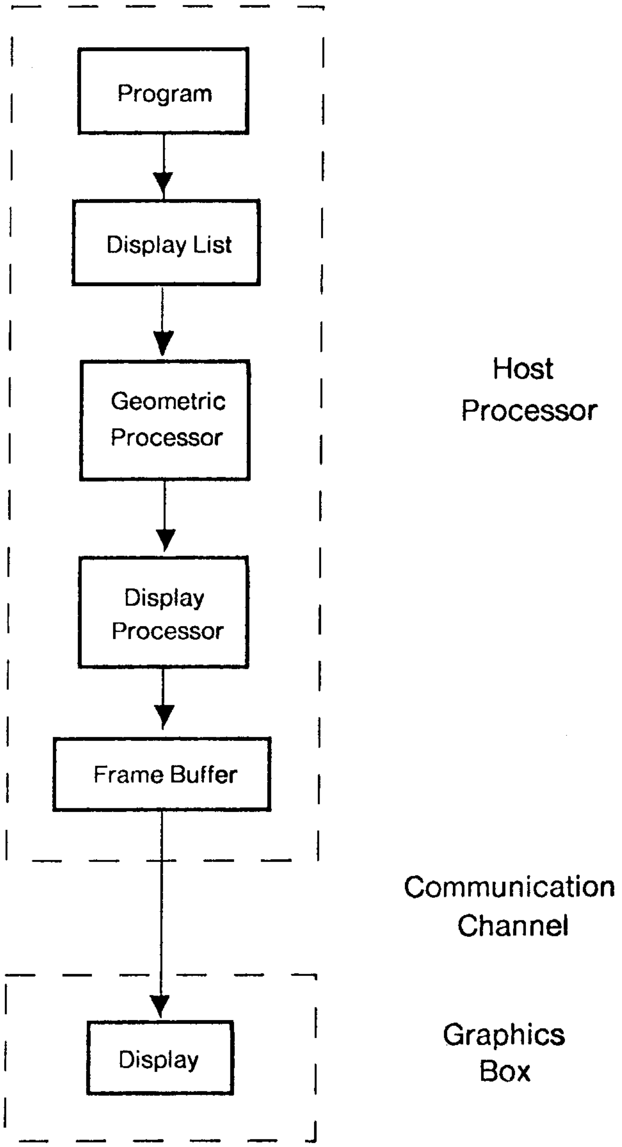


Figure 3-2: Architecture of a Typical Microcomputer-Based 3-D Graphics System

Other existing configurations involve the integration of more components in the graphics box. An example of this is an arrangement in which the display list, composed of a manipulation and storage portion, is divided between the boxes. The host maintains the program and the display list manipulation components while the graphics box has the display list storage module as well as all the successive components. The architecture is shown in Figure 3-3. An example of this type of system is the Silicon Graphics IRIS[2] Graphics System [6]. It uses a Motorola 68000 microprocessor-based graphics processor to control the components of the graphics box. The geometric processor contains special purpose hardware in the form of custom VLSI chips [5]. Among other uses, this hardware handles mathematical computations, floating point transformations, and clipping operations. The special purpose hardware makes this system appropriate for real-time graphics applications. While this thesis develops a general purpose framework for specifying animation, a specific implementation is necessary to prove the feasibility of the idea. The framework described in this thesis has been implemented on an IRIS Graphics System.

3.1.1 Minimizing Processing Bottlenecks

While all of the described configurations are acceptable systems for specifying motionless scenes, the architectures are not appropriate for complex real-time animation applications. In the microcomputer example, speed and memory usage are important considerations when trying to use a microcomputer to produce animation. Limited applications for real-time animation can be achieved by writing the programs in assembly language [17], thus enabling the programs to run faster and occupy less memory.

On more powerful systems, the trend of development has been to implement

[2] IRIS stands for Integrated Raster Imaging System.

In spite of this additional speed advantage, these systems still can have considerable bottlenecks which are not usually noticeable when generating motionless graphics scenes using polygon animation model techniques. The bottlenecks in this arrangement are the communication channel between the host processor and the graphics box, as well as the host's inability to provide real-time processing power. For a motionless scene, the object manipulation portion of the display list module does not significantly affect system performance. The user creates the description of the scene on the host using imaging model primitives, and the data is transferred to the graphics box. From then on, the only responsibility of the graphics box is to refresh the screen. The network data transfer takes place only when a new scene is displayed, and the host only has to compute the object positions once. Thus, the bottlenecks have a negligible effect on the performance of the system.

However, the use of this architecture to create images exhibiting kinematic behavior places the burden of animation entirely on the host processor. The host processor must supply enough processing power to the calculate new positions and orientations for the animated objects every 1/20 - 1/30 of a second. Whether or not the host can accommodate this demand on processing power depends on the system load. The following example, in the C programming language, illustrates a box rotating 10 degrees about the z-axis each time through the loop.

```
for (i = 0, i < 1000, i++) {
        clear  screen
        rotate  10° about  the  z-axis
        draw  box
    }
```

The smoothness of the motion is determined by the consistency with which the host can supply the processing power. The actual rate of animation is bounded by the execution time of the **for** loop and the speed at which the new information can be transferred to the graphics box. In addition to rotating and drawing the box, the screen must be cleared each time to erase previously drawn boxes. The host must continuously transfer this data to the the graphics box, so the network connection can also become a significant bottleneck, especially when a high-speed data link is not used to transfer the animation information. This burden on the host processor is not necessary when the task can be accomplished in hardware through a modification of the existing architecture. If the mathematical calculations are computed in hardware in the graphics box, the computationally intensive tasks are handled more efficiently, and the data link will not become bogged down.

3.2 Architecture Supporting Animation

The effect of the limited bandwidth between the host processor and the graphics box as well as the host's inability to consistently supply real-time processing power can be alleviated by creating a new imaging model which includes animation, as described in Section 2.1. A modification to the architecture of Figure 3-3 is necessary for processing the animation specified by the imaging model. The change involves the addition of an animation module to the graphics box. The new architecture is illustrated in Figure 3-4. The animation module receives instructions from the host processor which describe the trajectories and speeds of the objects. This information need be sent only when the process is started or when changes in the animation instructions are desired. The large quantities of data produced by the animation module can now be processed more efficiently. Moreover, for many applications, the entire display list component can be included in the graphics box. The net effect is to transfer the burden of display list manipulation from the host to the graphics box where it can be accomplished more efficiently using hardware. The system's dependence on the communication channel has also been reduced significantly.

Furthermore, if the host is a time-shared computer system, the speed, smoothness and consistency of the animation is greatly dependent on the ever-changing system load. A heavily loaded system executes much more slowly than a lightly loaded system. Forcing

