Interactive Multiple Representation Editing of Physically-based 3D Animation

Wayne A. Christopher
Computer Science Division (EECS)
University of California at Berkeley

Report No. UCB/CSD 94-813

May 29, 1994

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Interactive Multiple-Representation Editing of Physically-based 3D Animation

by

Wayne Alfred Christopher

Dissertation submitted in partial satisfaction
of the requirements for the degree of
Doctor of Philosophy in Computer Science
in the Graduate Division of the
University of California at Berkeley

This research was sponsored by the Defense Advanced Research Projects Agency (DARPA), monitored by Space and Naval Warfare Systems Command under Contracts N00039-88-C-0292 and MDA972-92-J-1028, and in part by the National Science Foundation Infrastructure Grant number CDA-8722788.

Interactive Multiple-Representation Editing
of Physically-based 3D Animation

Wayne Alfred Christopher

Abstract

In recent years, much work has been done on realistic animation. It is now possible to simulate and animate a variety of different types of objects and physical effects. However, little attention has been paid to the problem of interactively creating and editing realistic animated scenes. Developing and modifying such scenes is still a batch-oriented process that depends on trial and error.

Animation editing for non-realistic scenes has also undergone rapid development. Presentation graphics systems, keyframe-based editors, and virtual reality-based interactive tools for a variety of purposes have been designed. However, physical realism is missing from these systems.

The principal result of this dissertation is the design and implementation of the Asgard animation system, an editor for creating physically-based 3D animation. Asgard provides the user with a flexible multiple-representation editing interface, which allows both direct-manipulation graphical definition and modification of the scene, and textual editing of a formal description of the objects, forces, and other parameters of the animation. Multiple instances of each type of view can be created, all of which are synchronized. Such an interface allows the user to utilize the most convenient and effective tools for each task in the definition of the scene.

Asgard also includes a differential equation solution technique that is adapted to the problem of physically-based animation, which uses an event-driven scheduling algorithm to take maximum advantage of the differing time-varying properties of different components in the scene while at the same time minimizing the penalties involved with collision detection and response. This algorithm partitions the state equation graph of the scene into strongly-connected components, which also allows Asgard to preserve as much previously-computed data as possible when editing changes are made by the user.

Asgard is part of the multi-media document processing system Ensemble. Animated scenes can be integrated with multi-media documents, as well as developed on their own.

Acknowledgements

I would like to thank my committee members: my advisor, Michael Harrison, for his support, guidance, and patience during my long and somewhat chaotic quest for a PhD, Susan Graham, for her advice and encouragement, and Mary Kay Duggan, for some very stimulating discussions on the history of printing and the Reformation.

My research would not have been possible without the help of the rest of the Ensemble group, including Steve Procter, Vance Maverick, Tom Phelps, Ethan Munson, Roy Goldman, Brian Dennis, Kannan Muthukkaruppan, Robert Wahbe, Tim Wagner, and Bill Maddox. Ming Lin, Ioannis Emiris, and John Canny significantly contributed to the Asgard project, in the areas of collision detection and convex hull calculation. I also owe thanks to Armin Wulf, Mike Hohmeyer, and the rest of the ICEM CFD team for providing a great place to work and some very good ideas while finishing this dissertation.

I would like to thank my aunts Alice, Elene, Mary, and Dorothy, my uncle Fred, and my cousins Bill, Ron, and Fred, for all they have done for me, and also Tom, Matt, Pat, and especially Ginger.

Finally, I would like to acknowledge all the people who have been important to me in numerous ways over the last few years, without whose influence I would have finished a bit sooner but had a lot less fun. Including but not limited to, and in no particular order: Pete French, Tom and Linda O'Toole, Sonny and Stan Friedman, Maureen Master, Carla Markwart, Birgitta Sylvan, Rosemary Busher, Leah Slyder, Amy Lee, Chris Loverro, and Ylva Lindholm.

Contents

List of Figures

Chapter 1

Introduction

Physically-based animation is the simulation and rendering of the motion of objects according to the laws of physics. Such animation is useful for a wide variety of applications, such as engineering, robotics, and education, as well as movies and games. The goal of the research described here is to develop algorithms and user interface techniques that make it easier for animators to rapidly create and modify animated sequences. As part of this research, I have written a prototype animation editor, called **Asgard**, which uses these algorithms and techniques.

There are two main issues that must be addressed in any animation system. These are the design of the user interface and the algorithms used for calculating the motion of the objects being animated. The organization of this dissertation reflects this fact: after the introductory, background, and architectural chapters, the user interface and the motion simulation algorithms of Asgard will be discussed separately.

This introduction will first describe some application areas for physically-based animation, and then discuss the problems with existing systems. Next, a brief overview of how Asgard deals with these problems will be given, together with an outline of the major components and capabilities of the system. Finally, the organization of the rest of the dissertation will be described.

1.1 Applications of physically-based animation

There are many engineering and scientific applications for physically-based animation. Most of these applications rely heavily on the accurate simulation of motion, which is the core of any animation system. Researchers in aeronautics and space sciences use *motion simulation* to determine how planes and satellites will move in flight. Mechanical engineers also use motion simulation to determine whether a machine operates as required, and to correct any problems, such as inadequate clearance for a moving part, before any physical models are constructed. Fields such as computational fluid dynamics use somewhat different algorithms than those used for discrete body simulation, for somewhat different purposes, but they still fall into the category of motion simulation.

Robotics uses motion simulation for the purpose of *motion planning*. One must be able to calculate how a robot will move, given particular control forces and object configurations, in order to determine whether the desired objectives are reached without collisions. An important part of these calculations is reverse dynamics, which calculates the joint torques given the desired ending positions. Motion simulation, in the form of forward dynamics, is required for verifying the results of a planned movement and for calculating the motion of objects in the robot's environment.

Physically-based animation is also used extensively in *science education*, often at the level of elementary physics classes. Systems are available which allow a teacher to set up a configuration of objects and forces, and then demonstrate how the motion takes place in this simulated world. More importantly, the students can modify the scene and experiment for themselves, often with situations that would be difficult or impossible to create in the real world. They can also view graphs of the motions of the objects, and qualitatively compare the simulated behavior with what the equations of motion predict.

In the entertainment industry, physically-based animation is becoming central to creating a convincing illusion of reality. Recent movies such as *Jurassic Park* use motion simulation extensively, and video games such as *Hard Driving* perform real-time calculations to create an accurate simulation of some aspects of reality. Until a few years ago, most available techniques were too slow to perform interactively in real-time, but modern hardware and new algorithms have made it possible to add physical realism to many types of games.

Finally, a very interesting area where physically-based animation is just beginning to appear is in *simulated environments*, also known as "virtual reality". The computational requirements for such applications are especially challenging: the motion must appear realistic, and must be calculated in real time with a short latency between the user input and the computed motion. Furthermore, it is especially difficult to predict the computational and communication requirements of virtual reality applications in advance, and the algorithms must be able to handle the availability of less bandwidth or CPU power than they actually want.

The research described here is not targeted to any of these specific application domains. Asgard is a prototype system for creating physically-based animation sequences, and the techniques employed by its user interface and motion simulation subsystem can be applied to any of these types of problems.

1.2 Problems with existing editing techniques

Currently, there are two basic ways to create animation that appears to obey physical laws. The first way is to use a system that allows the animator to specify motion in terms of explicit paths and pre-determined velocities, called a *kinematics-based* animation system, and to use one's own intuition or perhaps external simulation mechanisms to determine the motion. This sounds straightforward, and a great deal of physically realistic animation is currently produced this way, but there is no guarantee that the motion will be correct. In addition, most people's judgements about how objects actually move are faulty for all but the simplest cases: viewers can usually tell the difference between correct and incorrect motion, even if they can't create correct motion themselves. Finally, doing animation this way takes a long time and a great deal of effort, and if changes are made to initial conditions or object parameters, the animator must redo the motion from scratch, since the system cannot determine automatically what changes are necessary.

The second way to create physically-based animation is to describe the system at the level of *dynamics*, which means that in addition to a specification of the shape and appearance of objects, the input to the animation program also includes the mass, moment of inertia, and other properties of each object that would affect its movement, as well as a specification of the forces, torques, and initial conditions and other constraints that apply to it. Depending on the capabilities of the motion simulation system, other types of information might be required, such as deformability or elasticity.

This information is usually provided to the animation system in the form of a *textual* description. An animator typically must create an input file using a text editor that contains a complete description of the scene to be animated. He then must run it through the animation system to determine the result, mentally correlate the motion on the screen with the textual description, and then modify the input and rerun the motion simulation until it is satisfactory. This results in a tedious and inefficient development process for animated scenes of even moderate complexity. The conceptual distance between the static, textual input and the dynamic graphical output is too large for convenient editing. In addition, when an animator wants to introduce some kinematic elements, where the desired motion of the objects is known in advance and must be explicitly specified to the system, a textual interface is generally inappropriate and difficult to use.

There are many advantages to using a textual scene specification, however. Some elements of a physical configuration are difficult to specify any other way but symbolically, such as arbitrary force laws between objects and parameterized or procedural descriptions. Also, many users want to see the various inputs to the simulation process described explicitly, and many of these inputs cannot be specified using only graphical means in a reasonable way.

The reason that animation editing is such a problem is that the set of capabilities that a user requires is not easily provided by a single editing interface. The animator wants physical realism, but he also wants some amount of detailed control over the final results, and must be able to determine the balance between these two himself. A graphical, kinematic interface is best for control, since the animator is working very close to the final product and has the ability to create precise motion definitions. However, a textual interface is best for creating physically correct motion, since force laws and constraints are usually best manipulated textually.

1.3 Asgard's solution

The goal of the research described here is to develop a system that reconciles the sometimes incompatible requirements for precise control and for realistic motion. This system must allow interactive editing of animated scenes, which means that efficient motion simulation mechanisms and incrementality must be exploited whenever possible.

The basic approach that Asgard takes is to provide *both graphical and textual editing tools* simultaneously. That way, the types of interaction that are best done graphically, such as approximate spatial positioning, shape editing, and trajectory specification, can be done using a graphical editor, and those tasks that one wishes to perform using a language-based interface, such as definition of general forces and parameterized objects, can be done using a formal specification language.

The keys to making such a system usable are synchronization and incrementality. When a change is made in the graphical viewer, it should be immediately reflected in all other viewers, and the same is true, to a certain extent, for changes made in a textual viewer. A user must be able to move an object in a graphical viewer and watch the numeric value of the position change in all the textual viewers that are present, and likewise should be able to add new objects in a textual viewer and, after indicating that he has finished editing, see them appear in the graphical windows. As with many other editing and user interface applications [105], rapid and intuitive feedback is very important.

Because motion simulation requires a significant amount of computation, one would like to preserve as many results as possible when changes are made to the scene, either graphically or textually. Then, when the user wishes to update the motion of the objects, only data that is actually unavailable, or has been invalidated by changes the user has made, should be calculated. The motion simulation algorithms used in Asgard make this possible, while at the same time allowing for efficient simulation of large and complex scenes, with collision detection and response.

1.4 Overview of the Asgard system

The goal of the Asgard project is to develop user interface paradigms and low-level motion calculation algorithms that are appropriate for a wide variety of animation application domains. Each of these domains has a different set of requirements for a fully-featured editing system: to make movies, one needs a great deal of support for physically realistic rendering, and for robotics, appearance is less critical, but particular object modeling formulations for articulated bodies are required. Since Asgard is not intended to be a production system for any particular area, the types of objects, forces, and rendering facilities provided were chosen to be complex enough to expose the research issues, but simple enough to be tractable in the context of this project. For example, articulated bodies are provided, but using a simple spring-based model rather than the more complex analytical formulations used in robotics.

1.4.1 Types of animated scenes supported

The kinds of objects that can be created, simulated, and displayed in Asgard are rigid polyhedral and spherical bodies. They can be connected by stiff springs, which are a form of soft or inexact constraints, to form articulated bodies. The user can define arbitrary forces and torques, which can be time-dependent or can depend on any of the state variables, such as position, rotation, and velocity, of any of the objects in the scene. The system simulates Newtonian dynamics, with collisions but without static contact forces. Extending the system to handle phenomena such as contact forces and deformable bodies would not be hard in theory, since such problems have been the subject of much recent research [115, 14], but would be a great deal of work that is only tangential to the research described here.

Kinematic constraints can also be specified, using time-dependent trajectories. These trajectories can constrain either the position and rotation of an object, or their derivatives.

They can be active for any interval of time. More complex constraints, such as inequality conditions and constraints that involve functions of more than one state variable, can usually be defined using forces, and although they will not be exact, they can be made as accurate as necessary by adjusting the magnitude of the forces.

1.4.2 Editing functionality

The basic user-interface principle in Asgard is that of multiple-representation editing. Systems that use this model of interaction present the user with more than one way to look at the document or presentation being created, and allow editing in any of these views, maintaining synchronization between them as closely as possible. In most such systems, two major types of views are supported: a language-based view, which presents a formal description of the structure and underlying semantics of the objects, and a graphical view, which presents the document in a fashion that is close to its final form. An early multiple-representation editing system was Vortex [30], a predecessor to the Ensemble project described in section 4.4.1. Other multiple-representation systems include Pan [9], Lilac [22], and Juno [82].

Asgard follows this model, and presents two types of views. In the language-based view, one can edit a scene description written in a language that makes explicit all the elements of the animation: shapes, objects, forces, trajectories, and so forth. In the graphical view, one is presented with a view of the animation, which has viewing transformation and time controls to allow flexible playback and previewing. He can edit the scene in either type of view, and all the other views will update themselves as soon as possible so that all remain in synchronization.

Most existing animation systems provide only one view, which is either direct-manipulation and graphical or language-based. A number of these systems are discussed in Chapter 2.

Since motion calculation is a potentially time-consuming process, an animation system must be careful about *when* it recomputes the motion of the objects. Asgard allows the user to indicate when he has finished editing and wants to calculate the updated motion. In addition, it is careful to preserve as much data as possible from the previous run, and incrementally calculate only the data that has changed.

1.5 The organization of this dissertation

This dissertation has four major parts. The first is a discussion of previous work in animation editing and an architectural overview of the Asgard system. Then the two main aspects of this research are described in detail: the user interface and the motion simulation algorithm. Next is a description of the integration of Asgard with the Ensemble multi-media system. Finally, an evaluation of the Asgard project is presented and future work is discussed.

Chapter 2 describes some previous animation systems, and also related work that has been important in the development of Asgard. The systems discussed in this chapter include editing frameworks, graphical animation editors, and batch-oriented systems. A tabular summary of the systems is provided that compares them in terms of their interfaces and capabilities. In addition to the discussion in this chapter, more specific background information is provided in other chapters where appropriate.

Chapter 3 contains a high-level description of the types of problems that Asgard is designed to handle and of the process that an animator would use to define a simple scene. An architectural overview is also provided, which describes the various parts of the system and how they interact. The interesting details of these subsystems are left for the next two chapters.

Chapter 4 discusses the principles of multiple-representation editing and how it is used in Asgard. The two main user interfaces, the graphical editor and the textual editor, are described in detail, and the algorithms used to synchronize them and interface with the low-level motion simulation modules are given. A discussion of some other systems that have used similar editing techniques for other types of media is also provided.

Chapter 5 contains a discussion of the motion simulation algorithms used by Asgard. First, an overview of the physical principles involved is presented, with a description of how they are transformed into low-level systems of equations. Then differential equation solution algorithms are discussed, and the approach used by Asgard to achieve incrementality and handle collision response is described in detail. Finally, the collision detection and response algorithms of Asgard are discussed.

Chapter 6 provides a brief overview of the Ensemble multi-media document editor, and the issues involved in the integration of the Asgard system with Ensemble.

The concluding section of this dissertation, Chapter 7, contains a discussion of future research problems suggested by the Asgard project, and evaluates and summarizes the contributions of this project.

Chapter 2

Background

Computer animation systems can be characterized in a number of ways: they can be interactive and provide graphical editing capabilities, or batch-oriented and utilize a textual description of the scene to be animated; they can compute object motion in real time, near-real time, or much slower than real time; they can allow the specification of motion using kinematics, dynamics, or behavioral descriptions. This chapter will first explain these terms in more detail than in Chapter 1. A number of animation systems will then be described, characterized according to the properties listed above, and compared with Asgard, with regard to capabilities and overall approach to the animation editing problem.

2.1 Types of animation systems

Interactive animation systems are most useful for rapidly creating animations that are approximate and not necessarily physically realistic. Since the animator works with an image that is close to the finished product, he can create an animation in a relatively short time, without a time-consuming trial-and-error iterative process. On the other hand, in order to achieve precise results or model complex phenomena, he must either work with a predefined and limited set of facilities provided by the program, such as discrete grids and buttons that turn gravitational forces on and off, or modify parameters that are attached to all objects of interest one by one, perhaps using property sheets, which can be tedious and error-prone.

Language-based animation systems use textual or program-like descriptions to characterize the system being animated. One might create a file with a list of all the objects of interest, the forces that act on them, the initial conditions, and additional information needed to perform the animation, and then run a program that reads the file and produces a sequence of frames as output. Until a few years ago, such systems were more common than graphical ones, because processing and graphics speeds were not high enough to support the kind of interaction required for graphical editing of complete scenes. This is no longer a problem, but there are still some advantages that language-based systems have over graphical systems. The principal one, from the point of view of this dissertation, is that they allow more complex types of animated phenomena to be specified precisely and efficiently.

The speed of an animation system is largely a function of the types of phenomena that are being modeled, and the size of a particular problem. When only kinematic motion simulation is being performed, and very little real computation is being performed per frame, real time performance is usually easy to achieve for all but the largest models. On the other hand, performance for dynamic or behavioral systems can range from real time to thousands of times slower than real time. For example, some problems involving contact forces are NP-complete [14], and many types of behavioral descriptions require techniques such as optimal control [122], which can be very slow. However, it is often possible to calculate simple dynamics-based motion of relatively small models in real time on a fast workstation.

The different ways of specifying an animation range from low-level detailed motion specification to high-level rules or constraints that implicitly, and in most cases only partially, define the motion. The categories of kinematic, dynamic, and behavioral animation are in common usage in the literature. They focus on the type of information that is provided to the system from which to compute the motion, as opposed to how it is presented to the system by the user.

Kinematic animation requires the animator to explicitly specify the motion of the objects in the scene. This can be done using *keyframes*, which are snapshots of the scene at particular times between which the program must interpolate, or *motion paths*, which are curves in 3-space parameterized by time that describe how each object should move.

In dynamic animation, the animator provides a description of the forces that act on the objects, the constraints that must be satisfied, and the other laws of physics that apply, and the system calculates the motion from this information. This calculation is usually done by means of numerical integration of the final force expressions.

Behavioral animation essentially includes everything that is higher-level than dynamics. Behavioral systems may accept input in the form of rules, constraints, or goals. Mechanisms for creating motion that satisfies these conditions include optimization, neural networks, genetic algorithms, and a variety of other techniques.

Zeltzer [125] has defined three levels of animation description which focus more on the way this information is supplied: *guiding* level, where the motion is described explicitly, *program* level, where it is described algorithmically, and *task* level, where implicit and declarative descriptions in the form of goals and constraints are given. These levels are, to some extent, independent of the classifications of kinematics, dynamics, and behavior. For example, a kinematic description might be at the guiding level, in the case of a motion path, the program level, in the case of a symbolic expression that gives the position as a function of time, or the task level, in the case of *morphing*, where the animator specifies two figures and the goal that one should change into the other, and the actual process is done by interpolating the key features.

The axes used here for characterizing animation systems are (1) user interface, (2) speed, (3) input type, and (4) interaction level. These are conceptually orthogonal – there exist both graphical and language-based systems that fall into each of the kinematic, dynamic, and behavioral categories. Furthermore, most systems are hybrids at the input specification level: dynamics systems usually allow some elements to be specified kinematically, and many behavioral systems are built on top of a dynamics substrate.

The rest of this chapter will describe a number of animation systems, and characterize them according to the criteria outlined above. The types of objects and physical phenomena they support will also be indicated, since this is a major factor that influences the speed and complexity of the system, and whether they support incremental motion update. This information will be summarized in a table at the end of the chapter.

In terms of these categories, Asgard is both a graphical and language-based editing system, it is near real-time for many types of scenes, it provides both kinematic and dynamic control, and can be viewed both as a guiding-level and as a program-level system.

2.2 Animation editors

Animation editors allow an animator to interactively define how objects should move, to request that the motion be calculated, and to view the resulting scene. He can then modify initial conditions, constraints, or other parameters of the scene, and recalculate the motion. The editor may support off-line high-quality rendering, incremental update of motion, or simultaneous viewing of the scene from multiple positions in time and space.

One aspect of animation editing is the problem of 3D object design. This is a large area that is mostly independent of animation editing, so nothing more will be said about it, other than that animation editors provide solutions that range from the minimal facilities provided by Asgard, which are described in Section 4.1.1, to very sophisticated CAD functionality such as that found in commercial systems such as Alias Animator.

2.2.1 ThingLab

ThingLab [20] is a constraint-based simulation system. It is embedded in SmallTalk, and takes advantage of the graphics and interpretive language capabilities provided by the language. The user can define objects and constraints in the language and view them on the screen, and can also manipulate them graphically. It is thus an example of a multiple-representation editor.

It solves systems of constraints using local propagation, which means that it starts with the variables that have been modified and successively solves the constraints that are connected to them and modifies other variables, stopping when no more constraints are unsatisfied. This type of algorithm is by nature incremental. Many animation problems can be formulated in terms of such constraints, but general motion simulation requires iterative solution mechanisms such as relaxation, which are somewhat harder to provide in a constraint-based framework.

Further work on ThingLab [21, 47] has extended the constraint solution mechanism and made it more efficient. Many other constraint-based systems have been developed, for example Sketchpad [112], Ideal [118], Bertrand [73], TBAG [41], and the constraint based version of Ensemble [31]. Work on constraints has also been done by Steele [109] and Gosling [54].

The constraints used for these systems are solved in different ways than constraints used in 3D dynamics systems, some of which are described below. The former are essentially symbolic, whereas the latter are numerical and must be solved using numerical methods.

2.2.2 Twixt

A simple system for animation editing was Twixt [53]. It used a combination of keyframes and motion paths to give the animator very flexible control over the motion of the objects in question. Any parameter of any object, including control points of deformable objects, could be attached to a linear or higher-order path. The initial specification of the scene was done using a simple language, but additional editing could be performed graphically, or a parameter could be attached to an input device. Twixt is a fairly light-weight system, both by design and because of the limits of the available hardware when it was written.

2.2.3 Alias Animator

Alias Animator [1] is a widely used system for creating highly realistic graphics for movies and other types of films, and is also used for industrial design. It provides an interface for the animator to define parametric motion paths in 3D, and also allows the camera to be treated as an animatable object. The control points of an object can be attached to a motion path, which allows some types of time-dependent deformations. No facilities are provided for physically-based animation, although it is possible to interface Alias with external motion calculation programs, which could provide motion paths for objects to follow.

Alias is a powerful system with a rather arcane interface, and it is designed for large projects where a great deal of animator time and computation can be applied to a project. It is not especially good for the casual user, who wishes to do rapid prototyping of animated scenes with realistic motion.

Similar animation systems are Wavefront [64] and Macromind 3D [76]. What these programs have in common is that they are basically 3D object and scene editors that make it possible for the user to attach motion paths to objects, and to interface to external motion calculation routines, with varying degrees of convenience.

2.2.4 Interactive Physics and Working Model

A recent product available for the MacIntosh is Interactive Physics II [104]. It is designed for classroom settings, and allows the user to set up complex physical configurations and simulate their motion. A teacher can create a model and let the students experiment with what happens under varying conditions, and construct machines quickly. Unlike Asgard, it is limited to two dimensions, but it does simulate a wider variety of phenomena, including contact forces. A similar program is Working Model [77], which can import shape descriptions from, and export motion descriptions to, a number of popular graphics systems. These systems are graphical and interactive, but it is unclear to what extent they can take advantage of possibilities for incremental recomputation, such as when a user modifies some parameters of objects which do not affect other parts of the system.

2.2.5 IGRIP

IGRIP [63] is a robotics simulation and programming system that is widely used for industrial applications. It can perform both kinematic and dynamic simulation, and detects collisions and near misses. The goals of motion simulation are somewhat different for robotics than for animation, since the results are used to determine how the robots will move, under the control of programs that can be generated from the simulation package.

IGRIP performs a number of functions that are often included in animation systems, such as inverse kinematics, where the user specifies a desired position and the system calculates how to move the various joints and other degrees of freedom in the robot to achieve this position, and inverse dynamics, where the user specifies either a position or a velocity and the system calculates what torques are required. Inverse dynamics is probably more useful for animation, since robots generally have closed-loop kinematics control which generates the required forces and torques automatically, while an animator may wish to model such effects as bulging muscles that depend on the force values.

2.3 Languages

In this section, a number of languages are described that have been used for describing animated scenes. The systems that use these languages are all batch-oriented. The systems included are those where the entire description can reasonably be written in the language by the user, as opposed to graphical systems that have textual components, such as blocks of code in a scripting language that can be attached to existing objects on the screen.

Most of the existing animation languages, including those described in this section, are fairly old, and some were grafted onto existing 3D picture description languages, such as Rayshade [69], Unigrafix[83, 103], and Mira [78, 79]. Little work has been done on animation languages in recent years, with the exception of constraint systems such as TBAG [41]. The reason for this seems to be that most graphical editing systems do not integrate well with language-based editing, and given a choice between one or the other, graphical systems are simpler and easier to use.

The prevalence of animation languages is actually much greater than the literature would indicate, however, because in research on animation where the focus is on the motion simulation algorithm, in most cases an ad-hoc language is constructed to describe the system and the animation is performed in a batch fashion [15]. The reasons for this are easy to see – it is a lot easier to create a language-based front end for an animation system than an interactive graphical one, and there are currently no widely available and generally-accepted graphical editing frameworks for physically-based animation. Also, the algorithms being developed are generally so slow that scene description is not a bottleneck in the edit-simulate-view cycle, and incremental update may be hard to implement for a given algorithm.

2.3.1 Script

The SCRIPT language [40] is an early language-based system for creating animations. No physical realism is provided, and the language itself is fairly low-level and depends on absolute frame numbers. However, it is possible to animate, or make time-dependent, any quantity used by the system, including both state variables such as position and appearance attributes such as color. The language is partly procedural and partly declarative, and although it is not a full programming language, it has proved very useful for commercial

```
define start 1.0                    : Define variables to denote the
define end 300.0                    : beginning and ending frames.
begin                               : Marks the start of the frame-drawing code.
        newframe                    : Initialize a new frame.
        push                        : Push and pop are transformation matrix ops.
                move z = 10.0       : Translate the camera position.
                view Δ start end    : Define the range of keyframes.
        pop
        push
                rot y               : Rotate about the Y axes, with the limits
                i start 0.0         : defined by the start and end keywords.
                i end 360.0
                drawbox             : Draw a box subject to the current transformation.
        pop
        render                      : Output the frame.
end                                 : Marks the start of the frame-drawing code.
```

Figure 2.1: A sample of the Script language

applications such as creating "flying logos". Script also provides a message-passing facility so that the user can include external functions (in FORTRAN) that compute variable values. An example of the Script language is shown in Figure 2.1. This code draws a rotating cube.

2.3.2 ASAS

Reynolds [98] developed a system called ASAS, which is based on Lisp. One writes programs in this language to construct pictures, and can create animations using *animated variables*, similar to the articulated variables of MENV, which is described below. Objects can be defined procedurally, which makes it easy to create fractal structures and deal with things that break apart or exist for short periods of time. The system also includes the notion of actors, as defined by Hewitt [59, 60], which makes it possible to define very high-level behaviors for objects. On the other hand, there is no support for dynamics built into the system, and no graphical editing capabilities.

A sample of an ASAS description is shown in Figure 2.2. Like the Script example, this also creates a picture of a spinning cube.

The ASAS system is a good example of what Zeltzer calls "program level". Reynolds has also done work on behavioral modeling [99] at the task level.

2.3.3 Gramps

Gramps [84] is an early kinematic system that uses a simple hierarchical picture description language with transformations that can be either time dependent, using keyframes and interpolation, or linked to input devices such as dials. For example, the command

```
(script spinning-cubes
        (local: (runtime 96)
                (midpoint (half runtime)))
        (animate (cue (at 0)
                        (start (spin-cube-actor green)))
                   (cue (at midpoint)
                        (start (spin-cube-actor blue)))
                   (cue (at runtime)
                        (cut))))

(defop spin-cube-actor
        (param: color)
        (actor (local: (angle 0)
                       (d-angle (quo 3 runtime))
                       (my-cube (recolor color cube)))
        (see (rotate angle y-axis my-cube))
        (define angle (plus angle d-angle))))
```

Figure 2.2: A sample of the ASAS language

`TRANSLATE SPHERE X [-100<D4<100]` causes the X coordinate of an object called `SPHERE` to be linked to dial number 4, with upper and lower limits of 100 and -100.

