MODELING THE FILM HIERARCHY IN COMPUTER ANIMATION

by

Michael S. Blum

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

The University of Utah

August 1993

Copyright © Michael S. Blum 1993

All Rights Reserved

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a thesis submitted by

Michael S. Blum

This thesis has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

Chair: Richard F. Riesenfeld

Robert Mecklenburg

Robert McDermott

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of The University of Utah:

I have read the thesis of Michael S. Blum in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to the Graduate School.

Date

Richard F. Riesenfeld
Chair, Supervisory Committee

Approved for the Major Department

Thomas C. Henderson
Chair/Dean

Approved for the Graduate Council

B. Gale Dick
Dean of The Graduate School

ABSTRACT

We introduce an integrated animation and storyboarding system that simplifies the creation and refinement of computer generated animations. The framework models both the process and product of an animated sequence, making animation more accessible as both a communication medium and as an art form. The system adopts a novel approach to animation by integrating storyboards and the traditional film hierarchy of scenes and shots into a computer modeling and animation system. Traditional animators initially storyboard key moments in a film to help guide the animation process. In a computer generated animation, a much more dynamic and integrated relationship between storyboards and the final animated sequence is possible. This hierarchical decomposition allows animators to reduce the complexity of long animations to manageable proportions. We also introduce the animation proof reader, a tool for identifying awkward camera placement and motion sequences using traditional film production rules.

CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would first like to thank Robert Mecklenburg for his unwavering moral support, inventive ideas, and help throughout this work. With his guidance, Alpha_1 revealed itself to be a powerful foundation on which to implement this thesis. I am also very grateful to the rest of my committee, Richard Riesenfeld, who had the faith to support me, and Robert McDermott, who helped motivate several ideas in this work.

Although the entire Alpha_1 staff and many of the Alpha_1 group have served as helpful resources, several deserve special mention. Marc Ellens proved to be an invaluable mentor, never too busy to answer the silliest system or C++ related question. Tim Mueller showed great patience in preaching the intricacies of the new display code. Greg Heflin and Mike van Thiel gave me boosts in moments of despair. Finally, I could always count on Beth Cobb's unbridled enthusiasm.

I would also like to thank Chih-Cheng Ho and Gershon Elber for their collaborative ideas in designing algorithms to extract key features from models and Lance Williams for corroborating my ideas.

Elaine Cohen deserves special thanks—without her help I would never have attended graduate school in Utah. Many thanks as well to David Johnson and Greg for their useful comments on early drafts of this thesis.

This work was supported in part by the DARPA (DOD) (N00014-91-J-4123 and N00014-91-J-4046) and the NSF and DARPA Science and Technology Center for Computer Graphics and Scientific Visualization (ASC-89-20219). All opinions, findings, conclusions, or recommendations expressed in this document are those of the author and do not necessarily reflect the views of the sponsoring agencies.

CHAPTER 1

INTRODUCTION

1.1 Introduction

The *Story* system is an integrated animation and storyboarding system that simplifies the creation and refinement of computer generated animations. The framework enables an animator to produce better organized and potentially more dynamic animations in a shorter amount of time than is currently possible. By structuring the animation process, we have taken a step towards making computer animation both a more accessible art form and a more useful communication tool.

The process of creating a computer animation is iterative in nature[9]. The individual or team producing the animation must develop a story, model the characters and props, decide the composition of each shot, script all dialogue or other sound effects, generate the motion of all the moving characters, and render each frame of the animation. All of these elements may be modified or refined before the final animation is completed. Enabling animation teams to make simple refinements quickly while minimizing the time consuming and computationally expensive operations required to revise models, generate motion, and render images will result in significant time savings.

This iterative technique is similar to filling in an outline. The structure of this outline, originally developed by the film and animation community[1, 25] for describing animated and live action sequences, forms the framework of our system. The framework consists of a hierarchy[18] that describes an animation as a sequence of *scenes* containing a number of *shots* (see Figure 1.1). Each shot consists of any number of individual *frames*. In particular, a scene is a piece of animation with a collection of objects in specific locations during a period of time. A scene may be broken up into a number of shots for dramatic purposes. Shots record the action as seen from a particular camera position. By editing these shots together, a complete scene is created.

The film and animation community have also developed techniques to help guide and organize the creative endeavor from the early stages of production. One such technique allows an animator to represent a shot by one or more *storyboards*. Storyboards, first used by Disney animators in the 1930s, are used to suggest the action, the locale, and the appearance of characters. These storyboards help guide the creation of the animation or film. In a computer generated animation, a much more dynamic and integrated relationship between storyboards and the final animated sequence is possible. In our system, storyboards are an integral part of

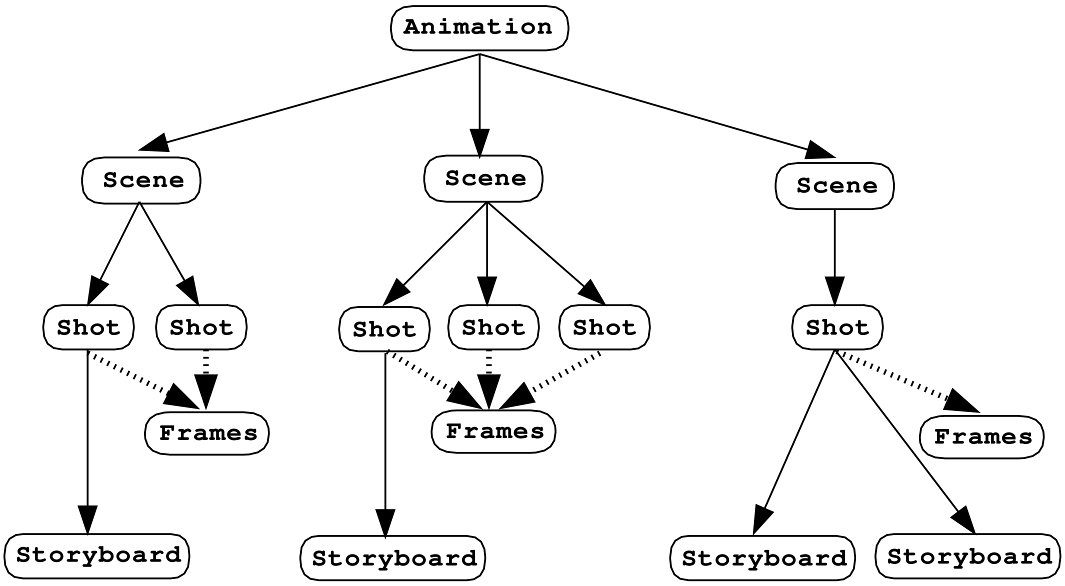

Figure 1.1. In traditional film, an animation consists of one or more scenes. Each scene consists of one or more shots. During film production shots are often represented by storyboards. Frames are eventually generated for each shot.

the animation creation process.

Computer animation systems have not adequately addressed the process of animation creation. By applying the scene, shot, storyboard hierarchy to computer animation, we gain several advantages. First, by using the structure established in the film and animation industries, we make computer animation more accessible to industry professionals. Second, we directly address the problem of managing long, animated sequences. Feature length computer animations will be technically possible in the near future; unfortunately, a practical framework for managing and refining such features has not been adequately explored. As early as 1978 Catmull[3] stressed the importance of using computers to manage storyboards and associated information, but we have not seen storyboards as traditionally used discussed in the literature[11]. Finally, the framework applies even to short animations because it stresses efficient use of computational resources by minimizing computationally expensive tasks such as rendering.

Eventually animation will become a "desk top" industry as publishing is today. (Multimedia authoring tools are already widely available.) Novices will have the power to produce animations but will lack the aesthetic judgment acquired by professional filmmakers through a lifetime of education and practice. Sound advice on animation aesthetics would help ensure a clear presentation of their ideas. Such advice, like advice on word usage, grammar, or spelling in publishing, might be provided by a set of automated tools that are able to identify and advise animators on problems in animated sequences.

Shot composition is of critical importance in creating clear and engaging animations. To help animators avoid shot composition problems, our user interface allows an animator to position easily and orient any number of cameras and then interrogate the cameras' views. The importance of careful shot selection is illustrated in the following example. Suppose we wanted to film a scene of a large diamond being thrown towards a cliff only to be saved at the last instant by a surprise catch from a boy standing behind a boulder at the edge. We might break the scene up into two shots; the first shot shows the diamond hurtling towards the cliff, and the second shot shows a hand rising above the boulder to catch the ball. If the first shot showed any part of the boy, the surprise would be given away.

The Story system is conceptually divided into five components. *Animation Production* facilities manage the hierarchical framework used by the rest of the system. *Camera Placement* tools provide an animator with a variety of mechanisms to create and position cameras and examine their views. The *Property Manager* is a flexible model manager that provides version and complexity control of models and other attributes in an animation. The *Rendering Manager* provides rendering and previewing features to produce frames of varying quality. Finally, the *Animation Proofreader* identifies awkward camera placement and motion sequences according to a set of traditional film production rules.

1.2 System Overview

The computer animation field has had varying amounts of success producing photo-realistic images, physical and nonphysical motion, simulations of natural processes (e.g., procedural modeling of fire and plants), and geometric models adequate for current needs. However, no monolithic animation program can provide all of these features and provide fast, efficient, and current algorithms. Therefore, a complete animation system providing all of these facilities must operate in a diverse environment and integrate a variety of tools.

Story, written in C++ and Motif, is an integrated component of the Alpha_1 modeling system[6]. Alpha_1 is centered around a NURB-based geometric modeler that can be used both interactively and through a network interface as a server. In server mode, the modeler may act as a gateway for transferring models and control information between various clients. Although oriented towards geometric modeling and manufacturing, Alpha_1 provides a tool-rich, flexible environment suitable for the construction of complex animations.

Services provided by the Alpha_1 system include scanline and ray traced rendering, physically and nonphysically based motion generation, interactive sculptured surface creation, and a common communications format for tool integration. The Story system coordinates access to these tools. Figure 1.2 provides an overview of how the system interacts with the rest of Alpha_1.

The Story program communicates directly with the geometric modeler, shape_edit, which mediates further communication with clients for physically-based motion generation and a key-frame system for nonphysically-based motion. Story's rendering module controls both the animation segments to be rendered and the quality of the rendered frames. The system can also import conventional images scanned in by an animator. By integrating this complex environment, we provide a flexible and powerful animation system which would not be possible with a more monolithic design.

The rest of this thesis is arranged as follows. We begin with a discussion of related and background work. Subsequent chapters discuss animation production, cameras, the property manager, rendering, and the animation proofreader. We then briefly discuss some of the results produced. Finally, we discuss the status of the system, future work, and conclusions.

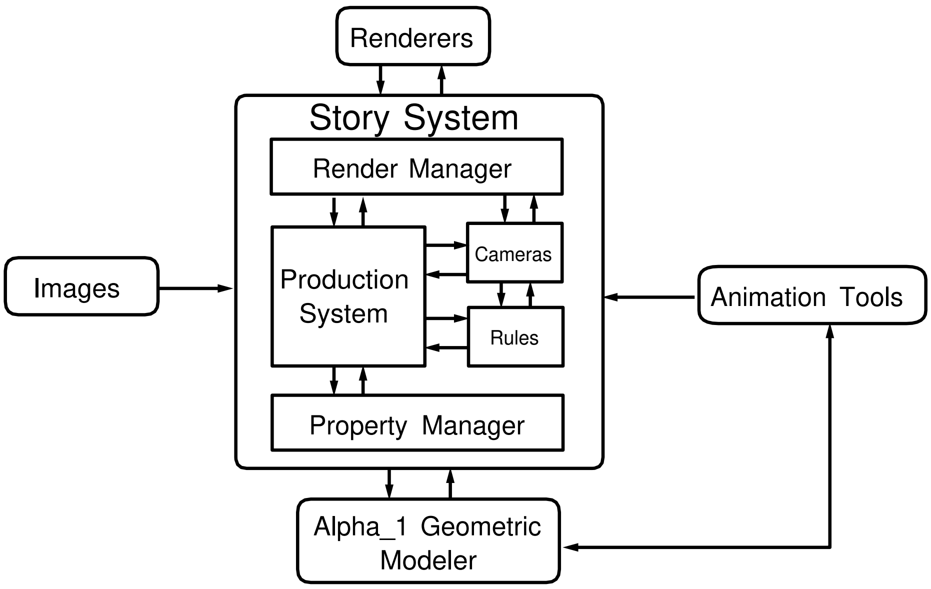

Figure 1.2. The animation system communicates with and relies upon external tools to provide models, motion curves, and rendering.