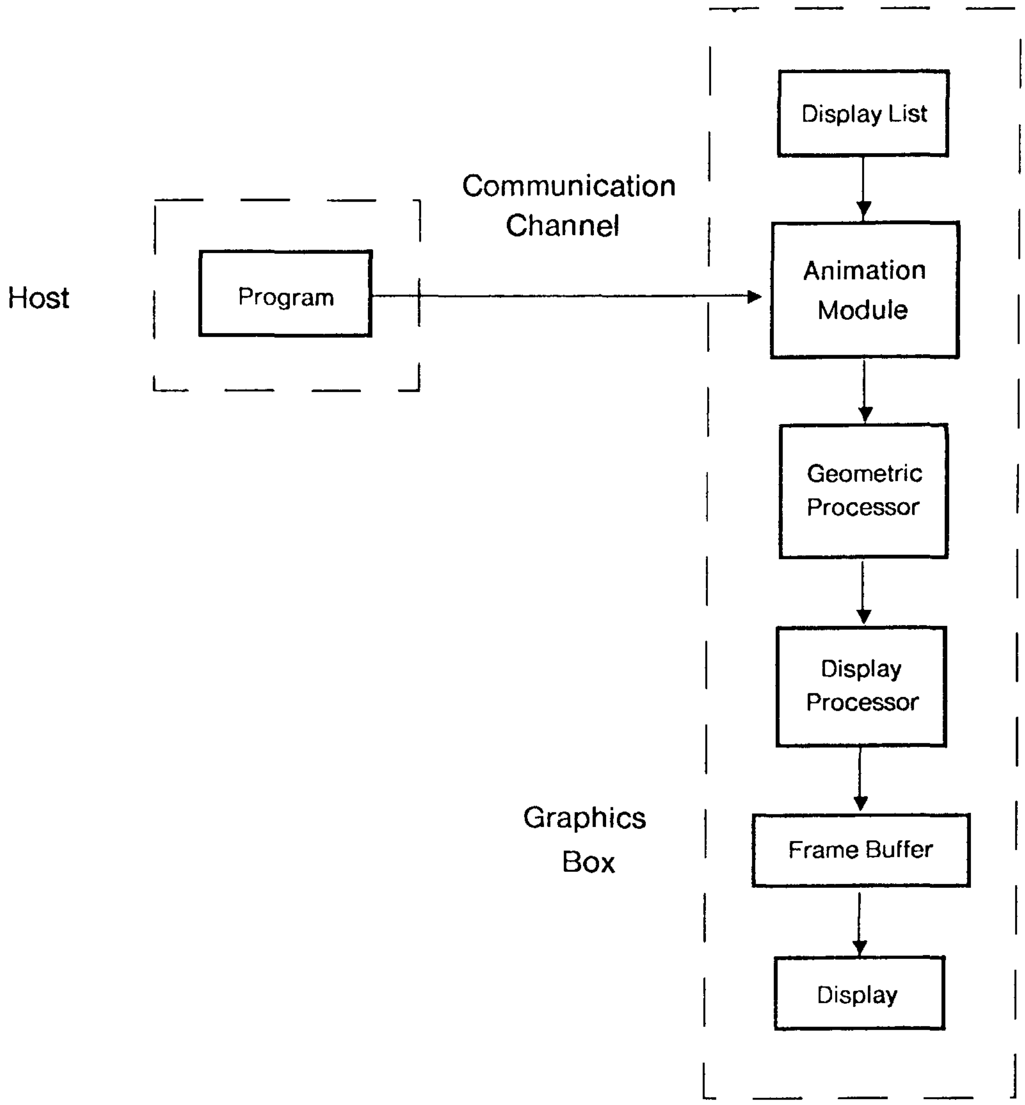


Figure 3-4: Architecture Modified to Incorporate Animation

the host to do the animation puts an even greater load on the system. The modified architecture will eliminate this undesirable dependence on the load of the host by using special purpose processors to provide the computing power for the animation. The new architecture lets the host's processing power be applied to other tasks.

3.3 The IRIS Graphics System

As mentioned in a previous section, the animation imaging model was implemented on an IRIS Graphics System. Since the goal of this project was to design an imaging model which allows the user to specify complex animated scenes rather than just motionless scenes, the IRIS was an appropriate system to use for the implementation. Animated scenes involve the application of large numbers of transformations to objects. Each type of animation (where type is defined by the pattern of motion and speed) requires a separate animation environment. As the number of types increase, the scene increases in visual as well as computational complexity. This implies that animation requires a great deal of matrix multiplication, a computationally intensive task for real-time applications. Furthermore, it emphasizes the need for a new imaging model which efficiently handles animation by remembering the positions of motionless objects from frame to frame.

For this specific implementation the special purpose IRIS hardware can be used to compute the matrix calculations. A more detailed illustration of the IRIS Graphics System [23] is shown in Figure 3-5. From the host side, the applications programmer has access to the graphics library. This graphics library contains routines such as point, line, and polygon drawing primitives; modeling, viewing, and projection transformations; routines for defining custom colors, textures, and fonts; and routines for object creation, modification, and deletion. These routines can be called from several high-level languages. The host uses a byte stream to transfer the graphics program to the graphics box.

As shown in Figure 3-5, this byte stream is received at the graphics box by the protocol reader, which sends it to the graphics library compiler. In the IRIS graphics library compiler, a routine or macro exists for each graphics library command found in the host's graphics library. The compiler uses these routines to build a display list. The

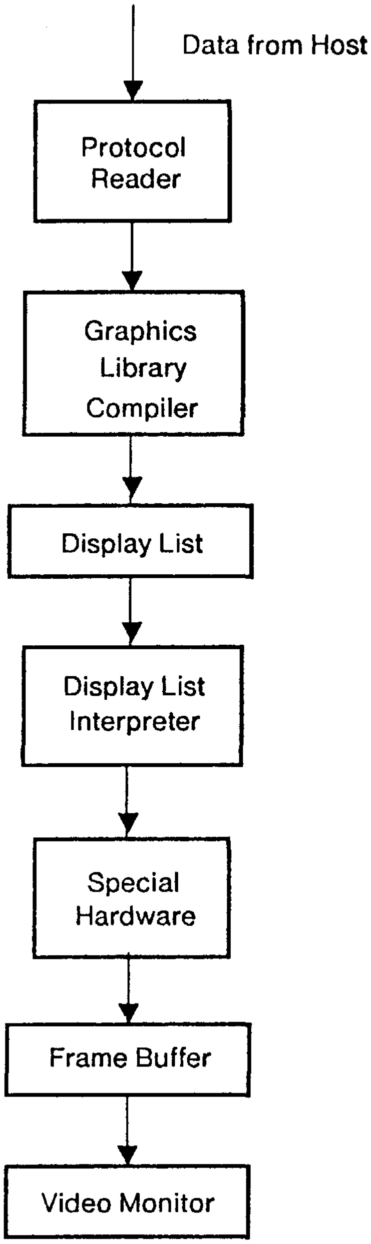


Figure 3-5:IRIS in More Detail

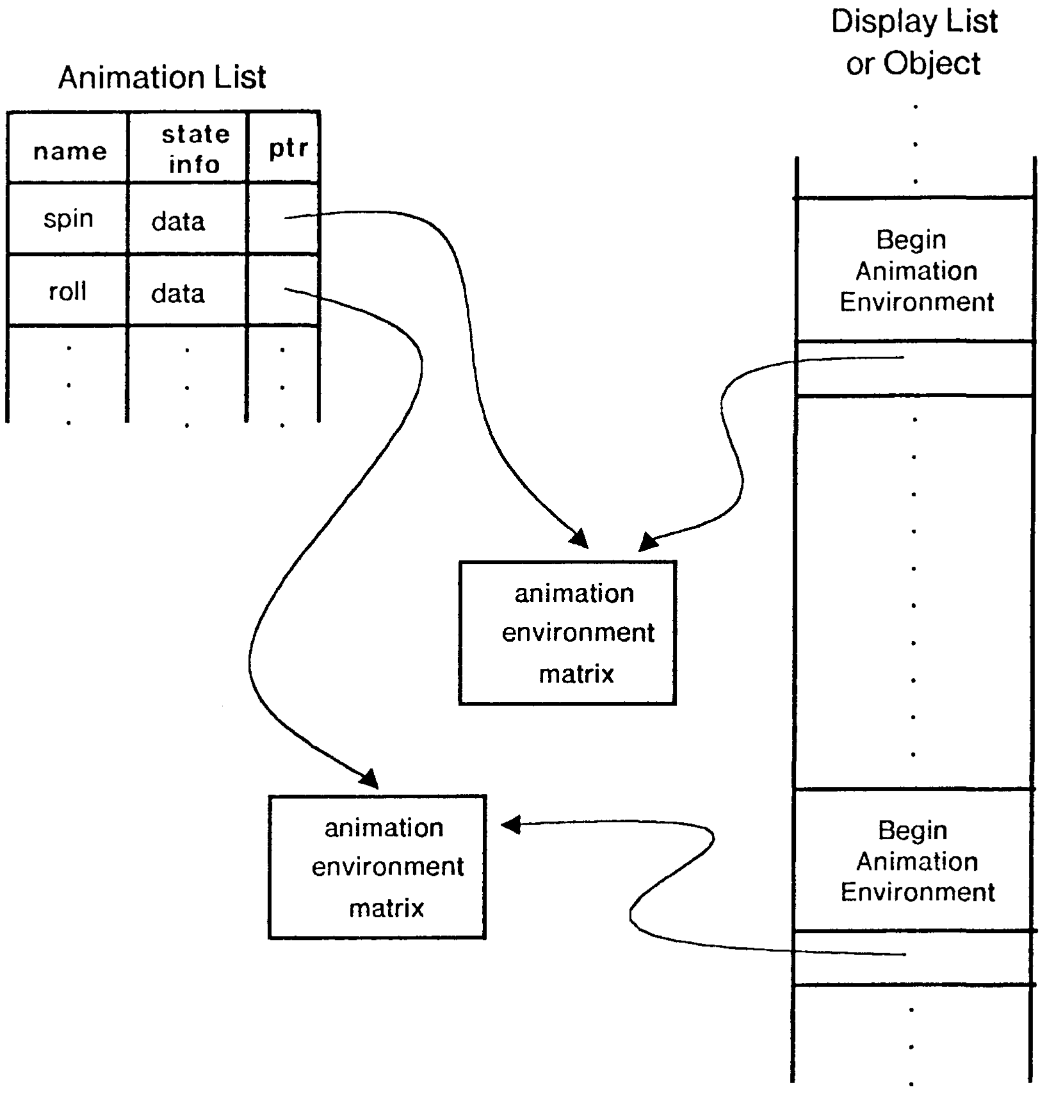
interpreter reads the commands from the display list and and translates them into a format which is suitable as input to the special purpose hardware. The hardware processes the computationally intensive calculations needed to render the objects. The resulting pixels are stored in a frame buffer and subsequently displayed on the screen.