Gramps was used for visualizing data such as molecular models and human figures. Its most interesting feature was that the user not only described how to draw an object and how it changed as a function of time using the language, but also specified how the user could interact with the system, using a simple and and elegant mechanism. This was a precursor to constraint-based interactive systems such as TBAG, described below, and to the Twixt system, described above, which also provided facilities for connecting object parameters to input devices.

2.4 Frameworks

A number of systems have been developed that are primarily frameworks for integrating independent animation components and supporting distributed interaction. These systems define common interfaces for modules which can simulate different object types, physical phenomena, and viewing and editing interfaces. Many of the issues involved in these systems are similar to those for other types of integration frameworks, such as software and document development environments [114, 33, 50].

2.4.1 The Brown animation framework

Zeleznik et al. [124] describe a framework for integrating different types of simulation and physical models in an object-oriented way. Each element of the scene is represented by an object, which communicates with other objects using messages. Caching is used extensively to maintain reasonable performance by avoiding the recomputation of expensive data. This system allows many different types of animation algorithms to be integrated in a convenient and flexible way, although it is not clear what types of interactions are hindered by the relatively loosely-coupled architecture. The same group has done related work on 3D interactive toolkits [35].

2.4.2 Menv

The Menv system [96], which has been used internally at Pixar for the production of a number of high-quality films, is a based on a modeling language called ML. This language allows objects to be specified hierarchically and procedurally, and relies on a mechanism called *articulated variables* to achieve animation. These variables have time-varying values, which can be controlled by external modules, which are linked to the main database using UNIX IPC mechanisms. Graphical editing of the animated scene is done primarily by altering the waveforms that control the articulated variables, which can be specified in a variety of ways, including splines and symbolic expressions. Menv does not directly deal with dynamics, although external modules that handle forces and constraints can be added to the system. One of its more interesting aspects is the way different program components are connected – the core of the system is a database framework that coordinates the actions of the different modules, which makes the system extensible and flexible, at the expense of certain functionality that requires tighter integration.

2.4.3 TBAG

Another constraint-based animation system is TBAG [41]. This is a toolkit that uses constraints, similar to those in ThingLab, to implement 3D objects that can be composed and manipulated much like widgets in existing 2D toolkits. It can perform dynamical motion calculation and numerical integration, like Asgard, but it is somewhat lower-level and does not provide collision detection or automatic generation of motion equations from object descriptions. It also provides both language-based and graphical editing, as does Asgard, but there are no facilities for updating the textual representation as the graphical representation is changed.

2.4.4 Bolio

Zeltzer et al. [126] describe an "integrated graphical simulation platform" called Bolio. This system allows the animator to mix the three levels of specification, guiding, program, and task. It also includes a constraint system, similar to those of ThingLab [20] and SketchPad [112]. Facilities are provided for the integration of external tools into the system, such as an interface to the DataGlove input device, and this interface can be used to support various interactive editors.

2.5 Physics simulators

There has been a great deal of work in the past decade on physically-realistic animation. Many researchers have developed computational techniques to simulate the motion of a wide variety of types of objects, such as articulated and deformable bodies, and physical effects, such as collisions and friction. Although Asgard supports only a minimal set of object types and effects, the editing interface it incorporates could be used for many other physically-based editing systems. This section will describe some of the more important recent works on physically-based animation, with an emphasis on research that has influenced or is relevant to Asgard.

2.5.1 ThingWorld

Pentland [90, 89] describes a number of techniques for achieving rapid dynamics calculations, which are part of a system called ThingWorld. The major contribution of this work is the representation of deformable objects by means of vibration modes – by using a small number of extra variables one can simulate deformations and oscillations of reasonably rigid objects with good accuracy. Also, collision detection is done using implicit surfaces, which are functions that are positive inside the object and negative outside. This makes it easy to perform collision detection when deformations have been applied to the objects. Two types of constraints are provided: soft constraints, implemented with stiff springs similar to those used by Asgard, and hard constraints, which must be holonomic – that is, expressible the form $f(\mathbf{u}) = 0$ where $\mathbf{u}$ is the state vector for the system and f is a scalar function.

2.5.2 Space-time constraints

Witkin and Kass [122] have built an animation system that uses a technique called space-time constraints to control dynamics-based animation. The problem they were trying to solve is how to mix dynamics with kinematic constraints, which may be more complicated than simple motion paths. The system may contain a number of free or partially constrained forces, which should be adjusted so that the motion satisfies the kinematic constraints while minimizing the work performed. This is a common problem in optimal control theory [32]. An earlier version of Asgard included code for solving this sort of problem using gradient descent methods, but this approach proved to be too slow and unstable for use with animation, mainly because of the large number of variables involved. The time complexity is generally at least $O(n^3)$, where n is the number of variables, but if a poor initial guess is specified the computation may not converge at all.

The approach used by Witkin and Kass was somewhat different: they discretized the time over which the motion was to happen and used quadratic programming, a discrete method, to perform the optimization. The results were striking – it seems that this sort of goal-based optimization is very useful for producing "character animation", with exaggerated gestures and the appearance of intelligence. This technique seemed rather robust, but was very slow, and it was unclear whether it would generalize to other types of problems.

Cohen [34] extended the idea of space-time constraints to include the concept of "windows", which allow the system to perform the optimization step in stages. This reduces the time complexity of the problem, while giving up the ability to perform true global optimization.

2.5.3 Virya

Viryas [121, 120, 119] is a system for animating human figures and other articulated bodies. In order to animate such figures, one must implement joint constraints that ensure that the body components remain connected, and reduce the number of degrees of freedom of the system. For example, two hinged segments have 7 degrees of freedom, rather than the 12 that a pair of unconnected segments would have.

Virya uses a technique called the Gibbs-Appel formulation that reduces the number of degrees of freedom in the system by rewriting the equations of motion. Once this is done, there is a smaller number of unconstrained equations to solve, which makes the solution process easier, but since these equations are more complex, the speed of the simulation is less.

Other researchers, such as Armstrong [4, 2, 3] and Badler [7, 6] have also addressed this problem. Armstrong uses low-level "motion processes" to model per-joint torques, and incorporates higher-level guiding primitives to perform more complex motion planning. Badler has focussed on the problem of human motion, and includes a significant amount of anthropometric data and functionality.

2.5.4 VES

The Virtual Erector Set [101] is a system for simulating systems of articulated bodies. Many other systems have modeled articulated bodies with soft constraints, such as Asgard does, or by means of exact analytic methods, such as the Virya [119]. The VES system uses a recursive technique that has aspects of both methods. It treats the object as a tree, with cyclic links represented as soft constraints. This leads to a very efficient simulation mechanism, that can handle small articulated models in close to real time.

2.5.5 Jack

The Jack system [92, 91] is an example of goal-directed interactive animation. It tries to minimize the value of a penalty function in an iterative way, and updates the current state of the simulated system as new values become available. The constraints on the objects generally lead to an underdetermined system, where there is generally more than one way to satisfy all the constraints and minimize the penalty function, but since Jack is used for interactive positioning, the use can be relied upon to continue modifying the configuration until it is satisfactory. Projects with similar approaches are Scuplt[111] which is used for interactively modeling and large molecules, and Lee's work in strength-guided motion [71].

2.5.6 Dynamo

Isaacs and Cohen [66, 65] describe a system for creating mixed kinematic and dynamic simulations. Their system, called Dynamo, can also handle behavioral motion specifications. The problem specification is similar to that of Space-time constraints, as described above, but the solution mechanism is quite different. Dynamo allows the user to impose *complex kinematic constraints* on a problem, which are general functions of all the generalized coordinates, including joint angles, and of time. These constraints may be conditional, which allows them to be active only when a collision or contact takes place. The system is then solved using Lagrange multipliers at each time step, with behavioral control applied before each iteration.

The algorithms used in Dynamo are very general and powerful, and a similar approach was tried in a previous version of Asgard. However, the solution process can be rather time-consuming: an example involving a marionette requires one hour for each second of simulated time.

2.5.7 Other research

Many other researchers have done important work in physically-based animation, much of which has been more oriented towards solving particular problems than building integrated general-purpose systems. Baraff [11, 13, 12, 16] has done a great deal of work on simulating friction and contact forces between different types of objects, including rigid and deformable bodies. Terzopoulos [116, 115] has described techniques for handling deformable objects of various types, including elastic, viscous, or plastic objects. Badler [7, 6] has developed a system called Tempus for creating and simulating human motion. Calvert [25, 26] has also done work in simulating human motion, with a focus on computer-aided choreography tools. Mirage [113] and NPSNET [128] are animation frameworks that address problems of distributed simulation and real-time interaction between agents.

2.6 Summary

This chapter has described some of the basic criteria that one might use to classify animation systems, and has listed some significant animation editing systems that have appeared in the past decade. Table 2.6 summarizes this data, comparing the various systems with regard to the types of animation they can perform and what sort of user interface they provide.

The "Interface" column indicates whether graphical or language-based editing is provided. A few systems provide both, but with the exception of Asgard, none allow graphical and language-based editing to be done simultaneously – most allow the initial specification to be performed textually and then allow graphical modification of the scene.

The "Speed" column indicates whether the system performs motion simulation, as opposed to playback, in real time, batch mode, or near real time, which means "good enough for interactive editing". Of course, this usually depends on the complexity of the scene being edited and the types of physical phenomena being simulated, but what is most important

System name	Interface	Speed	Simulation type	Level	Increm
Twixt	both	real	kinematic	guiding	no
Alias[1]	graphical	real	kinematic	guiding	–
ThingLab	both	real	kinematic / constr	program	yes
Interactive Physics	graphical	near-real	dynamic	task	yes
IGRIP	both	varies	dynamic / kin.	task	no
Script	language	real	kinematic	program	no
ASAS	language	real	kinematic	task	no
Gramps	language	real	kinematic	program	no
Zeleznik et al.	both	real	mixed[2]	task	varies[3]
MENV	both	batch	kinematic[3]	program	no
TBAG	both	real	kinematic / constr	task	yes
Bolio	both	varies[4]	mixed[5]	mixed	no
ThingWorld	language	near-real	dynamic	task	no
Spacetime constr.	language	batch	dynamic / kin.	task	no
Virya	language	batch	dynamic	task	no
VES	graphical	real	dynamic	task	no
Jack	graphical	real	dynamic	task	yes
Dynamo	unknown	batch	mixed[5]	task	no
Asgard	both	near-real	dynamic / kin.	task	yes

1. This also applies to Wavefront and other similar programs.
2. Since the system is a framework for integration animation applications, many types of motion simulation algorithms can be applied to a problem.
3. External or user-supplied modules can be interfaced to the system, which provide additional functionality.
4. Some types of interaction are real-time, using the DataGlove for input.
5. Handles kinematic, dynamic, and behavioral specifications.

Table 2.1: Summary of animation systems

from our perspective is the design goals of the author, since they strongly influence the user interface.

The "Simulation type" category describes what kind of input the system receives from the user; in a kinematic system, explicit motion paths or keyframes are given, for dynamics a description of forces and other constraints is required, and for a behavioral system, one provides some type of higher-level task specification.

The "Level" column indicates which of "guiding", "program", or "task" the system would be considered using Zeltzer's taxonomy. Some of the assignments are a bit arbitrary, since most of these systems could be described at multiple levels. All dynamics systems are considered task-level, since force descriptions seem best described as implicit motion specifications.

Finally, the last column indicates whether the system is designed to handle incremental re-simulation of motion. For many systems, this is not a relevant question, since motion simulation and playback are identical operations.

These systems seem to fall into a few major categories. First, there are primarily kinematics-based graphical editors that are built on top of a general-purpose CAD system, such as Alias. These generally have facilities for interfacing with external motion calculation routines, but the primary focus is not dynamics. Second, there are scripting languages such as ASAS that are also primarily kinematic, and that feature some interactive control, but not full-featured graphical editing. Third, there research systems such as Virya that focus more on the algorithms and simulation techniques than the editing interface. Finally, there are heavy-weight frameworks, such as MENV, that coordinate the actions of a number of tools, but do not directly implement physically-based motion simulation.

Asgard embodies aspects of all of these categories. It is possible to design shapes and attach motion trajectories to them, to edit a textual representation of the animated scene, to simulate physically-realistic motion, and to provide different views on a scene simultaneously. It takes a somewhat different approach from much of the previous research, however, in considering *editing* to be the primary activity of the animator, and organizing the other functionality of the system in such a way as to enhance the editing capabilities of the animator.

Chapter 3

Overview of the Asgard System

This chapter provides an overview of the goals and architecture of the Asgard system. First, a description of the goals of this research will be given, in greater detail than in Chapter 1. Then, in order to provide a concrete example of an animated scene that can be created in Asgard, a very simple model will be presented, along with the steps that one would use to create it. Next, an overview will be given of the kinds of physical phenomena that can be simulated by the system, and finally a brief outline of the architecture of Asgard will be provided.

3.1 Design goals

The purpose of Asgard is to enable an animator to create realistic animations rapidly and easily. To create a system that fulfills this purpose, one must first identify the tasks that are involved in creating and editing such animations; second, determine what tools are best suited for those tasks; and third, design an overall system that incorporates these tools in a systematic and natural way. This system must not only present a good user interface, but it must also be fast enough to be usable for interactive editing.

The tasks involved in creating realistic animations include the following:

- Defining and editing shapes, which are used in later object creation operations. This task could also be considered a part of object creation, but Asgard treats it as a separate step.

- Creating instances of objects and determining their initial conditions. These conditions include values for position, orientation, velocity, and angular velocity at the starting time ($t = 0$).

- Specifying forces and torques that act on bodies. These might be described in a variety of ways. The forces comprise the *dynamics* of the simulated system.

- Specifying path constraints, both exact or inexact. These constraints may be links between objects, or paths that objects must follow over an interval of time. Other types of constraints have been implemented by some systems, such as expressions that involve state variables of several objects which describe conditions that must be met at each point in time, but Asgard requires that such constraints be described as forces, if possible.

- Viewing the resulting animation from various points of view, and saving it for later playback or exporting it to off-line higher quality renderers.

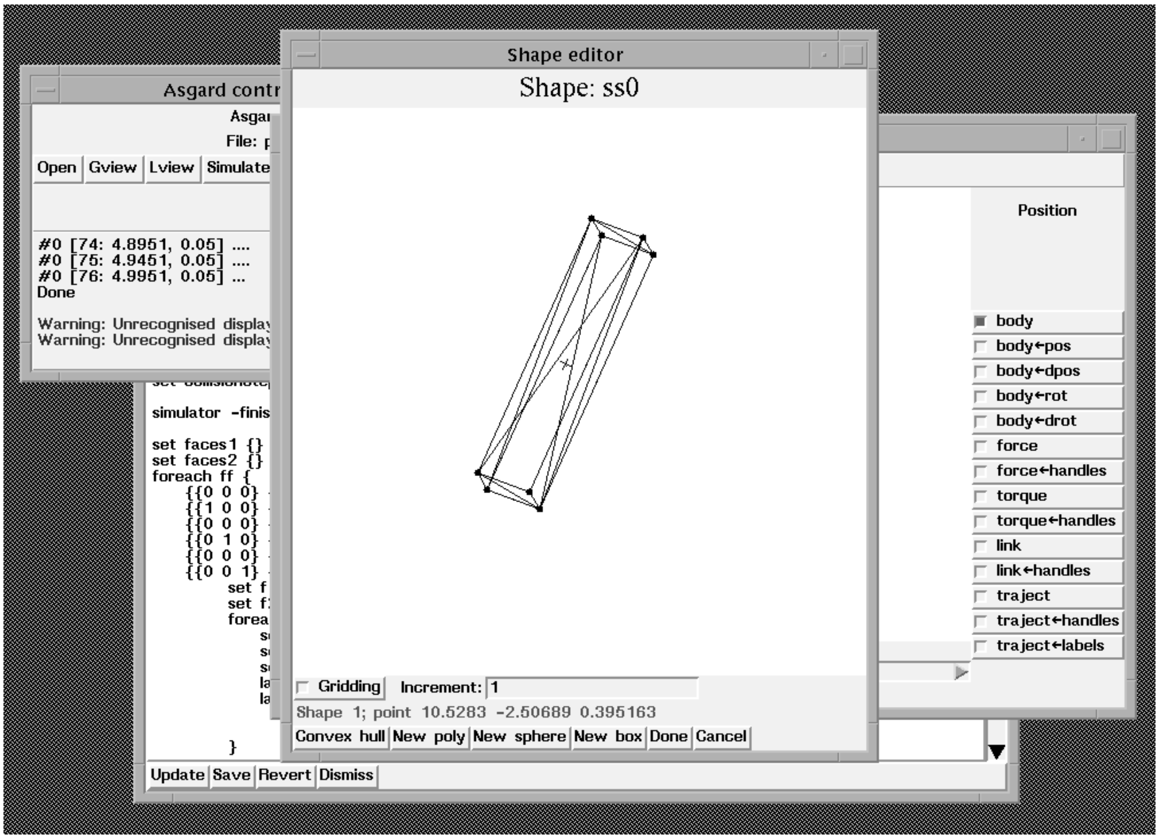
Two major considerations when providing facilities for accomplishing these goals are first, the suitability of the interface for the task at hand, and second, the support for interactive editing. Interface components must be handled on a case-by-case basis, but one can try to provide some generalized facilities, such as the multiple-representation paradigm that Asgard uses, that are appropriate for a broad range of applications. Support for interactive editing requires attention to the algorithms used for motion simulation, or more generally, presentation maintenance, at all levels, from the high-level language analysis stage to the lower-level differential equation solution techniques used. These algorithms should be both as fast as possible and incremental enough to respond quickly to editing changes in near-real time.

3.2 Sample problem – a two-part pendulum with a collision

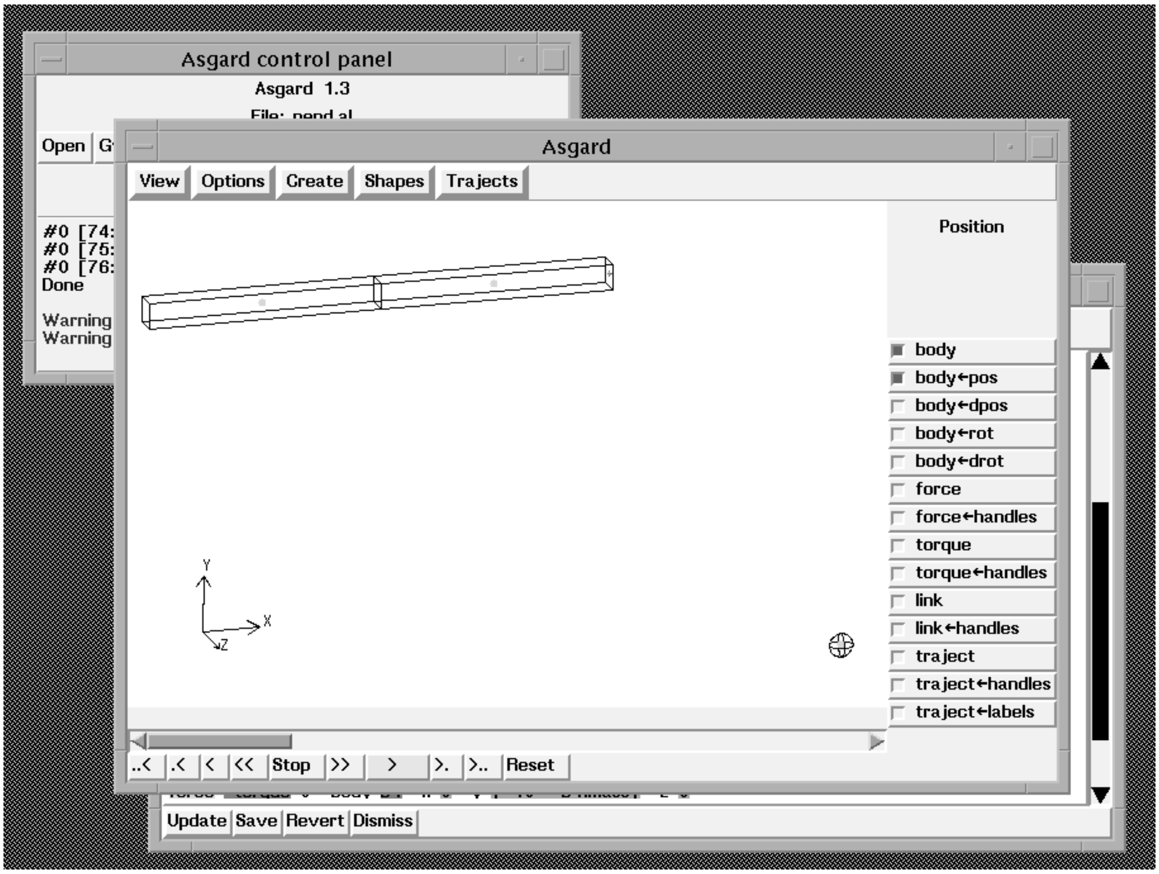
Asgard is quite flexible and can be used to create a variety of animated scenes. Let us consider an example that is simple, but illustrates most of the interesting features of the system. A full diagram of this example is shown in Figure 2 in Chapter 5. It consists of a pendulum that has two parts, which are connected by a link or hinge. It is also connected to a fixed point in space at the top. It swings down, under the influence of gravity, and collides with a sphere that is moving across the bottom of the scene, unaffected by any forces. This sphere then flies off to the right.

This simple example will be used in a number of places throughout this chapter and the next ones. It illustrates most of the physical simulation capabilities of Asgard, including forces, links, and collision detection and response. A later discussion will modify this example to show the use of trajectories.

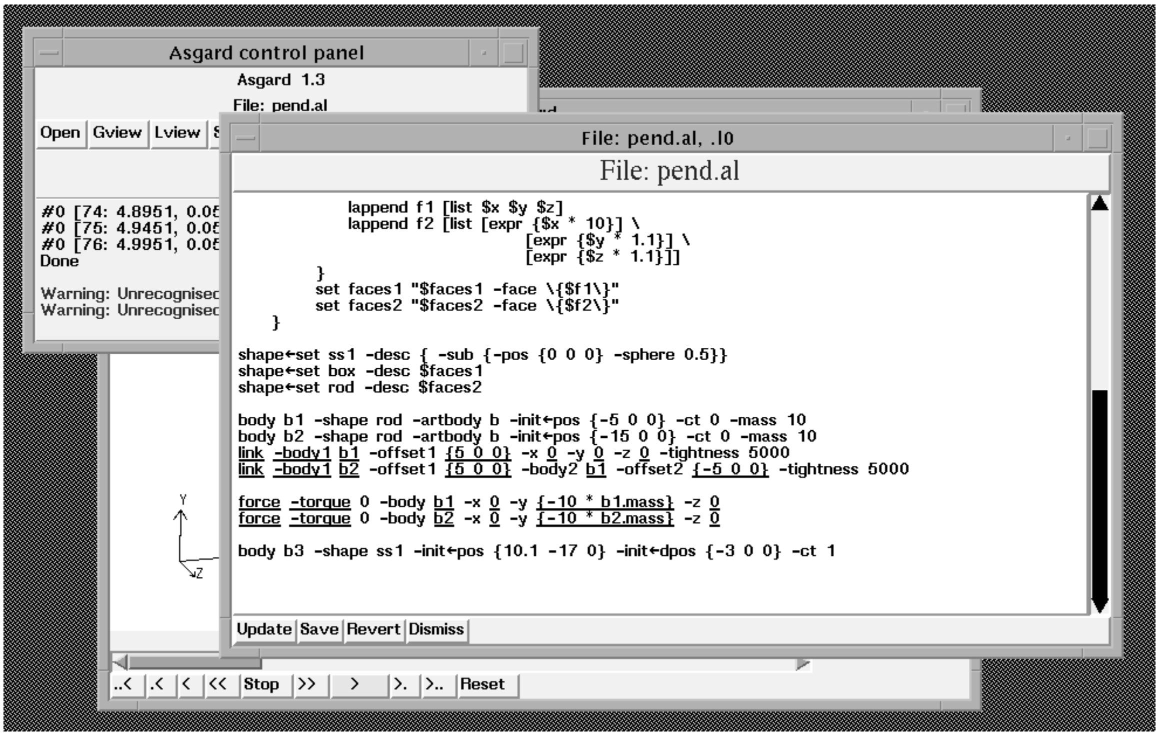
A user would typically construct a scene such as this using the following steps, which are illustrated in the accompanying figures. The description of what each figure shows appears below the figure.



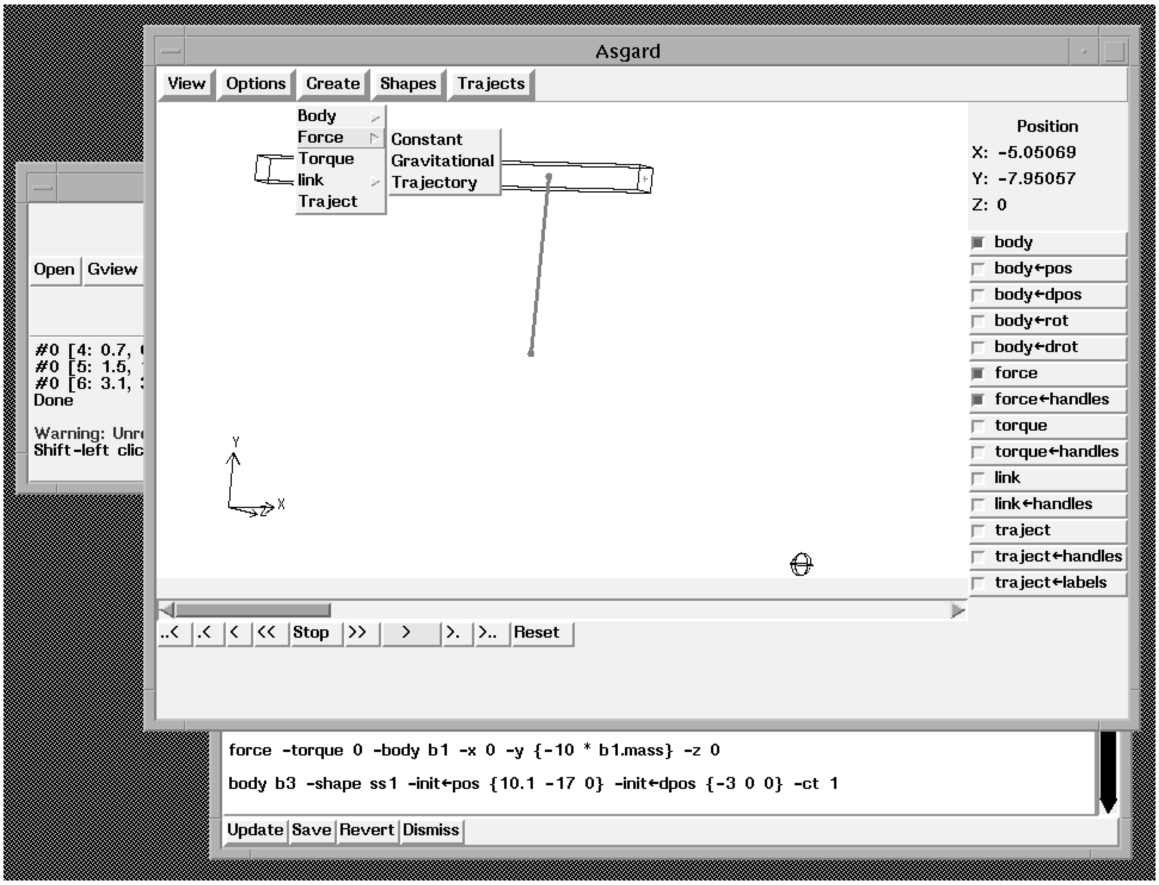
1. Create the basic shapes that are to be used in the scene using a *shape editor*, which allows the user to create shapes as unions of convex polyhedra and spheres. Alternately, the user could import shapes from a library. In the figure, the user is creating a rod from scratch.