CHAPTER 2

PREVIOUS WORK

This thesis draws upon a variety of past graphics research efforts. We describe these works as they relate to the relevant sections of the thesis.

2.1 Animation Production

The utility of storyboards as a graphical outlining tool has long been known to conventional animation production houses[25], and some computer generated animations have used hand-drawn storyboards as well[18]. In general, however, computers have not played a role in managing storyboards and associated information, although as far back as 1978 Catmull[3] stressed the value of using computers for this purpose. Recently, there has been some interest in creating interactive storyboard systems[8], but at this time, no such system is discussed in the literature.

2.2 Camera Placement and Control

Positioning synthetic cameras is one of the more complex tasks that the scene creation interface must handle. A good deal of literature on this subject has been produced during the past few years. Ware and Osborne[27] evaluate three metaphors for controlling cameras in virtual environments. All control is performed using a six degree of freedom input device. For instance, the "eyeball in hand" metaphor allows the user to view a scene from the vantage point of a virtual eyeball. The "scene in hand" metaphor transforms the whole environment in correspondence to changes of the input device. If the input device is twisted to the right, the whole environment is rotated right. The last metaphor, "flying vehicle control," allows the user to control a virtual vehicle. The authors were unable to declare any of these metaphors as the best under all circumstances; each appears well suited to certain tasks. "Flying vehicle control" was chosen as the metaphor best suited for movie making, probably because it corresponds most closely to the way a camera operator manipulates a real camera.

Gleicher and Witkin[10] have implemented a collection of techniques that permit a user to manipulate a camera by constraining features in the image seen through the camera's lens. The techniques can be used independently of the underlying camera parameterization.

Mackinlay et al. have developed a three-dimensional (3D) viewpoint movement scheme appropriate for examining a specific target[17]. Their system will automatically reorient the synthetic camera to present a maximal view of some specified object of interest. Controlled navigation is achieved by allowing the user to move a camera rapidly when the target object is far away and logarithmically slower as the camera approaches the target.

More recent work by Phillips et al.[19] augments 3D direct manipulation of cameras by automatically adjusting the view to ensure smooth viewing transitions and avoid viewing obstructions. Drucker et al.[8] have developed a system that allows for high-level procedural descriptions of camera specifications to be written. These descriptions can encapsulate different camera control paradigms such as Ware's; can allow for specification of standard film industry camera moves such as pan, tilt, and roll; and can be extended to produce very complex camera moves.

A topic related to camera control is general interactive 3D control of objects. We are specifically interested in 3D control using two-dimensional (2D) input devices. Chen[4] conducted experiments comparing the speed and accuracy with which novice users were able to perform rotations using five different mouse-based controllers. These controllers varied from the traditional horizontal sliders that allow for rotation about one axis at a time to a more complex virtual sphere controller that simulates the mechanics of a physical 3D trackball. Chen found that sliders work best when accurate rotations are required or for simple rotations about one axis. However, users felt that the complex controllers behaved more intuitively, and tests proved they completed complex rotations more quickly with them.

2.3 Property Manager

The property manager is a database designed to allow efficient access to objects needed for animation. An interesting component of the database allows models of varying complexity to be freely substituted for one another. Hitchner[13] has developed techniques for reducing the complexity of data in order to interactively visualize fly-bys of Mars. He makes dynamic substitutions of portions of the landscape based on such factors as distance from the eye, smoothness of the surface, and closeness of a patch to the center of view. This approach is more sophisticated and accurate than the standard approach used by flight simulators in which model complexity is based solely on the distance of the model from the eye. The closer the object to the eye, the more complex the model used. Both of these approaches perform model substitution automatically.

2.4 Animation Proofreader

Although the idea of an animation proofreader appears to be novel, there has been some attempt to incorporate cinema, stage, and dance knowledge with traditional computer animation[14, 16, 20, 21, 30]. Ridsdale and Calvert[20] have developed a set of rules that attempt to ensure the theatrical concept of *focus*. For instance, to shift an audience's attention to the next speaker, the other actors may move to the back of the stage or may form a line that points towards the speaker.

The character movements needed to ensure this focus are inferred from a script. Karp and Feiner[14] have developed an expert system that uses cinematic knowledge to construct coherent descriptions of a scene. Their knowledge-based system automatically selects appropriate camera positions and movement based on cinematic considerations. We have abstracted many of these cinematic considerations into rules that we will discuss in relation to our animation proofreader.

Below is a description of how characters can be animated using the Alpha_1 system.

2.5 Animation With Alpha_1

Alpha_1 has a variety of tools to animate models. Although each system may use a unique internal representation for the created motion, all generate standard Alpha_1 *motion curves* to represent the animation. A motion curve is a function, represented as a NURB in the Alpha_1 modeling system that specifies the values of the transformation over time. Motion curves in Alpha_1 can specify rotation, translation, scaling, and other transformations such as quaternions. The x-axis corresponds to time, and the y-axis specifies the value of some transformation at each point in time. Time is represented by the parameter, t. For example, $rx(t)$ is a curve representing rotation about the x-axis.

Alpha_1 has several tools capable of generating motion curves. These tools are representative of both common and state-of-the-art animation techniques. Alpha_1's physically based modeling tools can simulate the dynamics of rigid link/joint systems, create goal-directed motion of linked figures using spacetime constraints, and animate flexible surfaces such as cloth[7, 5, 24]. The nonphysically based tools include a simple key-framing system and the Alpha_1 geometric modeler, shape_edit, which can be used to create models and to procedurally animate them. Rlisp[6], a Lisp variant, serves as the interface to shape_edit.

CHAPTER 3

ANIMATION PRODUCTION

The heart of the Story system supports the animation hierarchy. This support is provided by four objects: movie, scenes, shots, and storyboards. These objects model the traditional film hierarchy and structure the film, providing a focus for various production facilities (see Figure 1.1). By reducing production complexity, we increase the amount of time an animator can spend on creative tasks, allowing longer and more intricate animations to be produced.

We introduce storyboards as a design paradigm to oversee the entire animation process.

3.1 The Design

In the 1930s animators at Disney studios began to use story sketches, or storyboards, to suggest action, relationships, locale, and the appearance of characters[25]. Director and producer Woolie Reitherman states, "When you look at a board, it should reflect the feeling of the sequence so the viewer starts to pick up some excitement and stimulation"[25]. These storyboards were passed on to the layout artist whose job was to stage and dramatize each scene. This artist would design backgrounds, suggest motion paths for the characters and camera, and indicate the camera positions that would tell the story in the most entertaining fashion.

In addition to storyboards, animators developed other charting techniques to control the large amount of information generated during the animation process.

The *route sheet* lists the length, vital statistics, and name of the person in charge of every scene. The *exposure sheet* lists the dialogue, background, characters and various character attributes such as color and tone, and camera position for each frame in the animation[3].

The Story system adopts features of conventional storyboards, layouts, and route sheets and disperses the traditional information in scene and shot objects, and in the geometric models themselves. In computer animation, characters become 3D models and character attributes become model attributes such as texture and surface quality. The scene, shot, and storyboard objects in this section refer to C++ data structures. A scene object consists of a written description of the scene and its importance to the animation, the length of the scene (in seconds or frames), the group of people in charge of the scene, and the names of the props in the scene.

Each scene may also contain any number of shot objects. A shot object contains a variety of annotations that detail the people in charge of producing the shot and a description of the action. In addition, shots include a start and end time and the duration of the shot in the final scene and may also contain a camera, a list of props used, and a list of storyboards that initially inspired the shot. We discuss how the start and end times of a shot are determined in Section 6.1.

We provided the prop name list in scene objects to ease the task of assigning props to shots. In our implementation, a scene's prop names are the union of props contained in its shots. When an animator creates a new shot, prop names are inherited from the parent scene. This is a reasonable first guess because, by definition, all of a scene's shots take place in one location.

Storyboards contain an *image object* that consists, in part, of a pointer to a run-length-encoded image file. The image is intended to capture the same characteristics as traditional storyboard images including the layout of visible objects in the shot and the intended camera locations and motion. In addition to the actual file name of the image, image objects contain annotations detailing the creator of the image, version number, and comments about the image as well as other attributes that enumerate the renderer used (scanline or raytrace), whether the image was input from an external source or generated as output, and the time at which the image was rendered (for images generated from objects with motion curves). Accompanying each image object is a written description of the shot, sound effects, dialogue, and the name of the person responsible[29]. A storyboard may also include lists of props and cameras used in the storyboard.

Our system supports two types of storyboard images: those generated by the animation system and those imported from other sources. Initial storyboards may be scanned in from photographs or sketches. These images drive the model creation process and shot composition refinements. When the geometry associated with an imported image is complete, the objects may be composed in the storyboard. Objects contained in props are added to a storyboard by selecting them from the workspace. Story allow users to create and position cameras (see Chapter 4) to aid in viewing objects from different orientations. These cameras can be added to a storyboard in the same fashion as other objects. The animator can then generate a synthetic image of the composed objects using one of several renderers.

An animation is described by a strict object hierarchy, but the user is free to create an animated sequence in any order. For instance, if the user defines a shot before a storyboard, the system will automatically create movie and scene objects that contain the shot. Later, the user can rename and reorder these components to fit the evolving sequence.

Many postprocess techniques can be used to complete scene-to-scene transitions such as *dissolves*, *fade-ins*, *fade-outs*, *white-outs*, and *wipes* that our system might incorporate in the future. For instance, a fade-out (where the screen gradually darkens) might be implemented by gradually interpolating between the screen's pixel colors and black. This technique, usually completed in the final stage of the editing process, could then be integrated into the animation design cycle.

3.2 The Interface

A display window in the storyboard widget (see Figure 3.1) allows users to view wireframe representations of objects added to the storyboard. Users may interactively manipulate the view of objects displayed in this window, and if a camera has been added to the storyboard, the user may view the objects from the camera's perspective. Multiple cameras can be added to a storyboard allowing an animator to see the locations of several cameras while viewing the entire scene from another camera's perspective. Because the models in a storyboard are actually

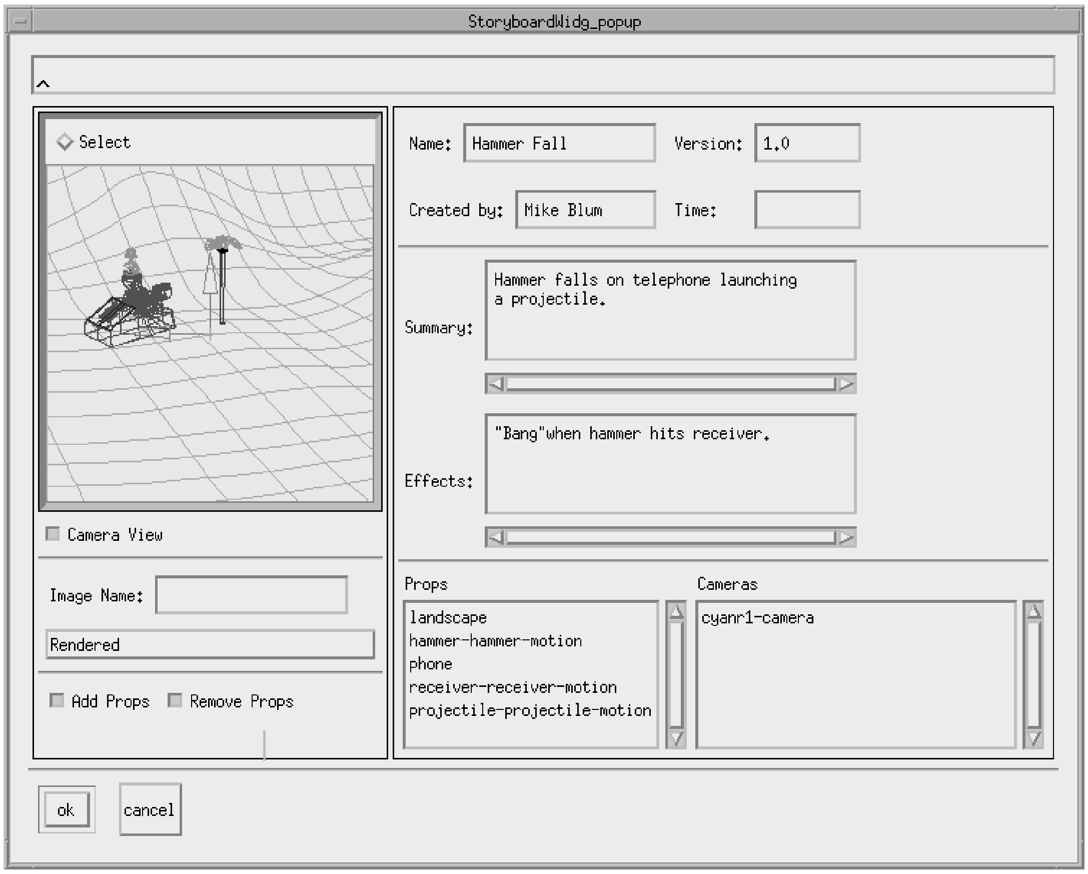

Figure 3.1. The storyboard widget. Objects are being viewed from the camera's perspective.

props, the system will automatically update the objects in the display window to reflect any changes in the component props (see Section 5.1.4). A straightforward modification to the system would recreate synthetic images when changes to the component props occur.