3.3.1 Adding Animation Capabilities to the IRIS

The method chosen for implementing the animation imaging model on the graphics system requires the integration of an animation model into the existing architecture. This module creates a process which modifies the matrix for each animation environment, places each new matrix in the display list, and asks the interpreter to rescan the display list. The approach also requires double buffering [20], whereby the interpreter writes to one buffer, while the screen displays the contents of the other buffer. The buffers are swapped during the first vertical retrace period following the completion of the interpreter's scanning of the display list. Thus, the new image is created invisibly while the old one is viewed on the screen.

Our animation algorithm employs a list, the *animation list*, which maintains the state information for each animation environment. When the state information changes, the change must be reflected in the display list. This is accomplished by letting the animation list and the display list interpreter share each animation environment matrix, as shown in Figure 3-6. Both have pointers to the memory locations which are storing the matrices. The animation process accesses the animation environment matrices from the animation list, updates them, and then places the updated values in the shared memory locations. The next time the interpreter scans the display list, it applies the new matrices to the objects in the scene. Thus, the animation list always writes into the shared memory locations, and the display list interpreter always reads from them. Note that the animation environment matrices cannot be stored directly inside the display list because the display list memory can be reallocated. If reallocation occurred, the animation list would lose track of the location of the matrices and therefore, could not modify them.

The block diagram of the modified IRIS graphics system is found in Figure 3-7. One advantage of this approach is that it can be implemented using existing routines in the graphics library compiler. This approach does not require any modifications to the



Animation List

name	state info	ptr
spin	data	
roll	data	

Figure 3-6: Implementation for the Animation Module

graphics library interpreter. The animation environment matrix is implemented as an object whose only job is to apply the matrix to the current top of stack matrix, which is the matrix applied to all subsequent drawing commands. The display list can access the animation environment matrix by instantiating an instance of the object. This is a convenient implementation for the animation list because it can also use the name of

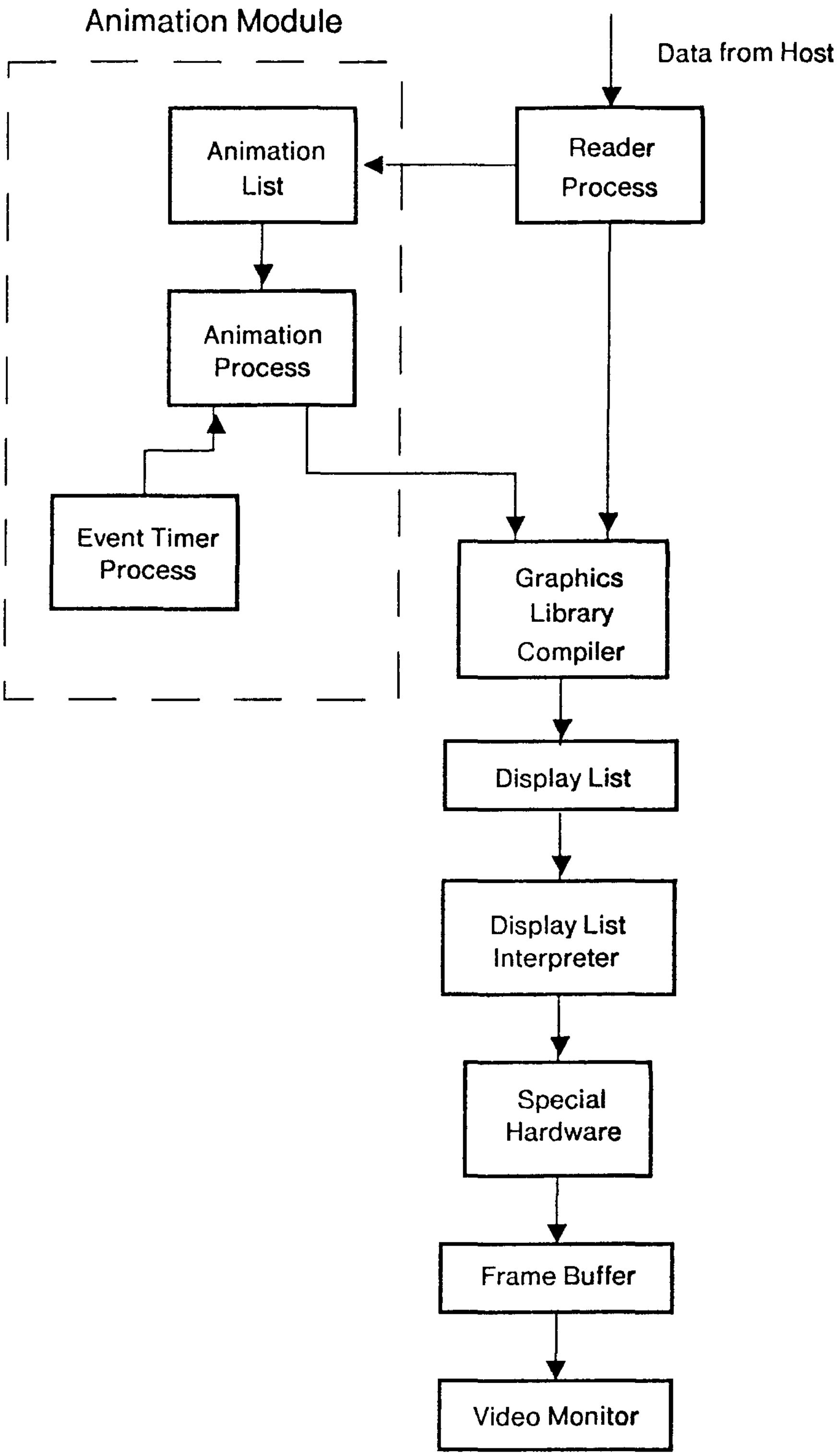


Figure 3-7: Approach Using Existing Graphics Library Compiler Routines

the animation environment as the name of the object, as well as the name of the tag which identifies the matrix command within the object. Thus, it does not need an explicit pointer to the memory location. To modify the matrix in the object, a dedicated process [21] called the *animation process* simply edits the object. Unfortunately, in the display list, a call to an object, *grow*, is not stored as a pointer to the object. Instead, the call to *grow* is stored in a display list, as shown in Figure 3-8. The interpreter then uses the name, *grow* as the tag for a symbol table which contains the starting address of each object. The interpreter then jumps to this address to read the commands stored in the object's definition.

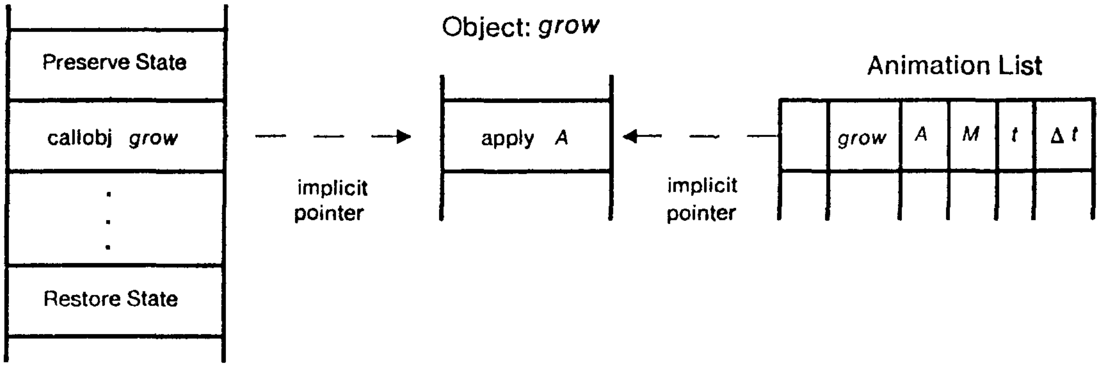


Figure 3-8:Accessing the Animation Environment Matrix

While the chosen approach is not the most efficient means of implementing the algorithm, it does not require modifications to the interpreter, which is a very complicated piece of code, implemented largely in 68000 assembly language. As stated earlier, the goal of this implementation is to prove the feasibility of the imaging model. Once that goal has been reached, efforts towards more efficient implementations can be researched.

3.3.1.1 The Animation List

As mentioned in the previous section, the animation list, stores the animation information for each animation environment. Each entry is a structure containing five elements. Three elements are user specified. They are the name, the animation matrix, M, and the time interval, Δt, which is specified in units of 1/30 of a second. The two remaining elements describe the current state. They are the matrix, A, which is the matrix actually applied to the specific animation environment and the time, t which indicates the next time A should be updated. A global modulo counter, T keeps track of the current time. The code illustrated below is a sample structure definition in C.

```
typedef   struct  AniList   {
          int  name;                          /*  name  of  Envron.   */
          int  CLockTicks;                    /*  Δt */
          float  AniMatrix[4][4];             /*  M */
          int  LapsedTicks;                   /*  t */
          float  AniEnvMatrix[4][4];          /*  A */
          }AniList,    *ALptr
```

After n applications of M, the state is as follows: $A = M^{n+1}$ and $t = n\Delta t + T_0$ where T_0 is the time when the animation environment was originally created.

