5. The LEMAN System

An interactive animation system is presented for studying layered character models with elastic components. The system, called LEMAN (Layered Elastic Model ANimation), allows three-dimensional animated characters to be built up from successive layers of skeleton, muscle, fat and skin in a completely interactive, direct-manipulation environment, using a variety of multi-dimensional input devices. This system has been used to implement the elastic surface layer model, in which a simulated elastically deformable skin surface is wrapped around a kinematically modeled articulated figure. The character may then be animated by moving the underlying articulated figure, either interactively using forward or inverse kinematics, or by interpolating a sequence of key postures. Once a motion sequence has been specified, the entire simulation can be recalculated at a higher surface resolution for better visual results.

5.1. Introduction

Practical three-dimensional character animation requires interactive tools for both construction and animation of characters. Whether animating simple animals or realistic-looking humans, character animation is an essentially creative process and it is necessary that the software tools be accessible to non-technical, creative animators. Layered construction techniques which model anatomical features have shown promise in creating character models that deform automatically around an articulated skeleton. But purely geometric models, although they can be very expressive, usually require too much user intervention to achieve realistic-looking results. Physically-based elastic models provide more realistic behavior at the price of CPU resources and difficulty of control. With proper use of constraints, however, deformable models can be controlled by kinematic geometrical models. Recent improvements in processor speeds now make it possible to simulate certain kinds of physically-based models in real-time.

For these reasons, we believe that a hybrid approach in which layered models are constructed using a combination of geometric, kinematic and physically-based techniques, is the most promising one. The ideal 3D character model should provide a good compromise between interactive speed and realism, and between animator control and physically realistic behavior. The exact details of such a model are no more important, however, than the types of interactive technique used to construct and animate it. A variety of multi-dimensional input devices are now available, such as the spaceball, MIDI keyboard and dataglove. Finding the right kinds of interaction metaphors by which these devices can control layered, physically-based models requires experimentation with many of the various possibilities.

To aid this kind of research, we have developed an in-house environment at the Computer Graphics Lab of the Swiss Federal Institute of Technology in which a large variety of different interactive devices are easily available and can be rapidly combined with different types of interaction metaphor for testing. For the remainder of this paper, we will describe a system we have built using this environment which can be used to implement layered elastic models and study interaction metaphors for constructing and animating them. In particular, we will use the layered elastic model introduced in Chapter 4 called the elastic surface layer model [Turner 93].

5.2. Implementing The Elastic Surface Layer Model

As described in Chapter 5, the development of the elastic surface layer model is an attempt to minimize computational effort by using for each layer the modeling techniques which are most appropriate. Since the skin is the outermost layer and the only one directly visible, we concentrate CPU effort on this by modeling it as a simulated deformable elastic surface [Terzopoulos 87]. The underlying layers are then modeled using geometric and kinematic techniques which act on the surface as force-based constraints. In particular, reaction constraints prevent the surface from penetrating the underlying layers, pushing the skin out, while point-to-point spring constraints pull the surface in.

The skin surface is implemented as a simulation of a continuous elastic surface discretized using a finite difference technique [Terzopoulos 87]. The surface is represented as a rectangular mesh of 3D mass points, together with their physical characteristics (e.g. mass, elasticity) and their current state information (e.g. position, velocity). When the numerical solver is turned on, the state is evolved over time at a fixed simulation time step. At periodic intervals, usually at least five per second, the surface is rendered on the screen, resulting in a continuous simulation, usually at some fraction of real-time.

The surface is bounded at its poles and constrained by reaction constraints, point-to-point spring forces and other environmental forces such as gravity and air pressure. At each time step, the spring forces acting each surface point are calculated, according to the hookian spring constant, and added to the total applied force for that point. Then the point is checked to see if it is inside any of the muscle layer surfaces, in which case reaction constraints are applied. Rather than simply adding forces, reaction constraints remove undesirable forces and replace them with forces that move the elastic surface towards the constraint surface with critically damped motion [Platt 88].

5.3. User Environment

Unlike many research animation systems, the LEMAN system was intended to demonstrate a system usable by non-programmers, in particular, by professional animators. To them, the process of creating an animation should be as tangible and direct as modeling a piece of clay. Therefore, I have tried to make the user interface as much as possible an intuitive, direct-manipulation one in the style of the Macintosh, without any scripts, configuration files or other components that hint of programming. There is, therefore, a very firm boundary between what the user sees and what the programmer sees. This section will describe the ideal LEMAN system from the user point of view.

The animator sits in front of a high-performance Silicon Graphics Iris workstation which can display lighted, texture-mapped surfaces containing many thousands of polygons at interactive update rates. He looks at a window containing a single, lighted, perspective view into the three-dimensional world in which the animation is to take place. Surrounding this window, or perhaps on a separate monitor sitting next to him, is an arrangement of windows containing two-dimensional widget panels which can be used to issue command and adjust parameters of the animation. In one hand he holds a mouse, in the other a spaceball. Optionally, he may also have a MIDI keyboard in front of him or off to one side, and a audio system which can play synthesized prerecorded digital sounds.