Although images currently exist only in storyboard objects, images in scene or shot objects might prove helpful in describing the overall view of a scene. This has not been implemented because the benefits gained by such an implementation were not worth the added interface overhead. Users can still create images that describe the overall feel of a scene and by placing the image in the first storyboard of a shot, they can derive many of the benefits sought without unnecessary complexity. If this implementation proves insufficient, integrating image objects with scenes or shots would be a simple procedure.

Adding, deleting, and reordering storyboard, shot, and scene objects are performed by highlighting the object from a list of object names and selecting the desired action from a menu. The ordering of elements at each level of the hierarchy dictates the order of the final animation. These features allow an animator to sketch out the structure of a movie using simple mouse clicks. To reorder a shot, the user first selects the shot to be moved and then selects the new location of the shot. A small message area in the main interface widget gives the user instructions to make manipulating these objects easier. For instance, the system will prompt the user to "Pick a new location" when attempting to reorder a shot.

CHAPTER 4

CAMERAS

4.1 The Design

Cameras in our system are defined with a from-to-twist specification[9]. The camera is positioned at the *from* point; the center of view is at the *to* point. The camera may be rotated about the axis defined by $to - from$ by $twist$. A view matrix for each camera is calculated from this specification as follows:

1. Determine the plane passing through the *from* point of the camera with a normal coincident with the z-axis of the camera (defined as $to - from$).

2. Project the world coordinate system y-axis onto this plane and rotate this vector by $twist$. This vector is the y-axis of the synthetic camera.

3. A cross product of the y and z camera axes gives us the camera's coordinate frame, and the view matrix translates the camera's coordinate system into world coordinates.

Cameras also have a pseudo-zoom feature that is implemented by scaling the view matrix by a user controlled constant. A zoom produces different visual effects from translating or "trucking" the camera along its z-axis [29]. This difference manifests itself in the camera's 4x4 view matrix. A scale affects the rotational component of the matrix, whereas a translate affects only the translation vector of the matrix.

Next, we describe a variety of tools that help the animator position cameras.

4.2 The Interface

Each camera has a separate viewport that displays the view of the workspace from the camera's perspective. To prevent cluttering the display, this extra viewport may be selectively displayed and undisplayed by the user. Much as a television director may choose a particular camera angle from a palette of monitors, so too an animator has the ability to examine objects from different camera locations.

Cameras have two different display representations. They may be displayed as a trihedron of vectors indicating position and orientation or as a more familiar icon resembling a reel-to-reel movie camera. The first representation wastes little screen space and is quickly drawn, whereas the latter representation offers better depth cues but occupies more screen space and takes significantly more time to display.

Two additional features provide visual cues that help the animator place cameras. First, the *center of attention* or *to* point may optionally be displayed. Second, a skeleton representing the view frustum of a camera may be displayed. Figure 4.1 shows several objects including a camera being displayed in the workspace window. To help the animator distinguish one camera from another, we associate a color and a name with each camera. For instance, a blue camera might be named "blue1-camera." Users may change the default camera colors. To help the user match a camera with its viewport, a colored marker is drawn in each camera's viewport.

4.2.1 Positioning Cameras

Users may position and orient cameras using several mechanisms. The most primitive of these allows a user to enter numerical values for the *to* and *from* coordinates and for the *twist* angle. This form of control is useful if the camera's exact coordinates are known a priori or when small adjustments are needed. Sliders,

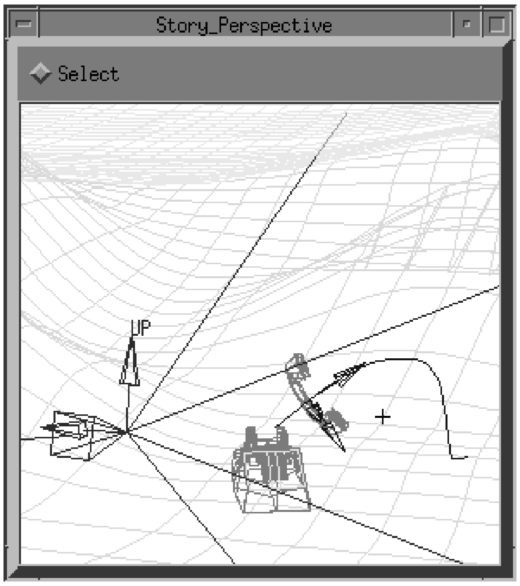

Figure 4.1. The Story workspace window. The camera is rendered as the trihedron on the left with the viewing frustum expanding to the right. The "up" vector is labeled. The small cross on the right is the camera's center of attention. The receiver's motion curve is displayed as a polyline and an orientation trihedron.

a somewhat less cumbersome tool, can also control each coordinate. In practice, neither method is very convenient.

A more intuitive technique allows either the *from* or *to* points to be snapped to objects in the workspace. This is implemented by casting a ray from the eye, selecting the intersection point on the closest object, and making this point the specified camera coordinate. This technique allows the general position and orientation of a camera to be quickly established. The current implementation allows for cameras to be snapped to surfaces and certain primitives such as cones, tori, and spheres.

Our system also supports the "flying vehicle control" metaphor for cameras that Ware and Osborne[27] chose as the metaphor best suited for movie making. The user can "fly" the camera from its own perspective (coordinate system) through the scene. The flying mode is selected from the same set of buttons used to change the view in the workspace. However, instead of mouse movements altering the view, they change the position and orientation of the camera. The user may translate or rotate the camera along or about any of its three axes. These tools make camera positioning straightforward.

CHAPTER 5

THE PROPERTY MANAGER

5.1 The Design

A typical animated film will use many geometric models, light sources, texture maps, and motion curves. In addition, each prop may have several *versions*, and different props may be designed to be interchangeable with one another. For instance, we may have two versions of a gun: one version may have a plain stock and the other an engraved stock. Models might also have versions of varying *complexity*. We might have a bounding box version of the gun sufficient for viewing a rough sketch and a complete trimmed NURB surface version we will use when rendering the final frames of the animation. By providing animators access to models with varying levels of complexity, we can more efficiently use computational resources, but we also introduce a significant model management problem. We address this problem by introducing a simple but flexible model database—the *property manager*. The property manager has three main functions: associate a name with an object, link an object with a motion curve, and generate a simple approximation for a complex model. The object created when we associate a name with a model, motion curve, light source, etc. is called a *prop*.

As an example, we can construct a gun prop. Suppose we have three different gun models: a Colt, a Luger, and a BB gun. We indicate to the property manager that each one is a possible *instance* of a gun prop and that the Colt is the currently active instance (see Figure 5.1). The active instance refers to the instance loaded in

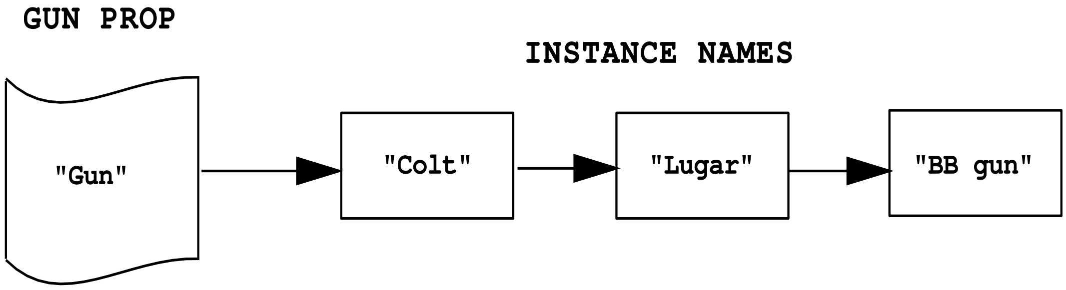

Figure 5.1. A "gun" prop has three instances with the Colt being active.

memory. Each model is now contained in a *prop instance*. Although the "gun" prop name has been bound to a Colt object, the version number and complexity measure are free to change. Further, the same Colt instance may be used in multiple props. Figure 5.2 gives a visual representation of the property manager data structures. Section 5.2.1 describes how such a property manager can be created.

An animator can now take advantage of the property manager's control features to construct an animation involving the gun prop. Initially, the animator might only have a crude model of a generic handgun. This model is added to a prop instance and used to make a rough "pencil test" of, for example, an old western shoot-out. Later, a more sophisticated model of a Colt is created and added to the gun prop. This new prop instance is used to refine the animation to ensure that the gun does not intersect the cowboy's hand. Refinements are initially verified by previewing the animation in real-time wireframe mode (see Chapter 6); however the refresh rate of the display device might be unacceptably low. In order to preview the animation at a faster rate, the animator might design a new model of the Colt that has roughly the same shape as the original but is not as complex (see Section 5.1.3). This less complex model would be added to the gun prop and used to preview the animation. When the animator renders frames for the sequence to make accurate timing tests, the more complex version of the Colt can be made active. Further, other individuals might make refinements to the Colt model during the animation