3.3.1.2 Time Slot Data Structure

It is important to find a data structure which lets the animation process efficiently access the items that need to be updated at the current time, T. There are several ways this list could be implemented. They include unsorted and sorted linked lists [19], heaps [12]. For this implementation, however, a less traditional data construct, the "time slot structure," was chosen. If it is assumed that the maximum time between animation events is one second, and there is a finite number of updatings per event, the time slot structure is a simple and appropriate one. If slower animation is desired, a matrix specifying smaller increments of motion can be used. As mentioned in the previous chapter, this implementation of slow animation is preferable to that of applying larger increments of motion over longer time intervals because it will provide a smoother and more realistic perception of motion.

For this time slot structure, the minimum time between the events is one vertical retrace period which is 1/30 of a second. Therefore, there are 30 time slots in which events can be scheduled. An event is defined as the updating of an animation environment matrix. A modulo 30 counter determines the current slot; this counter is incremented each vertical retrace period. If the slot is empty, there are no matrices which require updating. Any animation environments stored in a slot must be updated as soon as the slot counter reaches the counter value corresponding to that slot. Once A has been updated in an animation environment, the new slot location for the animation environment can be determined by adding the Δt to the current slot modulo 30 position. Thus, repositioning takes constant time, and T, the next time A requires updating, is implicit in the data structure; it does not need to be stored in the animation structure.

3.3.1.3 Implementation of the Time Slot Data Structure

The implementation of the slot concept is very straightforward. The 30 time slots are represented by a one dimensional array of length 30. Each entry of this array contains a pointer to a structure containing state information for that particular slot. The code shown below is a sample slot structure definition in C:

```
typedef  struct  SlotInfo   {
         ALptr  ALlist;        /* ptr  to  animation   struct   */
         int    SLOTentries;   /* # of  entries   at  slot  */
         }
```

The slot events are stored as linked lists of animation definitions, as shown in Figure 3-9. The slot structure contains a pointer to this linked list. Since linked lists imply that each structure in the list contains a pointer to the next animation structure in the list, the described animation structure must be modified to include a pointer to the next structure in the **ALlist**. To facilitate inserting, deleting, and moving structures, the animation structure also contains a pointer to the previous structure in the list. Hence, the events are actually stored in a doubly linked list.

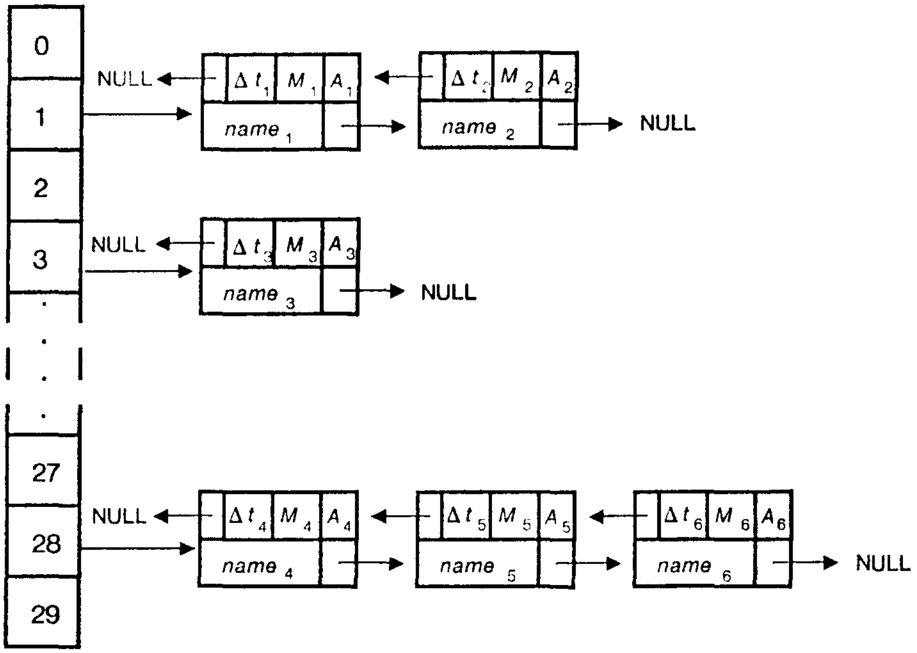


Figure 3-9: Slot Method of Implementing the Animation List

The other information stored at the slot is the total number of animation structures there at any given time. Once the maximum number of updates occurring on a vertical retrace period has been determined, the number of structures allowed per slot at any given time can be limited. This limit is determined by the capabilities of the display list interpreter and compiler. If a slot is full, any additional events for that slot can be scheduled either one slot earlier or later. The effect on the resulting scene is negligible since the change would only involve a 1/30 of a second time difference. In this implementation, if a slot is filled, the process places the structure in the next succeeding slot that has space. A ceiling on the total number of animation environments protects the system against errors accumulating over time because of filled slots.

3.3.2 A Passive Animation List

Our implementation has one additional animation list. It is a passive animation list which is used to store animation environments which have been halted by the **HaltAnimation** primitive. This allows an environment to stop its motion temporarily. This list stores the animation structure for the environment until it asks to be reinstated in the active animation list (slot structure) where by it will resume being updated. The passive animation list is implemented as an unsorted set of pointers to animation structures.

3.3.3 The Dedicated Process

The dedicated process, mentioned earlier, has actually been implemented as two processes: the *animation process* and the *event timer process.* This two process system was chosen because it allowed easy integration into the V-kernel operating system (the IRIS' operating system) by using V-kernel message primitives [3]. Upon receiving a message, the *animation process* immediately sends back a reply because the event timer process, described in a later section, blocks until it receives the reply. A global flag, *done*, indicates that the animation process is waiting for the event timer process to send it a message. Since the animation list and display list are shared data structures, the process also contains locks to insure that a second process does not modify a shared data structure until the first process has finished accessing it.

3.3.3.1 The Animation Process

The animation process is the process that changes the display list to reflect the animation. When called, it executes the procedure shown in Figure 3-10. Following the flowchart in Figure 3-10, the process first increments the modulo 30 slot counter. It then checks the current slot selected by the slot counter. If it is empty, the process becomes dormant and waits for another message from the event timer process.