Starting from scratch, the animator can build up a three-dimensional character model, adjusting parameters and moving it interactively throughout the process to test its appearance and behavior. Once the character is constructed, he can save it to a file for future use. The character can then be animated using an interactive key frame animation technique and these animation can be saved as well. Any number of animation sequences can be created for a single character.

5.4. Constructing a Character

The LEMAN system allows layered elastic characters to be constructed and animated in a totally interactive, direct manipulation environment, using, multi-dimensional input devices such as the spaceball, valuators, and MIDI keyboard. Traditional desktop-metaphor interaction techniques are also available such as the mouse and widget panels. Such a variety of input techniques allows several possible working configurations. At one extreme is the familiar mouse-and-widget metaphor, usually using a virtual trackball for 3D interaction. From here, one can add other devices, such as the spaceball, valuators, MIDI keyboard, and even dataglove.

Since all 3D input operations can be expressed in the form of either an absolute or a relative 4x4 homogeneous transform matrix, the various 3D input devices can more or less be interchanged at will. We therefore concentrate our software development efforts on creating a collection of 3D interaction metaphors, or *manipulators*, and let the user assign which 3D input device controls them at run-time. As a minimal configuration, we usually have two 3D input devices available: the spaceball and the virtual trackball, which is normally bound to the mouse. Most common operations can be performed using the spaceball with one hand and the trackball with the other. For example, the spaceball can be used to position and orient the character while the trackball can be used to move the joints. Less frequently performed operations are accomplished using the widget panels. If we wish to avoid using the widgets altogether, to make more screen real-estate available for viewing the scene, their functionality can be easily bound to keys on the MIDI keyboard or to dataglove gestures.

Figure 5.1 shows a sequence of several stages of the interactive construction process for an elastic surface layer character. Starting from scratch, the animator first builds up an articulated skeleton, then adds the muscle layer as links attached to the skeleton joints. Then an elastic surface skin is wrapped around the articulated figure and the physical simulation is started. Finally, the fat and connective tissue layer, which controls the attachment of the surface skin to the underlying layers, is adjusted. The process is entirely iterative, that is, the animator may step back to any point in the process without losing work.