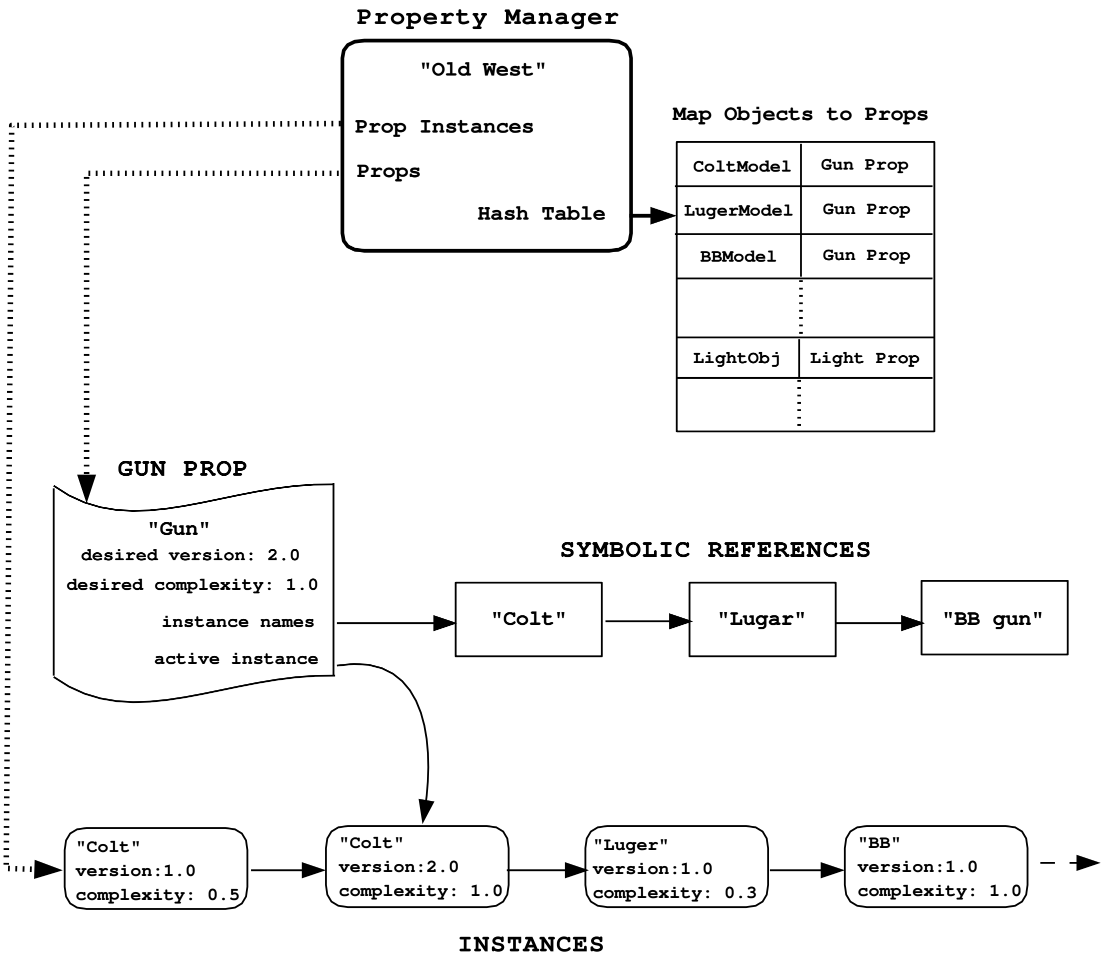

Figure 5.2. A detailed representation of the property manager data structures and the "gun" prop. The hash table uses virtual addresses to map objects to props.

design loop. We can keep the original model by placing the refinements into new prop instances with varying version numbers.

As Figure 5.2 describes, each prop has a notion of the desired version and complexity of its active instance. When the user varies these values, the property manager attempts to load the instance that most closely matches the requested version and complexity as follows:

```
closest = NULL;
for each <prop_instance> in <prop manager>
   if <prop_instance->name> == <desired name> &&
      <prop_instance->version> == <desired version>
      if NOT closest
         closest = prop_instance;
      else
         diff_current = fabs(prop_instance->complexity -
                             <desired complexity>);
         diff_closest = fabs(closest->complexity -
                             <desired complexity>);

         if ( diff_current < diff_closest )
             closest = prop_instance;
```

Because the version attribute of a prop instance is a string, we force the desired version and actual version to match exactly.

5.1.1 Automatic Propagation of Changes

Objects may be passed from Story to a shape_edit server (see Section 5.2.4). These objects may be modified in shape_edit with changes propagated back to Story. This type of interaction presents some problems when the objects that are being modified are contained within props. This difficulty arises because both Story and the shape_edit server assume they have absolute control over objects in their memory space.

Models in the server exist within a directed dependency graph. When an object in the graph is modified, all of its dependent objects must be updated. The server updates these objects by constructing new objects and deleting old ones. Story, however, uses virtual addresses as keys to map objects to props (see Figure 5.2). We use virtual addresses because they uniquely identify objects and can be extracted with very little overhead. When the server constructs a new model, the old key becomes meaningless. We update the address map by intercepting an event sent by the server to our client. The event contains pointers to both the old and the updated object; if the old object corresponds to a key in the address map, we delete the old hash table entry and construct a new one associating the updated object with its prop.

In general, the property manager does not create models but rather manages associations between models and names and provides flexible access through specification of name, version number, and complexity measures. There are, however, two instances under which the property manager will create models. The first occurs when associating a model with a motion curve and the other when generating simple versions of models. We discuss these two cases next.

5.1.2 Linking Props and Motion Curves

The Alpha_1 modeler supports an *instance* data structure that consists of a geometric model and a transformation. A transformation can be a collection of simple matrices or motion curves. In order to maximize the flexibility of models and motion curves, the animator can separately register models and motion curves with the property manager and link them dynamically while composing an animation. This linking is achieved by creating an instance whose geometry and transformation are both property names, not actual geometry or transformations. Because the references are symbolic, by changing the value bound to a property name we change the representation of the linked object (see Figure 5.3). For instance, suppose we have a "projectile" prop and a "trajectory" prop. Initially the "projectile" prop might have an active "jet" instance. We can create a new prop, called "flying" prop, that links "projectile" and "trajectory." When we preview the animation using the tools described in Chapter 6, we see a flying jet.

Now we can switch to another instance of the "projectile" prop, perhaps a "train" instance (we can make anything fly in our world!); the corresponding "flying" prop now has the train attached to the motion curve. This facility allows an animator freedom to mix and match geometry and motion quickly and easily.

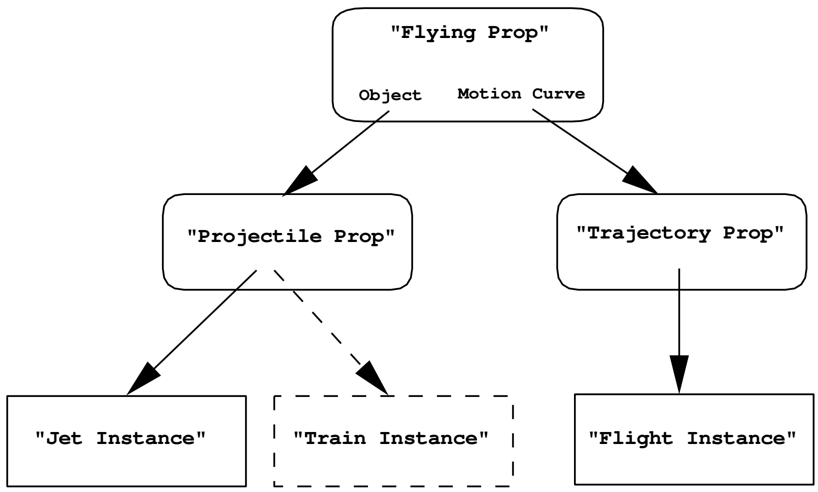

Figure 5.3. Symbolically linking two props. The current configuration would show a flying jet but could easily show a flying train.

Another advantage of using symbolic reference is that many props may share the same motion. Users could construct an animation of a column of marching soldiers using the following technique. First, construct a model of a soldier and the motion curve to animate her. Then construct other instances of the soldier offset from the original model. Finally, create props for each model and the motion curve and then link each model prop with the motion curve prop. The transformations of the motion curve are applied after the offsetting transformation of each instanced soldier, so when the props are animated, a column of marching soldiers in perfect lock-step will be previewed.

5.1.3 Generation of Simple Models

The property manager can store and load objects modeled at varying levels of complexity. Switching between these models on-the-fly allows fast display rates on low performance graphical devices as well as better time/quality trade-offs during rendering. The dimensions of the actual object should be closely approximated in the less complex models. For example, both a simple and complex model of a hammer should be the same height. We can use the simple model to quickly rough out the animation and substitute in the complex model for fine tuning the motion and generating final images. If like models do not share dimensional correspondence, substitution may be meaningless. We present several algorithms that maintain correspondence by generating lower complexity approximations from full geometry.

Although some work has been done on generating simple models from polygonal data [13, 22], our algorithms are designed to work on an arbitrary set of, possibly trimmed, NURB surfaces.

The first two algorithms take an unsophisticated approach to generating simple models by extracting either a bounding box or *boundary curves* from each surface. Boundary curves are the curves constructed from the control points of the first and last row and column of a surface control mesh. To extract simple bounding boxes:

```
simpleModel = NULL;
for each <srf> in <model>
    bbox = extract_bounding_box( <srf> );
    simpleModel = append( bbox, simpleModel );
```

To extract bounding curves:

```
simpleModel = NULL;
for each <srf> in <model>
    curves = extract_boundary_curves( <srf> );
    simpleModel = append( curves, simpleModel );
```

These algorithms are insufficient because they generate models that are typically as complex as the full model. The resulting "simple" models are often more cluttered than the original and show relatively poor data reduction qualities. However, because these models are easily polygonized, it takes less time to raytrace or scanline render these models than their NURB counterparts.

Figure 5.4 shows a wire frame rendering of a NURB telephone base. Figure 5.5 shows a version with a bounding box for each surface.

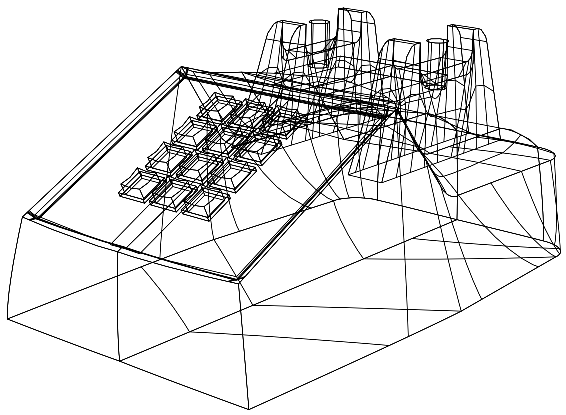

Figure 5.4. A wire frame rendering of a complex B-spline model.

To be truly useful, simple models should extract the key features of the full model. We determine key features by calculating the area for each surface and comparing this value to the surface area of the entire model. If the ratio of the surface's area to the total area is greater than some user defined total percentage, then a bounding box or boundary curves are extracted from the surface. These algorithms work quite well, preserving the general shape of a model, while greatly reducing complexity. The ratio of a surface's area to total surface area also serves as a complexity measure for the property manager.

Figure 5.6 shows a bounding box version of the telephone with a complexity of 0.020. This measure means that all surfaces with 2% or more of the model's total surface area have been extracted. The smaller the complexity value, the more detail contained in the extracted model. Figure 5.7 shows derived models with

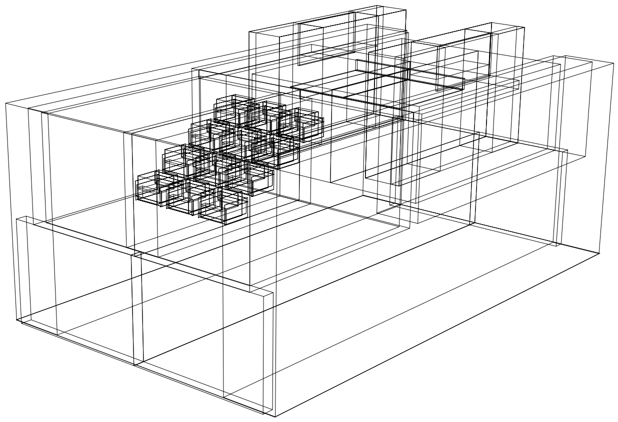

Figure 5.5. This model was generated by constructing a bounding box for every surface.

complexities of 0.034 and 0.020, respectively.

Surface area is approximated by subdividing each surface (within some tolerance) into polygons and summing the area of each triangle generated in the subdivision process. The algorithm that extracts boundary curves is summarized below; the bounding box variant is similar.

```
complexity = user defined value [0-1].
totalArea = compute_surface_area( <model> );
simpleModel = NULL;
for each <srf> in <model>
  area = compute_surface_area( <srf> );
  if ( area / totalArea >= complexity )
    curves = extract_boundary_curves( <srf> );
    simpleModel = append( curves, simpleModel );
```

While the boundary curve algorithm can generate models that closely approximate the key features of their full complexity counterparts, it does not closely

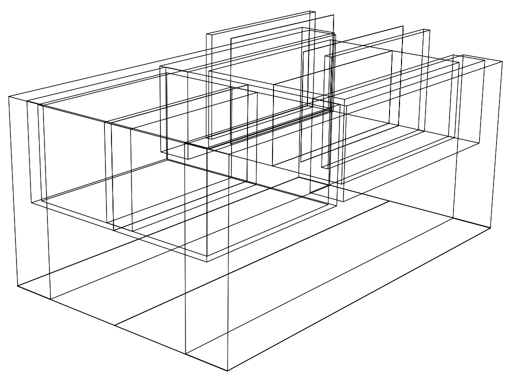

Figure 5.6. This model was generated with a complexity measure of 0.020.