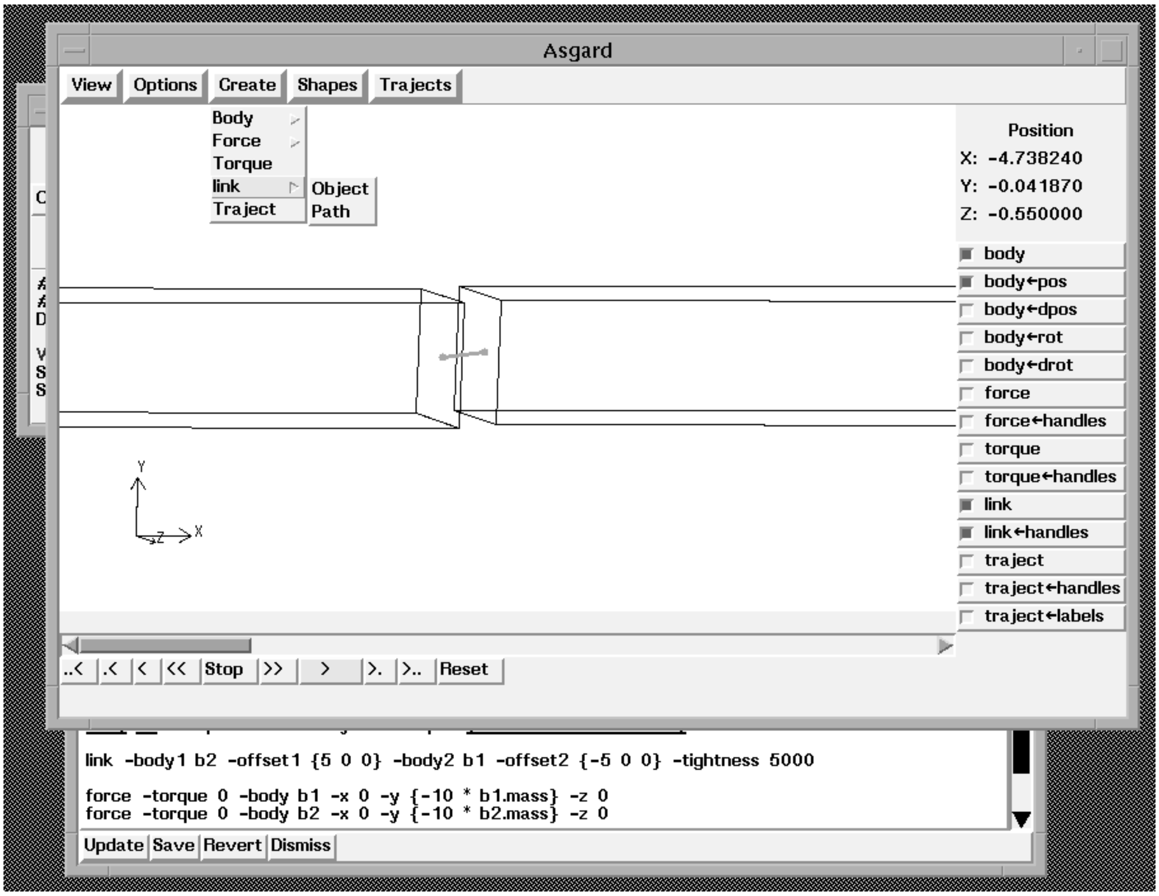
2. Define the objects in the scene. To create an object, one must specify at least two things: the shape and the initial position. The shape is chosen from a menu of the shapes that are currently defined, and the position is specified by clicking on a location on the screen. Other initial conditions and physical properties such as mass can also be given at this time. In this example, two rods and one cube are created.



3. Add additional object information using the language viewer. An example of this is the `-artbody` option, which stipulates that the objects should be considered as components of the named articulated body. This information allows the collision detection routine to ignore interpenetration between the components in the area of the joint, as illustrated in Figure 5.4. Other information that can be added in the language viewer includes the masses and moments of inertia of the objects.



4. Define the forces that act on the objects, such as gravity. This can be done in a variety of ways, but in the example we select the **Gravitational** option from the **Force** submenu. This type of force has a magnitude that is proportional to the mass of the object, which is specified directly by the user. We then click on the center of mass of each object, and on a point approximately 10 units in the negative Y direction from the center. All editing done in the graphical window is approximate – if the user wants the magnitude to be the precise value of the Earth's gravity, he can use the language viewer to modify it textually.



5. Define the links between objects, which are stiff springs connecting specific points on the shapes. There are two links in this scene – one between the two rods and one between the top rod and the origin. This is done in a similar manner to the definition of forces.

6. Specify the time range, by adding a `simulator` command using the language viewer, and ask for the motion to be calculated by clicking on **Simulate** in the control panel.

7. The user can now view the animation, using the viewpoint and time controls in the graphical viewer. The input can also be modified and the motion simulation redone.

The details of how these operations are performed and the reasoning behind their designs are given in Chapter 4.

3.3 Problem domain

A wide variety of systems have been developed which can simulate many different sorts of physical phenomena. A number of these are described in Chapter 2. In designing Asgard, I have tried to select a set of editing tools and user interface features that can be usefully applied to a variety of simulation frameworks and problem domains. The types of objects

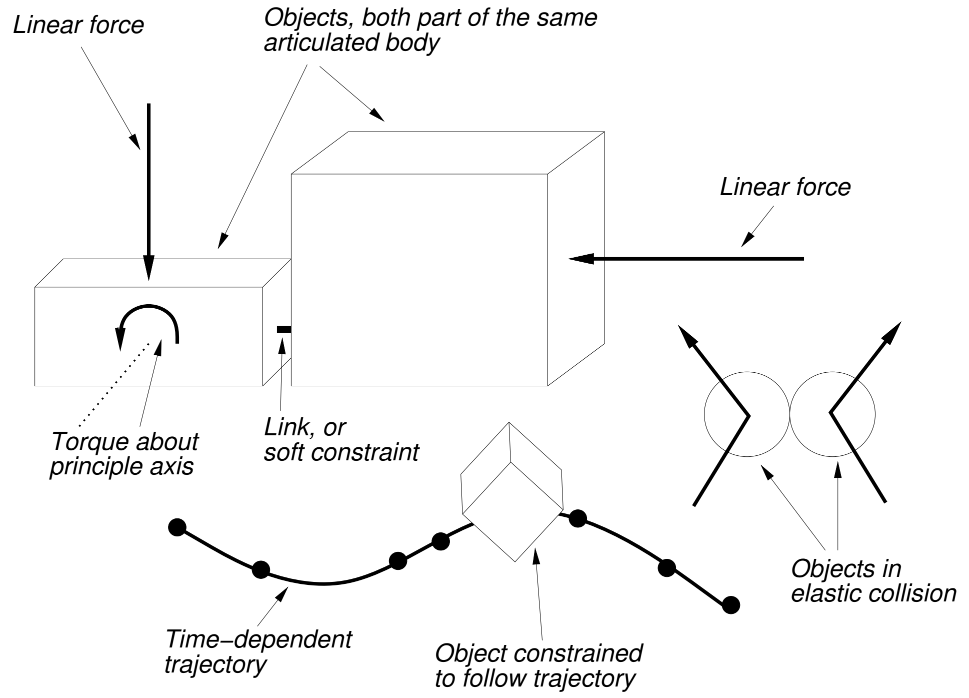


Figure 3.1: The Types of Elements Handled by Asgard

available and the set of physical phenomena that can be simulated have been limited somewhat, in order to make the implementation of Asgard tractable while still providing enough capabilities to justify the utility of these features and interfaces.

The types of elements that can be handled by Asgard are described below, and are represented in Figure 3.1. These include objects, forces, links, and trajectories.

3.3.1 Objects

In the following discussion, objects are assumed to be limited to rigid polyhedral and spherical bodies. Non-rigid and breakable objects can have an arbitrary number of degrees of freedom, or parameters that are required to describe their state, which can result in much higher complexity than that of rigid bodies. Pentland [89] has used modes of vibration to represent deformations very economically, in the same way that Fourier transforms can be used to represent mathematical functions, but in the general case, deformable objects require a great deal of state information. The same is true for simulation techniques such as finite-element methods [127] that represent objects as systems of springs and masses. Rigid bodies require only one set of positional and rotational state variables, which yields twelve scalar parameters when derivatives are included. This simplification was made to make the implementation of Asgard tractable, but the most of the algorithms discussed in this thesis apply to the more complex object representations mentioned above.

Spheres and polyhedrons are also simple to represent and to use in geometrical calculations. There are a wide variety of alternatives available for modeling solids, such as spline-based surfaces and volumes [45, 67] and implicit functions [90]. Editing techniques for polyhedra are more easily implemented, however, and most existing collision detection algorithms work only for convex polyhedra. Collision detection is discussed further in Section 5.3.

In Asgard, articulated bodies are considered to be collection of rigid bodies with hinges or links between them. A number of techniques have been developed by researchers in animation and robotics for enforcing exact constraints between the articulation points of different components of an object [119, 2]. Most of these approaches require the connection graph of the object components to have a tree structure, and they can have the effect of increasing runtime substantially. The advantages include exact solutions and the correspondence of the number of variables in the system with the number of degrees of freedom of the object.

A simpler way to implement articulated bodies is by using "soft" constraints, which can be represented using springs. This is the technique used by Asgard. Such constraints translate directly into forces and torques, and thus require less support from the lower-level motion simulation code. Since they are inexact, they may be noticeably violated at certain points in the animation, but this can be ameliorated by adjusting the spring constants appropriately. It is not clear whether they lead to a more or less efficient solution process than the exact constraint methods mentioned above – the system of equations is larger with soft constraints, but the individual equations are simpler. Also, stiff springs can limit the size of time steps that can be taken in numerical integration, but this has not proven to be a problem in practice. Asgard uses soft constraints because of their simplicity and their suitability for integration with Asgard's overall model of physics.

3.3.2 Principles of motion

Even with the restriction to rigid polyhedral and spherical bodies, there is a wide variety of physical effects that we might want to simulate. These include such phenomena as friction and static contact forces [14]. All of these effects can be represented as forces, but the calculation of the values of these forces can be quite difficult, and in some cases is an NP-complete problem [16]. Asgard limits the types of forces that can be handled to those whose values can be expressed as algebraic functions of time and the state variables of the objects in the scene, and their derivatives. Such forces include gravitational, spring, and sliding friction forces, but not contact forces or other inequality constraints. The advantage of this limitation is a relatively simple simulation framework, since the problem reduces to a system of simultaneous ordinary differential equations. The one exception to this rule is the handling of collision detection and response, which requires transient forces to be applied at a point in time that must be determined independently of the differential equation formulation. An exception is made here because collisions are quite important for realistic animation, and can be handled in a simpler way than the other cases. The disadvantage is that the types of motion that can be simulated is somewhat limited.

3.3.3 Control mechanisms

In addition to simulating physical systems, an animator wants to be able to control certain objects in ways that are independent of the laws of physics. For example, one might want a human or robot to move in a particular way, or a rocket to follow a certain trajectory, determined by its thrust. This sort of control is referred to as *kinematic*, as opposed to *dynamic*, which deals with the forces that act on the system. Asgard makes it possible for the animator to specify simple trajectories for objects to follow for particular time intervals. These trajectories are given as unions of Bézier curves defined over disjoint intervals of time, which can be used for any of the state variables of one or more objects. Other researchers have developed more complex types of constraints, such as space-time constraints [122] or optimization-based methods [92], but these require a great deal more mechanism. Using kinematic constraints combined with fast motion simulation algorithms, the animator can develop scenes by an interative trial-and-error procedure. This type of approach is likely to be more efficient than a more analytical top-down method, where the animator specifies detailed constraints and other declarative information, and then must wait a long time for the results. Also, many objects in the real world tend to be either mostly dynamic, where they have no control over their own motion and merely respond to external forces, or mostly kinematic, where they have detailed control and can specify their motion precisely, guided by goals and plans. Algorithms such as optimal control do not really model how most things work in reality.

3.3.4 Appearance

The creation of realistic 3D images using computers is a well-developed science, and issues of rendering are essentially orthogonal to the concerns of Asgard. Both immediate and off-line graphical output is provided by Asgard. Immediate output is what is displayed in the graphical viewer, which can currently utilize both the X11 or the IRIS GL [106] graphics systems. With X11, only wireframe graphics are supported, but GL provides shading, texture mapping, flexible lighting specification, and visible surface calculation. The output may be viewed using the stand-alone Asgard viewer, which is discussed in Chapter 4, or from within the Ensemble multi-media editing system, as described in Chapter 6.

Off-line graphical output can be produced as PostScript line drawings, PPM [94] or input for Rayshade [69], a public-domain raytracing program. PostScript output is fairly minimal, but Rayshade can accept a wide variety of parameters, including various surface properties and lighting arrangements.

Asgard allows the user to specify colors for particular objects, but does not include facilities for texture mapping and per-face rendering specification. One reason for this is that the faces are generated as a side effect of the vertex positions, as described in Section 4.1.1. There are no conceptual difficulties with adding texture mapping facilities, however. It is also possible to define directional and point light sources, which are used if possible and ignored otherwise. Their specifications are simply passed to the underlying graphics system.

3.4 Architecture

The major components of Asgard can be roughly divided into editing interfaces and motion simulation code. The user interacts with the editing interfaces, which include both the major graphical and textual viewers and also the minor editors such as those for shapes

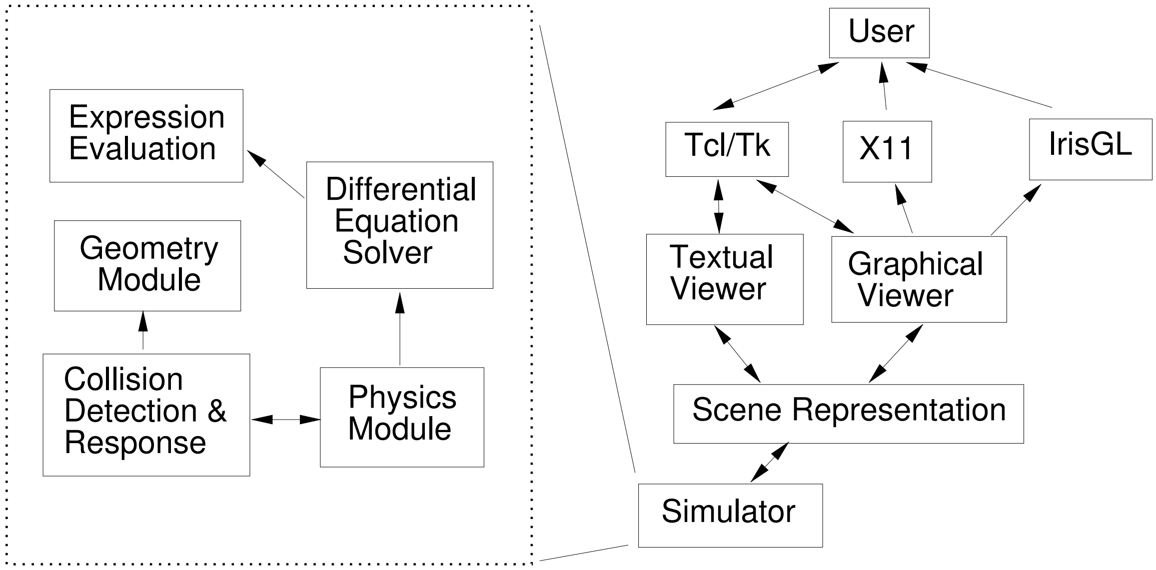


Figure 3.2: Block Diagram of the Asgard System

and trajectories. These editors make changes to the shared state using the scene representation module. The scene representation module has responsibility for tracking changes and determining what data is invalidated by these changes. Then, when the user indicates that he would like to recompute the updated motion, the simulation module performs this computation. The overall block diagram of Asgard is shown in Figure 3.2.

The task of motion simulation is fairly complex, and the substructure of the simulation module is shown to the left. The *physics module* is responsible for translating from the force, link, trajectory, and body specifications provided by the user into a set of first-order differential equations. These equations are solved by the *solution module*, which performs the partitioning and event-driven scheduling described in Chapter 5 and numerically integrates the equations, currently using the trapezoidal method. As the values for the variables are being computed, the *collision detection module* is periodically called to determine if any collisions have taken place. This module, also described in Chapter 5, uses the *geometry module* to determine the closest points between pairs of objects over time. If a collision takes place, the physics module must be invoked to determine the resulting reaction forces, which are then applied to the objects at a single point in time. Finally, the *expression evaluation module* performs symbolic differentiation and simplification of the equations being evaluated, and for large equations, translates them into C code, compiles them, and dynamically loads them for speed. All of the code in the simulation subsystem was written specifically for Asgard, except for the sparse matrix library, which was obtained from Ken Kundert [70]. The collision detection algorithm was developed by Ming Lin [74], although it was reimplemented in C++ for Asgard.

Asgard is implemented using C++ and Tcl/Tk [86]. C++ is used for the more computation-intensive tasks, such as motion simulation and low-level rendering, and Tcl is used for the higher-level user-interface sections of the system. This division of tasks gains the benefits of efficiency where it is needed and flexibility where efficiency is not a concern. Also, the use of Tcl/Tk greatly reduces development time and code size, compared to compiled user interface systems such as Motif[123].

Chapter 4

Animation Editing Techniques

Asgard is a multiple-view editing system. Such systems are designed to present more than one representation of the document, which is in this case an animated scene. In this chapter, the term *document* refers to a compound multimedia document, which may contain a variety of media, both static and dynamic. Each representation of a document has a specific visual or audio presentation style and a specific set of operations that are available for modifying the basic document. In Asgard, the two major types of views are a graphical view, which shows a picture of the scene at a particular point in time and from a particular viewpoint, and a language view, which shows the formal description of the input to the animation system. Multiple-representation systems that have been developed for other media include Vortex [29, 30] for text, and ThingLab [20] and Juno [82] for graphics. These systems are described at the end of this chapter.

The key to making such a system usable is *synchronization between the viewers*. When a change is made in the graphical viewer, it should be immediately reflected in all other viewers, and the same is true, to a certain extent, for changes made in a textual viewer. A user must be able to move an object in a graphical viewer and watch the numeric value of the position change in all the textual viewers that are present, and likewise should be able to add new objects in a textual viewer and, after indicating that he has finished editing, see them appear in the graphical windows.

While there is no *a priori* reason that the editing interfaces should be limited to a graphical and a textual viewer, these two do embody a basic distinction between two ways of looking at the editing process, for any medium. A graphical or direct-manipulation interface presents the document that is being edited, which in the case of Asgard is an animated scene, in a way that is close to how the intended audience will view it. This allows the author to accurately judge the effect it will have on the viewer and modify its appearance accordingly.

A formal language-based view of a document, on the other hand, may or may not be intended to be seen by the final audience. It exists to make explicit the underlying structure or phenomena that determine the content of the document, which is important if one of the functions of the document is to help the viewer visualize these structures or phenomena. On the other hand, in the case of a textual document or in many applications of physically-based graphics, the "final output" is adequate for the audience and the formal language view is useful mostly for the author.

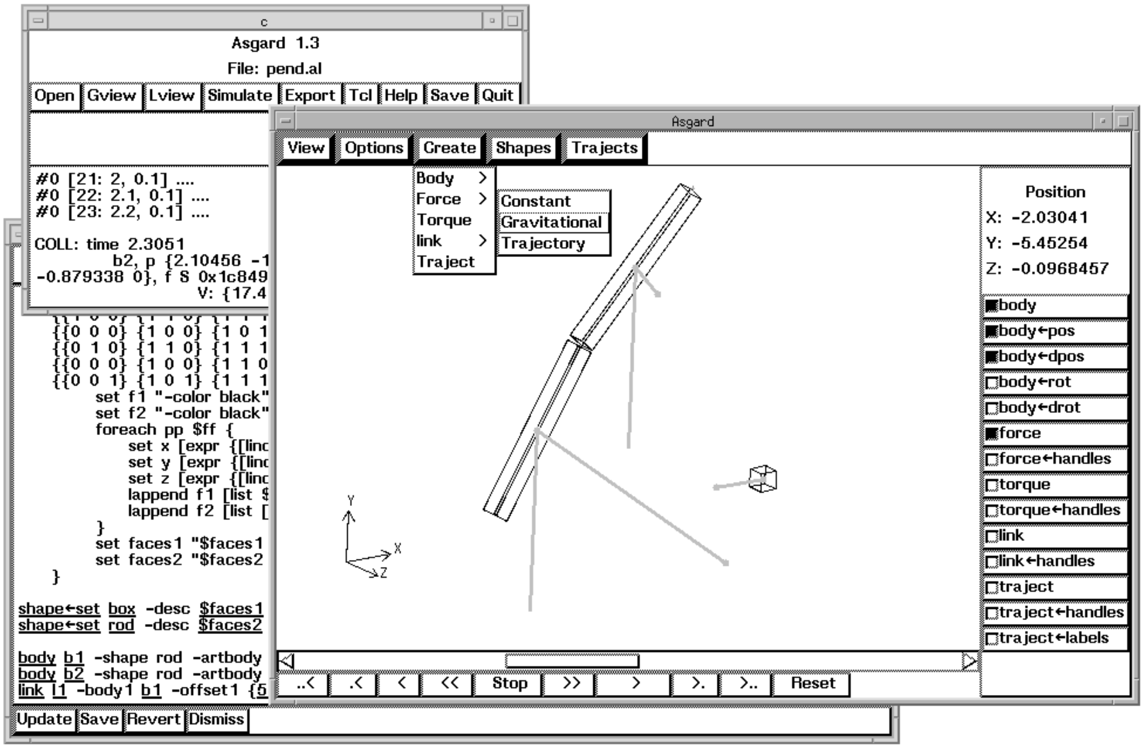


Figure 4.1: A Typical Asgard Editing Session

In the case of physically-based animation editing, the graphical view allows the animator to define certain parameters of the scene, such as shapes, initial conditions, colors, and viewing state, whose effects are related to their values in a very immediate way. The language view, on the other hand, allows the animator to define other scene parameters, such as forces and constraints, whose effects are less directly related to their specifications, which themselves can be fairly complex. One can also use the language view to create procedurally-defined and parametric models. In one case, a direct interface is appropriate because the values being manipulated are concrete and the feedback is immediate and intuitive; in the other, a formal language editing mechanism is appropriate because the qualities being defined are more abstract and the feedback is more indirect.

Figure 4.1 shows a typical Asgard editing session. This picture contains all of the major interface components of Asgard. These are as follows:

- In the upper left-hand corner, the control panel. This contains menus and dialog boxes that are used for managing the overall state of Asgard.

- In the upper right-hand corner, a graphical viewer, with a typical scene displayed. There may be any number of graphical viewers open at any time, with different spatial and temporal viewpoints.

- Below the graphical viewer, a trajectory editor, used for defining paths in space for objects to follow. One of these may be active at any given time.

- Below the control panel, a language viewer, which displays the formal description of the current scene. Any number of language viewers may be present, and although they are not synchronized on a keystroke basis, they can all be brought up to date using an "Update" button on any one of them.

- Beneath the language viewer, a shape editor, which allows the user to create and modify shape definitions. One shape editor may be active at any time.

The rest of this chapter describes these various tools in detail. It also discusses some previous work in multiple-representation editing for other media.

4.1 Graphical display and editing

The graphical editor in Asgard allows the user to directly create, delete, and manipulate the objects being animated, view their motion, and define initial conditions and other parameters interactively. An important consideration is to provide as much editing functionality as possible without trying to cover areas that are better handled by the language viewer. For example, the graphical editor provides shape creation facilities and interactive 3-D positioning, but not text-based interfaces for precise numerical values, since the latter can be done using the language viewer. The user can define rough positions graphically, and then adjust them precisely textually if necessary.

Many of the editing functions described here utilize the mouse. There are two sets of operations, which can be classified as *positioning* commands, which allow the user to modify the overall scene transformation or camera position, and *selection* operations, which are used for selecting bodies and other objects on the screen and for indicating positions. The positioning commands use the unshifted mouse buttons, and the selection commands require the user to hold the shift key down while the operation is performed. There are a number of possible ways to organize different sets of operations in an editing system such as Asgard, and there has been a lot of human-factors work on determining what style is easiest to use [105]. The style described above was chosen because having separate positioning and selection modes that the user would be forced to explicitly toggle between turned out to be rather awkward and difficult to use.

A picture of the graphical viewer is shown in Figure 4.2. A menu bar is at the top of the window, and directly below it is the display area. To the right of the display window is a status area that contains status information and visibility controls, and at the bottom of the screen is a set of time control buttons and a time scrollbar. The slider inside of the time scrollbar is designed so that the left edge is at the current time, and the length corresponds to one second. Directly above the time scroll bar is an area used for editing trajectories, which is described below.

The operations that the user can perform using the graphical viewer include the following.

- Create and edit definitions of shapes. When the Shapes/New menu option is selected, a shape editor, described below, is invoked. When the user is finished editing the shape, he can define bodies that are instances of this shape, and attach non-geometric

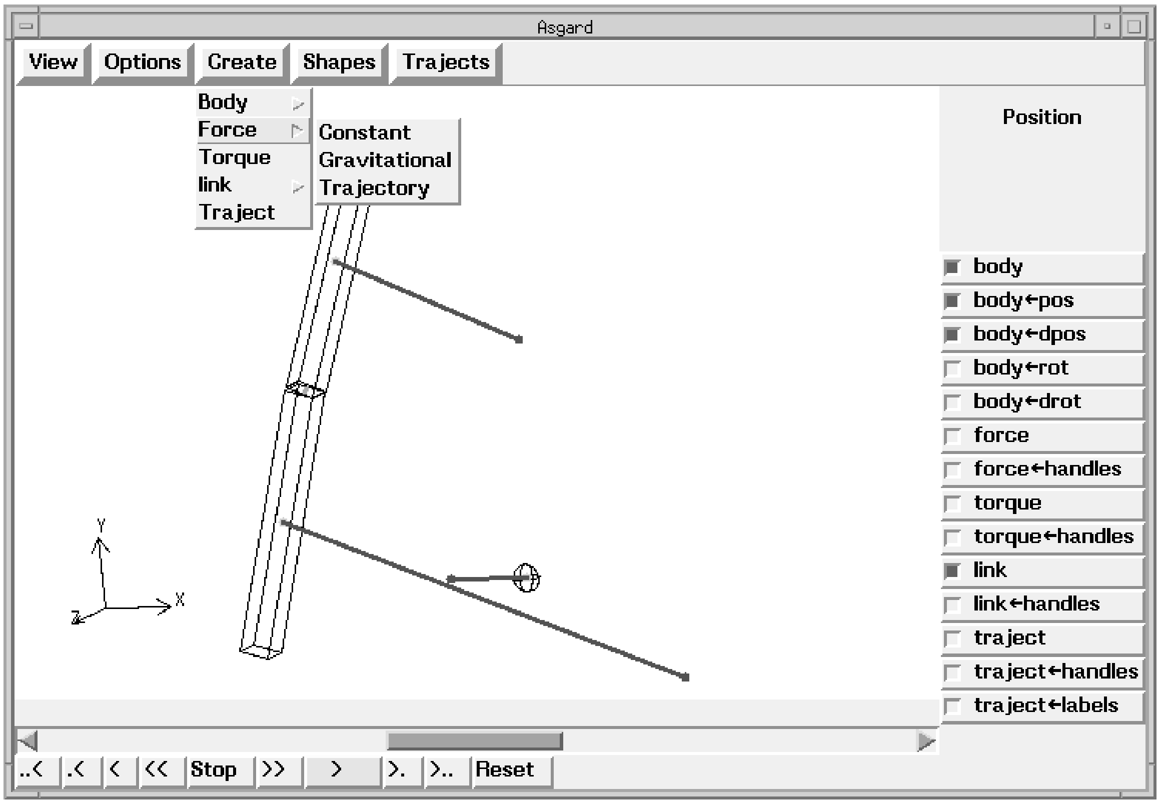


Figure 4.2: The Graphical Viewer

properties to them using the language viewer. Existing shapes can also be modified using the Shapes menu.

- Create scene elements such as bodies, forces, and links. This is done using commands in the Create menu. In general, when an element is created, the user will be prompted to indicate spatial and object parameters by clicking on locations on the screen. Non-spatial parameters can then be entered textually.

- Edit scene elements. Using the element type check buttons on the right of the viewer, the user can enable or disable the handles that are used to control each element type. If the handles for a particular element are visible, they can be moved on the screen to edit various parameters such as initial conditions and link and force offsets.

- Create and edit trajectories using a trajectory editor. These trajectories can then be attached to objects and used in other contexts where vector values as functions of time are required.