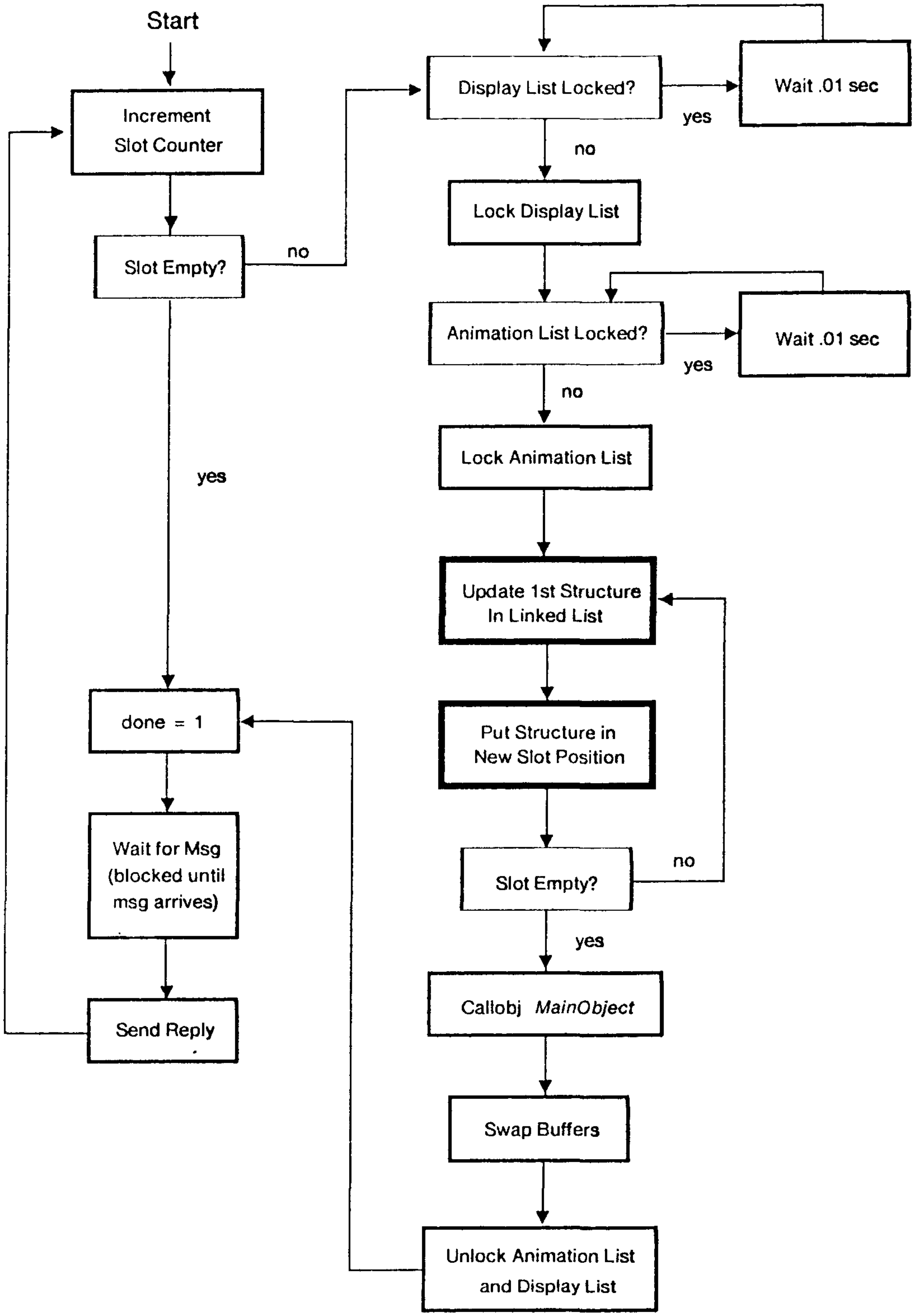


Figure 3-10:Flowchart for Animation Process

Otherwise, it checks to see if anything is modifying the display list. If so, it waits for the display list lock to be released. It then locks the display list, and follows the same procedure to insure that the animation list is not being modified. Once it is able to lock the animation list, the process updates the first animation structure in the slot by concatenating M onto A and stores the result as A. In other words, $A \leftarrow MA$. It then updates the display list by modifying the object containing the matrix which is applied to that particular environment. This object has the same name as that associated with the animation environment and is called immediately upon entering the environment. The state of the transformation matrix stack is preserved prior to entering the animation environment and subsequently restored upon exiting the environment. This updating procedure is illustrated in Figure 3-11. The repositioning algorithm, shown in Figure 3-12, is also very simple. The new slot location is calculated by using modulo arithmetic to add Δt to the current slot position. The structure is then moved from its current list to the front of the linked list of the new slot.

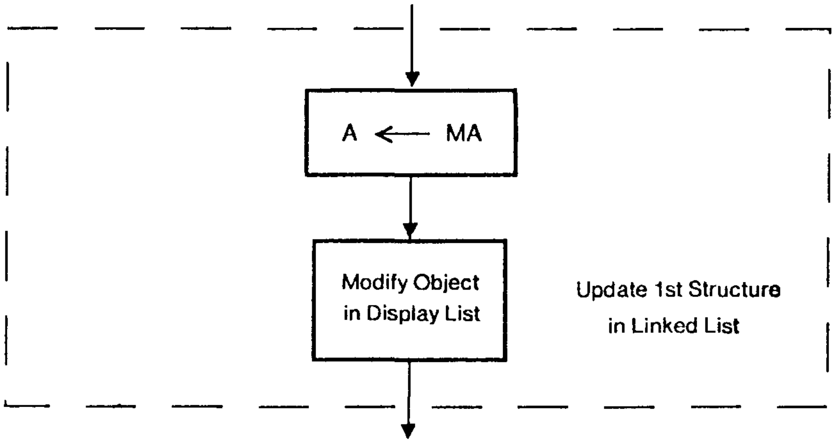

A ← MA
Modify Object in Display List
Update 1st Structure in Linked List

Figure 3-11: Update Algorithm for Animation Process Corresponding to the First Highlighted Box in Figure 3-10

After the structure has been repositioned in its new slot position, the current slot is checked to see if it has any more animation structures. If so, the entire updating and

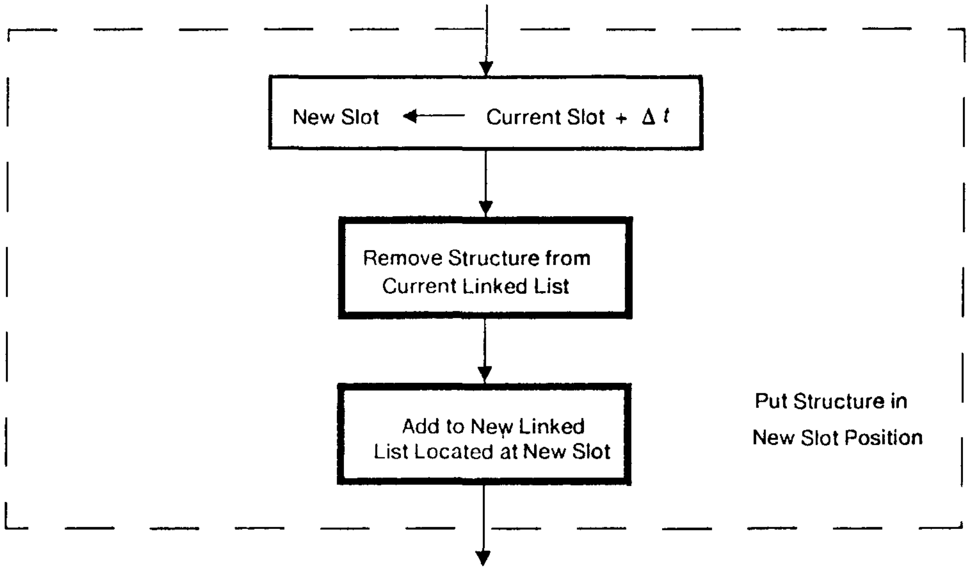

New Slot ← Current Slot + Δ t
Remove Structure from Current Linked List
Add to New Linked List Located at New Slot
Put Structure in New Slot Position

Figure 3-12: Positioning Algorithm for Animation Process Corresponding to the Second Highlighted Box in Figure 3-10

repositioning process is repeated until the slot is empty. This iterative checking is necessary because multiple animation environments may need updating during the same frame time. Once all the animation structures have been updated, *MainObject* is instantiated and the interpreter is called to rescan the modified display list. The newly rendered scene is then stored in the non-visible buffer. The two buffers are then swapped during the next vertical retrace period. Vertical retraces actually occur once every 1/60 of a second, but the screen is only refreshed once every 1/30 of a second because of interlacing. Thus, it is not necessary to swap buffers any sooner than a vertical retrace period, defined as 1/30 of a second. Swapping more often could also change the picture halfway through the refresh cycle.

The final step in the animation process is to unlock all the shared data structures. It then activates *done* and blocks until it receives a message from the event timer process, at which point it sends a reply and repeats the entire routine.

3.3.3.2 Event Timer Process

The event timer process is used to unblock the animation process. This timer is implemented as a vertical retrace event whose only purpose is to wake up the animation process after it has completed execution for the current slot counter value. At the start of every vertical retrace period, it checks the *done* flag. If *done* has been activated, the process deactivates the flag and sends a message to the animation process. It is blocked itself until the animation process replies. Once the reply is received, the animation process begins updating environments for the next time slot. A flowchart of the event timer process is given in Figure 3-13.

3.3.4 Implementation of Animation Primitives

Having described the new graphics system, the actual imaging model primitives can be implemented. The basic polygon modeling primitives, also described earlier, are used in the implementation of the animation primitives; the functions and groupings of these animation primitives themselves are described in detail in Chapter 2.

3.3.4.1 BeginAniEnv(*name*)/EndAniEnv()

BeginAniEnv and **EndAniEnv** is analogous to MAKEOBJ/CLOSEOBJ. The main difference is that when space is allocated in the display list, **BeginAniEnv** actually does a PUSHMATRIX followed by a CALLOBJ(*name*). **EndAnvScene** is simply a POPMATRIX. The name of the environment, *name*, is an object which contains just a MULTMATRIX(*m*) where *m* is the animation environment matrix, *A*, described earlier. The object, *name*, is created by **Animate**, which must be issued before the animation environment will cause any motion. Otherwise, **BeginAniEnv** and **EnvAniEnv** simply act as PUSHMATRIX and POPMATRIX respectively. A sample use of the animation environment as well as the resulting display list is shown in Figure 3-14. Note that *name* is a transformation, stored as an object merely for address purposes.