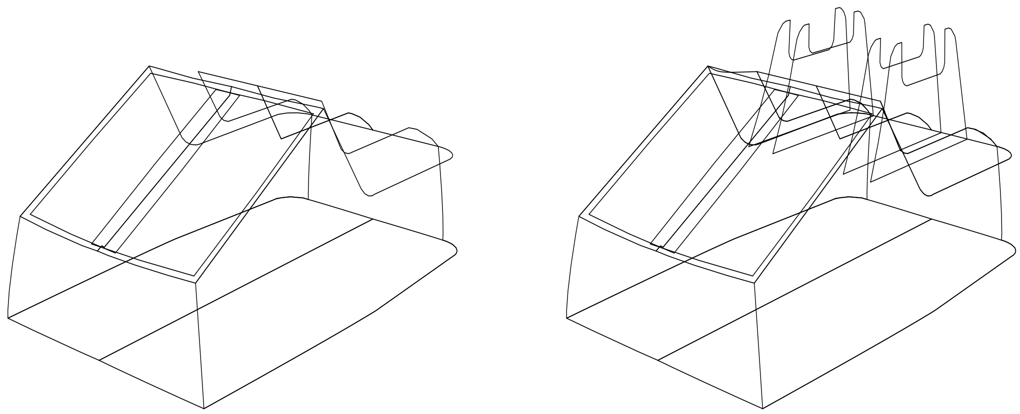

Figure 5.7. Two boundary curve versions of the telephone at complexity 0.034 (left) and 0.020 (right).

approximate all models because a NURB surface can vary arbitrarily within its boundary curves. For instance, the boundary curves of a surface with a large "bump" in the middle will not accurately represent that feature. Animators are encouraged to model with this limitation in mind.

Although boundary curve models work well when displayed in wireframe, they are not as useful when rendered with hidden surface algorithms. For "pencil test" renderings in which an animator checks for occluding surfaces, these simple models yield little information. An easy enhancement to this algorithm would extract surfaces instead of boundary curves.

The computation time needed to generate a simplified model from even a fairly complex NURB model is small enough to make these algorithms acceptable for interactive work. The telephone base contains 172 surfaces and can be approximated in under one second on an HP 9000/730 workstation. Any number of simplified models can be generated for a given model.

5.1.4 Additional features

The property manager is designed to be as flexible as possible while still providing a useful mechanism for structuring animation. For this reason, models may be referenced through a variety of different prop names. For instance, one animation may use a model of a pair of scissors as a "murder weapon" prop, whereas another animation uses the scissors as a "barber shop" prop. By allowing quick substitution of one prop instance for another, an animator can easily experiment with different ideas. In an extreme case, a scene might consist of the three props: landscape, hero, and action. The property manager allows the scene to be a cowboy walking down the street of a western town or a ball falling down a set of stairs. In addition, the ability to rapidly substitute one level of model complexity with another simplifies progressive refinement of an animation.

Complex geometric models in Alpha_1 are on the order of 100K bytes, and manipulating them quickly requires careful use of physical and virtual memory. To make efficient use of memory, the property manager lazily loads models associated with props and retains only the active prop instance. This optimization means that only one model per prop resides in memory at a time and that when the user changes prop instances, the memory used by the old model is freed and the new model is loaded. Reducing memory requirements improves overall performance of the system at the cost of short delays when a model is first loaded.

The following sections describe the various facilities Story provides to construct a property manager and change the active prop instance.

5.2 The Interface

A property manager can be created in two ways: The contents of the database may be specified as a text file that the system will parse to generate the database, or a user may interactively construct the database within Story.

5.2.1 Constructing a Property Manager with a Text File

The text file format requires instances to be specified before props and is defined as follows:

```
inst <instance-name> <version> <complexity> <path to model>
....
prop <prop name> <prop-instance> <desired version> <desired complexity>
....
```

The `prop-instance` field in the prop specification is the name of the prop instance to associate with the current prop. If more than one instance is associated with a prop, multiple entries for a prop can be made. The last entry will be the active instance. A "projectile" prop with both a "shoe" and a "rocket" instance could be defined as:

```
prop projectile shoe 1.0 0.00
prop projectile rocket 2.0 0.3
```

where "`rocket`" is the active instance.

5.2.2 Interactively Constructing a Property Manager

Story also allows a property manager to be built interactively. Users may import models directly into the system, and if a selected model is not found in the property manager, the user is asked to add the object to the database (see Figure 5.8).

The user can enter the name of the prop, the prop instance, version, and complexity number before adding the object to the property manager. If the object has a "name" attribute, the system will use that value as the default prop and instance name. This default choice allows even a novice user to build a property manager quickly. If the object being added is a simple model generated from an existing prop, the system will use the prop's name and instance name as

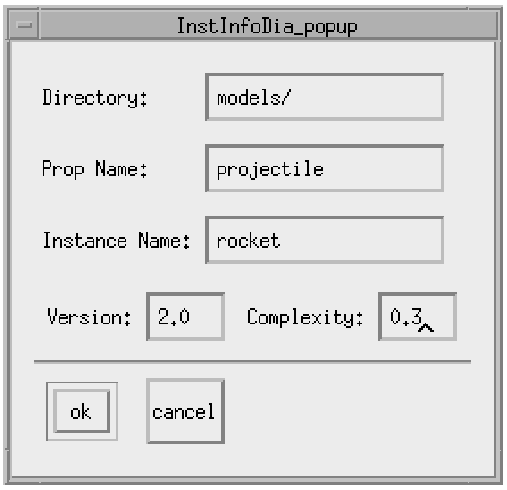

Figure 5.8. Users may interactively add models to the property manager.

defaults. The system assumes a default version of 1.0 and a default complexity of 0.0; however, if the object being added is a simple model, a default complexity equal to the user defined complexity is used. Any of these default values may be overridden by the user.

Because of time constraints, we chose to implement a simple interactive property manager editor. A more sophisticated interface would include editing and deletion support of property manager objects. The current interface can dump the property manager out in text format, and by hand editing this file, users are able to perform all functions. Once editing has been completed, the system can parse the text file and reconstruct the database. The text file interface is also useful as a backup mechanism in case the binary version of the database is somehow corrupted. For instance, the full path of each object is stored in the property manager, but if an object is moved or deleted, Story will fail when attempting to load that object. We could avoid this problem by editing the text file to reflect the change and reconstructing the binary property manager. Remember that not only geometric models but also lights, textures, and motion curves objects can be stored in a prop.

5.2.3 Changing Active Prop Instances

Story provides an interactive facility for changing the active prop instance. The user may open a separate prop information widget that contains desired version and complexity figures for the prop as well as a list of all of the prop's instances. The user may change active instances by highlighting a different instance entry (see Figure 5.9).

5.2.4 Importing Models

Models and motion may be imported into our system in several ways. An animator can import Alpha_1 data structures stored in the standard Alpha_1 format

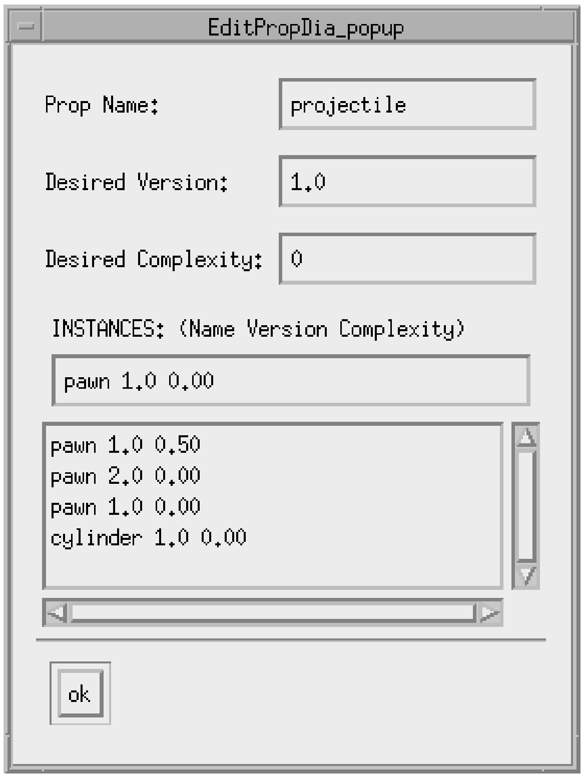

Figure 5.9. Users can query the property manager for prop information and can interactively change a prop's active instance.

or can access a shape_edit server directly through network interprocess communication. This second option allows the animator to perform modeling and animation operations in Rlisp and interactively display and then incorporate these models into the property manager. Further, because multiple modeling and animation tools may be in communication with the server, we can import geometry and motion from these other tools by using the shape_edit server as a gateway. This second mode of communication has not been used because most Alpha_1 clients currently do not support the necessary client/server protocols.

Once an object has been imported, the system will display the object in the *workspace*. The workspace is the main display and interaction area of the system.

Objects can exist without being associated with a prop, but if such an object is imported into Story in either of the two ways mentioned above, the animator can view or modify the object but cannot perform any system specific tasks such as previewing or rendering an animation, generating simple models, or selectively displaying and undisplaying an object. Users can view and construct objects within our system without the additional overhead involved in associating these objects with props (see Section 5.2.1), but once the user selects an "unknown" object, Story will attempt to create a new prop. Users can override this prop creation process.

We further exploit the capabilities of the server by allowing objects to be passed between Story and the server. For instance, a model could be directly imported via a binary file and then sent to the server to be modified. Objects modified in the server are automatically propagated back to the Story system to ensure correct correspondence. Both models and motion curves may be modified in this fashion.

CHAPTER 6

RENDERING

We refer to rendering as viewing any part of an animation in either wireframe or shaded image mode. We first discuss the "fast render" preview and editing functions. Previewing and editing typically use wireframe renderings of models but can use shading if hardware support is available. Next we discuss the high quality rendering facilities.

6.1 Preview and Edit Design

Motion quality and the timing of scenes are critical elements[16] in the creation of both film and animation. Therefore, the animator should get a feel for the final motion as early in the production as possible. To facilitate this, we have implemented a comprehensive set of previewing widgets. The previewing module has the ability to display not only geometric models but also such abstractions as cameras, viewing frustrums, and motion curves (see Figure 4.1). We can also preview storyboards, shots, scenes, or the entire animation from the director's vantage or through the camera's lens.

Producing computer animation has advantages over either live action or traditional animation. We can preview both cameras and motion curves and, unlike animate objects, their motion can be reproduced exactly. As described earlier, a camera can be represented as either an oriented arrow or a more realistic looking camera model. If a camera has been added to a shot, then the shot will be previewed from the camera's perspective; otherwise, the system's last viewing location is used as a default. The animator may interactively modify this default view. The view through several cameras can be displayed simultaneously, thus providing an animator with shot selection flexibility. A camera can display its viewing frustum to help an animator visualize the space visible by the particular camera. Motion curves are displayed by sampling the transformations over their valid range and forming a polyline through the resulting positions. A moving trihedron, representing the rotational component of the curve, is attached to the translation path.

A motion curve is defined by a 2D NURB whose x-axis is time and whose y-axis is the transformation's value at each time. By searching the control polygons of all of a shot's motion curves for the minimum and maximum x-coordinate, we determine the initial time range of the shot. The animator may then edit the default start and end times for the shot by either typing in new times or by setting one of these times to the current preview time. Editing a shot's time range changes the range over which the motion curves are sampled by the rendering software and effectively incorporates editing into the animation production cycle. Note, that we may overlap shots for emphasis or we may skip sections of time to achieve compression.

In our system, a shot's time range can be altered only while the shot is being previewed. The mechanism used to edit shots together could be greatly improved. The user should be able to manipulate interactively the start and end times for each shot as a slider. By displaying all sliders simultaneously, the user gains a more global view of the animation. Figure 6.1 is an example of a more sophisticated shot sequence editor. Sliders are represented as the end-marked dashed lines beneath the shot labels.