4.1.1 Object definition

To create an object, the user must specify two types of information: a shape description and additional properties such as appearance and physical attributes. In Asgard, the shape description is created using a graphical interface, and the additional properties are given textually using the language editor. In many systems such as Alias [1], the user can select shapes from a pre-defined list of geometric primitives, which include spheres, cylinders, parallelepipeds. Since the basic primitives that Asgard uses are union of convex polyhedra and spheres, rather than presenting the user with a list of convex polyhedra to choose from, the system includes a general shape editor for these types of objects. This editor, however, does contain shortcuts for creating spheres of a particular radius and boxes whose sides are aligned with the coordinate axes, since creating such objects is a frequent task.

Many techniques have been developed for defining and editing shapes graphically. These generally fall into two categories: surface-based editing and volume editing.

In surface-based editing, the user defines 2-dimensional surfaces, which may be polygons or spline patches. He can then select sets of surfaces and create volume objects from them by grouping them together and, if necessary, indicating which should be opposite faces and which surfaces should be joined together as sub-faces to make a single face. Examples of this style of object construction are the ICEM Mulcad system [36] and Swivel 3D [88]. Many other CAD programs offer similar functionality.

In volume editing, the user typically starts with volume primitives and modifies them to form the shape he desires. Some examples of this are hierarchical B-spline refinement [45], where the user defines an initial shape and then interactively drags control points, allowing the system to determine where subdivision is required, and hyperpatches [67], which is an extension of 2D spline surface calculation techniques. Constructive solid geometry [44] is another technique for incrementally defining shapes.

Asgard provides a third model of shape creation that is tuned to the object model used by the simulation subsystem. The primitive objects that it uses are spheres and convex

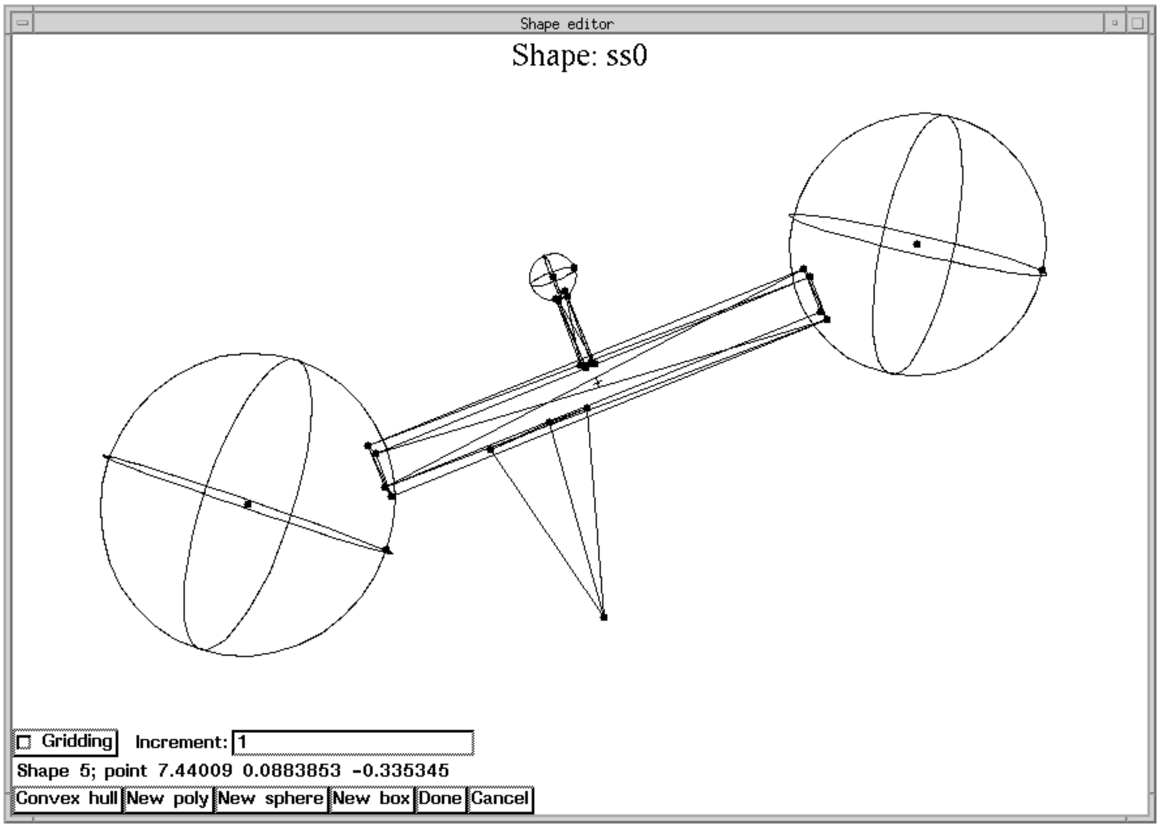


Figure 4.3: Shape Editor

polyhedra, and the values that define these objects are, in the case of spheres, a center point and a point on the surface of the sphere, and in the case of polyhedra, a set of vertices, which define the shape by their convex hull. These shapes are grouped together to form "shape sets", which are then bound to objects. The shape set is defined as the union of its component shapes – if necessary this scheme could be extended to a general constructive solid geometry model, where intersection and subtraction operations are also provided.

Shape sets are defined using a special shape editor. A picture of this editor is shown in Figure 4.3. This editor displays a set of spheres and polyhedra. Each object is represented by its vertices, which are drawn using small balls that can be moved by the user. A vertex is moved by clicking on it with the left button while the shift key is held down and dragging it with the mouse. Once a vertex is clicked on, it and all the other vertices that belong to the same object, along with the lines that connect them, are drawn in color, and the object is considered *selected*. Other objects are drawn in gray. When a polygon is selected, new vertices can be added to it by clicking the middle mouse button at the desired position, and a vertex can be deleted by clicking the right button on it, both with the shift key held down. Spheres have two control points, the center and a point on the surface, and these can also be moved using the shift-left button operation. New spheres and polyhedrons can be created using buttons at the bottom of the editor window. The unshifted keys control the scene transformation in the shape editing window, just as in the graphical viewer.

A polyhedron can have any number of vertices, and is defined by their convex hull. The convex hull is computed automatically, when the user switches from one selected component to another and when he clicks on the "Convex Hull" button in the shape editor. An external program, written by Ioannis Emiris [43, 42], is used to perform this computation. Whenever a vertex moves inside its component, and thus does not lie on its convex hull, it is colored white. When the shape editor is exited, all vertices not on the boundary will be deleted from the shape description.

There are a number of reasons why Asgard provides a special viewer for shape editing, as opposed to integrating this functionality into the graphical viewer. First, the operations that are performed on bodies, or instances of shapes, which include positioning and rotating in world space and defining initial conditions, are different from the operations that one performs on shape definitions, and it would be awkward to overload the semantics of mouse operations to accommodate both in the graphical viewer or to define a separate editing mode for shape editing. Second, there may not be an instance of a shape that the user wishes to edit currently visible in the graphical viewer – a user will generally want to create a new shape first and then to create instances of it. Third, it turned out to be cleaner from a software engineering point of view to separate the implementations of these two types of editors, although significant modules are shared between them.

There are a few tradeoffs involved in providing automatic convex hull calculation as a basic shape editing mechanism. On the positive side, the amount of information that the user must provide is rather minimal and direct, and it is possible to create various types of polyhedra rapidly with little effort. On the negative side, sometimes the user would rather work with faces than vertices, and impose constraints on these faces, such as requiring the edges of a face to remain parallel or keeping two faces a constant distant from each other. Asgard attempts to mitigate some of these problems by allowing the user to define a spatial grid for vertex positions, which makes it a bit easier to line up vertices in a plane parallel to the coordinate axes. Some CAD systems such as P-cube [37] have very powerful constraint systems, which are very useful but also are quite a lot of work to implement.

Per-shape parameters, such as color and other optical properties, can be created and modified using the language view. In previous versions of Asgard, colors could be assigned to specific faces, but since faces are now only an artifact of the convex hull operation, colors are defined on a per-object basis. Asgard could allow the user to attach colors to vertices, since high-end graphics systems can accept a color for each vertex and use these colors to perform Goraud shading [19], but this does not seem very useful. If one wants to assign separate colors to different faces of an object, this can be done by defining one shape component for each face and positioning the shapes accordingly. Since these polyhedra are all part of the same object, from the point of the simulation system, this formulation does not affect the mass or other physical properties of the object. Texture maps [44] could also be used for this purpose, although Asgard does not support them.

Other properties that can be attributed to objects are mass and moment of inertia. These are attached to bodies rather than shapes, and can be explicitly specified via the language viewer. If they are not given by the user they are automatically calculated assuming a constant unit density. This is very easy for convex polyhedra, since they can be decomposed into tetrahedra, the volumes and centroids of which can then be combined to yield the total for the shape.

4.1.2 Instance creation and modification

Instances can be created using the Create/Body menu item. The user is asked to select an existing shape definition, and the body is created with its initial position at the origin. It can then be moved and rotated, either by modifying the `-init_pos` and `-init_rot` values using the language viewer, or by clicking on the appropriate handles and dragging them in the graphical viewer, as described below.

Initial values for velocities and angular velocities can also be modified using handles. Angular velocity is represented as a vector through the center of mass, where the direction defines the line about which the object is rotated, and the magnitude defines the speed of rotation, using the right-hand rule (that is, counterclockwise about the vector when it is pointing at the viewer). Since it is not possible to correlate rotational values with vector magnitudes as easily as in the translational case, textual labels are attached to the ends of these handles which show the values.

4.1.3 Handles

There are two mechanisms in the graphical viewer for modifying positions and other spatial parameters such as velocities. Objects can be transformed using the shifted mouse buttons while in *positioning* mode, as described below. For other types of editing, which include the modification of vector values such as velocities, trajectory points, force vectors, link offsets, and lighting controls, objects called *handles* are used. A handle is represented as a small sphere on the screen, using a color that indicates what kind of quantity it is tied to.

Additionally, while a handle is being manipulated, pertinent information is displayed in a label in the upper right hand corner of the viewer.

While the viewer is in selection mode, the left button can be used to move handles. When it is depressed, the nearest handle within a certain distance is selected, and when the mouse is moved, the handle tracks its position in the plane of the screen. The 'z' and 'x' keys can be used to push the handle into the plane of the screen or move it towards the viewer while the mouse button is depressed, and the current position of the handle is shown in the upper right hand part of the screen. In some cases, such as handles that control the offset of a link, the default is to constrain the handle position in some way, such as to be within the shape of the object. If the shift key is held down, this constraint is not enforced, and if the next time the handle is moved the constraint is not satisfied, no attempt is made to enforce it until it becomes satisfied as a result of the user's modifications.

4.1.4 Trajectories

A trajectory is a path in space which is parameterized by time. Trajectories can be used wherever variable references are called for, and can also be used to define the paths of instances over specific intervals. Trajectories can be used for position, rotation, and their derivatives, and can be defined for all of time or for a particular interval. In the latter case, all forces that affect the constrained state values are ignored, and the object is treated as having infinite mass for the purposes of collision response. If two such bodies collide, it is not possible to determine correct behavior, since both constraints are considered to be strict by the system. Discontinuities may result in the state of an object when a trajectory is imposed, if the position or rotation of the object at that point is different from the required one. The motion simulation code takes this fact into account when timesteps are chosen, and the playback routine treats this discontinuity as a very short duration displacement.

Trajectories can be edited in the graphical view by interactively adding, deleting, and moving control points. By default, trajectories are not shown, but they can be displayed using a checkbutton in the right panel.

Each point along a trajectory is represented by a handle. A point can be moved using the shifted left button. The right and middle buttons can be used to create and delete control points. To create a new control point, the user must click on an existing one with the middle button and then move the mouse in either direction along the trajectory. This will create a new point either before or after the existing point, depending on the direction taken. If there is only one existing control point, then the new one will come after it. When the button is released, the trajectory will remain where it was placed. To delete an existing control point, one can click on it with the right button. If all points on a trajectory are deleted, the trajectory will be deleted.

When a trajectory is being modified, a scale appears above the time scrollbar at the bottom of the screen that contains a mark for each control point, which shows the time that control point corresponds to. The user can click on one of these marks with the left button and then drag it to alter the time value of the corresponding control point. The time value for each trajectory point is displayed textually next to the point on the screen.

By default, the trajectory time scale is the same size and corresponds to the time scroll

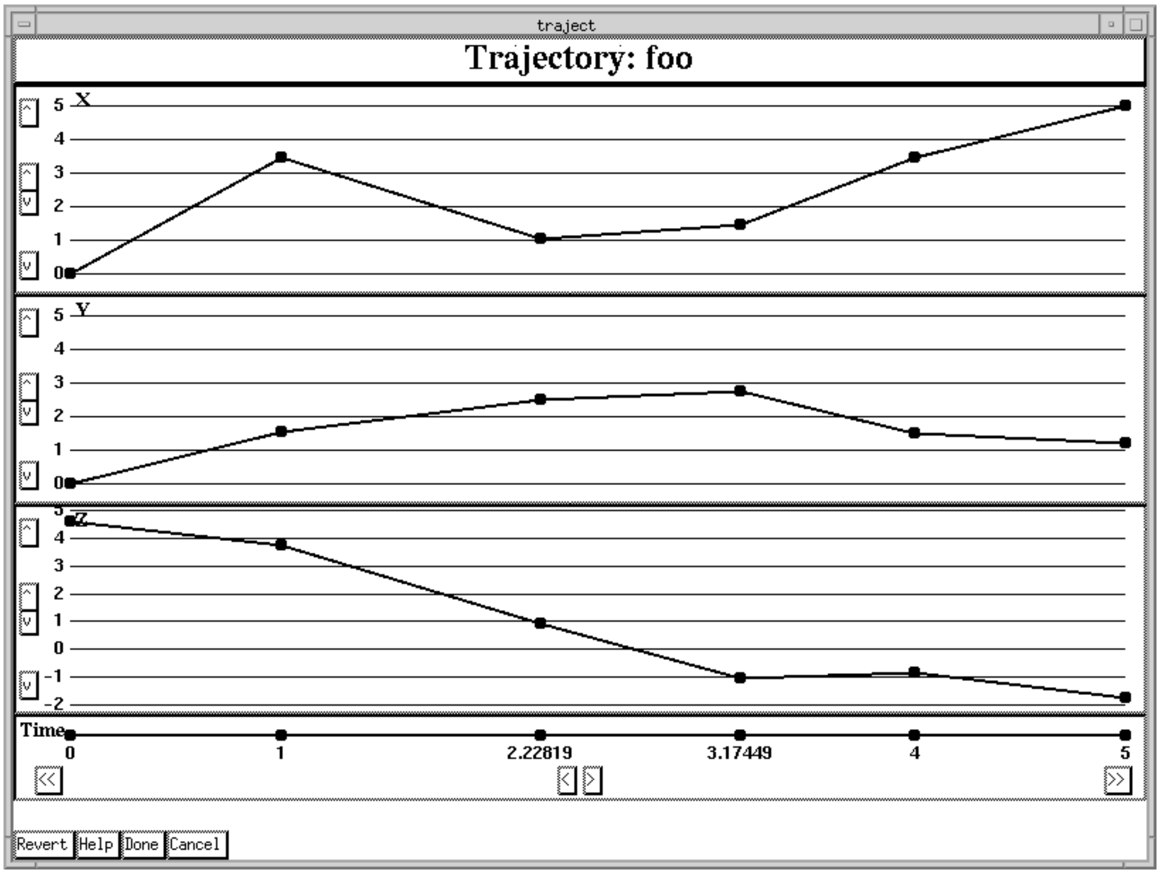


Figure 4.4: Trajectory Editor

bar. Using the zoom controls at the left of this scale, however, one can edit the time at a finer scale than is possible otherwise.

In addition to directly editing a trajectory in the graphical view, one can also edit a trajectory in a component-wise fashion using a special trajectory editor, which is shown in Figure 4.4. This is especially useful for rotation trajectories, which may not be easy to edit in 3-space. The trajectory editor presents an interface simular to that described above, except that it is 2-dimensional and there are three trajectory components shown instead of one. The Y axis in each sub-window represents the component value and the X axis represents time.

4.1.5 Forces and torques

A wide variety of forces and torques can be specified textually, and the more complex ones can only be described using that interface. However, simple forces, like gravity or constant forces, or forces that are defined by a trajectory, can be defined graphically.

The Create/Forces menu item contains a submenu of force types, such as "Constant", "Gravitational", and "Trajectory". In the first two cases, the force is created with unity magnitude, in the negative Y direction. The user can then edit the value in the language viewer, or modify the vector using handles, similar to those available for setting the initial value of the velocity. In the third case, the user will be prompted to select a trajectory on the screen, or request a new one with one point which can then be edited as described above.

A force vector is graphically represented using two handles – one at the base, which determines the point at which it is applied, and one at the head, which determines the direction. If the force is not constant or mass-dependent, a vector will be drawn that gives the value of the force at the current time point, but no handle will be provided since the vector cannot be modified graphically. If a force is not applied to the center of mass, the system transforms it into a linear force and a torque, using simple mechanics identities [52].

Either constant or trajectory-defined torques can be created using the Create/Torque menu item. The graphical representation of a torque is a vector about which the torque acts, where the magnitude of the vector is proportional to the magnitude of the torque, using the right-hand rule. This is the same representation as the one used for the initial value for the angular velocity. Torques can be defined about the center of mass, or any other point on the body and axis through that point. In the latter case, the system internally decomposes the torque into a pure torque component and a linear component, similarly to the non-central force case.

4.1.6 Links

A link is a critically-damped spring that is used to connect body components to one another and to points in space. The Create/Link menu item contains a list of possible variations on these link types: object-to-object, object-to-point, or object-to-trajectory. Other types of object-to-point links, where a general expression rather than a trajectory is used, can be created using the language viewer, or an object-to-point link can be created and then edited to replace the constant point value with a set of expressions.

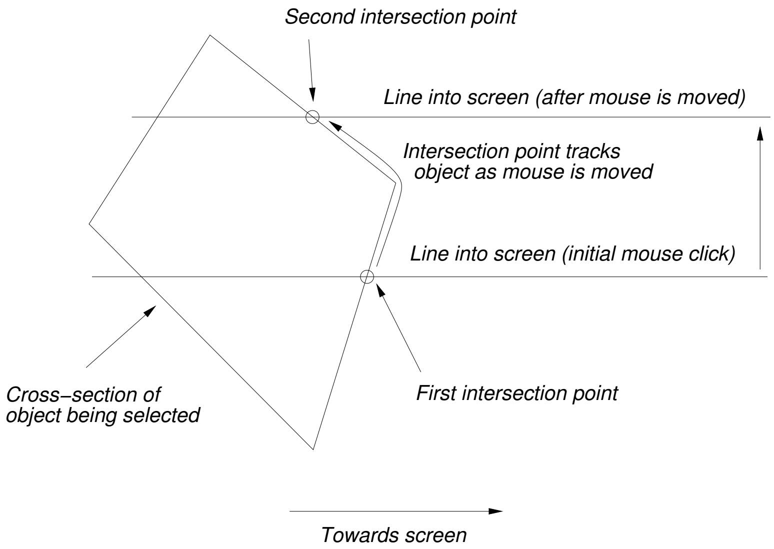


Figure 4.5: Object intersection constraints

An object-to-object link has two object parameters and one handle for each object, whose position is defined with respect to the coordinate system of the object. When such a link is created, the user is prompted to select one object and a location on that object, and then another object and location. Object selection is done by clicking the left mouse button anywhere on the shape, and the location is determined by finding the nearest point on the surface of the shape that is covered by the mouse position. The user can move the handles corresponding to the endpoints of the link, and they will track the surface of the shape. This is the most common case, but the user can move them off of the surface of the shape depressing the "a" key while performing this operation. In this case, the movement is constrained to be in the plane that passes through the initial point position and is parallel to the screen. This is the default behavior for other types of handle-based editing that are not subject to constraints, such as modifying the magnitude of a constant force vector.

Constraining the movement of a point to follow the surface of an object has been used in other systems [18, 17]. In the case of Asgard, the operation is fairly simple. Whenever the user moves the mouse while he is dragging a handle, the system must determine a new position for the handle. One constraint that must be met if possible is that the point be on the line that passes through the mouse position and is normal to the plane of the screen. The remaining degree of freedom for the position is along this line. This operation is illustrated in Figure 4.5, which shows a 2-dimensional cross section of an object, perpendicular to the screen.

In the unconstrained case, the depth of the point is taken to be the previous depth. One could also constrain the movement to be in one of the coordinate planes, but this tends to be hard for the user to visualize and control, and can be easily done by using one of the orientation commands to make the plane of the screen be a coordinate plane. In the case where the movement is constrained to a surface, one must simply find the point on the surface that intersects the cursor line, and is closest to the screen. This is an easy operation for simple polyhedra – one need merely intersect the line with each face. In complex cases one could use a nearest-feature collision detection technique such as the Lin-Canny algorithm [74, 75], which is used in the motion simulation subsystem of Asgard.

The process for creating a link between an object and a path or point in space is similar. One selects an object and a point on the object with one mouse click, and selects a trajectory or a point with the second. The point can then be moved graphically, or the link can be modified in arbitrary ways using the language viewer.

4.1.7 Viewing state

There are four basic vector values that determine the how the scene is viewed. These are position, rotation, scale, and the center of rotation and scaling. The first three can be manipulated using the mouse: the left button rotates the scene using a virtual sphere model[28], the middle button translates the scene in the plane of the screen, and the right button scales the scene either up or down, if the mouse is moved vertically, or rotates it about the Z axis, which points out of the screen, if the mouse is moved horizontally. This is the model used by the DDN CAD system [61].

The center of rotation and scaling is by default the centroid of the scene. It can be moved by typing "c" with the mouse positioned above the desired point. If the mouse is within a few pixels of any shape vertex or handle, the center point then becomes the location of that entity. It does not track the position of this entity if it is later moved, however. If the mouse is not near any vertex or handle, the center becomes the point under the mouse that is midway in depth between the near and far bounding planes for the currently displayed objects.

Rotations are represented internally by the viewer using quaternions [58, 93]. These are 4-dimensional vector quantities that can be thought of as a vector/scalar pair. Rotations in 3-space about a particular vector can be represented by the quaternion $< \cos(\Theta/2), \vec{v} * \sin(\Theta/2) >$, where $\vec{v}$ is the vector and Θ is the angle of rotation about it, using the right-hand rule. Quaternions are useful for representing rotations because one can compose rotations by multiplying the quaternions, whereas there is no easy way to compose rotations represented by Eulerian angles. Also, interpolating between two quaternions corresponds to interpolating the rotations in a correct way, which is otherwise very difficult. Rotations are composed when the overall scene transformation is modified, and interpolation is performed during playback when the time points are further apart than the frame rate.

In addition to using mouse controls to modify the viewing parameters of a scene interactively, these parameters may be tied to trajectories or arbitrary expressions. This allows for fairly sophisticated camera motion effects. An "eye point / look direction" model would be useful also, but would be harder to provide using the available Asgard primitives for rotation.

4.1.8 Lighting

By default, a directional light source is defined which points into the plane of the screen. This may be replaced with point and directional light sources defined by the user using the "lightsource" command. They can also be created using the Create/Light source menu item. These may or may not have an effect on the image, depending on the graphics system or output format being used – if Rayshade [69] output is being written, or if the graphics system is GL [19], then lighting will be enabled, and otherwise it will have no effect. In the case of Rayshade output, shadows will be generated, and in the case of GL the lighting will only contribute to the shading of the objects in the scene.

Light sources can be moved using handles. Point sources have one handle, and directional sources have two. Actually, a directional source only needs one, but it is more convenient to be able to put the representation somewhere on the screen and then move the direction vector.

4.1.9 Motion interpolation

When motion is played back after being computed, one must somehow match the frame rate of the display with the time scale of the data available. The two major problems with this task in Asgard are the unpredictability of the graphics speed and the differing time scales for state variables. One can never be sure that a given frame rate is achievable, especially on a multi-tasking system using X11 and a software implementation of 3D graphics operations. Also, each variable may have a different set of time points, which is a consequence of the differential equation solution technique used.

Asgard solves these problems in two ways. First, it plays the motion as fast as possible, checking the real-time clock before each frame and determining at what point in the simulated time flow the scene should be drawn. This allows it to adapt to different hardware and software capabilities. Second, it interpolates the values of all state variables, using linear interpolation. This is appropriate since the integration method used, trapezoidal integration, uses a linear approximation to the function value. For a higher-order integration method a different interpolation scheme would be necessary.

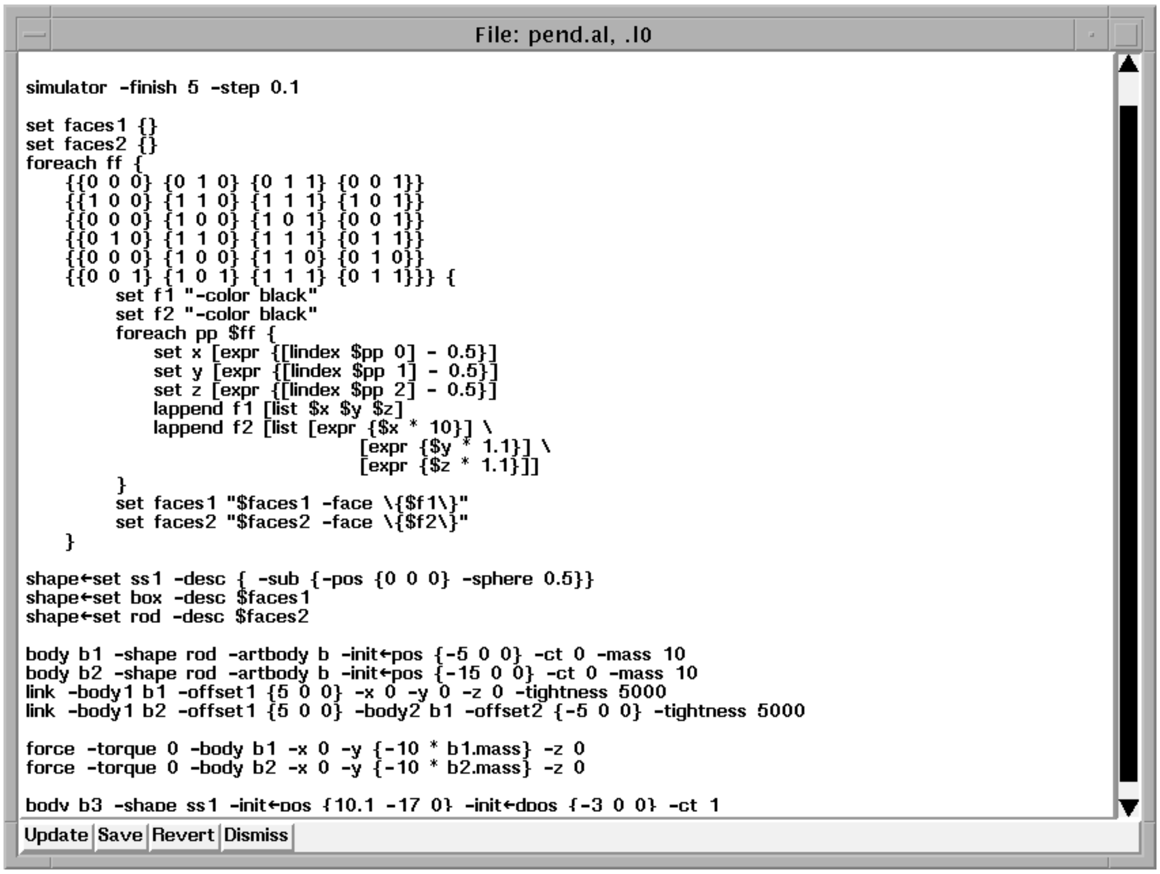
4.1.10 Image export and off-line playback

Images can be exported in a variety of formats, using the "Export" button on the control panel. Postscript [62] and PPM [94] formats are available for individual frames. For saving a sequence of frames that can be played as a movie, a file that contains sequences of graphics commands can be created and played back, using a special playback program written for Asgard. Images and movies can also be written in JPEG and MPEG formats, using publicly available conversion tools that accept PPM format.

In addition, input files for the Rayshade [69] renderer can be created for off-line ray-traced pictures. This program produces pixmap images in the Utah Raster Toolkit RLE format [117], which can then be manipulated with a variety of tools and played back in movie mode.