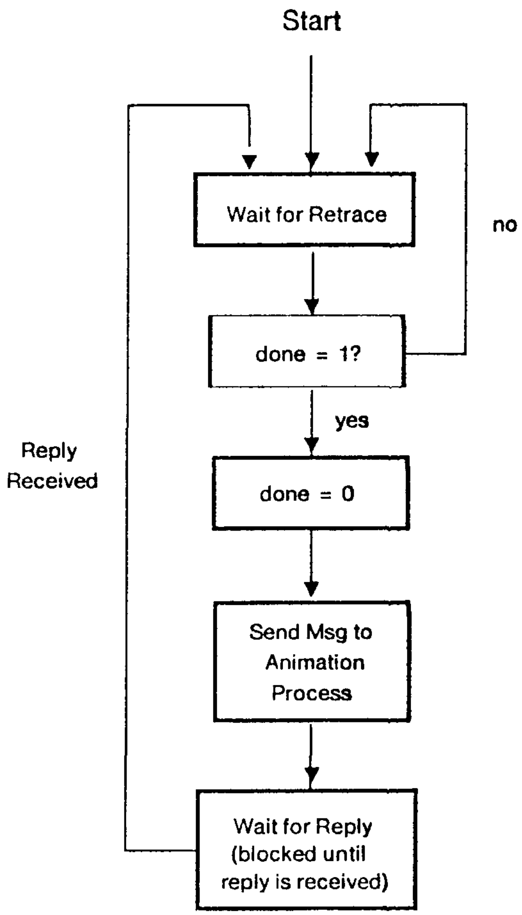


Figure 3-13: Flowchart for Event Timer Process

3.3.4.2 StartAniProc(*MainObject*)

StartAniProc is the routine which puts into motion, the process of animating the scene: In addition to instantiating the first instance of *MainObject*, **StartAniProc** must initialize the slot counter to 0, create the animation process and set up the event timer process.

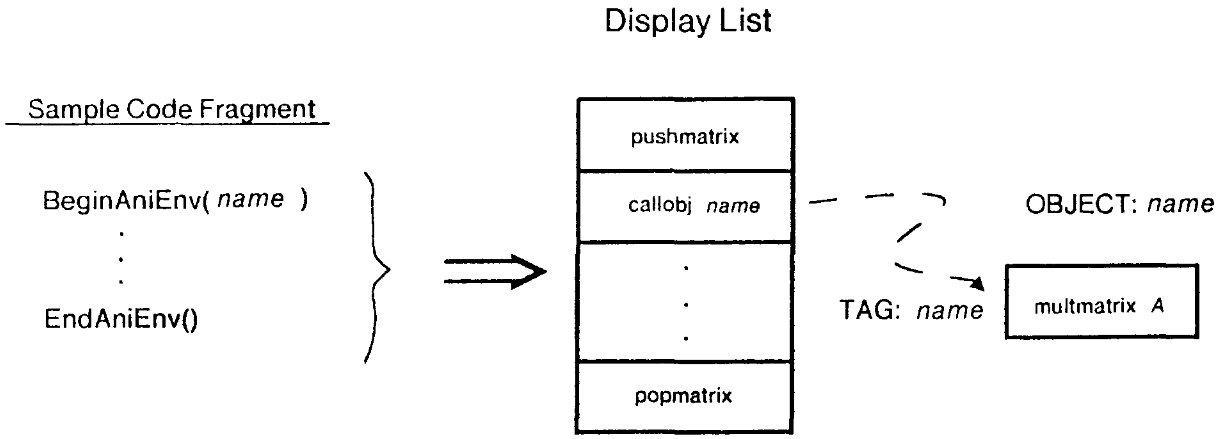


Figure 3-14:Sample Code Fragment and Resulting Display List

3.3.4.3 Animate(*name*,Δt,*M*)

Animate manipulates an animation environment, *name*, with matrix A, multiplying A by the animation matrix M every Δt time units. If *name* has previously been defined by another **Animate** command, **Animate** simply modifies the existing animation list entry. The existing M and Δt are replaced by the new values of M and Δt and the position of the entry in the slot structure is readjusted relative to T, the current slot counter value. In other words, the change in animation will occur in $\Delta t + T$ time intervals.

The current implementation requires that when using **Animate** to restore motion to a halted animation enviroment, the user must keep track of the original values of M and Δt for the halted environment or else use **GetAniTime** and **GetAniMatrix** to obtain them. The most commonly used modeling transformation matrices are provided in Appendix A.

3.3.4.4 HaltAnimation(*name*)

HaltAnimation halts the motion of the animation environment leaving it in the position last specified by *name*. There are several methods of implementing this primitive. The simplest is to change the animation matrix to an identity matrix. Thus, each time it is applied to A, A remains unchanged. This approach, however, forces the animation process to update matrices needlessly. Our implementation removes each halted animation definition from the active animation list and places it in an passive list that is not checked by the animation process. As described earlier, the passive animation list merely stores the definitions of halted animation environments. The animation can easily be restarted by issuing a new **Animate** command for that animation environment. This lets that entry be moved back to the active animation list and placed in the proper slot.

3.3.4.5 KillAnimation(*name*)

KillAnimation removes the entry, *name*, from either the active or passive animation list, in whichever it resides. This means the object can no longer be animated. As with **HaltAnimation**, the position of the objects within the environment is that described by the most recent animation environment matrix, A, for *name*.

3.3.4.6 GetAniTime(*name*)/GetAniMatrix(*name*)

These routines allow the user to obtain the parameters of an animation list entry. For an animation definition, *name*, **GetAniTime** and **GetAniMatrix** return the values of Δt and M respectively. The primitives look up the animation list entry for *name*, and return the value of the requested parameter.

3.3.4.7 GetSlotCounter

A modulo counter is used to keep track of the current slot number. This primitive returns the current value of the slot counter.

3.4 The Resulting Implementation

While this implementation has been designed on the IRIS, the approach used is a general, system-independent method. Now that the parts have been described in detail, a full illustration of the implementation can be drawn, as shown in Figure 3-15. It is essentially a combination of Figures 3-8, 3-9, and 2-2c. The main components are the animation list, which contains the state of the animation environments *travel*, *spin*, and *roll*, the main display list, which just contains an instantiation of *MainObj*, and the object, *MainObj*. *MainObj* shows a simple use of an animation environment, *roll*, which causes all objects within the environment to exhibit rolling. The second animation environment, *travel*, is a nested environment of the type illustrated in Figure 2-2c, containing the actual object *frisbee* and transformation "objects" *travel* and *spin*. This combination yields a spinning frisbee, which is simultaneously following a trajectory specified by the *travel* environment.

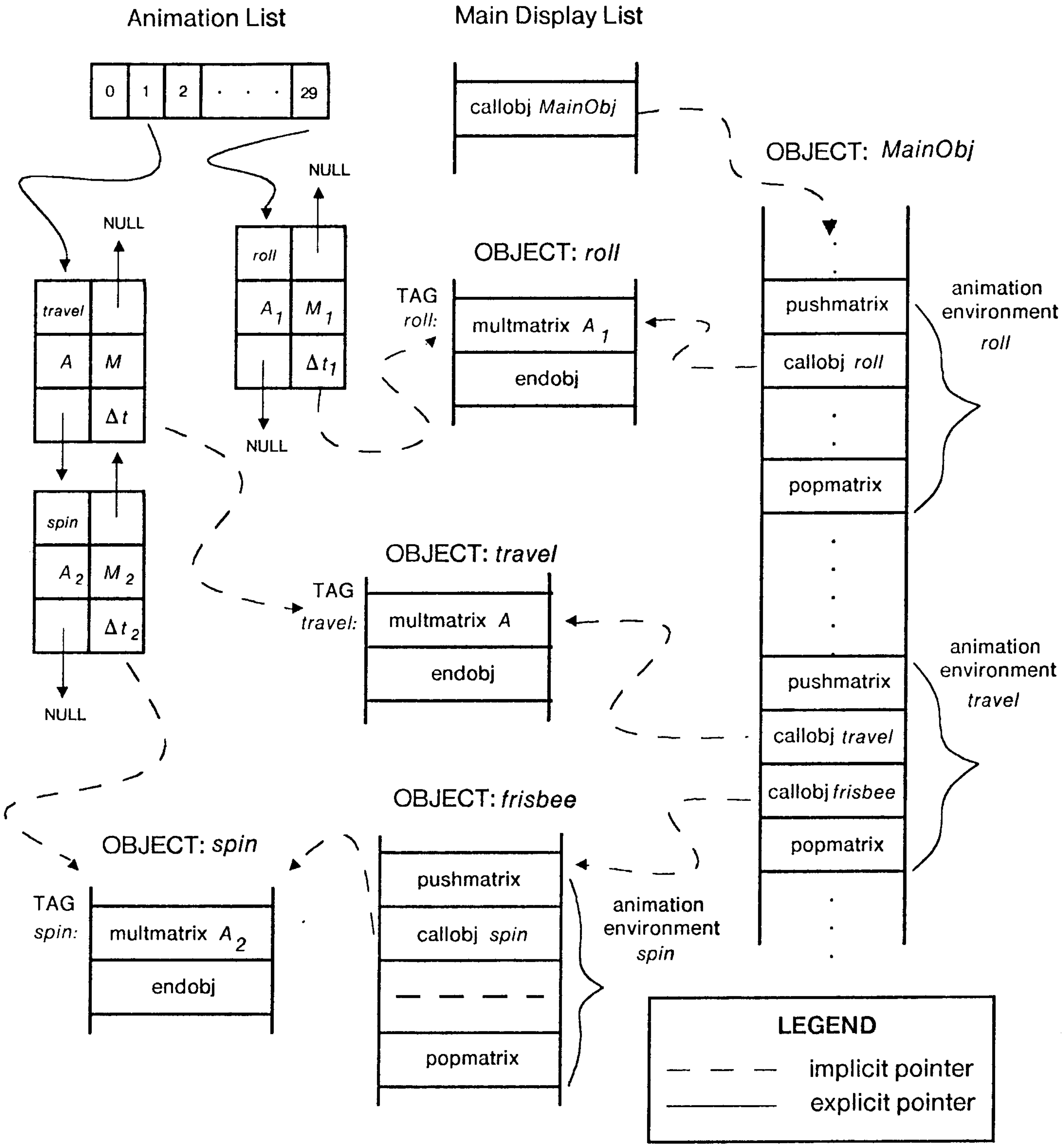


Figure 3-15: Display List with Animation

Chapter Four

Application Observations

4.1 Using the IRIS

This section describes the method used to test the animation module and animation primitives.