In addition to altering start and end times, animators may scale the actual length of each shot. For instance, the user may specify that motion begin at time 2 and end at time 4. When the shot is rendered, the number of frames generated

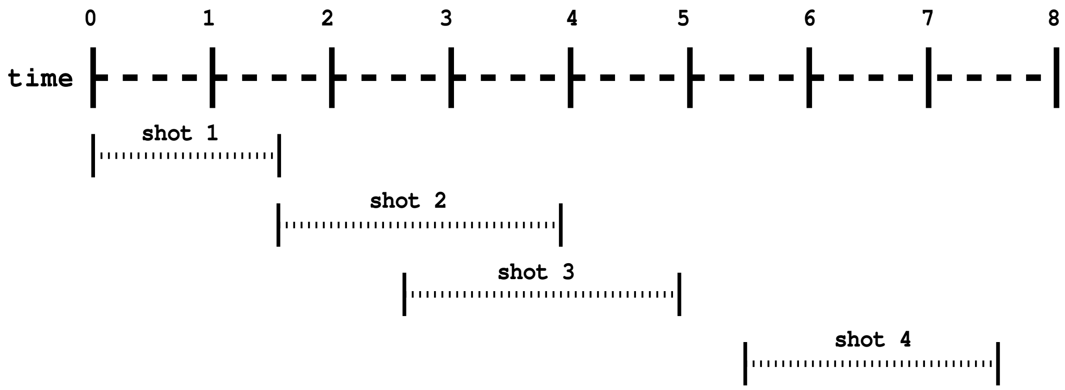

Figure 6.1. An example shot sequence. The overlap between shots 2 and 3 and the omission of time between shots 3 and 4 is evident.

is based on the 2-second time differential. If the animator requests that the shot last for 3 seconds, the action slows down; if the shot lasts 1 second, the action speeds up. Alternatively, this type of control could be implemented by modifying the motion curves themselves. Alpha_1 has a variety of utilities that generate and modify motion curves. The architecture of our system encourages application developers to develop new programs and modify existing ones to take advantage of the integrated support provided by Story.

As the animator modifies the start and end time for shots, these values are permanently stored in the shot object and are used when generating frames with one of Alpha_1's high quality rendering tools. Because our system does not provide timing control during real-time previewing (see Section 8.3.3), total time specification only affects the number of frames rendered with one of the auxiliary tools.

6.2 Preview and Edit Interface

The preview window is an integrated facility allowing the user to edit together and preview a sequence of shots in real time (see Figure 6.2). An animator can interactively specify a specific shot or a range of shots to preview. Time control within a shot is controlled by either the familiar set of VCR buttons indicating

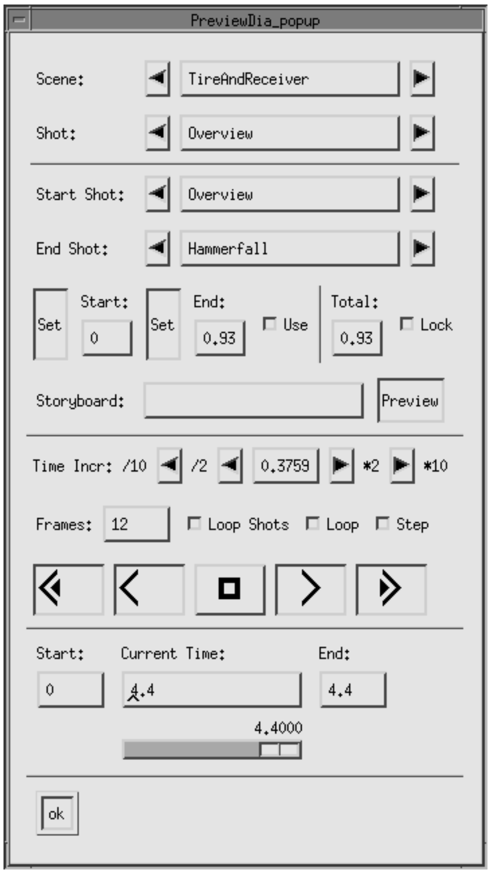

Figure 6.2. The preview control window.

stop, play forward, play backward, fast forward, and fast backward, or a slider bar. The tape player buttons include options that allow an animator to loop through a sequence of shots or step through the animation one frame at a time. The animator may change the time increment used to sample the motion curves, but this only changes the apparent length of the shot in preview mode, not the number of final rendered frames.

To interactively modify a shot's time interval, the animator must request that the previewer use the modified times (see the "Use" button in Figure 6.2). Once this request is made, the user is unable to preview the animation outside of the edited range. Otherwise, the shot's full time range can be previewed.

These tools merge previewing and editing into a tight interaction loop. Once several shots have been edited, the animator can preview the entire sequence by selecting the starting scene and shot and the ending scene and shot.

We have also provided limited storyboard previewing facilities. An animator can request that Story invoke an external display program and view all of an animation's storyboard frames in sequence. The storyboard frames can be a mixture of hand drawn and machine generated images.

6.3 High Quality Rendering

Our system supports high quality rendering by providing access to the rendering tools in the Alpha_1 modeling system; these include a fast scanline algorithm and a more sophisticated ray tracer. Both utilities provide options for changing the image size and quality (e.g., antialiasing, texture maps). The ray tracer also supports shadows, material properties, varying ray cast depth, and motion blur. The animator controls the renderers used, image size, and image quality through a rendering control panel (see Figure 6.3).

The animator can render a storyboard (a single frame), shot, scene or the entire movie. The timing of an animated sequence can be controlled in several ways. As with the previewing tools, the default start and end time of a shot are determined by examining its motion curves; by using the preview and edit tools, the animator can alter these start and end times to shorten or lengthen the shot. The rendering control window also allows the animator to indicate the length of a shot as either frames per second or frames per shot. The former is intended to be used when generating final frames, and the latter is useful when performing "pencil tests."

When the animator has set all rendering parameters, Story generates a Unix shell script to invoke the Alpha_1 rendering programs. Scripts have several advantages, foremost being that the actual rendering can take place at a later time in "batch" mode. Clearly, it is unacceptable to require the animator be logged in during the

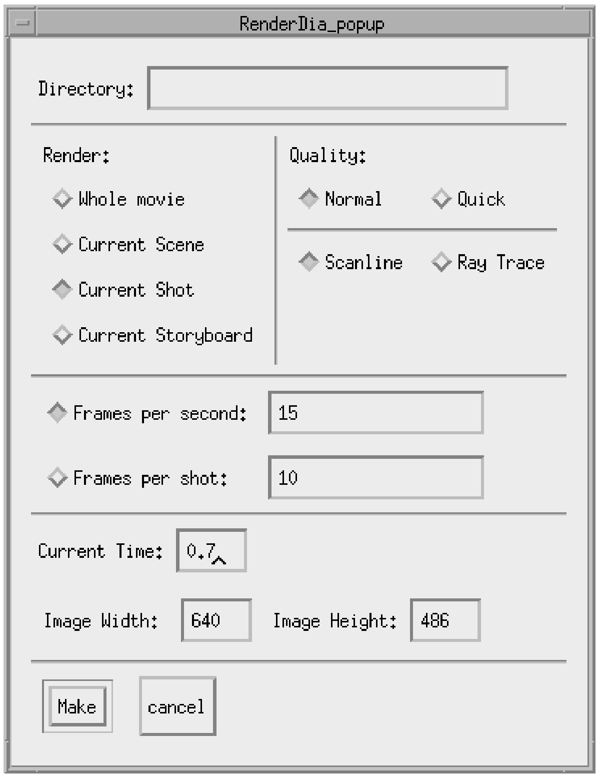

Figure 6.3. The high quality rendering control window.

time-consuming process of rendering, so frames must be generated without requiring a windowing system to be active. Scripts also have the advantage of being able to run on a different host from the one used to generate them. We can make better use of computational power with this facility. Also, the scripts are structured to allow the frames of a movie to be rendered in parallel, with each shot's frames computed on a different workstation.

CHAPTER 7

AN ANIMATION PROOFREADER

As computational power increases and rendering and animation packages become more robust, even amateur filmmakers will have the power to create animated sequences. These novices will need tools to aid them in producing quality animations. One such tool is a proofreader able to identify awkward camera placement, motion sequences or shot transitions. By identifying these problems, a proof reader would, among other things, help an animator choose camera locations, shoot multiperson dialogue, and edit action sequences.

Automating what many may hold as a purely artistic endeavor may seem futile. Indeed, much of a filmmaker's or animator's job is creative in nature. However, such an individual filters his or her creativity through a set of established rules in order to communicate effectively to an audience. A filmmaker who fails to obey these rules risks irritating if not confusing an audience[1]. An example is a scene in which two actors are having a conversation, and each time a character speaks, the audience sees a close up of the speaker's face. If these close ups are filmed so that the speakers appear gazing in the same direction, the audience will believe that the characters are talking to a third person and not to each other. Our goal is to identify "bad" animation practice and then devise algorithms or heuristics to uncover instances of these in an animated sequence.

Arijon[1] defines three different types of scenes. Scenes may have dialogue with action, dialogue without action, or action without dialogue. Because of the difficulties associated with natural language recognition, we attempt to analyze only the last classification of scenes.

Some relevant terms from film nomenclature are first defined.

7.1 Film Definitions

A *player* or *character* is an object in a scene on which the audience focuses attention. Any object, including a camera, may move within a shot. In the context of computer animation, any of these objects may have an associated motion curve (see Section 2.5). Motion in a *neutral direction* is either towards or away from the camera.

A shot's *center of attention* (COA) is the character that is the focal point of a particular shot. The *line of interest* (LOI) is a straight line in the direction of the COA's gaze or movement. If two characters were speaking to each other, the LOI would be based on the directions of their gazes. By alternately editing together shots of different centers of attention, the audience will get a sense of conflicting or related story lines. For instance, the audience will understand that two characters are speaking to one another if shots of each character are interleaved. The placement of the LOI is not always achieved as simply as in the previous example. If the two characters above were moving, the LOI would also need to move. Or the two characters may be having a conversation with a third person, requiring the shot to use multiple LOIs. Such shot sophistication is beyond the scope of this thesis.

The proofreading system must have an understanding of the scene-shot animation hierarchy as well as knowledge of each shot's COA and LOI. The proofreader must be able to query the speed and direction of motion for any object in a scene. For anthropomorphic objects, a representation of *direction of gaze* (DOG) may also be important.

7.2 The Triangle Principle

The triangle principle is a method for designing camera positions that allow clear visual coverage of the players in a scene[1]. We define a triangle that has one side parallel to the LOI. If we place a camera to one side of the LOI and two more cameras at either end of the LOI, we see that the cameras form a triangle. Another triangle may be formed by placing a camera on the other side of the LOI. Any visual composition may be covered by orienting these cameras in one of five variations[1]. Figure 7.1[1] illustrates these variations; the backs of the players are represented by the rounded blobs and the dotted line between the players represents the LOI.

These variations only dictate the placement of the two cameras at the base of the triangle. One option places these cameras in *external reverse angles*. In this case, the cameras are pointed inwards towards the players. The second option places the cameras close to the LOI and points them outwards covering each player individually. The cameras in this variation use *internal reverse angles*. By placing the internal reverse cameras back to back on the LOI a subjective point of view of the player excluded from the shot is produced. *Parallel position* cameras are deployed close to the LOI, oriented perpendicular to the LOI. This type of shot also covers players individually. These three options may be combined to increase the potential number of camera placements. The fourth variation places cameras at *right angle positions*. In this situation, the two cameras are pointed in directions 90^o apart. Finally, by advancing one of the cameras along its direction of view (either physically or with a zoom), we obtain cameras on a *common visual axis*. This arrangement is useful for emphasizing a player after showing it in its surroundings.