4.2 Language-based editing

The Asgard language viewer allows the formal description of the animated scene to be edited. The viewer operates as a simple text editor, with the usual command bindings – it uses the Tk text widget [85] – and can be used to edit animation descriptions without syntactic or




```
simulator -finish 5 -step 0.1

set faces1 {}
set faces2 {}
foreach ff {
    {{0 0 0} {0 1 0} {0 1 1} {0 0 1}}
    {{1 0 0} {1 1 0} {1 1 1} {1 0 1}}
    {{0 0 0} {1 0 0} {1 0 1} {0 0 1}}
    {{0 1 0} {1 1 0} {1 1 1} {0 1 1}}
    {{0 0 0} {1 0 0} {1 1 0} {0 1 0}}
    {{0 0 1} {1 0 1} {1 1 1} {0 1 1}}} {
         set f1 "-color black"
         set f2 "-color black"
         foreach pp $ff {
             set x [expr {[lindex $pp 0] - 0.5}]
             set y [expr {[lindex $pp 1] - 0.5}]
             set z [expr {[lindex $pp 2] - 0.5}]
             lappend f1 [list $x $y $z]
             lappend f2 [list [expr {$x * 10}] \
                                     [expr {$y * 1.1}] \
                                     [expr {$z * 1.1}]]
         }
         set faces1 "$faces1 -face \{$f1\}"
         set faces2 "$faces2 -face \{$f2\}"
    }

shape←set ss1 -desc { -sub {-pos {0 0 0} -sphere 0.5}}
shape←set box -desc $faces1
shape←set rod -desc $faces2

body b1 -shape rod -artbody b -init←pos {-5 0 0} -ct 0 -mass 10
body b2 -shape rod -artbody b -init←pos {-15 0 0} -ct 0 -mass 10
link -body1 b1 -offset1 {5 0 0} -x 0 -y 0 -z 0 -tightness 5000
link -body1 b2 -offset1 {5 0 0} -body2 b1 -offset2 {-5 0 0} -tightness 5000

force -torque 0 -body b1 -x 0 -y {-10 * b1.mass} -z 0
force -torque 0 -body b2 -x 0 -y {-10 * b2.mass} -z 0

body b3 -shape ss1 -init←pos {10.1 -17 0} -init←dpos {-3 0 0} -ct 1
```

Figure 4.6: Language Viewer

semantic guidance or restrictions from the system. When the user has made changes to the text and wants to update the internal representation and the other viewers, he can click on an "Update" button, which causes the scene description to be parsed and evaluated. This brings the internal database up to date with the description, which also causes the graphical views and the other language views to be similarly updated. If there is some error in the description, the offending textual element is flagged in the text, and for each statement that is successfully parsed, certain arguments, which are the ones that are tied to parameters that can be modified graphically, are highlighted in distinctive colors. The language viewer is shown in Figure 4.6.

The motivation for providing a general-purpose text editor as the mechanism for creating and modifying animation descriptions comes from the Ensemble system [55]. Ensemble is a successor project to both the Pan language-based editor [10] and the Vortex document processing system [29], which treated documents as both formal language descriptions and visual representations. The design of the language editing components of these systems was guided by the observation that users do not want to be limited to using only a strict syntax-directed editor, such as the Synthesizer Generator [97] – rather than being limited to transformations that operate on well-formed descriptions, users prefer to edit programs and document descriptions as regular text, and then indicate when the editor should analyze the text as a formal description. Pan, Vortex, and Ensemble are all designed to provide the best of both worlds – syntax directed editing and free-form textual editing can both be used as needed. Although the structure of the Asgard language is simpler than that of a general purpose programming language, the language viewer uses the same approach. Perhaps more than most other types of media, animation benefits from the close coupling of formal language and direct representations.

There were two approaches that were used for implementing language-based editing in Asgard: first, integration with the Ensemble system, which contains the language-based editing components of Pan, and second, via a separate editor that is independent of Ensemble but is closely integrated with Asgard. The second option is described in this chapter, and the integration with Ensemble is discussed in Chapter 6.

4.2.1 The Asgard animation language

The animation language is based on Tcl [86]. An animation description is read as a Tcl command file, and as such can contain variable definitions, control constructs, procedures, and any other elements of the Tcl language. This capability is important for defining large and complex scenes, or for defining objects procedurally. An example of the animation language is visible in the language viewer shown in Figure 4.6.

It is important to note that all aspects of the scene that cannot be recreated are represented in the language, since this is both the formal description that the user sees and the file storage format. For example, since state variables as functions of time can be recomputed, they are not visible in the language view. Trajectory values, on the other hand, are input to the animation process and thus they must be explicitly represented. In many cases these representations are not especially enlighting or easily editable. For this reason, certain parts of the language can be elided, or replaced in the language viewer with an icon. If the user clicks on this icon with the right button, the full text is shown, and if he again clicks on the text, which is drawn with a distinctive background, it reverts to the icon. The parts of the language that are currently candidates for this sort of elision are the points in a trajectory and in a complex shape: if the text is larger than a certain threshhold value, currently about half a line, it will be elided by default.

In the following discussion, the term *element* refers to any component of the animation description that is meaningful to the simulator or viewer. Elements include shapes, bodies, forces, links, and trajectories. Things that appear in the description that are not element definitions are generally parts of the Tcl language: control constructs, comments, and variable definitions.

4.2.2 Parsing and semantic analysis

The major function of the language editor is to translate the textual representation into the internal format of the scene, in such a way as to allow a change in this internal format to be mapped back into a textual change. This type of problem arises in other types of multiple-representation editing systems, some of which are described at the end of this chapter.

The Ensemble system contains fairly sophisticated facilities for handling general program editing, including incremental lexical analysis, parsing, and semantic analysis. This is especially useful for languages that have a complex structure, like C++ or Ada. The Asgard animation language, however, is relatively simple, and easy to parse and format. It is more important for a system like Asgard to give the user feedback on particular parameters and tokens that are directly tied to the graphical state than to provide assistance for ensuring syntactic and semantic correctness.

There are two main types of functionality that we wish to provide. First, we must perform incremental change analysis – when a new version of an animation description has been completed, either from scratch or by modifying an existing one, we must find out what elements have been added, deleted, and modified, and use this information to update the internal scene representation. Second, we must be able to identify and modify language elements when the underlying database objects have been changed by the user, either graphically or textually.

The key to performing these two functions is being able to perform reverse mapping from internal objects to their textual representations. For some types of objects, this is relatively straightforward, because they are identified by a unique name, and this makes it easy to find them in the text stream. For others, such as forces and links, this is not the case, and we must resort to more complex and less elegant strategies to locate the textual descriptions. Asgard relies on certain features of the Tk text widget for this: it attaches tags to places in the text where the non-named elements are defined, and then keeps track of the lines that it is currently executing while it evaluates the text, so that it can go back and figure out which tag is attached to the line being evaluated, and thus which element definition is being read.

Another possibility, which was used in an earlier version of Asgard, is to require every command that corresponds to an element of the database to be of the form

element_type name options ...

Even elements that are never referenced elsewhere in the language by their names, such as forces, would have to be given names. This had some advantages, such as more robust identification of places where elements are unchanged, but the requirement that everything have a unique name given by the user was judged to be too onerous.

Once the reverse mapping analysis is done, Asgard can then make the appropriate changes to the internal state of the graphical viewers and the simulation modules. In the case of the motion simulation subsystem, an additional level of incremental analysis must be performed to determine what motion needs to be recalculated – this is described in Chapter 5.

While the process of parsing is taking place, the system also remembers the textual positions of certain language elements such as the values for the initial positions and orientations of objects. These values are tagged in the textual stream using Tk text widget tags, and are highlighted to aid in the clarity of presentation. The tags are saved with the internal representation of the scene, and can be used to modify the textual values when they are changed graphically. This is done using an event-based model that relies on the Tcl variable trace facility.

Another problem is one that arises in many multiple-representation editing systems – that of one-to-many mappings between formal language elements and graphical objects. Part of the power of a formal language is that one can create objects procedurally, for example inside of a loop. For example, one might create four boxes using a command like:

```
foreach i {10 20 30 40} {
    instance obj_$i box -init_pos "$i 0 0"
}
```

In general, there is no reasonable way for the system to identify a parameter that corresponds to the initial position of object `obj_10`, which could be modified when that object is moved.

We handle cases like this by keeping track of the number of elements created by any one command in the text stream. If this number is more than one, it is not safe to modify the parameters of the elements graphically, since this would interfere with the other elements that came from the same command. In the above example, if the user tries to graphically modify `obj_10`, it is ambiguous what should happen to the text, and it is a difficult problem even to identify the reasonable choices and ask the user which is desired. For this reason, graphical editing is not possible for such objects.

Additionally, when an object is defined with a string literal, but the position and orientation parameters are not coordinate values but rather are more complex descriptions such as expressions, the parser will refuse to tag them and they will be marked as read-only for the graphical viewer. The alternative, which is to replace the expressions with the literal values, is probably not what the user wants.

Although this parsing scheme has only been described for instances and simple coordinate parameters, it applies to all objects which can be edited graphically. The basic rule of thumb is that something is safe to modify graphically if and only if it's easy to parse.

4.2.3 Errors

Tcl has a general mechanism for dealing with errors: any procedure can signal an error, either by calling the `error` built-in function, if it is a Tcl procedure, or by returning an error status, if it is implemented in C. When errors occur as the result of executing an Asgard animation description, they may come from either source. Since the only indication of where an error has occurred is a human-readable string, it is hard for a system such as the Asgard language viewer to always show the user exactly where the error has occurred.

However, one can identify two types of errors that might take place during the evaluation of an animation description. The first is a regular Tcl error that occurs outside of the evaluation of an element definition, such as `instance` or `force`. An example of an error like this would be a random line in the file that causes an "unknown command" error. Not much can be done about this besides presenting the user with a message window that contains the error message.

The second type of error is one that occurs within the evaluation of an element definition. A simple example is the line `body b1 box -init_pos`, where there is no argument for the `-init_pos` option. The routine that implements the `body` procedure does a certain amount of bookkeeping to determine whether this is a new or changed object, and at the point when it actually creates or modifies the body, it uses the `catch` function to trap any errors that occur. In certain cases, as described above, it can find the text of the offending command in the viewer and highlight it, so that the user can see where the error took place. If it cannot find it, it uses the fallback method of just presenting the error message without a location marker.

4.3 Performance

Another issue that arises when maintaining multiple representations is performance. When the granularity of the editing changes recognized by the system is small, such as one keystroke, or when the amount of update required to maintain the representation is large, such as simulating the motion of a set of objects or parsing and evaluating an entire animation description, it may not be possible to keep the representation up to date all the time.

In the case of Asgard, there are several places where efficiency is important. One is in the basic text editing functionality of the language viewer. If any significant work were to be done between each keystroke, such as analyzing the changes made, performance would be an issue, but since the text is treated as simple text from an editing standpoint, and only analyzed when the "Update" button is pressed, there is no problem. This is not just a compromise to achieve acceptable efficiency: while editing a description, the text may pass through many stages that are not well-formed or not what the user wants, and any feedback based on these invalid states would be a hindrance rather than a benefit.

Another area where performance is a concern is during the parsing, semantic analysis, and modification of the internal database that takes place when the update function is invoked. One would expect this to take at most a few seconds for a reasonable-sized scene description, which turns out to be the case for the examples that Asgard has been used for until now.

Updating the graphical state when the database has changed is also fairly fast, at least for the initial conditions, as is updating the language representation when the database changes as a result of editing actions in other views. Updating the full motion of objects over all time is the one type of activity that is potentially very time-consuming, and for this reason it is only performed when the "Simulate" button on the control panel is pressed. This aspect of the system is described further in Chapter 5.

4.4 Previous multiple representation work

To better understand the philosophy of the Asgard editing system, one should consider similar editors for other types of documents. The multiple-representation paradigm was first described by Chen and Harrison [30]. Since then, a number of systems have been designed that allow textual documents to be created using both direct manipulation of a facsimile of the final typeset output and language-based editing of a formal input language, some of which are described below. Work has also been done in multiple-representation of graphics, user interfaces, and musical notation. To my knowledge there has been no work before Asgard in multiple representation animation editing.

4.4.1 Vortex

The Vortex system [29] provides two types of views onto documents written using the TeX language [68] – a formatted view, which is the output of an incremental formatting program based on TeX [87], and a language view that can be edited using a traditional text editor. Whenever the user wants to update the formatted view after editing the language view, he can give a command that causes the formatter to compare the differences between the old version of the input file and the new one. It then regenerates pages that have changed, detecting when no further changes in the output are possible as a result of changes in the input.

A number of lessons were learned from the design and implementation of Vortex. The major problem encountered was the difficulty of analyzing descriptions written in TeX. Since TeX is a macro-based language, it is very hard to determine what changes in the output a given input change will induce. A functional document description language with no side-effects or global variables would be ideal for such analysis, these are hard to design and few exist [57]. Also, for interactive usage, a tightly coupled system is preferable to a looser collection of cooperating programs, since there is a great deal of shared state that must be maintained between the pieces of the system.

4.4.2 Lilac

The Lilac editor [22] is another multiple-view document editor, which is based on the ideas used in Vortex. It provides two windows – a direct manipulation view similar to that of FrameMaker [46], but with a stronger model of structured editing, and a language window, which displays a formal description of the document structure and content. The language view may be edited in a very constrained way, using the same sorts of commands that are used for structured editing in the direct manipulation view.

Brooks notes that after using Lilac for some time, one finds that the language view is seldom if ever used. This is partly a result of the power of a good direct manipulation editor – most users of good editors like FrameMaker don't seem to miss having a language view.

Another factor may be the quality of the language description – Lilac's language seems to be a literal representation of the internal data structures, and certainly not as editable as LaTeX, for instance. Also, the usefulness of a language view is strongly dependent on the nature of the medium – the structure of text is a lot easier to infer from its appearance than the structure of an animated scene, input of textual information in a direct manner is much more straightforward than most of the elements of an animation description (or perhaps we have just been thinking about the problem longer).

4.4.3 Pan

The Pan system [8, 9] is a language editor that recognizes syntax and semantics, and performs incremental parsing and consistency maintenance. Pan makes it possible to edit programs and formal descriptions in a variety of languages at the same time. The syntax of a formal language is described using Ladle [24], and the semantics is provided using a description written in a logic programming language called Colander [8]. Internally, Pan maintains a number of representations of the document, which include an abstract syntax tree representation and a textual stream with references back to the syntax tree. The user is able to edit the text stream using familiar text-editing commands, but can also use structure-editing commands that refer to statements, expressions, and so forth.

4.4.4 User interface construction tools

Many direct-manipulation user interface mechanisms have been developed in the last few years, for example XF [39] and NextStep [49]. Avrahami et al. [5] describe a system that allows this kind of editing, along with language-based manipulation of the user interface description. The direct manipulation view is not precisely the final form, since hints are given to the designer about the nature of the widgets and the layout, so a third view is provided that is not editable and shows the final interface exactly as it will appear to the user.

Multiple-representation editing is especially useful for user interfaces, because most of the work in designing an interface consists of formally specifying the behavior of the interface objects, but one wants to use a direct-manipulation editor to describe the appearance of the interface. It is equally inappropriate first to design an interface graphically and then edit the generated code to put behavior in, and to program every aspect of the appearance of an interface in a non-graphical programming language with a long compile-debug cycle, such as Motif.

4.4.5 Graphics

A good example of multiple-representation graphics editing is ThingLab [20], a system that uses constraints for enforcing relations between elements in pictures. For example, one might define a square using a constraint that the sides of a figure must have the same length, and the angles between them must be 90°. Then whenever one side of the square is moved, the other sides move appropriately to maintain the constraints. This example is underconstrained, since many changes could satisfy it, but the system only guarantees a correct solution, rather than the "best" one.

ThingLab is written in SmallTalk, and constraints are described textually in a SmallTalk-like format. Objects are defined in this language, and they are also displayed graphically. The user has the option of editing the language description or the graphical view, but some operations, such as constraint definition, can only be performed textually. For example, constraints cannot be defined graphically but must be described textually. This is similar to the restriction in Asgard that force expressions can only be edited in a language viewer.

Another system that combined direct-manipulation graphical editing with language-based editing is Juno [82]. This system, like ThingLab and Sketchpad [112] before it, uses algebraic constraints to express geometric relationships. As the graphical representation is edited, the language description is also updated, but unlike Asgard, the language used is declarative rather than procedural, using *guarded commands* to express relationships.

4.4.6 Music editing

Multiple representation editing has also been used for the creation of musical scores. One such system is Arugula [80]. It provides specialized editors for different parts of the musical editing process, such as the design of tuning systems. Among the mechanisms it uses is a constraint solution system that was part of an earlier version of Ensemble [31]. Rhythm is also specifiable using an editor which allows adaptive subdivision of time intervals.

Music is also a good candidate for multiple-representation editing, since the usual representations used while a piece is being created are graphical and very formal, although the final output is a different medium altogether: sound. Because any visual or textual representation of music must be a significant abstraction, one might conclude that different abstractions probably work better for different tasks, which can be seen from the wide variety of musical notations that have been used in the past. Allowing a composer to use multiple different representations simultaneously is a major advantage.

The principle that the further the representations of a piece are from the final form, the more useful multiple-representation editing becomes, is applicable to other non-graphical media such as dance [25], which has a great deal in common with both music and animation.

Chapter 5

Motion simulation

The core of any animation system is the motion simulation algorithm that it uses to transform the user's specifications into detailed motion paths for the objects in a scene. For keyframe animation systems, where the position and other state variables are given for particular points in time, this algorithm must interpolate between these points to determine the full paths. For physically-based animation systems, however, motion simulation is much more involved, since the movement of the objects is a result of the physical laws that apply to the scene.

The physical model that Asgard presents to the user is described in Chapter 3. This chapter will first describe the primitives that are used by the motion simulation code and how the user-level elements are mapped onto it. Then the manner in which the resulting system of differential equations is handled will be discussed, with special attention to the problems specific to interactive animation and how Asgard addresses them. Finally, a description will be given of other alternatives for motion simulation, and they will be compared to Asgard's approach.

5.1 Problem formulation and primitives

The model of the animated scene that Asgard presents to the user is in terms of high level concepts: rigid bodies with user-specified shapes, forces, links, and constraints. At the lowest level, however, the primitives are differential equations and piecewise-linear functions of time, and the high-level concepts must be mapped into these primitives. This section describes how this mapping is performed.

5.1.1 Objects and forces

Articulated bodies are composed of *components*, which are rigid bodies, connected by links. A link can be thought of as a connection between a point on one component and a point on another. For the rest of this chapter, the term "object" will be used to refer to these components, since internally all the components of a body are treated as separate entities, except for the purpose of collision detection.

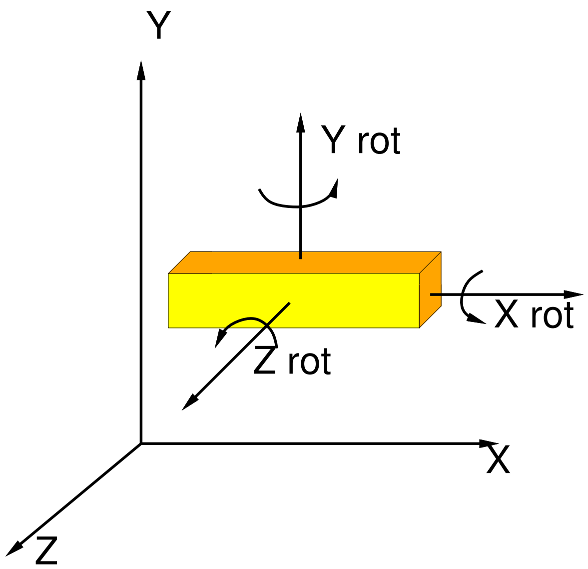


Figure 5.1: Degrees of Freedom of a Rigid Body

Each object has six degrees of freedom – three of position and three of rotation, about each coordinate axis, as shown in Figure 5.1. These degrees of freedom are determined by second order ordinary differential equations, according to Newton's Second Law of Motion: $\vec{F} = m\vec{a}$. For the positional components, one simply adds up the forces, in each of the X-, Y-, and Z-directions, and divides by the mass to get the acceleration in that direction. For the rotational components, one must add up the torques about each axis and divide by the appropriate component of the moment of inertia to get the rotational acceleration.

In the case of linear forces that are applied to points in the object that are not the center of mass, a torque is induced. This torque is given by the expression $\vec{F} \times \vec{d}$, where $\vec{F}$ is the force vector and $\vec{d}$ is the vector from the center of mass to the point of application. The direction of this vector is the axis about which the torque is applied and the magnitude is the magnitude of the torque. It is straightforward to extract the components about the coordinate axes from this formulation. A configuration with a number of forces, including links that induce forces away from the center of mass, is shown in Figure 5.2. In this diagram, there are two links and gravitational forces on both objects. The forces and torques that act on the objects are illustrated.

Also, torques that are applied to axes that do not pass through the center of mass of the object give rise to linear forces. If a torque $\vec{T}$ is applied about an axis $\vec{A}$, which has displacement $\vec{d}$ from the center of mass (that is, if the center of mass is $\vec{M}$ and the point on $\vec{A}$ that is nearest to $\vec{M}$ is $\vec{P}$, then $\vec{d} = \vec{P} - \vec{M}$), then the linear force induced by this torque is $\vec{F} = \vec{d} \times \vec{T}$. Figure 5.3 illustrates both this case and the off-center linear force case. Asgard automatically computes the correct forces and torques for these situations, although the off-center torque case is rare.

Instead of trying to solve a system of second order equations directly, we turn each

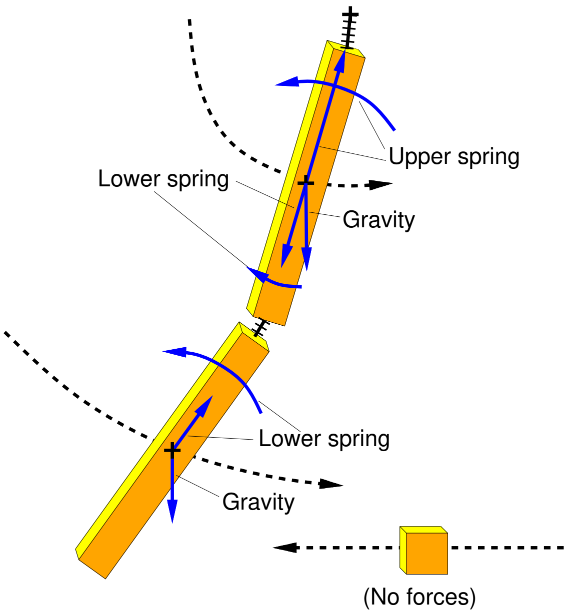


Figure 5.2: Configuration of Objects and Forces

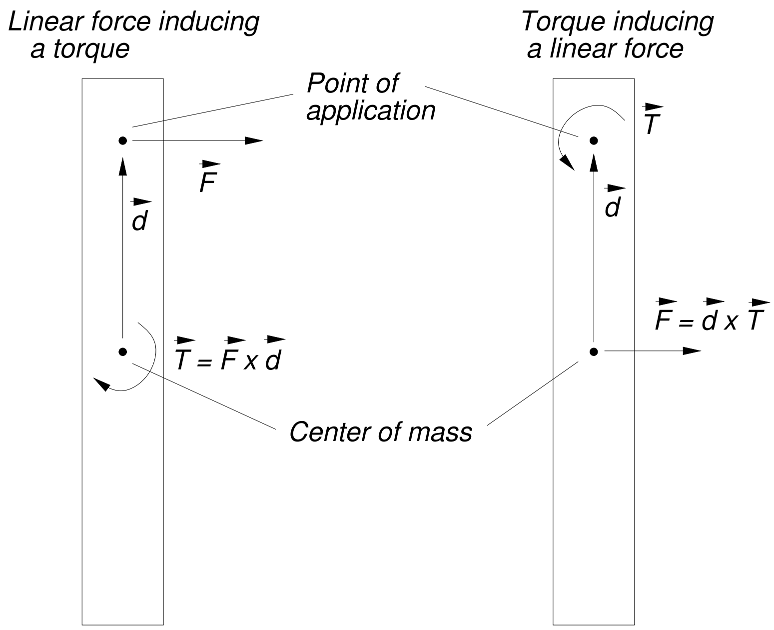


Figure 5.3: Off-center Forces and Torques

equation into a pair of first order equations. For every equation of the form

$$\frac{d^2x_i}{dt^2} = F(t, x_0, x_1, \ldots, x_n, \frac{dx_0}{dt}, \frac{dx_1}{dt}, \ldots, \frac{dx_n}{dt})$$

where the x_i are all the state variables in the system of equations, we can write

$$\frac{dx_i'}{dt} = F(t, x_0, x_1, \ldots, x_n, x_0', x_1', \ldots, x_n')$$

$$\frac{dx_i}{dt} = x_i'$$

with the variables x_i' representing the linear and angular velocities of the objects. There are two reasons for doing this. First, algorithms for numerically integrating first order systems of equations are better known and simpler that those for higher-order equations. Second, the initial conditions and trajectories that the user can specify for objects are position, orientation, linear velocity, and rotational velocity, which correspond to the first order equations. The transformation from second to first order equations is performed symbolically by Asgard during the preprocessing phase of motion simulation.

5.1.2 Links

Links can be defined either between points on two objects or between a point on an object and a point in space. The link constrains the points to be coincident, or more accurately, exerts a force that pulls the points together when they get too far apart. There are many ways to implement such constraints. One class of of constraints is "hard", in the sense of not allowing any violation whatsoever. Some of these techniques are described in Chapter 2. Asgard uses "soft" constraints, which are fairly simple and flexible, and serve most animation applications well enough.

Links are implemented by springs that are critically damped: that is, there is a friction term calculated to maximize the speed that the spring returns to its rest position without oscillating. This term is computed using the mass of the object being attached. It turns out that in many cases the effective mass will be different because other objects or forces may act on the object, and this will cause the spring to be underdamped, but as long as there is some damping term, numerical instability will not result. Other types of soft constraints could easily have been implemented, but there did not seem to be a need for them, and in any case the animator could easily define explicit forces to obtain them if necessary.

Each link is implemented by either one or two forces, depending on whether it is object-to-point or object-to-object. They are given by $\vec{F} = -k\vec{x} + 2\sqrt{k * m}\frac{d\vec{x}}{dt}$, where $\vec{F}$ is the restoring force, k is the "tightness" value, which can be given by the user as a parameter to the link statement, $\vec{x}$ is the difference between the two points, and $2\sqrt{k * m}$ is the damping term. The sign of $\vec{x}$ is defined so that the force on each object will pull it towards the point on the other object, or towards the point in space. This $\vec{x}$ term can be very complicated – the coordinates of a point on an object as a function of time depend on the position and rotation in a non-linear way, since there are transformation matrices involved that contain

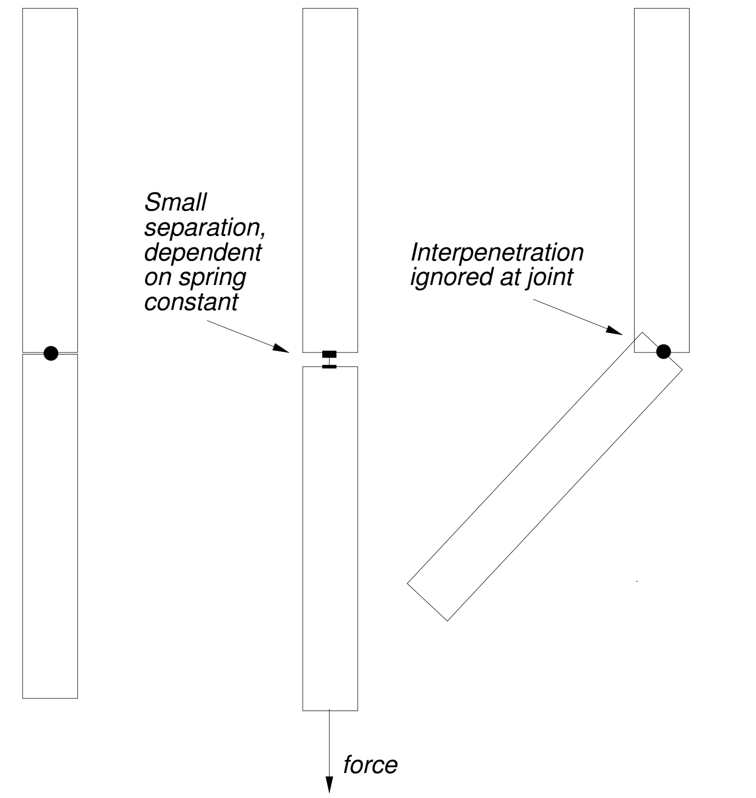


Figure 5.4: Joints Between Components

sine and cosine terms. Also, in the case of object-to-point constraints, the point can be an arbitrary expression or a trajectory.