4.1.1 IRIS/Host Configuration

The network topology of the IRIS allows it to communicate with host computers using several methods - an RS232 serial line, an Ethernet using IP/TCP or XNS protocol, or an IEEE 488 parallel port. Multiple hosts can be used to access the IRIS. Our particular configuration has 2 hosts, a VAX[3] 11/750 running Unix[4], and a Symbolics 3600 LISP Machine. The VAX communicates with the IRIS via an Ethernet interface using XNS while the LISP Machine uses a 9600 baud RS232 serial line. Because the IRIS does not currently support LISP, the host's graphics library and input/output routines were translated from C to LISP.

Despite the slow communication channel, the LISP Machine performs animation better than the VAX when using the conventional polygon imaging model. This superior performance occurs because the LISP Machine is a single user system; it does not experience the same load problems as a VAX 11/750 which is supporting several users simultaneously. By relying on the host to cause the animation, the speed of the motion is directly dependent on the how quickly the host can execute the animation loop; the

[3] VAX is a trademark of the Digital Equipment Corporation.

[4] Unix is a trademark of Bell Laboratories.

loop is the means by which the host can continuously change the transformations affecting the objects. On a mainframe such as the VAX, the speed of execution depends on many parameters such as the number of users on the system and what programs are being run. These parameters are not constant, and hence animation should not directly depend on them.

4.1.2 Testing the Primitives

One of the advantages of the chosen implementation was that most of the code for the animation module and animation primitives could initially be tested on the VAX; the code was written in C, and therefore could be tested using the host's graphics library. Furthermore, source level debuggers and Unix environment could be used to debug the code. A sample animation test program was written for the VAX to test the new animation primitives. The program would draw a flower and cause it to roll (rotate and translate) across the screen. The animation process and timer events were simulated by a **while** loop. Because the VAX was heavily loaded at the time of the test, the varying load had a noticeable effect on the excuting animation program. The flower moved in a very jerky manner; the motion was neither smooth nor continuous and the speed was constantly changing. Executing the same program when the system was not as heavily loaded improved performance considerably. The flower still moved in discrete jumps, but the rate was much more consistent. Decreasing the time interval would cause the motion to appear smoother and more continuous, but it would also slow it down. Motion which was both rapid and smooth could not be achieved with this particular choice of host.

A slightly different test was run on the LISP Machine. Since the animation module was written in C, it would not run on the LISP Machine without explicitly translating the code to LISP. However, for testing purposes, similar results could be produced without explicit use of the animation primitives. The LISP Machine did not have any difficulty producing the motion at a consistent rate, but had the same problem with producing rapid and smooth motion. This is mainly because the loop cannot execute quickly enough to produce the continuous motion. By causing motion to happen at a rate as quickly as 1/30 of a second, the animation would appear smoother. Furthermore, by executing on the IRIS rather than the host, the motion would be independent of the load on the host system.

4.2 Observations from the IRIS Implementation

The actual IRIS implementation was very difficult to completely debug because of limited debugging facilities. Nevertheless, once the animation module was integrated into the IRIS architecture, tests showed that motion was consistent, without the jerky effects produced by host animation. This was the expected result. However, a somewhat surprising result was that the overhead for updating the animation matrices and display list was negligible compared to the overhead for instantiating the new instances of *MainObject.* The frame repaint time was heavily dependent on the complexity of the scene, rather than the number of animation environments. The more complex the scene, the slower the animation for the same animation specifications. Thus, the refresh rate for the animated scene is bounded by the complexity of the scene rather than the complexity of the animation specification.

Chapter Five

Conclusions

5.1 Imaging Model

A new imaging model, based on a conventional polygon model, has been proposed. The model, which provides a set of animation primitives that can be used to specify certain types of motion, allows the user to easily animate parts of scenes. Once the scene has been specified, the graphics subsystem brings about the animation automatically in a manner which does not require recomputing the positions of objects every frame time. Without this unique feature, real-time animation would not be possible with current technology, unless very expensive and specialized hardware were used.

5.2 Enhancements to the Animation Imaging Model

There are many improvements which can be made to the animation imaging model implementation which would enable the system to bring about animation more efficiently. While most improvements would enhance this implementation, they are device dependent changes and therefore do not extend the capabilities of the imaging model. This section focuses on enhancements to the imaging model.

5.2.1 Supporting Other Types of Motion -- Acceleration/Deceleration

One added feature would be the ability of the model to support accelerating and decelerating motion. This can be accomplished by using an algorithm which requires increasing or decreasing the time interval Δt by a user-specified quantity when updating the application matrix. Besides being less computationally intensive than the one described in Chapter 2, this approach has the added advantage of allowing acceleration or deceleration of any type of motion that can be specified by one matrix. The previously described approach is restricted to applications of acceleration in a straight line. This algorithm allows cases such as the following example:

```
A ball  is rolling  in a straight  line.  Its rate  of
rotation  gradually  slows  down  with  time  until  it
stops  completely.
```

This approach also produces smoother motion when moving in small increments at a faster rate. If the object is decelerating, the effect of movement in large increments is especially noticeable; as the Δt increases, the object may appear to be moving at discrete time intervals instead of displaying continuous motion.

This feature can be implemented by specifying two additional parameters in the **Animate** command. These parameters are r, the number of retraces which should elapse before Δt is changed, and i, the increment by which the time interval, Δt, will change every r retrace periods. The only major change to the described implementation would be a modification to the updating algorithm, which would be more complicated because there are limits as to how small or large Δt can become. If the object is accelerating, the Δt can not reach a value less than $\Delta t = 1$. Therefore, the algorithm must know that if $\Delta t = 1$, the time interval cannot be updated anymore. It can, however, still update the animation environment matrix, A, so the net effect is an object moving at a terminal velocity. If, on the other hand, the object is decelerating, it should eventually stop moving; therefore, a ceiling on Δt must be determined. When $\Delta t = \Delta t_{ceiling}$, the animation process should transfer the animation list entry to the passive animation list since it should not require movement.

5.2.2 Specifying Animation to Start and Stop in the Future

Another possible enhancement to the imaging model is the capability of specifying two parameters which. would enable animation to commence and end at a future time, relative to the current value of the slot counter. These parameters specify the times in terms of vertical retrace periods, allowing predetermined changes in the specified scene without host intervention. Animation list definitions which are not yet active or are already deactivated can be stored in the passive animation list. By specifying a limit on the future starting and ending time intervals, a method similar the slot method can be used to keep track of future events. This slot structure could be maintained by another auxiliary process whose purpose is to insert and remove animation entries from the passive and active animation lists as specified by the applications program.

5.2.3 Collision Detection

An important feature, missing from the animation imaging model, is a means of detection collisions between objects. Collision detection is a necessary task, especially when the animated scenes involve dynamic interactions. This capability would allow the host to be informed, should an exceptional circumstance occur. Such cases include a collision between two or more objects, or an object moving out of view. The former occurs when boundaries of objects overlap, while the latter case is modeled as the intersection of the object with the boundaries of the current viewing window. Once the host has been informed of the occurrence of a collision as well as the objects involved, it can proceed with three possible actions: alter the scene to take into account the effect of the collision; ignore the collision completely; or process the collision later.

There are many ways to implement a collision detection module. An approach taken by Evans and Sutherland for their Improved Scene Generator [8] used a separate graphics subsystem to process the collision detecting and reporting information. A simpler approach is to incorporate checking for collisions into the imaging model. If each object is surrounded by a bounding box, the scheme would just need to determine if any two parts of a bounding box overlapped. This information could be determined by a new module, the *collision detection module*, which obtains collision information and reports its results to the host. The architecture of a system which incorporates this module is shown in Figure 5-1.