7.3 Background

The artists at Disney studios developed a set of guiding principles in the late 1920s and 1930s[25] that enabled traditional animation to become more sophisti-

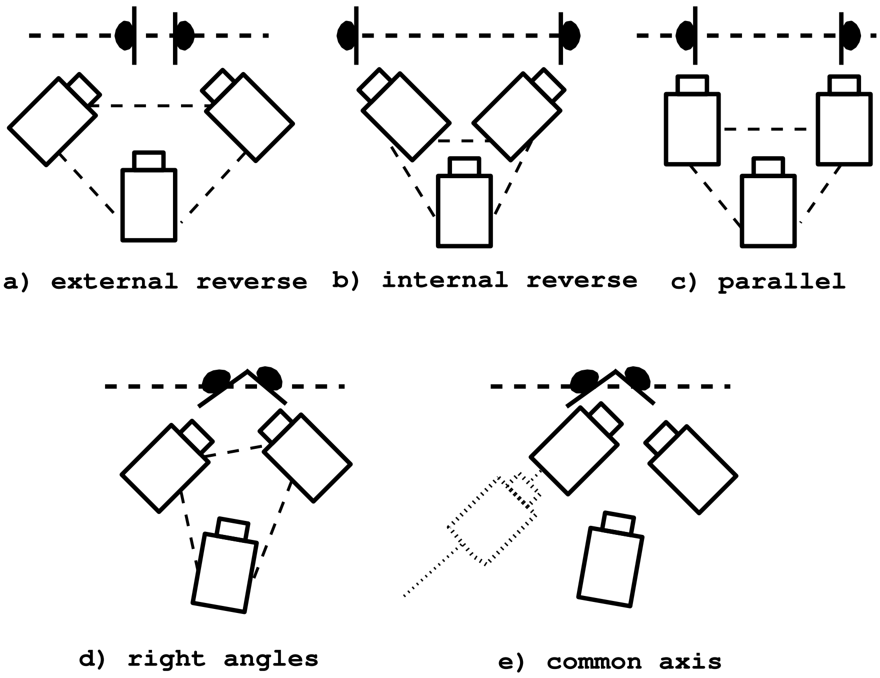

Figure 7.1. The five variations of the triangle principle.

cated and realistic. Lasseter summarized these principles and described how they applied to 3D computer animations[16]. These principles are:

1. Squash and Stretch—distortion of an object during motion.
2. Timing—speed of action to define weight and size of objects.
3. Anticipation—prepare the audience for an action.
4. Staging—clear presentation of an idea.
5. Follow Through and Overlapping Action—keeping continuity of motions.
6. Straight Ahead Action and Pose-To-Pose Action—two approaches to creating movement.
7. Slow In and Slow Out—avoid jerky starts and stops, especially for cameras.
8. Arcs—natural path of motion.
9. Exaggeration—accentuating key elements in an action.
10. Secondary Action—resulting action of another action.
11. Appeal—creating the elements people like to see.

All of these principles can be satisfied with current animation tools. For instance, spring/hinge dynamics systems can create objects that squash and stretch according to physical laws[24]. Just as cartoon animations often exaggerate these laws for entertainment purposes, physical models can also produce exaggerated motion if artificial laws are provided. As a consequence of obeying physical laws, objects demonstrate effects of timing and follow through; a variety of systems[2, 5, 7, 24, 28] have modeled these laws. For instance, when a crane with an attached wrecking ball is suddenly stopped, the ball will continue to swing after the crane comes to rest. In the hands of an artist, the effects of spline interpolation are characteristic of both Anticipation and Follow Through. Straight Ahead Action has its parallel with iterative techniques[5], while Pose-To-Pose action may be thought of as keyframing.

How can a piece of software determine what objects should obey these principles and to what degree they should obey? In general, a rock should not squash like a rubber ball when it hits the ground, but perhaps that is exactly the effect an animator desires. A dynamics system can model a man skidding to a sliding stop, but how exaggerated should the slide be, and what pose should the man assume to be "appealing"? Although there may be ways of measuring quantity and quality of these principles (see Section 7.5.6 for a possible "appeal" metric), we believe these problems do not lend themselves well to computer analysis, and any algorithm that is designed will probably be too clumsy to yield useful information.

Automation, however, can play an important role in resolving some of Disney's other principles. The linking of storyboards to camera selection will help the animator decide the location of objects or *staging* of each scene, and an animation proofreader should be of great use in identifying staging problems such as the COA being completely occluded by a foreground object.

Many other rules have been formalized by Arijon into a film grammar. Although the rules were written for makers of live action films, they are meant to ensure the clear presentation of any story in a film setting and are therefore just as important to the computer-assisted animator.

In order for a computer program to be able to interpret many of Arijon's rules, the system would need to grasp such difficult concepts as dialogue, storyline, expression, mood, and emotion. For instance, if an actor has a far away, dreamy look in his eyes, an audience would not be surprised to see a cut to a dream sequence. If the audience does not recognize the expression and thus anticipate the context switch, the dream sequence would be confusing. How, though, can a program recognize day dreaming or understand what day dreaming may imply? Such issues are topics for current artificial intelligence and expert system research and do not appear to be tractable problems at this time.

There are rules, however, that are more amenable to software analysis. Most of these rules ensure clear staging and editing of animations. The proofreader still requires information about the animation in order to check for violation of these rules, but this information may be easily specified by the user.

7.4 Rule Primitives

The following are primitive pieces of information the proofreader must determine in order to analyze an animation.

7.4.1 Screen Sectors

The *screen* is defined to be the view plane of the animation. This view plane could be a monitor or movie screen, but in general, most analysis will be done using the projection plane of the viewing frustum, so this too will be considered a screen. Most rules require knowledge of object location in screen space, and although some complicated space partitioning structure[9, 23, 26] might seem necessary at first glance, live action filmmakers generally do not require much accuracy[1] and therefore break the screen up into just two or three vertical screen sectors. We adopt this convention, although we admit that the reason film makers have not required higher accuracy might be a function of not having appropriate tools available to them. When discussing the screen split into two sectors, we will refer to the sectors as *half screens*.

7.4.2 Locating Objects

Because many rules require *screen sector coherency* between shots, the proofreader must be able to query the screen location of any object at any time. Screen sector coherency dictates that an object in one screen sector in one shot must often be in the same sector in the next shot. This check must be made at every time step for moving objects and for all objects if the camera is moving.

We can make this check by projecting each object onto the view plane. New objects are formed by intersecting the vertical lines that divide the screen with the projected model. The area to the left and right of the vertical line is then calculated. An object is in a particular screen sector if the majority of the projected object is in the sector. As a rough approximation, we could project the object's bounding box. A more accurate approach would project the object's full geometry. Heflin's[12] shadow volume work could be useful in forming this second type of projection.

7.4.3 Object Direction and Velocity

In addition to determining each object's screen location, the direction and velocity of moving characters with respect to the screen must be determined. Many rules are dependent on whether a player is moving horizontally across the screen or in a neutral direction. Also, certain types of shot transitions may appear jerky if the speed of a player differs in the two shots, hence the need for velocity information.

To check for this, we need to be able to compute the velocity of a player with respect to the screen. Given a motion curve, we could project the true velocity vector onto the view plane to calculate this screen velocity.

7.5 The Rules

The number of rules that a proofreader needs to check to adequately analyze a shot increases dramatically as player motion and sophistication increases. Sophisticated motion means multiple objects moving in a shot, or wildly erratic motion. Player sophistication refers to how geometrically complex or elaborate a player is. The more detailed a player, the more difficult it will be to analyze shots involving the player.

Just as today's language grammar checkers often give superfluous and shortsighted advice to fluent writers, an animation proofreader could be expected to be similarly limited. This seems unavoidable. Any rule may be broken by someone who understands them, but novices break rules without understanding the repercussions of their decisions. If a program can identify and warn a novice user when some guideline is violated, the user can at least make an informed decision. If we can formulate rules of thumb and can generate algorithms capable of detecting violation of these rules, the system should generate useful information.

With this in mind, we introduce our rules.

7.5.1 Simple Rules

One film convention dictates that the COA should remain unoccluded in the foreground of the shot. We can relate what constitutes foreground to camera position and check that the COA falls within some acceptable range from the camera. Surface occlusion can be determined with standard shadow volume algorithms[12, 9].

Not only should the the COA be unobstructed, but the area behind the COA should remain relatively uncluttered. We could clip each object against a bounding box of the COA and if more than a heuristically determined number of clips are detected, the frame could be tagged as "cluttered." Similarly, the COA should be colored in a way that distinguishes it from the background. We can check that the color of the foreground and the COA are significantly different. These rules ensure that an audience can easily identify the COA.

7.5.2 Switching Sides of the LOI

As a general rule, a scene should only be filmed on one side of the LOI so as not to confuse the audience. This rule can be verified by computing the dot product of the LOI and a vector from the COA to the camera.

Unfortunately, there are exceptions to this rule, which complicate matters to some degree. There are several tricks that a director can employ to gracefully switch sides of the LOI. Each of these methods relies on the fact that visual slight-of-hand easily fools humans.

One technique is to use a *cut-away* shot before making the switch. A cut-away is a diversionary shot that does not contain the COA and makes the audience forget the current viewing direction. For instance, if we wanted to switch sides during a scene in which a man is typing on a computer, we could film him from the right, use a cut-away shot of the screen he is typing on, and then film him from the left.

If a scene has a player moving in a neutral direction, we may also safely switch sides of the LOI. Presumably a player moving obliquely towards or away from the camera may also be used for the switch, but Arijon does not specify how oblique this angle can be without confusing the audience.

Another way to switch sides without confusion is to keep the moving player in the same half screen for both shots. For instance, in shot one, a player is seen moving from the right towards the center of the screen. Before the player passes the center of the screen, a switch to the other side of the LOI is made and the player is seen moving off to the right. This check can be made if we know the direction of movement of the player and if we can determine in which half screen a player resides at the beginning and end of a shot.

7.5.3 Character Motion

Motion is a necessary component to any animation, but with indiscriminate use, the audience may not focus their attention on each shot's COA. The eye will be attracted to a moving object in a still scene and to a still object in a busy scene[16]. So, the proofreader should check that if a shot's COA is moving, the rest of the shot's objects are generally still, and vice versa. We are not sure at this point whether it is more important for "still" objects to move more slowly than the COA or simply not as far in screen space.

The following choices ensure a smooth transition from one shot to another when animating motion (see Figure 7.2[1]). Motion may be repeated in the same screen sector either in the same or opposite direction, or a moving player may use both half screens if one of the following guidelines are followed. Motion can begin and end at the center of the screen, or motion can start on one side and finish on the opposite side of the screen. Finally, movement may either converge towards or diverge from the screen center.

Interestingly, the animator need not allow a player to move off screen before making a cut. In fact, faster and more dynamic transitions are often made by cutting before the player exits.

If the the motion from a single player is split into two shots, the player's speed should be similar in both shots. If this is not practical, then the speed in the second

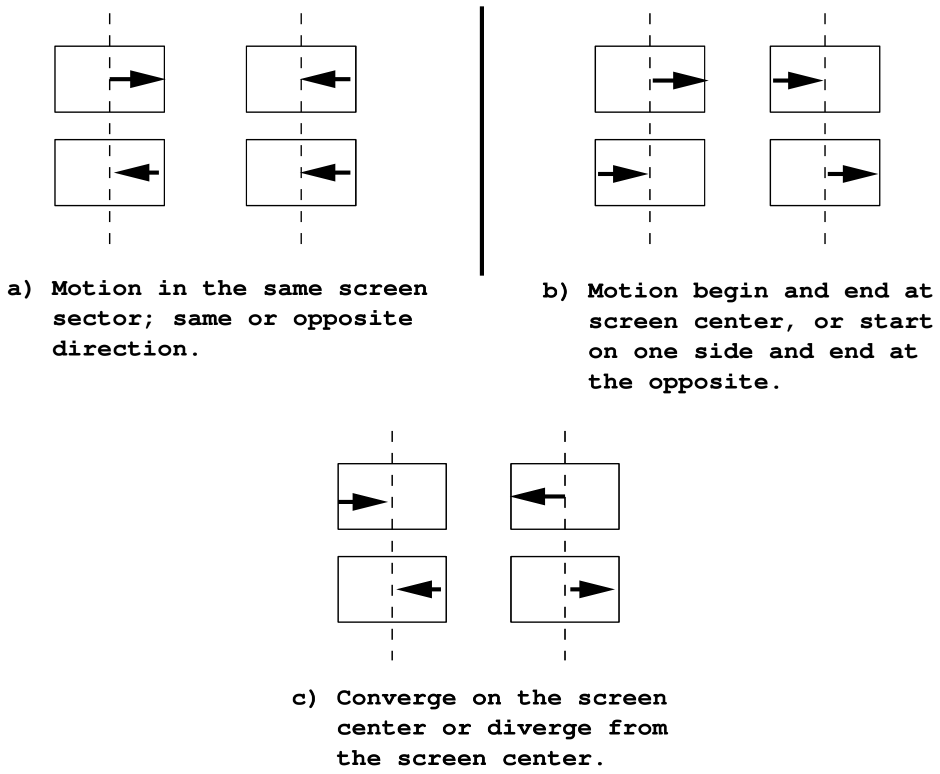

Figure 7.2. A director has several choices to ensure smooth shot-to-shot transitions.

shot should increase. Once the proofreader can determine the velocity of objects, this rule can be implemented with a few simple comparisons.