Given a link definition, Asgard computes the forces that implement the link symbolically, and then adds them to the force lists of the appropriate objects. These forces are generally not applied to the centers of mass of the objects, so torques are also created. For a moderately complex object that has five or six links attached to it, the force expressions can be several pages long, and compiling several such expressions takes a few minutes on a Sparc2 workstation. However, once they are compiled and loaded, the evaluation overhead becomes very reasonable. This compilation is only necessary when the forces themselves change, so when other objects are edited in the scene, or modifications are made to the relevant objects that do not invalidate the force expressions, no preprocessing overhead is required when the scene is incrementally resimulated.

Figure 5.4 shows two components of a single articulated body, under three conditions. At rest, the ends of the joints are coincident, since the rest length of the link spring is zero. If a strong force is applied to one or both components, they separate a small but sometimes noticeable amount, which depends on the tightness of the spring. If the joint bends, interpenetration of the components around the point of the link connection is allowed, since the collision detection algorithm treats these components as one object.

5.1.3 State variables and trajectories

There are two places where Asgard needs to represent spatial values, either positions or velocities, as functions of time. These are the representation of the state variables of objects, and the representation of trajectories.

For state variables, Asgard uses a piecewise-linear representation – points that are computed by the solution mechanism are stored for each time, and values are linearly interpolated in between. The reason that linear interpolation is used, as opposed to splines or other polynomials, is that the integration method uses linear extrapolation to compute the values of variables at each time step, using their values at the previous step. This method is currently trapezoidal integration, but many other methods also treat the functions as piecewise-linear, and calculate error tolerances based on this approximation. If we used a different scheme for playing back the motion, we would have no guarantee that the visible error would be within the tolerance used by the solution algorithm – for example, if spline curves were used for playback while piecewise linear functions were assumed by the integration module when computing the numerical error, the displayed motion would not be the same as the computed motion.

State variables are represented as lists of floating-point values. A time scale is attached to each variable, which makes it possible to look up values by time and to interpolate values that are between time points. This is important because in order to calculate the value of one variable at time t, we may need to get the value at t of a different variable that doesn't share the same time scale.

For trajectories, we use Bézier curves. The important consideration here is that the curves be easy for users to edit. Bézier curves are flexible and familiar to most users, and are in common use in many graphics utilities. The implementation of trajectories is hidden from the rest of the system, as is that of state variables – the primary interface in both cases takes a time value and returns the function value at that point.

Attaching trajectories to objects is a bit tricky, since a number of trajectories may control the same state value over different intervals of time, and there may be intervals where no trajectory is active. The motion simulation algorithm does two things to deal with trajectories. First, it ignores the computed values for the constrained state variables and instead uses the values determined by the trajectories whenever they are required. Second, it avoids taking steps past points in time where new trajectories come into effect. When such a point is encountered, the system must reset the simulated time to that point and then continue, using the trajectory value rather than the computed value.

Figure 5.5 shows an example of this process in one dimension: the X axis is time and the Y axis is the spatial coordinate. The solid curve is the trajectory, which is defined for a limited interval, and the dotted curve is the path of the object as seen by the differential equation solver. The line at the bottom shows the timesteps that the solver would like to take. Since the trajectory ends at t'_2, it takes a step from t_1 to t'_2 instead of to t_2, and continues from there. After t'_2, the value is determined by the physical laws, and the velocity is set to the derivative of the trajectory at t'_2, so the coordinate continues to increase linearly – in this example there are no forces acting on the object. When the time reaches t'_4, another trajectory comes into effect, and the timestep is reset, similarly to before. There is a discontinuity at this point,

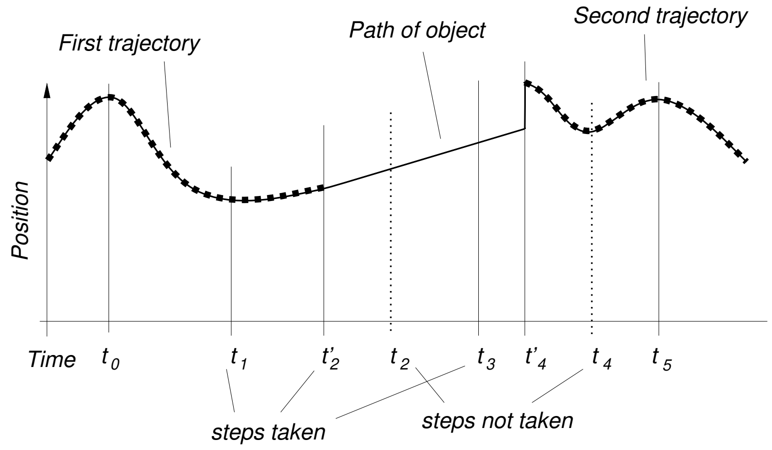


Figure 5.5: Dealing with Trajectories during motion simulation

but this is unavoidable unless we somehow look ahead and perform a smooth transition. This type of trajectory blending is done in some of the more powerful kinematics-based systems [96], but was not implemented in Asgard – since dynamics is available, a more complex trajectory that required such blending could be defined by attaching the trajectory to the endpoint of a link rather than to the position directly.

Trajectories can also be used as values in expressions. When an expression containing a trajectory is evaluated at a time outside of the interval over which the trajectory is defined, its value is linearly extrapolated from the closest endpoint and the derivative of the curve at that point. Other types of extrapolation are possible but linear seems to be the most straightforward.

5.2 Differential equation solution

The basic operation of the motion simulation process consists of numerically integrating the state equations, which determine the values of the state variables, over the time period of interest. These equations may contain references to any state variables of objects in the scene, and also to the independent time variable t. They may also contain trajectory references, which are treated as "black box" functions of time.

The techniques available for solving systems of ordinary differential equations, or ODE's, can be classified as either *explicit* or *implicit* [95]. An explicit method uses only the values at the current time point to extrapolate the values at the next time point. Examples of explicit methods are Forward Euler or Runge-Kutta. Implicit methods, on the other hand, solve expressions that contain the values at both the current and the next time points; they are implicit in that one cannot solve for one unknown at a time in terms of known values. One must perform a matrix inversion to solve for all the new values at once. Examples of implicit methods are Backward Euler and Trapezoidal.

In physics applications and motion simulation, explicit methods such as Runge-Kutta are common, whereas in circuit simulation, implicit ones such as the trapezoidal method predominate. Asgard uses the latter, because of its better stability and because it requires fewer evaluations of the functions per timestep – an Nth-order Runge-Kutta method requires N times as many evaluations as does the trapezoidal method, with the most common and best value of N being four [95]. This is important when the equations are fairly complex and difficult to evaluate. Runge-Kutta has the advantage of not requiring a matrix inversion, but as we shall see, the decomposition algorithm used by Asgard makes this less of a concern. To a first approximation, however, the choice of integration method does not affect the algorithms described below.

Another distinction that can be made between ODE techniques is that between *direct* methods and *relaxation-based* methods. To understand this difference, one must consider the basic structure of an ODE solution algorithm.

A direct method can be written as a set of three nested loops:

```
integrate time from start to finish
    linearize system at the current time
        solve the matrix obtained by the linearization
```

The integration is performed using an implicit or explicit method, as described above. The linearization is done by an approximation method such as Newton-Raphson iteration – this is what gives rise to the piecewise-linear nature of the computed trajectory. Methods also exist that make polynomial approximations to the curve, using mechanisms similar to Newton-Raphson. The matrix solution is required because the linearization produces a system of simultaneous linear equations: the variables are the values of the functions at the next time point. If the integration method is explicit, then every variable is a function of known values, at the previous time point, so no matrix inversion is necessary. In this case only two loops are required.

A relaxation-based method, on the other hand, performs the operations of integration, linearization, and matrix inversion somewhat differently. For example, waveform relaxation [72] can be written as:

```
repeat until all functions of time converge
    integrate each function from start to finish
```

The effect of this algorithm is to bring the iterative approximation outside of the integration and integrate each function independently, using a separate time scale. This improves performance for the same reason that the partitioning scheme described below does: each function is integrated independently and not constrained by the time steps required by the others.

Asgard uses the direct method approach, and does integration using the trapezoidal method. The value of the state vector $\vec{S}$ at time step n is related to its value at step $n-1$ by the equation

$$\frac{\vec{S}_n - \vec{S}_{n-1}}{h} = \frac{\vec{F}(S_n) + \vec{F}(S_{n-1})}{2},$$

where $\vec{F}$ is the state equation and h is the timestep, i.e., $t_n - t_{n-1}$. This is illustrated in Figure 5.6: the shaded area represents the contribution to the integral of the region of the function between times t_1 and t_2.

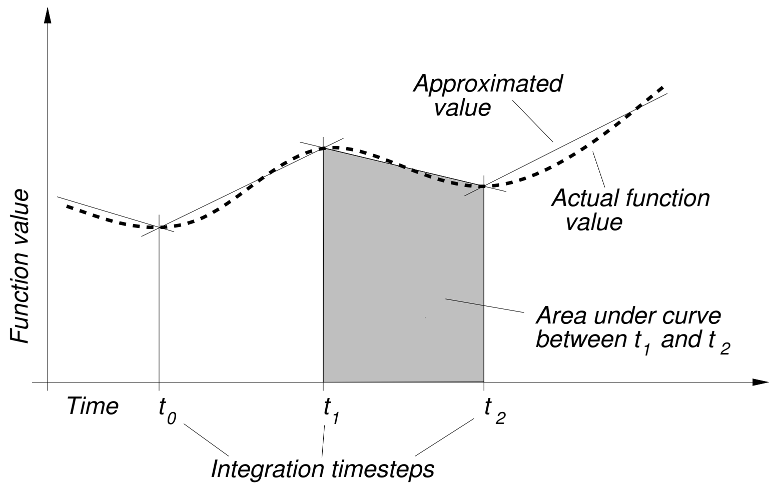


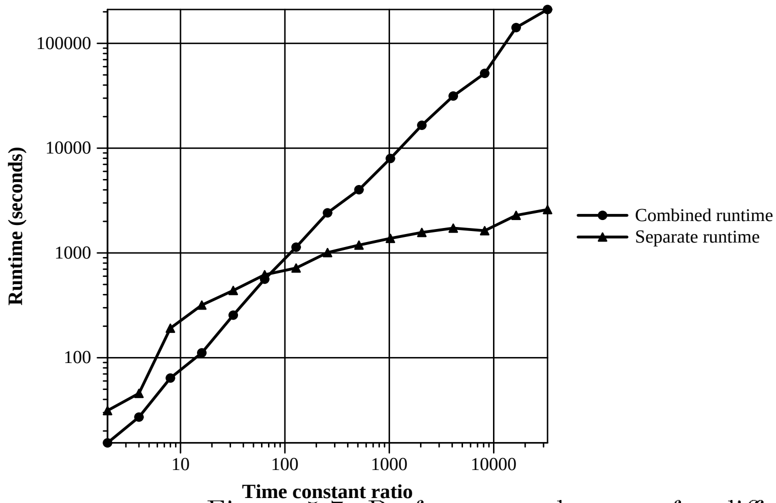
Figure 5.6: Trapezoidal Integration

In order to determine the state values at time t_n, we must solve the equation given above. It is generally non-linear, so Newton-Raphson iteration is used to find the solution, at each step approximating the equations by a line and calculating the X-intercept point. If the starting point is too far from the final solution, this process may not converge. The value from the previous timestep provides an initial guess, so if the timestep is not too big and the function is well-behaved (the derivative is continuous and bounded), we will obtain an accurate result. A more detailed discussion of this issue is given in Press et al. [95].

In order to obtain good performance, we use *adaptive timestep control* [95, 56]. If the iteration at a particular timestep fails to converge, we decrease the value of h, which means that we take a shorter step, and try again. This is done because in a shorter time period it is likely that the shape of the function over that interval will probably be more nearly linear, and will thus be more likely to converge. Also, if the system takes a large number of iterations, we decrease h for the next step, making the assumption that the variables are changing rapidly and thus require shorter steps to track them accurately. On the other hand, if it converges in a small number of iterations, we can increase h, so that we will finish the integration with fewer steps. This technique is essential for applications where the simulation routine does not know *a priori* what the numerical characteristics of the system will be, and does not want to assume the worst case. There is no good rule for calculating how much to increase of decrease h by, at least for trapezoidal integration, so we use a factor of 2 for simplicity.

Since $\vec{F}$ is a vector function, solving the linearized version involves inverting a matrix, or the equivalent, LU-decomposition[70]. This matrix has one row and one column for each equation in the system, and there are twelve equations per object component. This is an $O(n^3)$ operation, although with the use of sparse matrices its complexity can be reduced to approximately $O(n^{1.6})$ [70].

This solution process is reasonably robust and efficient for many applications. However, there are two sources of inefficiency when there are many bodies to simulate. The first is that different parts of the system may require different timesteps at different points in the



Problem Size	Combined Runtime	Separate Runtime
1	9.8	28.1
2	15.4	31.3
4	27.13	45.69
8	63.98	191.33
16	111.35	318.1
32	254.8	438.58
64	561.45	619.4
128	1135.23	717.97
256	2413.65	1008.09
512	4012.84	1188.08
1024	7973	1377.82
2048	16533.41	1569
4096	31442.86	1724.75
8192	51723.62	1630.72
16384	141446.5	2285.36
32768	210514.11	2594.32

Figure 5.7: Performance decrease for different time constants

integration, but since all the state variables are being calculated at once, we must take the smallest timestep required by any variable and use it for the entire system.

Figure 5.7 shows what happens when several bodies are simulated together, and one requires a much smaller timestep than the others. In each trial, several pendulums were simulated for 10 seconds, but one was given a much larger "gravity" value to make it swing faster and thus require a smaller timestep. The time it takes to make one oscillation is called the *time constant*, and is proportional to the required value of the timestep h. The runtime is plotted against the ratio of this small timestep to the one required by the rest. The upper graph shows performance when the objects are simulated together, and the lower one shows the sum of their runtimes when they are simulated separately. As one would expect, the runtime of the first case is proportional to the product of the size of the system and the fastest time constant of any equation, but in the second case it is proportional to the sum of these values.

For small time constant ratios, it is a bit faster to simulate all the objects together because of the lower overhead. However, as the ratio between the desired timesteps increases, one would like to be able to simulate them separately. In the current implementation the crossover point is about a factor of 100, but this could probably be lowered by careful optimization of the code.

The second problem with using implicit methods is that putting all the equations together in one system, building an $n \times n$ matrix, where n is proportional to the number of objects in the scene, and then inverting this matrix does not scale well. In general, the matrix could be re-ordered so as to be block-diagonal, and each block could be solved separately, but the technique described below obtains this effect more conveniently.

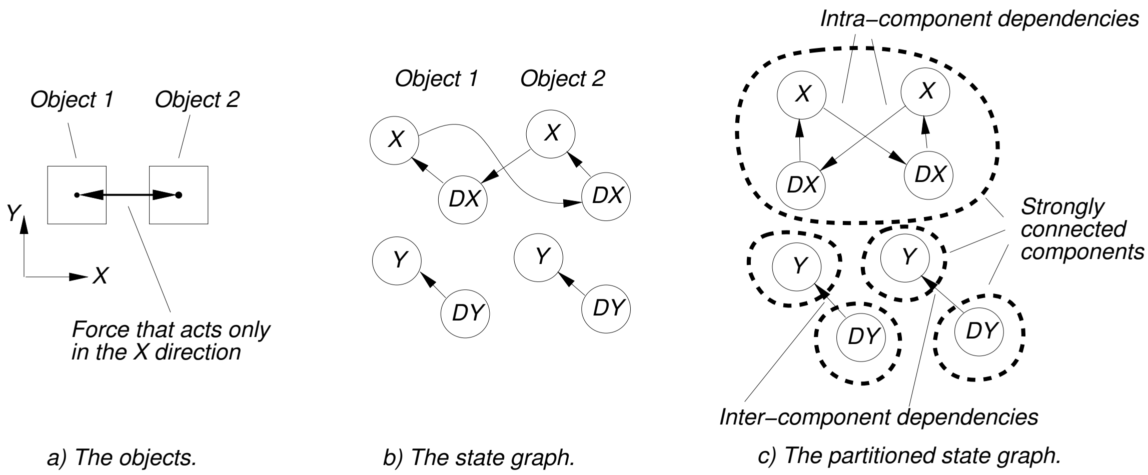


Figure 5.8: Example of a State Variable Reference Graph

5.2.1 Partitioning

The most obvious way to deal with the two problems stated above, the inflexibility of timesteps and the more than linear growth in matrix inversion complexity, is to somehow partition the system of equations into components that can be solved independently, in a particular partial order. A number of circuit simulation systems, such as Splice [100], perform this kind of partitioning.

When separating the system into components, there are two conditions we must satisfy. First, equations that depend on one another should be in the same component. Second, equations that do not participate in a circular dependency should be in separate components.

If we consider the system to be a directed graph, with the nodes being equations and the arcs being state variable references, what we want to do is identify the strongly connected components – that is, the segments of the graph with circular dependencies. Asgard uses Tarjan's algorithm [102] to do this. We can then identify the dependencies remaining between the components, which will form a directed acyclic graph. An example of such a graph is shown in Figure 5.8. Part (a) shows a very simple model: two objects with a mutually attractive force that acts only in the X direction, depends on distance, and is applied to the centers of mass. Part (b) shows a section of the state graph. The arrows point in the direction of data flow: since the value of the dx variable is used in computing the value of x, the arrow points from dx to x. Since the forces in the X direction depend on the position of the other object, there is a cycle in that part of the state graph. The Y components do not depend on each other. The rest of the graph, which includes the Z components and the rotational components, is similar to the Y section. Part (c) of the figure shows the partitioned state graph. Intra-component dependencies do not concern us at this point, but the inter-component dependencies induce a partial ordering on the components, which constrains the order in which we can solve them.

Relative to the other phases of motion simulation, building the state graph is very efficient, and is of complexity linear in the number of terms in the equations.

A body segment that is not connected to any other by forces or links will yield twelve

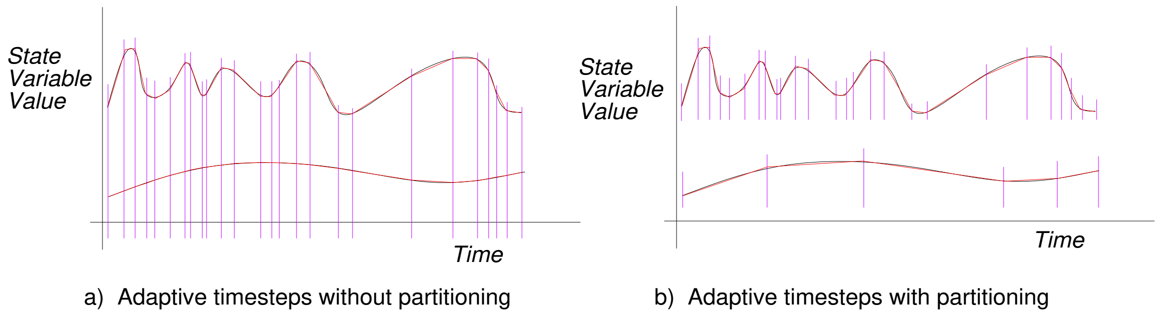


Figure 5.9: Integration Strategy

components, one for each state equation. However, a body with more than one connected segment will in general produce one large component, because the forces and torques induced by the off-center link forces generally involve all the state variables. If the link points coincide with any of the principle axes of inertia, there will be a few extra components, because of the simpler structure of the rotational state equations for those coordinates.

The solution process must be done in an order that is an extension of the partial ordering induced by the inter-component dependencies. If this were not the case, data would not be available for some components when they require it. Other than this requirement, the precise total order does not matter.

Each component has its own timestep, which can adapt to the particular characteristics of the equations therein. If there are k bodies, the runtime complexity decreases from $O(k^3)$ to $O(k)$, since the size of the components, and thus the matrices to be inverted, is constant. Clearly, such decomposition is essential if the solution process is to scale well. The "Separate" data in Figure 5.7 was actually obtained by partitioning the system of equations for the pendulums.

Figure 5.9 illustrates the advantage of this technique. In case (a), both functions are in one component and must take the same timesteps. In case (b), they are separate and can take the timesteps that are best for each one. The total runtime depends more than linearly on the number of times each function takes a step, so case (b) is clearly better.

5.2.2 Collision detection and response

In order to perform the partitioning process described above, one must know the structure of the equations of motion in advance. However, when collisions are possible, some interactions between bodies cannot be predicted until the actual motion is calculated. As soon as the motions of two objects become known over a particular time interval, Asgard checks to see if they have collided. If they have, transient reaction forces are introduced that push the bodies apart in the correct manner.

The problem here is that if we have partitioned the system and solved it in what was the correct dependency order with no collisions, a collision may invalidate earlier results. In general, there is no way to predict *a priori* which objects will collide before doing the simulation, so we cannot take this into account in the partitioning phase of the solution.

One possibility is to do the simulation as described above, and whenever a collision invalidates previously computed data, simply recompute it. This will yield correct results, but if the simulation interval is long and there are many collisions, this will result in a great deal of wasted computation.

5.2.3 Event-driven solution

Asgard handles the problem of collisions interfering with the partitioned solution process by changing the order in which the different components perform simulation timesteps. Instead of picking a total ordering and then solving the components entirely one after the other, it solves them in an interleaved fashion, with the goal of keeping the upper limit of the calculated time as synchronized as possible between the components.

Conceptually, we are allowed to solve the components in any order, subject to the following constraint: to solve component C_i over the time interval $[t_{is}, t_{if}]$, for every component C_j which C_i depends upon, C_j must have been solved at least to time t_{if}. That is, if one component depends on another one, the latter must have values defined for the entire interval we are going to calculate. Otherwise we would have to extrapolate from the data that is available, which will generally not be accurate.

Asgard is currently a sequential program, and the way it solves multiple systems of equations concurrently is by using a priority queue: it takes a component off the queue, performs one integration step, and then puts it back on the queue if it is not finished and if the components it depends on have been solved far enough to permit it to take another step.

The queue is ordered according to the last calculated time point of each component. At any given stage, it is possible that many components are in the queue, which means they can take a step. The maximum size of this next step is calculated at the end of the previous one, so it is a simple matter to decide which ones can proceed. We always pick the one that is furthest behind, in order to keep the entire system as well synchronized as possible without sacrificing performance.

Components are only added to the queue when they are eligible to take their next step. Whenever a component C has been simulated for one step, we check all the components C_i that depend on C and are not yet in the queue. Let t_f be the sum of the current ending time of C_i and its next timestep, which will be the ending time of C_i after it takes its next step. If all the components C_j that C_i depends on now have a current ending time that is at least t_n, then C_i is put onto the queue, since the data it requires is now available. We only check the C_i that directly depend on C, since these are the only ones which might be affected by a change in the state of C.

It would be fairly easy to parallelize this algorithm, since the partial ordering that provides the constraints on concurrent solution is usually far from being a total ordering. The degree of parallelism is quite dependent on the precise problem definition, but for typical scenes, hundreds of components could proceed in parallel, and the major bottleneck would probably be contention for the priority queue.

If a collision occurs at time t, we have to invalidate all results that were calculated based on the assumption that such a collision did not occur. This is done by looking at each component that contains state variables of either of the bodies that have collided, and if

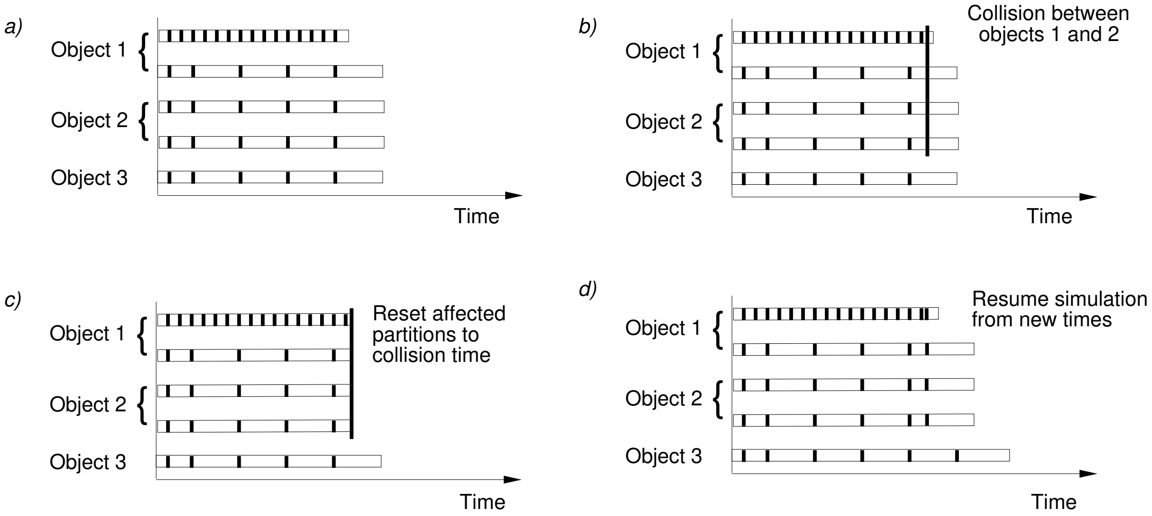


Figure 5.10: Progression of Partitioned Calculation with a Collision

that component has been simulated to a point beyond time t, backing up the ending time so that only the results up to t are retained. In certain cases we reset only some of the components, if the collision force is perpendicular to one or more of the coordinate axes.

Furthermore, for every component C that we back up, we must also back up all components that depend on C, and all components that contain state equations for objects that have collided with objects that have state equations in C after time t. It is only necessary to back up these components to the point of the first collision after t, however.

The basis task can therefore be described as follows: whenever data becomes invalid, track down all other data that were computed using it and invalidate that data also.

After a component has been backed up, we must also check to see if it is still eligible to remain in the queue. If it was backed up because of a collision that affected another component that it depended on, which was also backed up, then it will have to be removed from the queue, since data from the other component will no longer be valid for the interval it requires. On the other hand, if it was backed up because it participated in a collision, and it does not depend on another component that was also backed up, then it might remain in the queue.

An example of this process is shown in Figure 5.10. Three objects are simulated, and for the purposes of illustration we assume that objects 1 and 2 have two components, and object 3 has one. This could happen if the objects had more than one component, or were subject to certain kinds of frictional force. In addition, the first component of object 1 has much smaller timesteps than the other components. The tick marks on the bars represent timesteps, and the length of the bar represents the amount of time simulated for that component.

Part (a) shows the state of the simulation before a collision takes place. In part (b), the collision occurs. Note that we can only know about a collision when all components of the participating objects are calculated up to that point. Part (c) shows the components being reset to the collision time. In part (d), the simulation has begun again from the collision point.

A few points should be noted here: first, after the collision, the current time is reset to the collision time, and the timestep is reset to its initial, conservative value. This is because a discontinuity has been introduced in the function values, and we do not know whether we can continue taking a long time step. In either case, the system will automatically adapt to the new conditions as the simulation progresses. Second, the last object does not participate in the collision at all, so none of its data is lost.

The event-driven simulation algorithm can be more formally stated as follows:

```
Form equations from object descriptions and laws of physics
Partition system into connected components using Tarjan's algorithm
For each component $C$ that does not depend on another
   Insert $C$ into priority queue
While priority queue is not empty
   Remove component $C$ from the front of the queue
   Simulate $C$ for one timestep
   For each body segment $B_1$ that has a state variable in $C$
      Let $t_1$ be the minimum ending time for state variables of $B_1$
      For each body segment $B_2$ that might collide with $B_1$ (see below)
         Let $t_2$ be the minimum ending time for state variables of $B_2$
         Let $t_f$ be the last time collision detection was done between $B_1$ and $B_2$
         If $t_f < \min(t_1, t_2)$, then
            Check for collisions in the interval $[t_f, \min(t_1, t_2)]$
         If a collision has occurred at time $t_c$, then
            For each state variable $v$ of $B_1$ or $B_2$
               Let $C$ be the component that contains $v$
               Reset the ending time of $C$ to be $t_c$
            Modify the velocities of $B_1$ and $B_2$ appropriately
   For each component $C_i$ that depends on $C$, and also for $C_i = C$
      Let $t_i$ be the end of the next step $C_i$ will take
      If all components $C_j$ that $C_i$ depends on have an ending time $>= t_i$,
            and $C_i$ is not in the queue,
            and $C_i$ has not reached the end of the simulation, then
         Add $C_i$ to the queue
```

When forming the set of body segments that might collide with a given segment, one excludes segments that are part of the same articulated body. Also, one might use spatial subdivision techniques such as quad-trees to further limit the search for collisions, although this has not been implemented in Asgard.