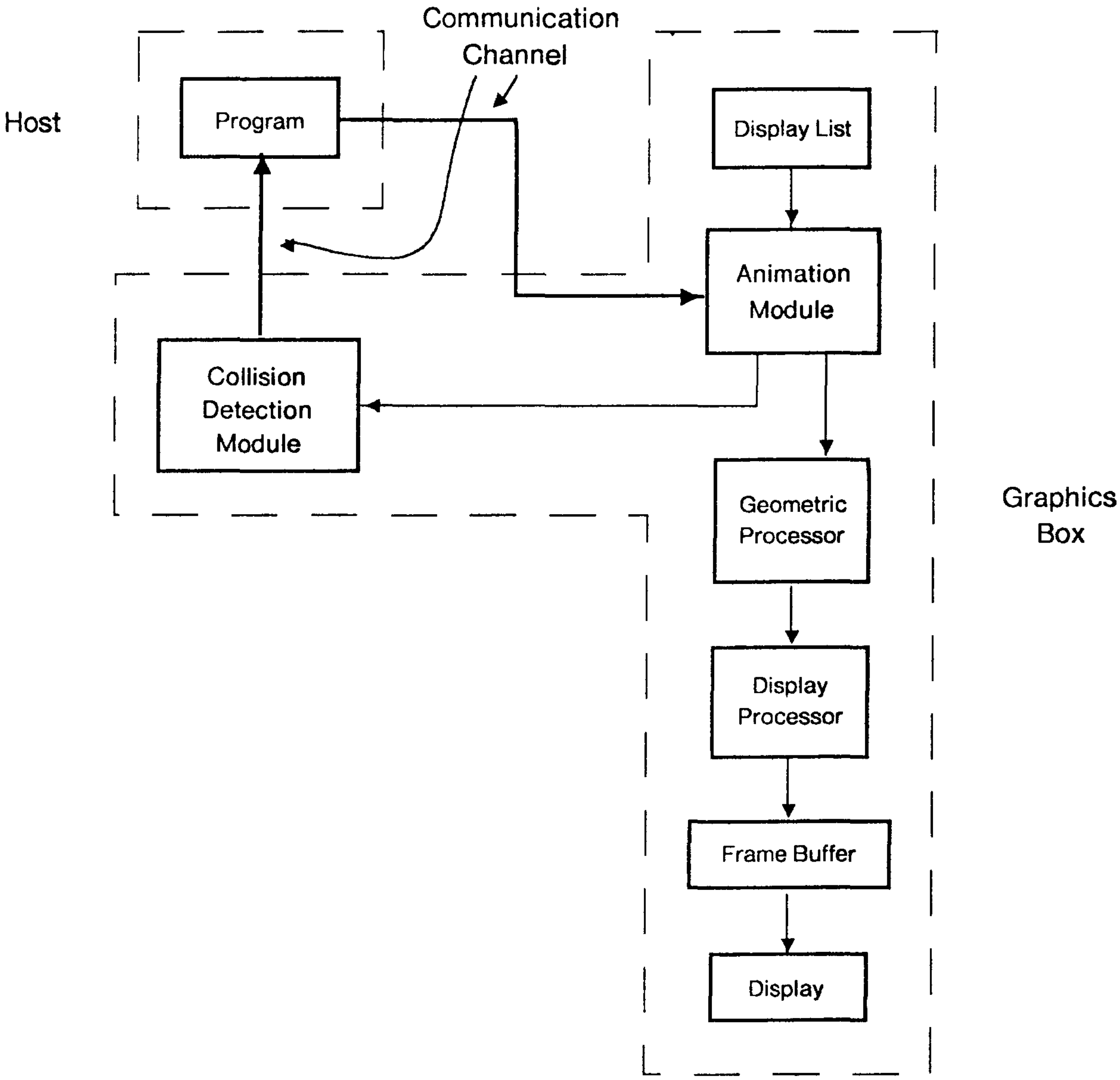


Figure 5-1: Architecture Incorporating a Collision Detection Module

5.3 Summary of Thesis

It is possible to separate animation generation from an animation application by incorporating animation into the imaging model. By enabling special purpose processors to take care of the animation tasks, the host was able to distribute its processing power to other tasks, and the dependence on the the communication channel was reduced considerably. The model also provides primitives which allowed the application programmer to easily specify how the objects should be transformed over time.

The main disadvantage is that the program is not aware of the positions of objects at all times, making collision detection capabilities essential for applications involving dynamic interaction of objects. Nevertheless, the animation imaging model provides a great deal of power. Its advantages include the fact that high performance animation can be specified simplistically by the applications programmer and that the animation capabilities in the imaging model are device independent.

Appendix A

Matrices for Modeling Transformations

A.1 Translation

$$\text{Translate}(T_x, T_y, T_z): \quad \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_x & T_y & T_z & 1 \end{bmatrix}$$

A.2 Scaling

$$\text{Scale}(S_x, S_y, S_z): \quad \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

A.3 Rotation about the X-axis

$$\text{Rotate}_x(\theta): \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

A.4 Rotation about the Y-axis

$$\text{Rotate}_y(\theta): \begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

A.5 Rotation about the Z-axis

$$\text{Rotate}_z(\theta): \begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

References

[1] Bentley, D. L. and Cooke, K. L.
Linear Algebra with Differential Equations.
Holt, Rinehart, and Winston, Inc, NewYork, 1973.

[2] Chang, K. Y.
Microcomputer Graphics and Applications with NAPLPS Videotex.
IEEE Computer Graphics & Applications :21-33, June, 1985.

[3] Cheriton, D. and Zwaenepoel, W.
The Distributed V Kernel and its Performance for Diskless Workstations.
Computer Systems Laboratory, Stanford University.

[4] Chuang, R. and Entis, G.
3-D Shaded Computer Animation -- Step by Step.
IEEE Computer Graphics & Applications :18-25, December, 1983.

[5] Clark, J. H.
The Geometry Engine: A VLSI Geometry System for Graphics.
Computer Graphics 16(3):127-133, July, 1982.

[6] Clark, J. H. and Davis, T.
Work Station Unites Real-Time Graphics with Unix, Ethernet.
Electronics :113-119, October, 1983.

[7] Crow, F. C.
The Use of Greyscale for Improved raster Display of Vectors and Characters.
Computer Graphics (Proc. Siggraph '78 12(3):1-5, August, 1978.

[8] Evans & Sutherland Computer Corporation.
Improved Scene Generator Capability.
1977.
Prepared for NASA, Lyndon B. Johnson Space Center, Houston, Texas.

[9] Foley, J. D. and Dam, A. V.
Fundamentals of Interactive Computer Graphics.
Addison-Wesley Publishing Co., 1982.

[10] Goldstein, R. A., and Nagel, R.
3-D Visual Simulation.
Simulation :25-31, January, 1971.

[11] Gouraud, H.
Computer Display of Curved Surfaces.
Technical Report Report CSc-71-113, University of Utah, June, 1971.

[12] Horowitz, E. and Sahni, S.
Fundamantals of Computer Algorithms.
Computer Science Press, Inc, Rockville, 1978.

[13] Ikedo, T.
High-Speed Techniques for a 3-D Color Graphics Terminal.
IEEE Computer Graphics & Applications :46-58, May, 1984.

[14] Kahn, K. M.
Creation of Computer Animation from Story Descriptions.
Technical Report Technical Report 540, Massachusetts Institute Technology AI Lab, August, 1979.

[15] Miller, R. R.
Simulation and Graphics on Microcomputers.
Byte 9(3):194-200, March, 1984.

[16] Newman, W. M. and Sproull, R. F.
Principles of Interactive Computer Graphics.
McGraw-Hill, NewYork, 1979.

[17] Newton, M.
Real-Time 3-D Graphics for Microcomputers.
Byte 9(10):251-286, September, 1984.

[18] Parke, F. I.
Computer Generated Animation of Faces.
Technical Report Report CSc-72-120, University of Utah, June, 1972.

[19] Reingold, E. M., Nievergelt, J. and Deo, N.
Combinatorial Algorithms: Theory and Practice.
Prentice-Hall, Inc, EnglewoodCliffs, 1977.

[20] Rogers, D. F.
Procedural Elements for Computer Graphics.
McGraw-Hill, NewYork, 1985.

[21] Saltzer, J. H.
Traffic Control in a Multiplexed Computer System.
Technical Report MAC-TR-30, Massachusetts Institute of Technology, July, 1966.

[22] Schumacker, R. A.
A New Visual System Architecture.
In *Proceedings of the 2nd Interservice/Industry Training Equipment Conference,* pages 94-101. Salt Lake City, UT, November, 1980.

[23] *IRIS User's Guide*
Version 1.0 edition, Silicon Graphics, Inc., Mountainview, CA, 1983.

[24] Sutherland, I. E., Sproul, R. F., and Schumacker, R. A.
A Characterization of Ten Hidden-Surface Algorithms.
Computing Surveys 6(1):1-55, March, 1974.

[25] Warnock, J. and Wyatt, D.
A Device Independent Graphics Imaging Model for Use with Raster Devices.
ACM Computer Graphics 16(3):313-319, July, 1982.