The audience should usually be given a clue as to where a player is moving. We know the player's final destination because we have the animation curves, so perhaps we could check that the general destination is displayed in at least one shot from the suspect sequence.

7.5.4 Camera Motion

In general, a lot of camera movement is distracting to an audience, so the animator should be warned once some predetermined threshold is reached. This threshold could be heuristically determined.

7.5.5 Constraints

Many of these rules could be used to drive a constraint solver to determine camera positions or paths that the system has determined to be "grammatically correct." The user should, of course, be able to override any of these default settings. Constraint techniques may also be useful in producing timing sequences that optimize some user specified emotional goal for a scene such as "surprise" or "tension." This is similar to the work of Kawai [15] in the field of rendering.

7.5.6 Strong Poses

Previously we mentioned determining an animation's "appeal." We now propose an algorithm for measuring the strength of a player's pose. Both traditional animation[16, 25] and live action[1] literature state that players should be posed to expose as much of their features as possible. For instance, a model of a woman should be staged so that the audience can see her face, arms, and both legs.

Shadow volume techniques[12] could be used to calculate the amount of self-occlusion in a particular shot. The animator is issued a warning when too much occlusion is detected. This problem is difficult because not all portions of a model should be considered self-occluding. For instance, the fact that the front face of an arm occludes its back half should not be included in a tabulation, but the fact that the entire arm occludes the chest should.

CHAPTER 8

RESULTS AND FUTURE WORK

8.1 Results

Several short pieces of animation have been completed using the Story system. The first segment showing a hammer falling on an inverted telephone receiver that in turn launches a chess piece was broken up into a series of shots to enhance the dramatic appeal of the scene. This is one scene of an intended longer sequence that follows a rolling tire around a Rube Goldberg-type landscape. Storyboards were created for several scenes. Initial storyboards were hand drawn and scanned into the system. Later, as models were designed, synthetic storyboards were created. We made simple pencil tests by first viewing the storyboards (as a mixture of hand drawn and system generated frames) in sequence and then by generating rough, sample frames of the animation. These pencil tests became progressively more sophisticated as models were completed and motion was fine tuned.

Another piece of animation, one that simulates the inspection of a mechanical part, illustrates the importance of animation in the manufacturing domain. By using a variety of camera angles, we were able to explain clearly the inspection process. By editing the long inspection, we were able to convey this information in a concise yet informative fashion.

Both sequences were constructed in less time than previously possible with the Alpha_1 system.

8.2 Current Status

The animation production (with storyboards), camera placement, and property manager features of the Story program are largely complete. The software includes support for X window and Silicon Graphics GL display devices and runs on a variety of hardware platforms.

8.3 Future Work

Although each module of the system (with the exception of the proofreader) is functional, each can be improved and expanded considerably.

8.3.1 Cameras

The camera positioning system, although adequate for many applications could be improved. The most obvious improvement would allow camera motion. A simple camera keyframing system could be added to either the Alpha_1 keyframing program or Story. Story would then need a few, relatively simple modifications to recognize moving cameras.

The current mechanism employed to snap the *to* point of the camera is deficient when attempting to position a camera in an area where there is no object. This difficulty could be alleviated by building a sphere widget temporarily placed around a selected object. The radius of the sphere is interactively controlled and the user could position the camera along any isoline of the sphere. A prototype of this interaction has proved very effective. We would eventually like to implement a more robust camera positioning system such as Mackinlay[17] or Gleicher[10] have developed. Such a system would require a constraint solver that could also be used to construct temporal constraints.

The physics of our cameras could also be improved to allow for such effects as field of view, focus, etc.

8.3.2 Property Manager

The property manager could be enhanced in several ways. For instance, we would like for the system to be able to handle multiple or linked property managers. If a property is not found in the active database, others would be searched. We would also like to integrate the property manager concept more fully into the Alpha_1 system so that, for instance, the ray tracer can work on a property manager with the most complex versions of objects loaded, while Story is using the least complex versions. A simpler mechanism of dealing with model attributes also needs to be investigated.

8.3.3 Previewing

One previewing enhancement would allow the animator to preview the entire animation in a single window with each shot being displayed as a storyboard, wireframe, or final shaded images according to the best representation available. Another important enhancement to the previewing module would allow accurate timing control. Previewing is implemented by sampling the motion curves for each object in a shot, calculating the position of the object along its motion curve, and displaying the object. Although this algorithm produces smooth motion, the apparent speed and duration in real time is dependent upon cpu speed and graphics performance. To properly handle timing while previewing, a system must monitor the update rate of the graphics display device and calculate the next frame according to this delay.

We also need to develop a more sophisticated time management interface. An animation has three distinct notions of time that may be manipulated. *Audience time* refers to the total subjective time that an audience spends watching an animation, *story time* refers to the actual start and end times that each shot is defined over, and *film time* refers to the order in which the various shots may be arranged in audience time.

8.3.4 User Interface

The addition of several interface widgets would improve the performance of the system. The first widget would relieve the user of the task of showing and unshowing objects in the workspace. The current implementation has a rather unsophisticated notion of display lists—individual objects are either in the display list or are not. The new widget should allow objects to be added to display groups so that layers of the model could be selectively displayed. For instance, we might place all dimensioning elements in one group and the actual model in another. Individual objects or groups of objects should be able to be displayed/undisplayed or moved to different display groups in a minimum of mouse clicks.

8.3.5 Animation Proofreader

Although the underpinnings of the animation proofreader have been largely worked out, little has been implemented. Clearly, we can only verify the utility of the proofreader through experimentation. Experimentation will require additional interface overhead to allow the user to specify or modify each shot's COA and LOI as well as code to analyze animation sequences. We believe that many common staging problems could be detected by implementing just a few rules.

CHAPTER 9

CONCLUSIONS

We present an integrated animation and storyboarding system which simplifies the creation of multiscene animations. Our system models both the process of animating a story and the product of the animation by introducing objects for movies, scenes, shots, and storyboards. These objects help animators quickly sketch out the structure of a story and provide a framework for step-wise refinement of that structure into a complete film. Camera positioning is simplified by several intuitive placement techniques and "through the lens" or "director's" views of the scene. The models used in the animation are managed by a flexible property manager, which allows easily substituting one model for another or for automatically generating simple approximations for complex NURB models. The rendering module manages previewing, editing, and the rendering of high quality final frames. Finally, the animation proof reader identifies awkward views or motion sequences and provides advice to novice animators. The system integrates a diverse collection of tools into a consistent animator's workbench. Several short animations have been completed with this software, leading us to believe our system makes computer animation more accessible to the traditional animator and filmmaker as well as making animation a more useful communication tool.

REFERENCES

[1] Arijon, D. *Grammar of the Film Language.* Hastings House, 1976.

[2] Baraff, D. Coping with friction for non-penetrating rigid body simulation. In *SIGGRAPH '91 Conference Proceedings* (Las Vegas, Nevada, 1991), pp. 31–40.

[3] Catmull, E. The problems of computer-assisted animation. In *SIGGRAPH '78 Conference Proceedings* (1978), pp. 348–353.

[4] Chen, M. A study in interactive 3-D rotation using 2-D control devices. In *SIGGRAPH '88 Conference Proceedings* (1988), pp. 121–129.

[5] Cohen, M. F. Interactive spacetime control for animation. In *SIGGRAPH '92 Conference Proceedings* (1992), pp. 293–302.

[6] Computer Science Department, University of Utah. *Alpha_1 User's Manual*, June 1991.

[7] Dovey, D. Integrating geometric modeling and motion simulation for mechanical systems. Master's thesis, University of Utah, December 1990.

[8] Drucker, S. M., Galyean, T. A., and Zeltzer, D. Cinema: A system for procedural camera movements. In *1992 Symposium on Interactive 3D Graphics* (Cambridge, Massachusetts, 1992), pp. 67–70.

[9] Foley, J. D., van Dam, A., Feiner, S. K., and Hughes, J. F. *Computer Graphics: Principles and Practice, Second Edition.* Addison-Wesley Publishing Company, 1990.

[10] Gleicher, M., and Witkin, A. Through-the-lens camera control. In *SIGGRAPH '92 Conference Proceedings* (1992), pp. 331–340.

[11] Gracer, F., and Blasgen, M. W. Karma: A system for storyboard animation. In *Proceeding Ninth Annual UAIDE Meeting* (1970), pp. 210–255.

[12] Heflin, G., and Elber, G. Shadow volume generation from free form surfaces. In *CGI '93 Conference Proceedings* (1993).

[13] Hitchner, L. E., and McGreevy, M. W. Methods for user-based reduction of model complexity for virtual planetary exploration. In *Proceedings of SPIE* (San Jose, 1993).

[14] Karp, P., and Feiner, S. Issues in the automated generation of animated presentations. In *Graphics Interface '90* (1990), pp. 39–48.

[15] Kawai, J., Painter, J., and Cohen, M. Radioptimization — goal based rendering. In *SIGGRAPH '93 Conference Proceedings* (1993).

[16] Lasseter, J. Principles of traditional animation applied to 3d computer animation. In *SIGGRAPH '87 Conference Proceedings* (1987), pp. 35–44.

[17] Mackinlay, J. D., Card, S. K., and Robertson, G. G. Rapid controlled movement through a virtual 3D workspace. In *SIGGRAPH '90 Conference Proceedings* (1990), pp. 171–176.

[18] Magnenat-Thalmann, N., and Thalmann, D. *Computer Animation: Theory and Practice.* Springer-Verlag, 1985.

[19] Phillips, C. B., Badler, N. I., and Granieri, J. Automatic viewing control for 3D direct manipulation. In *1992 Symposium on Interactive 3D Graphics* (Cambridge, Massachusetts, 1992), pp. 71–74.

[20] Ridsdale, G., and Calvert, T. The interactive specification of human animation. In *Graphics Interface '86* (Vancouver, 1986), N. Marceli Wein, Evelyn M. Kidd, Ed., pp. 121–129.

[21] Ridsdale, G., and Calvert, T. Animating microworlds from scripts and relational constraints. In *Proceedings Computer Animation* (Geneva, 1990), pp. 107–116.

[22] Schroeder, W. J., Zarge, J. A., and Lorensen, W. E. Inkwell: A 2.5-d animation system. In *SIGGRAPH '91 Conference Proceedings* (Las Vegas, Nevada, 1991), pp. 113–121.

[23] Thibault, W. C., and Naylor, B. F. Set operations on polyhedra using binary space partitioning trees. In *SIGGRAPH '87 Conference Proceedings* (1987), pp. 153–161.

[24] Thingvold, J. A. Elastic and plastic surfaces for modeling and animation. Master's thesis, University of Utah, March 1990.

[25] Thomas, F., and Johnson, O. *Disney Animation: The Illusion of Life.* Abbeville Press, 1984.

[26] Vanecek, G. Brep-index: A multi-dimensional space partitioning tree. Tech. rep., Purdue University, December 1990.

[27] Ware, C., and Osborne, S. Exploration and virtual camera control in virtual three dimensional environments. In *Proceedings of the 1990 Symposium on Interactive 3D Graphics* (Snowbird, Utah, 1990), pp. 175–184.

[28] Wejchert, J., and Haumann, D. Animation aerodynamics. In *SIGGRAPH '91 Conference Proceedings* (Las Vegas, Nevada, 1991), pp. 19–22.

[29] January 1992. Animation Interview with Lance Williams.

[30] Withrow, C. A dynamic model for computer-aided choreography. Tech. rep., University of Utah, June 1970.