The operation "reset the ending time of C to be t_c" can be more formally described as follows:

```
Truncate the state variables in $C$ so that their last time value is at $t_c$,
      interpolating their values if necessary
For each component $C_i$ that depends on $C$
   Recursively reset the ending time of $C_i$ to be $t_c$
If $C$ is in the queue, and for some $C_i$ that $C$ depends on, the ending time of $C_i$ is now
```

```
        less than the next ending time of C, then
    Remove C from the queue
```

Of course, when we recursively reset ending times of components, we are careful to reset each component only once for a given collision.

Remember that in the above discussion, a partition can depend on another in two ways: via a state equation dependency, or via a collision involving objects in the components that has taken place during the interval of interest. In both cases, dependence is transitive, which is why the backing-up operation is recursive.

5.2.4 Performance

To test the performance of these three simulation techniques, we created a simple test scene that consists of two pendulums and a ball that collides with one pendulum twice and the other once. We ran this example in several ways.

1. Collisions are ignored, and a single component is used.

2. Collisions are ignored, and multiple components are used.

3. Collisions take place, and a single component is used.

4. Collisions take place, multiple components are used, and event-driven scheduling is done.

5. Collisions take place, multiple components are used, but no event-driven scheduling is done. That is, after a collision, each component is run to completion before the others are dealt with.

In addition, we created a varying number of additional objects elsewhere in the scene which serve to increase the size of the problem, but which happen not to participate in any collisions. In a realistic scene there might be a large number of such objects. Figure 5.11 shows the runtime for the five algorithms, plotted on a logarithmic scale as a function of the total number of objects in the scene.

The adaptive case clearly outperforms the single-system case, especially when collisions are taken into consideration. In the first two cases, the values are fairly similar, because the time constants are not significantly different for the different objects, as was the case in Figure 5.7. The rest of the cases take more time, partly because of the overhead of checking for collisions and partly because of the wasted computation that results from collisions invalidating previously computed data.

One would expect the last case to be significantly worse than the others, but how bad it is really depends on how the components affect one another as a result of collisions. In this example, since the majority of the objects aren't affected by the collisions, the wasted computation becomes less significant as the problem size increases.

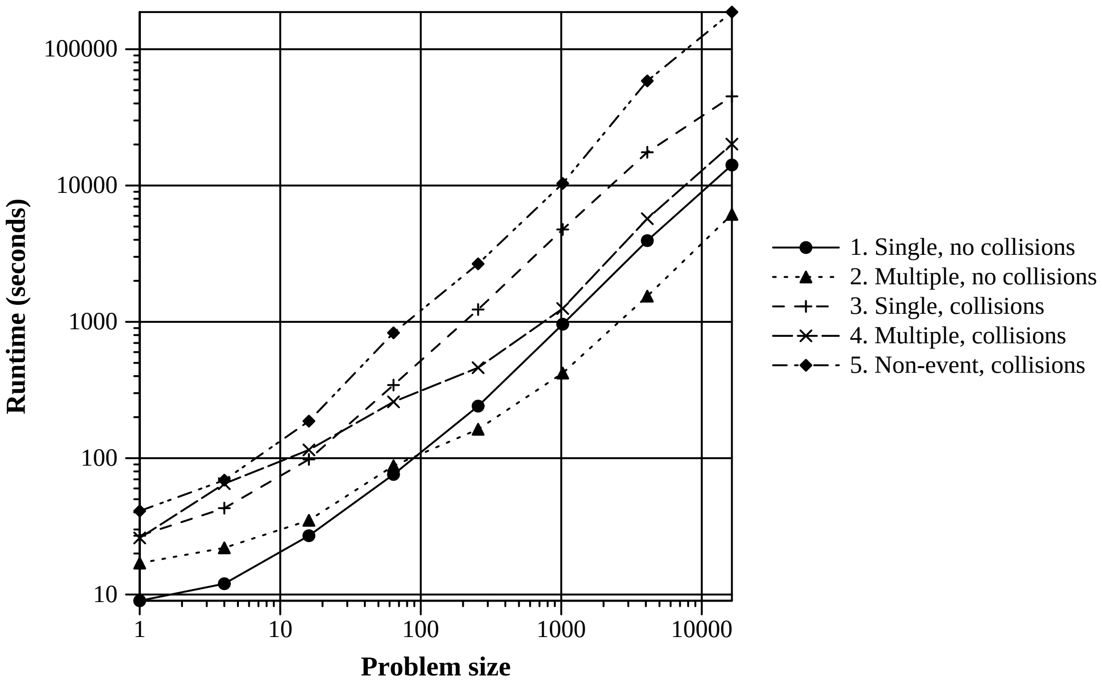


Figure 5.11: Graph of runtime for simulation algorithms

5.2.5 Interactive editing

In addition to correctly and efficiently handling collision response, the event-driven algorithm is ideal for interactive applications, where small changes are made to the scene description and fast update of the results is necessary. A typical change might be to alter one or more of the initial values for the state variables. Rather than invalidating the entire result set and resimulating, we can simply reset those partitions that contain the altered state variables, transitively reset the partitions that depend on them, via either the state equations or collisions that have taken place, and rerun the simulation algorithm, retaining the rest of the state variable trajectories. This has the effect of recomputing all data that is affected by the change in initial conditions, but no more. This is the same as introducing a previously unexpected collision at the start time.

5.3 Collision detection and response

Collision detection is the process of determining if and when a pair of objects collide, and the location of the first point of penetration. Collision response is the modification of the motion of these objects in a physically correct manner.

There has been much work on collision detection in recent years. Canny [27] and Moore [81] have both developed techniques for handling collisions between moving polyhedra. Baraff [16] and Pentland [90] have addressed this problem for curved surfaces, which are significantly harder to deal with than polyhedra, especially if they are concave and deformable. Typically, collision detection has taken up a large fraction of the time used for

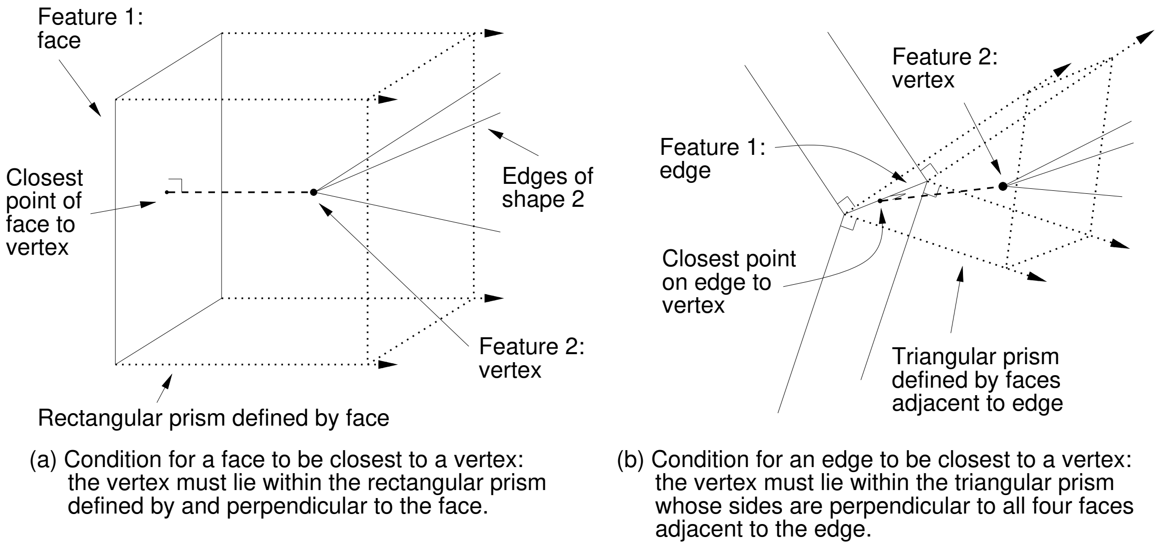


Figure 5.12: Closest feature conditions

motion simulation in animation systems.

5.3.1 The Lin-Canny algorithm

The basic collision detection algorithm that is used in Asgard was developed by Lin and Canny [75, 74], and was reimplemented as part of the Asgard geometry subsystem, since the original implementation was in Lisp. It is limited to convex polyhedra, which are one of the two basic shape types used in Asgard. A simple extension is used for dealing with the other shape type, which is spheres. One of the reasons for the limitation of the types of shapes allowed for objects in Asgard was that collision detection for non-convex polyhedra and curved objects is a very difficult problem [81, 12].

This algorithm is fairly simple, yet efficient – because of the aggressive use of prior results, the runtime is only proportional to the speed of rotation of the objects relative to one another. As a pre-processing step, Asgard first calculates the intervals of time during which the bounding spheres for the objects are intersecting, and only runs the full algorithm for these intervals. Spatial subdivision techniques such as oct-trees could also be used to filter out more collision checks, but this was not done in Asgard.

The Lin-Canny algorithm works by maintaining a record of the closest features between every pair of objects. Features can be vertices, edges, or faces. For each pair of feature types, there is a geometrical condition for a feature of this type on the first shape to be the nearest one of that shape to the other feature. Figure 5.12 gives a few examples of these conditions. There are a total of nine possibilities, but they are all similar to the ones pictured: each feature type defines an open-ended region of space, any point in which is closest to that feature.

The algorithm works by starting with a pair of initial features, one on each shape. It then performs the appropriate geometrical test to see if they are the nearest pair. If they are not, the test returns a pair of features that are closer. The pair is simple to determine as part of the test. The process now iterates, using the new pair. Since the distance always decreases, the iteration is bound to converge in time proportional to the sum of features on the two objects.

At each time point the collision detection is performed, the initial feature pair is taken to be the previous closest pair. If the objects are not rotating rapidly relatively to each other, the closest features will tend to remain the same, and thus only one test will need to be performed to compute them. The points at which we perform the test need not be the same as the numerical integration time points, and in fact, Asgard uses information about the speeds of the objects and their relative separation to determine lower bounds on the first intersection time – this can help to avoid a lot of unnecessary tests.

Spheres are handled by treating them as points for the purposes of spatial region testing, using the radius to determine the actual nearest point only after the other closest feature is established. This works because the other objects are guaranteed to be convex, and if a given point on such an object is the closest one to a point on the surface of a sphere, it will also be the closest one to the center.

It is straightforward to determine if the shapes are intersecting while the closest pair is being determined. If they are, then a collision has taken place somewhere between the last intersection check and the current one. Given the previous and current distances and times, we use a binary search to find the time at which the distance is zero, and the point of collision. This information is then given to the collision response routine.

Since a given shape may be a union of polyhedra and spheres, to determine if two shapes have intersected, we must test all pieces of the shape pairwise. In addition, since adjacent components of a single articulated body should not be considered to have collided if they interpenetrate, we do not perform these tests. The reason for this exception that in some cases we are modeling a complex joint such as an elbow with a simple point-to-point constraint between the ends of two rods. On the other hand, in some cases we may want objects to collide with themselves, as in the case of a chain that has many links. This is why the collision check is only skipped if the objects are directly linked.

5.3.2 Collision response

Once the point and time of collision has been determined, it is relatively straightforward to compute the correct reaction forces that push the objects apart in the correct way. Rather than determining the actual force values, Asgard takes the shortcut of directly computing the new velocities, and then substituting them for the old ones. This has a few advantages. First, if the forces were computed and applied over a finite duration, as actually happens with physical objects that are not perfectly non-deformable, we would have to consider what should be done when another collision happens during this interval. If the force is applied instantaneously, we can just handle all collisions that take place at once one after another, and the result comes out correctly even if one object collides with several others at the same time. Second, a very short and strong force would cause the timestep of the simulation to be reset to a very small value, which greatly decreases the simulation speed. Finally, the equations for reaction forces are actually in terms of momentum rather than acceleration, and it is simpler to avoid translating them into forces.

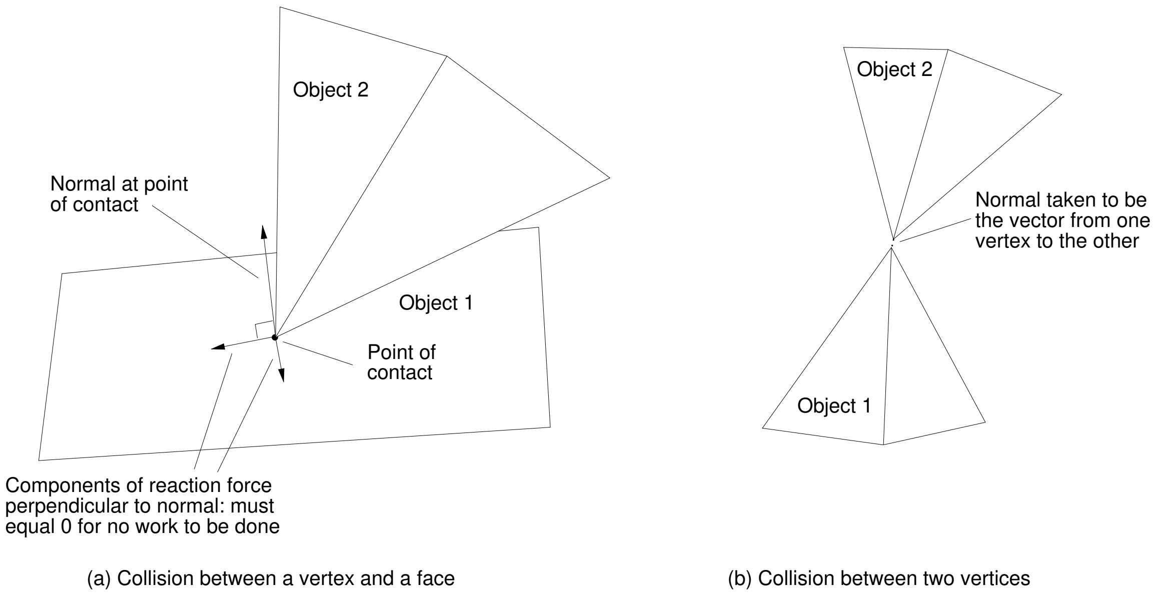


Figure 5.13: Determining the direction of the reaction force

Computing the correct velocities after a collision is an exercise in finding N equations that completely determine N unknowns. The unknowns in this case are the velocities after the collision, and there are 3 scalars for each object, which yield 6 total unknowns. The known quantities are the values of the momentum for the two objects before the collision, and the point of the collision. We also must know something about the geometry of the objects at that point, as we will see below.

There are two physical laws that must be satisfied by the calculation: the conservation of momentum and the conservation of energy. If the collision is fully elastic, then this energy is entirely kinetic energy. If it is not, then some proportion of the energy is dissipated as heat, and the conservation equation is written with a scaling factor to take this into account. Asgard treats all collisions as elastic, but this is not a significant issue. A full discussion of the physics involved can be found in any mechanics textbook, such as Goldstein [52]; what follows is an outline of the principles used in Asgard.

The three components of the conservation equation are as follows:

$$v_{1i}m_1 + v_{2i}m_2 = v'_{1i}m_1 + v'_{2i}m_2$$

where the v values are the velocities before the collision, the v' values are the velocities after, the m_j are the masses of the objects, and $i = x, y, z$. The equation for the conservation of kinetic energy is:

$$\frac{m_1}{2}|v_1|^2 + \frac{m_2}{2}|v_2|^2 = \frac{m_1}{2}|v'_1|^2 + \frac{m_2}{2}|v'_2|^2$$

Together these equations provide 4 out of the 6 necessary to determine the resultant motion. The other two equations must come from the geometry of the surfaces at the point of collision.

Because a reaction force must do no work, it follows that the direction of the force must be perpendicular to the normal of the surface at the collision point. Figure 5.13 illustrates this point. In case (a), the normal is defined by the normal of the face at the point of collision. If the collision were an edge-face or a face-face collision, the normal can be similarly defined with no ambiguity. In case (b), however, there is no surface normal, so we must pick something plausible, such as the vector from one vertex to the other immediately before the collision. This is not a physically possible case, since there is no such thing as an absolutely sharp point in nature, but it does come up sometimes in simulations. The third case is that of edge-edge collisions: in such a collision we use the plane defined by the two edges to obtain the normal, unless the edges are parallel, in which case we must use a mechanism similar to the vertex-vertex case to find one of the perpendicular directions.

Once we have a normal vector, we can write down two more equations. The dot product of the reaction force with each of the perpendicular vectors, as shown in Figure 5.13(a), must be zero. Given six equations in six unknowns, it is straightforward to find the new velocity values. Except for the vertex-vertex and parallel edges cases, which are not physically realistic, the resulting motion will be correct.

Chapter 6

Integration with Ensemble

Asgard was developed as part of the Ensemble project. One of the goals of Ensemble is the seamless integration of different media, including dynamic media such as animation, for both viewing and editing purposes. Most of this dissertation has described Asgard as a stand-alone system, which is appropriate since its user interface is specialized for the editing of animation, rather than of general documents. This chapter briefly describes the Ensemble system, and then discusses in detail the interface between Asgard and Ensemble, both in terms of what was implemented and what could be built on top of the basic framework described here.

6.1 The Ensemble System

The Ensemble system is designed to allow the creation and editing of compound documents, including structured media such as programs and reports. It has a powerful presentation engine that can control the formatting and display of the document components in a variety of styles, and can support multiple views of a single document, with different sets of presentation attributes. The presentation system is used by the Asgard medium in a rather minimal way. From the point of view of the enclosing document, an Asgard subdocument is simply a rectangular region, with opaque contents and internal structure – everything other than placement and overall window size is managed by Asgard's presentation mechanisms.

Figure 6.1 shows the structure of an Ensemble document, including multiple presentations and views. A *document* is composed of a tree of *subdocuments*, each of a particular *medium*, such as "Text", "Graphics", or "Animation". These subdocuments may themselves be tree-structured, but this is not required — Asgard subdocuments are treated by Ensemble as single nodes. Each subdocument may have one or more *presentations*, which associate the structural representation in the document with a visual or otherwise concrete representation. This association may be mediated by a *presentation schema*, which is analogous to the "style sheets" used by other document systems.

Views correspond on a one-to-one basis to top-level windows on the screen. A view contains a representation of an entire document tree, with sections potentially elided. The hierarchical structure of the overall document is mirrored in the view, with subdocuments in the document corresponding to *renditions* in the view.

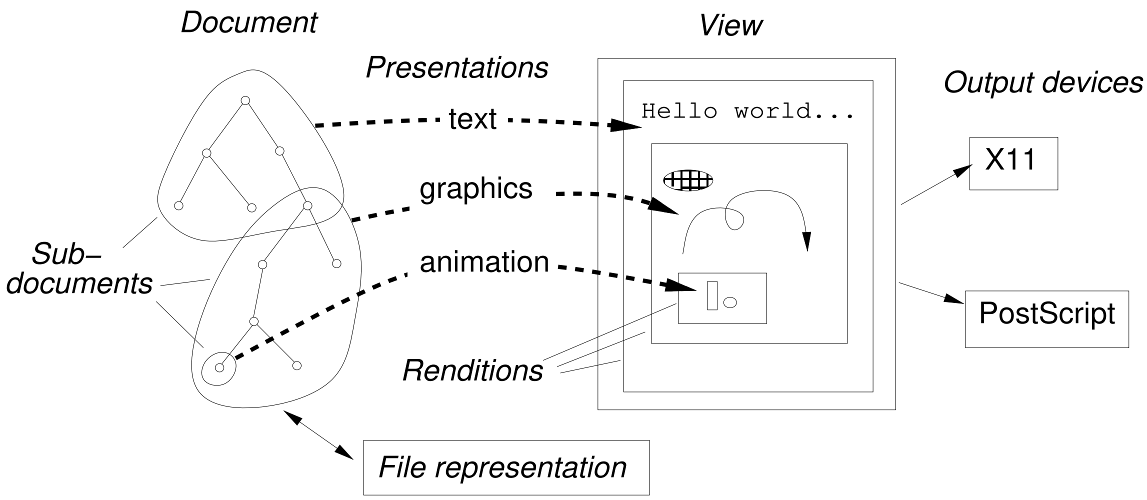


Figure 6.1: Document structure in Ensemble

Finally, documents can be represented in a persistent storage format, such as a UNIX file. Ensemble currently uses a file format similar to the Structured Graphical Markup Language (SGML) [51] to encode structured multimedia documents for storage and interchange. An alternative would be to use an object-oriented database such as Postgres [110] for this purpose.

The minimal set of facilities that a medium must provide in Ensemble is thus the following:

1. Routines to read and write the storage representation.

2. An implementation of subdocument nodes of the medium.

3. One or more presentations, which may employ presentation schemas to determine appearance.

4. An implementation of a rendition for each presentation type and for each output device.

Additionally, a medium may also provide user-interface functions that can be used in the editing or display of document components of that medium.

In the following section, the facilities that are provided for the integration of Asgard with Ensemble are described in more detail.

6.2 The Asgard Medium

When Asgard animations are embedded in Ensemble documents, the Asgard process is run as a slave to the Ensemble process, and handles all of the rendering and presentation, with the exception of the sizing and location of the animation window, and much of the user-interface functionality. The framework provided on the Ensemble side is thus fairly simple, and is also generic enough to be applicable to other loosely-coupled media that may be incorporated into Ensemble in the future. The components of this framework are as follows.

```
DOCUMENT CMEMO
<CMEMO>
<TO>{Whom it may concern}</TO>
<FROM>{Wayne}</FROM>
<SUBJECT>{Asgard within Ensemble}</SUBJECT>
<BODY>
<PARAGRAPH><DOCUMENT>
DOCUMENT ANIMATION
<ANIMATION>
<ASGARD>{
shape_set ss1 -desc \{-sub \{-pos \{0 0 0\} -sphere 0.5\}\}
body b2 -artbody 1 -shape ss1 -init_pos \{-1 0 0\} -init_dpos \{1 0 0\}
body b3 -artbody 1 -shape ss1 -init_pos \{1 0 0\} -init_dpos \{0 1 0\}
simulator -finish 3
}</ASGARD>
</ANIMATION>
</DOCUMENT></PARAGRAPH>
<PARAGRAPH><TEXT>{Thanks to Vance for putting this example together}</TEXT></PARAGRAPH>
</BODY>
</CMEMO>
```

Figure 6.2: A sample of an embedded Asgard scene in an Ensemble document

6.2.1 Storage representation management

Asgard subdocuments are represented in Ensemble document files by simply embedding the animation language description of the scene inside of special delimiters. Braces, which are used for grouping by the Asgard language, are escaped since they have special meaning to Ensemble. Figure 6.2 contains an example of a simple embedded Asgard scene within an Ensemble document. The tokens `<ASGARD>` and `</ASGARD>` delimit the Asgard input.

6.2.2 Subdocument representation

The Asgard interface is structured so that for every animation component inside of an Ensemble document, there is one Asgard process that handles the scene. Multiple renditions of this animation are managed by this process, rather than starting a separate process for each one, since Asgard has facilities for managing multiple views of a single scene. For this reason, the subdocument node is the locus of communication between Asgard and Ensemble.

When a new subdocument of the Asgard medium is created, a process is started and fed the animation description that was obtained from the Ensemble file. A UNIX pipe is set up to provide a bidirectional control channel between the processes. Currently, the only data that are sent from Asgard to Ensemble are window ID's for newly created animation renditions, as described below, but other data and commands could easily be sent if necessary.

6.2.3 Presentations and renditions

The only work that the Asgard presentation must perform is to inform the containing document node what size is desired for the animation window. Since Asgard has no built-in notion of what kind of viewport should be used to display scenes,the presentation arbitrarily selects a size for the window. A sophisticated layout presentation that manages the parent document of the Asgard window could then modify this size as necessary to accommodate the surrounding text and other media.

When an Asgard rendition is created for a new view of a document, it must send a request to the Asgard process to create a new graphical viewer, using the communication channel that is managed by the document object, and then embed this window inside of the Ensemble window that is used for drawing the document.

Although Asgard's viewing window is a child of a window owned by Ensemble, it still receives all the X events that it gets in stand-alone mode, which allows it to handle its own redisplay and mouse interaction, including both viewing transformation control and some editing operations.

6.2.4 User interface functionality

The user interface facilities presented by Asgard running under Ensemble can be grouped into two categories. The first is the direct-manipulation viewing transformation control described above, which is performed directly on the display window. Its implementation is straightforward because it does not require any additional widgets or controls outside of the viewing window. The second includes commands that are provided using such additional widgets in the stand-alone mode of Asgard, and it is somewhat more problematic to make them available within Ensemble, for a number of reasons.

First, it is not really permissible to embed the editing controls along with the viewing window in an Ensemble document, since a reader would not need or want them to be visible, and the basic principle of direct-manipulation editing is to minimize the gap between the document as seen by the author and by the reader. Otherwise, it would be hard to judge how the animation window appears in the context of surrounding material, for example.

Second, Asgard must share any available user-interface tool area, such as menu bars, with the other media that are also present in the document. This limits the amount of space that it can occupy much more than the looser constraints of stand-alone operation. One possibility under consideration for Ensemble is to multiplex a medium-specific tool area among all the subdocuments that are present in a document, and allow the one that currently has the view's cursor to occupy this area.

Third, if there is more than one animated scene within a document, the system may have to determine which one should receive user commands for that medium. For some types of operations, one might determine this based on the location of the cursor, but for others that are likely to be used in viewing, such as "Begin playback" and "Stop playback", it may not be reasonable to expect the reader to place the cursor on the subdocument before invoking the command, even if there is any sort of cursor present during viewing. One might restrict the commands to act upon the scenes that are currently visible in the window, which is certainly reasonable, but there may be more than one animation subdocument visible at one time. Another possibility is to attach floating controls to each Asgard subdocument, perhaps restricting them to locations away from the main viewing region.

The current solution used by the Asgard interface is rather minimal and does not address any of these concerns. A special "Asgard" menu is provided along with the other view menus, which applies to the first Asgard subdocument within the Ensemble document. This menu contains three types of commands: playback time controls, commands to bring up full-fledged Asgard graphical and language viewers, and a small subset of the editing commands available in the graphical editor. When additional viewers are created, they are naturally linked to the view in the Ensemble document, so that modifications can be made and immediately viewed in the context of the document.

The editing commands available from the Ensemble menu are currently limited to turning on the position and velocity handles, which can then be used to modify initial conditions, and a "Recompute motion" command, which has the same effect as the "Simulate" command on the main Asgard control panel. They are provided mainly to illustrate the feasibility of moving editing functionality from Asgard into Ensemble – this is very simple, partly because Asgard is built on top of the Tcl language, and all of the high-level control is done using textual strings.

These concerns apply generally to all dynamic media that can be embedded in multimedia documents, such as sound and video. Currently, work is under way to integrate these media with Ensemble, so these issues will be explored in more detail in the future.

6.3 Extensions to this interface

Most of the currently foreseeable extensions to the interface between Asgard and Ensemble are in the area of enhancing Ensemble's handling of editing functionality for different media, as described above. Because the communication channel between the two systems is a text stream, Tcl commands can be passed back and forth quite easily. In fact, it would be very simple to use the Tk `send` primitive to communicate directly between the Ensemble user interface code and Asgard, but that would be somewhat contrary to the spirit of the architecture of Ensemble.

Another possibility would be to allow the user to embed Ensemble documents within Asgard animated scenes. This would make animation a full-fledged hierarchical medium within the Ensemble multi-media system, since animation nodes would be able to contain sub-document trees rather than just being leaf nodes as they are now. A number of interesting questions would have to be answered: Would Asgard be responsible for applying 3-D transforms to the renditions of other media? How would time be synchronized between different levels of dynamic media, including animation, movies, and music? The architecture required to support a system like this would be an interesting design problem.

Chapter 7

Conclusions and future work

This chapter will briefly summarize the research described in this dissertation, and then discuss some future directions for work in this and related areas.

7.0.1 Evaluation of the Asgard project

The major research contribution of the Asgard project is the application of the multiple-representation editing paradigm to physically-based animation. Additional contributions include some of the specific mechanisms used for graphical editing of particular entities, such as objects and trajectories, the integration with the Ensemble multi-media system, and the motion simulation algorithm described in Chapter 5.

The Asgard program itself was intended to be a prototype animation system that illustrates these principles and algorithms. It has been used to make some simple animated sequences, but was not intended to be a production-quality piece of software, with all the options and capabilities found in commercial products. In a more advanced system, kinematic motion control would probably be emphasized to a greater extent than in Asgard, and the dynamics functionality, which forms a large part of the complexity, would be less central.

However, a few lessons learned from the Asgard project should be kept in mind when considering the design of any animation editor. First, the multiple-representation editing paradigm is a powerful one, and for a variety of reasons is even more useful for media like animation than the more traditional textual and graphical documents. This applies both to physically-based animation, such as that produced by Asgard, and to constraint-based and algorithm animation, which is provided by systems such as Macromind 3D [76] and Balsa [23]. These systems all handle complex objects with complex external structure that may be quite remote from the final appearance on the screen – this is a prime characteristic of problems that can benefit from multiple-representation facilities.

The second lesson that one can learn from the Asgard project is that for a dynamics-based system, one of the major issues that should be considered early in the design is the interaction between kinematics, dynamics, and possibly other forms of control and events that take place in different media. One of the issues that could be addressed in a follow-on project to Asgard is the potential for handling interactions between objects in the animated world and objects in the real world, and objects in other parts of the same compound document that a given animated sequence is part of. The first type of interaction has been explored by quite a few researchers, but mostly in the context of kinematic control rather than dynamics. The second type of interaction has not been considered in the past, partly because of a lack of a convincingly motivating example, but it seems likely that during the course of the development of a multimedia system such as Ensemble, such examples would naturally arise.

The third lesson is that for good performance of an editing system, incrementality must be exploited whenever possible. In many cases it is easier to discard all previous results and start from scratch than to determine what data can be kept and what is no longer valid, but that kind of approach is more appropriate for a batch system than an interactive editor. In the case of Asgard, the system benefited greatly from the fact that the same mechanism that handled invalidation of data after a collision could be used for dealing with the effects of editing changes performed by the user. This is not a result of any of the simplifications that were made in the implementation of the system – it follows from the nature of collisions in a physically-based system.

7.0.2 Future work

Physically-based animation is still a developing field, especially in the areas of user interfaces and editing technology. The ideas used in Asgard suggest a number of directions for future research, both in terms of editing and interface technology and in the underlying motion simulation algorithms. A promising area for future development that has not been extensively explored until now is the use of physically-based motion in simulated environments, or "virtual reality".

Editing

Currently, Asgard interfaces with the Ensemble system in a somewhat ad-hoc manner, as described in Chapter 6. A very useful project would be to provide a more general framework for the integration of external media with the Ensemble system, which includes both editing and viewing functionality. One could define a set of mechanisms and an interface which would be provided by Ensemble, that could be used by any medium. Ensemble would thus act as a high-level manager, dealing with integration issues and allowing the individual medium-specific editors to concentrate on issues specific to them.

The multiple-representation editing facilities provided by Asgard could be extended to include additional functionality, such as finer graphical control over positioning, lighting, and forces. Since the power of language-based editing is only limited by what can be expressed in the language, there will always be a way to create a given object, and as long as the language is well-designed it should not be prohibitively inconvenient. However, as it becomes clear that facilities are used frequently enough, graphical mechanisms should be implemented for editing them whenever possible. The multiple-representation framework already in place, which is discussed in Chapter 4, need not be extended, since it is general and can handle any object types that can be described in the language.

Motion simulation

The collision detection algorithm described in Chapter 5 works very well for convex polyhedra, but it has a few problems, such as the difficulty of dynamically determining the maximum allowable stepsize for collision tests, and is not easily generalizable to other types of shapes, including concave and curved objects. Some researchers [81, 12] have developed algorithms for curved surfaces, but these tend to be quite time-consuming and also difficult to generalize. Although the Lin-Canny algorithm is adequate for the objects that Asgard currently supports, if the system were extended to handle more general object types, another algorithm would be required. An algorithm that is currently under development that uses *recursive bounding spheres* is intended to address these concerns.

The model of physics used by Asgard and most other motion simulation systems is known as *Newtonian dynamics* – it uses a straightforward application of Newton's Second Law of Motion: $\vec{F} = m\vec{a}$. Constraints and conservation equations are all transformed into forces on particular objects, which are then added up and solved. An alternative formulation, which is known as *Hamiltonian dynamics*, uses constraints and conservation equations in a much more direct way to reduce a system of equations to a subset that has no constraints, and can be solved in a more straightforward way. Some dynamics systems use this or related formulations [119, 2], and it would probably be worthwhile to investigate the use of such formulations for the Asgard system.

Some researchers have investigated the use of optimization techniques, such as optimal control [32] or quadratic programming [122] for controlling animation. These techniques are useful because they take some of the burden of specifying detailed motion away from the animator, but it can be difficult to reconcile this with the degree of control that the animator desires, and to provide adequate performance – optimization techniques can be very slow. Providing incrementality in the context of optimization is another interesting problem – because most techniques use an iterative approach, previous solutions can generally be used as starting points for the new solution. Some projects such as the Virtual Erector Set [101] have addressed this problem, but further work needs to be done.

Currently, Asgard is oriented towards the production of visually realistic motion, involving a relatively small number of potentially complex objects. A different class of problems involves large numbers of simple objects – particle systems [99] are an extreme example of this. These problems occur both in engineering and scientific simulations and in animation for visual purposes.

In a more production-oriented version of Asgard, more sophisticated graphical output would be very useful. Currently, Asgard manages its own graphical database and handles rendering in a device-transparent way, but more functionality and better rendering performance could be obtained through the use of an object-oriented graphical database system, such as IRIS inventor [107]. Such systems provide a high-level model of object appearance which is much closer to that used by the Asgard language than the low-level graphical primitives available in X11 or IRIS GL.

7.0.3 Simulated environments

Providing physical realism in simulated environments, or "virtual reality", poses a number of very interesting problems. The major issue is real-time motion simulation – if objects do not respond to forces immediately and move fairly smoothly, the illusion of reality will not be maintained. Since this is a hard constraint, the only remaining parameter that can be decreased if needed is the accuracy of the motion, and the problem that must be solved is how to distribute this inaccuracy.

There are two effects of inaccuracy in motion simulation. The first is the immediate appearance – if the user is looking at an object that is not moving correctly, it may be obvious that this is the case. The second is the future motion path of the object. Unless corrected, any deviations from the correct motion will have an effect on the future of the system.

To take a somewhat contrived example, consider a user in a virtual basketball game. He might shoot the ball at the basket, and watch it only long enough to decide that it will go in. He might then turn to look at the other players, to see who is going for the rebound. If there is too little computational power, the system must decide whether to allow inaccuracies in the area that the player is viewing, or in the trajectory of the ball, where he is not viewing. If it chooses the former, the scene will appear wrong to him, but if it chooses the latter, the ball might not go into the basket, as it should.

Many other examples such as this can be constructed. In some ways, this problem resembles the visibility and detail computations done by the Soda Hall architectural visualization project at Berkeley [48] where great efforts were made to avoid wasting resources rendering objects that were not visible, or rendering objects at higher levels of detail than were necessary, given their apparent size. Resource allocation in simulated environments is the same problem in the temporal rather than the spatial domain.

Another set of issues arises when multiple users are interacting in a simulated environment. If all the computation is done by a single "motion simulation server", with per-user clients handling only the rendering, then the case is fairly similar to the single-user case. However, experience with text-based "virtual environment" systems such as IRC [38] and MUD [108] suggests that distributed processing at all levels is necessary if the system is to scale well. In a case like this, the questions that must be answered include:

- What is the best communication architecture for the kinds of interactions typically found in distributed simulated environment applications?
- If interaction occurs between users or objects that are being handled by different servers, when is it better to allow one server to do all the computation, as opposed to performing it in a distributed way?
- Should objects be allowed to "migrate" from one simulation engine to another? For example, one body may come into close proximity with several others in succession, and it may be worthwhile to do all the computation for that body's motion on the same servers as each of the others, for a particular interval of time.

This list can only suggest some of the problems that would arise in a distributed simulated environment system that handles physically-based motion. This seems to be a very promising area for future research.

Bibliography

[1] Alias, Inc. Alias Animator. Product Information Sheet, March 1990.

[2] W. W. Armstrong. The dynamics of articulated rigid bodies for purposes of animation. *The Visual Computer Journal*, 1:231–240, 1985.

[3] W. W. Armstrong, M. Green, and R. Lake. Near-real-time control of human figure models. In *Proceedings, Graphics Interface Conference*, pages 147–151, 1986.

[4] W. W. Armstrong and M. W. Green. Dynamics for animation of characters with deformable surfaces. In Nadia Magnenat-Thalmann and Daniel Thalmann, editors, *Computer Generated Images: The State of the Art*, pages 203–208. Springer-Verlag, 1985.

[5] Gideon Avrahami, Kenneth P. Brooks, and Marc H. Brown. A two-view approach to constructing user interfaces. In *SIGGRAPH*, pages 137–146, August 1989.

[6] N. I. Badler, J. D. Korein, J. U. Korein G. M. Radack, and L. S. Brotman. Positioning and animating human figures in a task-oriented environment. *The Visual Computer Journal*, 1:212–220, 1985.

[7] Norman Badler. Animating human figures: Perspectives and directions. In *Proceedings, Graphics Interface Conference*, pages 115–120, 1986.

[8] Robert A. Ballance. *Syntactic and Semantic Checking in Language-Based Editing Systems*. PhD thesis, University of California, Berkeley, CA 94720, December 1989. Technical Report UCB/CSD 89/548.

[9] Robert A. Ballance, Susan L. Graham, and Michael L. Van De Vanter. The Pan language-based editing system for integrated development environments. In *ACM SIGSOFT Symposium on Software Development Environments*, 1990.

[10] Robert A. Ballance and Michael L. Van De Vanter. Pan I: An introduction for users. Technical Report UCB/CSD 88/410, University of California, Berkeley, CA 94720, September 1987. PIPER Working Paper 87-5.

[11] David Baraff. Analytical methods for dynamic simulation of non-penetrating rigid bodies. In *SIGGRAPH*, pages 223–232, July 1989.

[12] David Baraff. Curved surfaces and coherence for non-penetrating rigid body simulation. In *SIGGRAPH*, pages 19–28, August 1990.

[13] David Baraff. Coping with friction for non-penetrating rigid body simulation. In *SIGGRAPH*, pages 31–40, July 1991.

[14] David Baraff. *Dynamic Simulation of Non-penetrating Rigid Bodies*. PhD thesis, Computer Science Department, Cornell University, Ithaca, NY, March 1992. Technical Report 92-1275.

[15] David Baraff. Personal correspondence, June 1993.

[16] David Baraff and Andrew Witkin. Dynamic simulation of non-penetrating flexible bodies. In *SIGGRAPH*, pages 303–308, July 1992.

[17] Eric A. Bier. *Snap-Dragging: Interactive Geometric Design in Two and Three Dimensions*. PhD thesis, University of California, Berkeley, CA 94720, April 1988. Technical Report UCB/CSD 88/416, also Xerox EDL 89-2.

[18] Eric Allen Bier. Skitters and jacks: Interactive 3d positioning tools. In *Workshop in Interactive 3D Graphics*, Chapel Hill, NC, October 1986.

[19] OpenGL Architecture Review Board. *OpenGL Reference Manual*. Addison-Wesley, Reading, MA, 1993.

[20] Alan Borning. *Thinglab — A Constraint-Oriented Simulation Laboratory*. PhD thesis, Stanford University, July 1979. Technical Report STAN-CS-79-746, also Xerox SSL-79-3.

[21] Alan Borning, Robert Duisberg, Bjorn Freeman-Benson, Axel Kramer, and Michael Woolf. Constraint hierarchies. In *Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 48–60, October 1987.

[22] Kenneth P. Brooks. *A Two-view Document Editor with User-definable Document Structure*. PhD thesis, Stanford University, May 1988. Also Digital SRC Research Report 33, November 1, 1988.

[23] Marc H. Brown and Robert Sedgewick. A system for algorithm animation. In *SIGGRAPH*, pages 177–186, July 1984.

[24] Jacob Butcher. Ladle. Technical Report UCB/CSD 89/519, University of California, Berkeley, CA 94720, November 1989. PIPER Working Paper 89-4.

[25] T. W. Calvert, C. Welman, S. Gaudet, and C. Lee. Composition of multiple figure sequences for dance and animation. In R. A. Earnshaw and B. Wyvill, editors, *New Advances in Computer Graphics*, pages 245–255. Springer-Verlag, 1989.

[26] Tom Calvert. Composition of realistic animation sequences for multiple human figures. In *Workshop on Mechanics, Control, and Animation of Articulated Figures*. MIT Media Laboratory, April 1989.

[27] John Canny. Collision detection for moving polyhedra. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(2):200–209, March 1986.

[28] Michael Chen, S. Joy Mountford, and Abigail Sellen. A study in interactive 3d rotation using 2d control devices. In *SIGGRAPH*, pages 121–129, 1988.

[29] Pehong Chen. *A Multiple-Representation Paradigm for Document Development*. PhD thesis, University of California, Berkeley, CA 94720, July 1988. Technical Report UCB/CSD 88/436.

[30] Pehong Chen and Michael A. Harrison. Multiple representation document development. *Computer*, 21(1):15–31, January 1988.

[31] Wayne A. Christopher. Constraint-based document presentation. Technical Report UCB/CSD 90/601, University of California, Berkeley, CA 94720, October 1990.

[32] Stephen J. Citron. *Elements of Optimal Control*. Holt, Rinehart and Winston, 1969.

[33] Geoffrey Clemm and Leon Osterweil. A mechanism for environment integration. *ACM Transactions on Programming Languages and Systems*, 12(1):1–25, January 1990.

[34] Michael F. Cohen. Interactive spacetime control for animation. In *SIGGRAPH*, pages 293–302, July 1992.

[35] D. Brookshire Connor, Scott S. Snibbe, Kenneth P. Herndon, Daniel C. Robbins, Robert C. Zeleznik, and Andries van Dam. Three-dimensional widgets. In *Symposium on Interactive 3D Graphics*, pages 183–188, June 1992.

[36] Control Data Corporation. *ICEM MULCAD User Reference Manual, version 3.0.3*. Publication Number 60000649.

[37] Control Data Corporation. *ICEM PCUBE User Reference Manual, version 3.1.1*. Publication Number 60000853.

[38] Helen T. Rose Davis. IRC Frequently Asked Questions (FAQ) list. Periodic posting to Usenet newsgroup `alt.irc`, November 1993.

[39] Sven Delmas. *XF: Design and Implementation of a Programming Environment for Interactive Construction of Graphical User Interfaces*. Techniche Universität Berlin, Institute für Angewandte Informatik, 1993.

[40] G. J. Edwards. Script: an interactive animation environment. In *Computer Animation*, pages 173–192, London, October 1987. Online Publications.

[41] Conal Elliot, Greg Schecter, Ricky Yeung, and Salim Abi-Ezzi. A system for interactive, animated 3d graphics based on continuous, high level constraints. Technical report, SunSoft, 1993.

[42] Ioannis Emiris and John Canny. A general approach to removing degeneracies. In *32nd IEEE Symposium on Foundations of Computer Science*, pages 405–413, 1991. Also to appear in the SIAM Journal of Computing.

[43] Ioannis Emiris and John Canny. An efficient approach to removing geometric degeneracies. In *8th ACM Symposium on Computational Geometry*, pages 74–82, 1992.

[44] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*. The Systems Programming Series. Addison-Wesley, Reading, MA, second edition, 1990.

[45] David R. Forsey and Richard H. Bartels. Hierarchical b-spline refinement. In *SIGGRAPH*, pages 205–212, August 1988.

[46] Frame Technology Inc. *Using FrameMaker*, 1990.

[47] Bjorn N. Freeman-Benson and John Maloney. The deltablue algorithm: An incremental constraint hierarchy solver. Technical Report 88-11-09, University of Washington, Seattle, WA 98195, November 1988.

[48] Thomas A. Funkhouser, Carlo H. Séquin, and Seth J. Teller. Management of large amounts of data in interactive building walkthroughs. In *Workshop on Interactive 3D Graphics*, pages 11–20, 1992.

[49] Simson L. Garfinkel and Michael K. Mahoney. *NeXTStep Programming*. The Electronic Library of Science. Springer Verlag, 1993.

[50] David Garlan and Ehsan Ilias. Low-cost, adaptable tool integration policies for integrated environments. In *SIGSOFT*, pages 1–10, December 1990.

[51] Charles F. Goldfarb. *The SGML Handbook*. Oxford University Press, 1990.

[52] Herbert Goldstein. *Classical Mechanics*. Addison-Wesley, Reading, MA, second edition, 1980.

[53] Julian E. Gómez. Twixt: A 3D animation system. In *EUROGRAPHICS*, pages 121–133, 1984.

[54] James Gosling. *Algebraic Constraints*. PhD thesis, Carnegie-Mellon University, Pittsburgh PA 15213, May 1983. TR CS-83-132.

[55] Susan L. Graham, Michael A. Harrison, and Ethan V. Munson. The Proteus presentation system. In *ACM SIGSOFT Fifth Symposium on Software Development Environments*, December 1992.

[56] Gary D. Hachtel and Alberto L. Sangiovanni-Vincentelli. A survey of third-generation simulation techniques. *Proceedings of the IEEE*, 69(10):1264–1280, October 1981.

[57] Bo Stig Hansen. On the design of a functional formatting language: FFL. unpublished manuscript, December 1988.

[58] R. Heise and B. A. MacDonald. Quaternions and motion interpolation: A tutorial. In R. A. Earnshaw and B. Wyvill, editors, *New Advances in Computer Graphics*, pages 229–243. Springer-Verlag, 1989.

[59] Carl Hewitt. Procedural embedding of knowledge in Planner. In *International Joint Conference on Artificial Intelligence*, pages 167–182, September 1971.

[60] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *International Joint Conference on Artificial Intelligence*, pages 235–245, August 1973.

[61] ICEM Systems. *ICEM DDN User's Manual*, 1992.

[62] Adobe Systems Inc. *PostScript Language Reference Manual.* Addison-Wesley, Reading, MA, second edition, 1990.

[63] Deneb Robotics Inc. IGRIP product literature. Auburn Hills, MI, 1993.

[64] Wavefront Inc. Wavefront user's manual.

[65] P. M. Isaacs and M. F. Cohen. Mixed methods for complex kinematic constraints in dynamic figure animation. *The Visual Computer Journal*, 4:296–305, 1988.

[66] Paul M. Isaacs and Michael F. Cohen. Controlling dynamic simulation with kinematic constraints, behavior functions, and inverse dynamics. In *SIGGRAPH*, pages 215–224, July 1987.

[67] Kenneth I. Joy. Utilizing parametric hyperpatch methods for modeling and display of free-form solids. In *Symposium on Solid Modeling Foundations and CAD/CAM Application*, pages 245–254, June 1991. Also in International Journal of Computational Geometry and Applications, Vol. 1, No. 4, December 1991, 455-472.

[68] Donald E. Knuth. *The TEXbook.* Addison Wesley, 1984.

[69] Craig E. Kolb. *Rayshade User's Guide and Reference Manual*, 0.2 draft edition, July 1991.

[70] Kenneth S. Kundert. Sparse matrix techniques and their application to circuit simulation. In Albert E. Ruehli, editor, *Circuit Analysis, Simulation, and Design.* Elsevier Science Publishing Company, 1986.

[71] Philip Lee, Susanna Wei, Jianmin Zhao, and Norman I. Badler. Strength guided motion. In *SIGGRAPH*, pages 253–262, August 1990.

[72] Ekachai Lelarasmee, Albert E. Ruehli, and Alberto L. Sangiovanni-Vincentelli. The waveform relaxation method for time-domain analysis of large scale integrated circuits. *IEEE Transactions on CAD of Integrated Circuits and Systems*, CAD-1(3):131–145, July 1982.

[73] Wm Leler. *Constraint Progamming Languages.* Addison-Wesley, Reading, MA, 1988.

[74] Ming C. Lin and John F. Canny. Efficient collision detection for animation. In *Eurographics Workshop on Simulation and Animation*, September 1992.

[75] Ming C. Lin and John F. Canny. A fast algorithm for incremental distance calculation. In *International IEEE Conference on Robotics and Automation*, pages 1008–1014, 1992.

[76] MacroMind, Inc. MacroMind Three-D, developer version. Product Information Sheet, June 1990.

[77] Macworld. Putting the moves on 2-D models. *Macworld*, page 83, July 1993. (Discussion of the Working Model animation program from Knowledge Revolution.).

[78] Nadia Magnenat-Thalmann and Daniel Thalmann. The use of 3-d abstract graphical types in computer graphics and animation. In T. L. Kunii, editor, *Computer Graphics: Theory and Applications*, pages 360–373. Springer-Verlag, 1983.

[79] Nadia Magnenat-Thalmann and Daniel Thalmann. Controlling evolution and motion using the cinemira-2 animation sublanguage. In Nadia Magnenat-Thalmann and Daniel Thalmann, editors, *Computer Generated Images: The State of the Art*, pages 249–259. Springer-Verlag, 1985.

[80] Vance Maverick. The Arugula computer-music score editor. In *Proceedings of the International Computer Music Conference*, pages 419–422, 1991.

[81] M. Moore and Jane Wilhelms. Collision detection and response for computer animation. In *SIGGRAPH*, pages 289–298, 1988.

[82] Greg Nelson. Juno, a constraint-based graphics system. In *SIGGRAPH*, pages 235–243, July 1985.

[83] Steven A. Oakland. Bump, a motion description and animation package. Technical Report UCB/CSD 87/370, University of California, Computer Science Division, University of California, Berkeley CA 94720, September 1987.

[84] T. J. O'Donnell and A. J. Olson. Gramps: A graphics language interpreter for real-time, interactive, three-dimensional picture editing and animation. In *SIGGRAPH*, pages 133–142, 1981.

[85] John Ousterhout. *Tk 3.3 manual pages*, 1993. Available for anonymous ftp from `ftp.cs.berkeley.edu`.

[86] John Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, Reading, MA, 1994.

[87] Derluen Pan and Michael A. Harrison. IncTeX: An incremental document processing system. Technical Report UCB/CSD 91/614, University of California, Berkeley, CA 94720, April 1991.

[88] Paracomp Inc. Swivel 3D. Product Information Sheet, 1990.

[89] Alex Pentland and John Williams. Good vibrations: Modal dynamics for graphics and animation. In *SIGGRAPH*, pages 215–222, July 1989.

[90] Alex P. Pentland. Computational complexity versus simulated environments. In *Symposium on Interactive 3D Graphics*, pages 185–192, March 1990.

[91] Cary B. Phillips and Norman I. Badler. Interactive behaviors for bipedal articulated figures. In *SIGGRAPH*, pages 359–362, July 1991.

[92] Cary B. Phillips, Jianmin Zhao, and Norman I. Badler. Interactive real-time articulated figure manipulation using multiple kinematic constraints. In *Symposium on Interactive 3D Graphics*, pages 245–250, March 1990.

[93] D. Pletincks. The use of quaternions for animation, modelling and rendering. In Nadia Magnenat-Thalmann and Daniel Thalmann, editors, *New Trends in Computer Graphics*, pages 44–53. Springer-Verlag, 1988.

[94] Jef Poskanzer. PPM - portable pixmap file format. UNIX manual pages, 1989. Available by anonymous ftp from export.lcs.mit.edu, in /pub/R5untarred/contrib/clients/pbmplus.

[95] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, 1988.

[96] William T. Reeves, Eben F. Ostby, and Samuel J. Leffler. The Menv modelling and animation environment. Pixar, San Rafael, CA, January 1989.

[97] Thomas W. Reps and Tim Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-based Editors.* Springer-Verlag, 1989.

[98] Craig W. Reynolds. Computer animation with scripts and actors. *Computer Graphics*, 16(3):289–296, July 1982.

[99] Craig W. Reynolds. Flocks, herds, and schools: A distributed behavioral model. In *SIGGRAPH*, pages 25–34, July 1987.

[100] Resve A. Saleh, James E. Kleckner, and A. Richard Newton. Iterated timing analysis in SPLICE1. In *ICCAD*, pages 139–140, 1983.

[101] Peter Schröder and David Zeltzer. The virtual erector set: Dynamic simulation with linear recursive constraint propagation. In *Symposium on Interactive 3D Graphics*, pages 23–31, March 1990.

[102] Robert Sedgewick. *Algorithms*. Addison-Wesley, Reading, MA, second edition, 1988.

[103] Carlo H. Séquin. The Berkeley UNIGRAFIX tools, version 2.5. Technical Report UCB/CSD 86/281, University of California, Berkeley, CA 94720, Spring 1986.

[104] Steve Shepard. Interactive physics II. *MacUser*, page 70, December 1992.

[105] Ben Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley, Reading, MA, second edition, 1993.

[106] Silicon Graphics, Inc, Mountain View, CA. *Iris Graphics Library Programmer's Manual*, 1992.

[107] Silicon Graphics, Inc. Iris inventor technical report. Technical report, Silicon Graphics, Inc, Mountain View, CA, 1993.

[108] Jennifer Smith. Frequently asked questions. Periodic posting to Usenet newsgroup `rec.games.mud.misc`, December 1993.

[109] Guy Lewis Steele, Jr. *The Definition and Implementation of a Computer Programming Language*. PhD thesis, Massachusetts Institute of Technology, Cambridge MA 02139, August 1980. Also MIT AI Lab TR 595.

[110] Michael Stonebraker and Lawrence A. Rowe. The postgres papers. Technical Report UCB/ERL M86/85, University of California, Berkeley, CA 94720, June 1987.

[111] Mark C. Surles. Interactive modeling enhanced with constraints and physics – with applications in molecular modeling. In *Symposium on Interactive 3D Graphics*, pages 175–182, June 1992.

[112] Ivan E. Sutherland. Sketchpad: A man-machine graphical communication system. In *Design Automation Conference*, 1964. Also in *25 Years of Electronic Design Automation*, 1988, ACM Order Number 477881.

[113] Mark A. Tarlton and P. Nong Tarlton. A framework for dynamic visual applications. In *Symposium on Interactive 3D Graphics*, pages 161–164, June 1992.

[114] Richard N. Taylor, Frank C. Belz, Lori A. Clarke, Leon Osterweil, Richard W. Selby, Jack C. Wileden, Alexander L. Wolf, and Michal Young. Foundations for the arcadia environment architecture, 1988.

[115] Demetri Terzopoulos and Kurt Fleicher. Modeling inelastic deformation: Viscoelasticity, plasticity, fracture. In *SIGGRAPH*, pages 269–278, August 1988.

[116] Demetri Terzopoulos and Kurt Fleischer. Deformable models. *The Visual Computer Journal*, 4:306–331, 1988.

[117] Spencer W. Thomas. *Utah Raster Toolkit Manual Pages*. University of Michigan, 1990. Contact `toolkit-request@cs.utah.edu` for availability information.

[118] Christopher J. Van Wyk. A high-level language for specifying pictures. *ACM Transactions on Graphics*, 1(2):163–182, April 1982.

[119] Jane Wilhelms. Virya – a motion control editor for kinematic and dynamic animation. In *Proceedings, Graphics Interface Conference*, pages 141–146, 1986.

[120] Jane Wilhelms, M. Moore, and R. Skinner. Dynamic animation: Interaction and control. *The Visual Computer Journal*, 4:283–295, 1988.

[121] Jane P. Wilhelms and Brian A. Barsky. Using dynamic analysis to animate articulated bodies such as humans and robots. In Nadia Magnenat-Thalmann and Daniel Thalmann, editors, *Computer Generated Images: The State of the Art*, pages 209–229. Springer-Verlag, 1985.

[122] Andrew Witkin and Michael Kass. Spacetime constraints. In *SIGGRAPH*, pages 159–168, August 1988.

[123] Douglas A. Young. *OSF/Motif reference guide*. Prentice Hall, 1990.

[124] Robert C. Zeleznik, D. Brookshire Connor, Matthias M. Wloka, Daniel G. Aliaga, Nathan T. Huang, Philip M. Hubbard, Brian Knep, Henry Kaufman, John F. Hughes, and Andries van Dam. An object-oriented framework for the integration of interactive animation techniques. In *SIGGRAPH*, pages 105–111, July 1991.

[125] David Zeltzer. Direct manipulation of virtual worlds. In *Workshop on Mechanics, Control, and Animation of Articulated Figures*. MIT Media Laboratory, April 1989.

[126] David Zeltzer, Steve Pieper, and David J. Sturman. An integrated graphical simulation platform. In *Proceedings, Graphics Interface Conference*, pages 266–274, 1989.

[127] O. C. Zienkiewicz. *The Finite Element Method*. McGraw-Hill, New York, NY, 1977.

[128] Michael J. Zyda, David R. Pratt, James G. Monahan, and Kalin P. Wilson. NPSNET: Constructing a 3d virtual world. In *Symposium on Interactive 3D Graphics*, pages 147–156, June 1992.