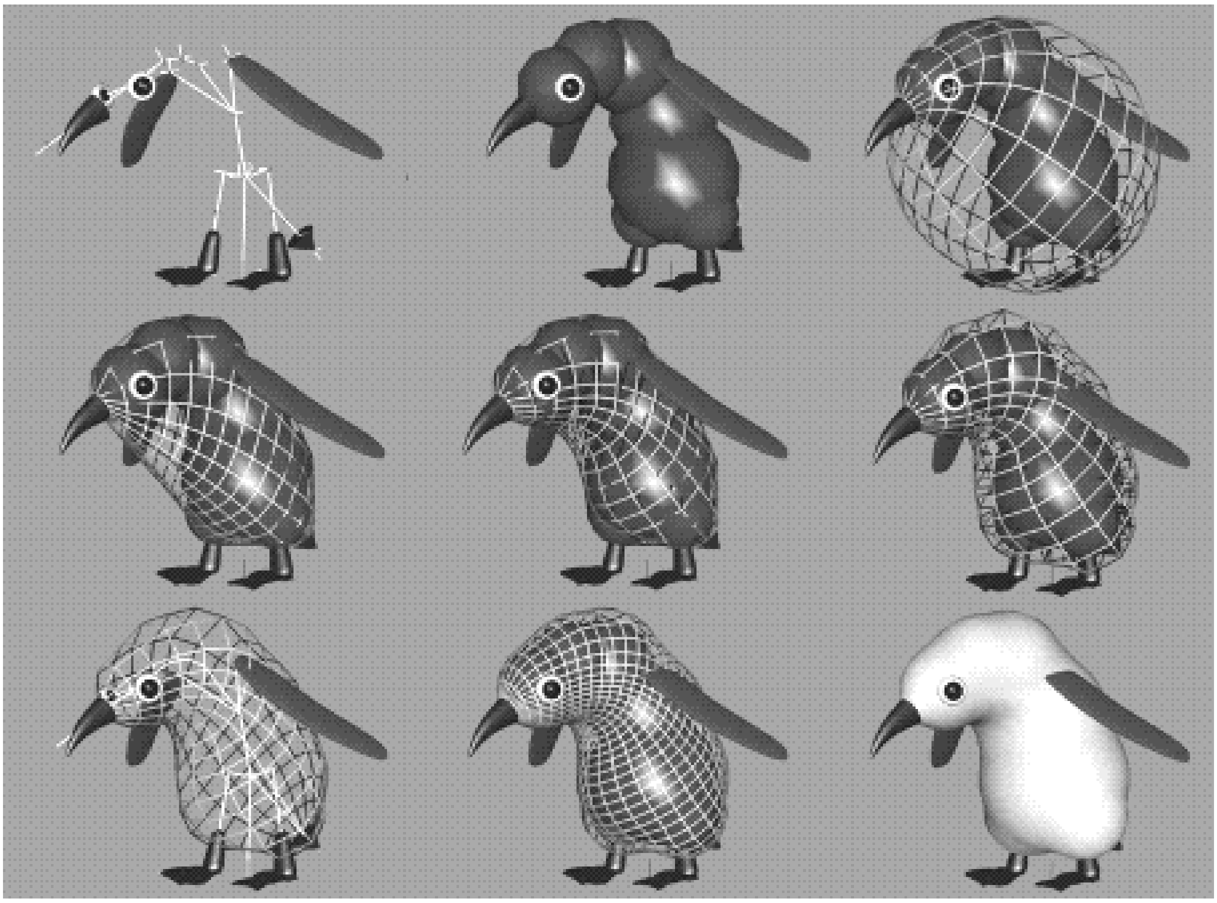

Figure 5.1: Stages in the Construction of a Layered Elastic Character

5.4.1. Skeleton Building

The skeleton is first built using the hierarchy-building tools, which provide basic operations on joints such as creation, deletion, copy and paste. A joint represents a 4x4 homogeneous space transformation together with a single rotational or translational degree of freedom. Articulated structures can be constructed by arranging the joints in hierarchies.

The current joint can be selected using the mouse. Then, by using one kind of 3D interaction metaphor, the *joint manipulator*, the local transformation of the current joint can be moved to its desired position and orientation. With a 3D local coordinate snap-to-grid mode, this can be done precisely. Joints are represented as a red-green-blue coordinate axis with a hierarchy line drawn from its origin to the origin of its parent. A pure hierarchy of joints therefore resembles a stick figure skeleton.

A second interaction metaphor, the *inverse kinematic manipulator,* can be used to move the current joint as the end-effector of a kinematic chain. In this way, the skeleton kinematics as well as its structure may be tested at any point in the construction process. The inverse kinematic manipulator takes relative transformations of the end effector (both translational and rotational) and multiplies them by the pseudo-inverse of the chain's Jacobian matrix to determine the differential joint angle values [Klein 83] When this manipulator is bound to the spaceball, for example, the chain can be directly manipulated in six degrees of freedom, as though the animator were simultaneously moving and twisting the end effector.

The kinematic chain can be specified by setting a root joint, which is considered the base of the chain and then treating the currently selected joint as the end-effector. As an alternative, an automatic root selection mode can be set, which walks up the hierarchy from the current joint until the first branch point is found, setting this to be the root joint.

Developing good interactive skeleton manipulation techniques is important because, once the character has been constructed, this constitutes the bulk of the work done by the animator to create the final animation. Since the motion can usually be visualized without displaying the skin surface, like a traditional pencil-test, this kind of interactive work can usually be done at very high screen update rates.

5.4.2. Adding Muscles

This stick-figure skeleton can then be fleshed out by adding muscle surfaces as links attached to the skeleton joints. Muscles are modeled as deformable implicit surfaces which prevent penetration by the skin layer. We currently use spheres and superellipses [Barr 82] together with global deformation functions [Barr 84], but any implicit surface for which a rapid inside/outside function exists could be used. To increase interactive speed, we try to use undeformed spheres whenever possible. Since the muscle surfaces push the outer layers out via reaction constraints, it is important to be able to test rapidly whether a point is inside a muscle surface.

Muscles can be created by the animator by attaching a shape object to the currently selected joint. It is also possible to create complex hierarchical muscles by attaching link subhierarchies to the joints. These link subhierarchies are made from static nodes each of which can contain a shape object. Shape objects can be edited to control the type of shape (sphere, cylinder, superellipse), the shape parameters, and the global deformation parameters, which can be adjusted using sliders. Shapes can be made visible or invisible and active or inactive. Active shapes push the skin to be on the outside, while inactive shapes do not affect the skin surface at all. Visible, inactive shapes can be used to represent external, non-deformable components of the character.

5.4.3. Attaching the Skin Surface

When the muscle surfaces have been added, the rectangular surface mesh (initially in a spherical shape) is created and connected at each pole directly to points on the muscle layer. The polar points can be connected either automatically, or by the animator on a point by point basis using the mouse.

At this point, the numerical solver can be started. Initially, the only force applied to the surface is a uniform internal air pressure force, which pushes the elastic surface outward like a balloon. The amount of pressure can be adjusted to ensure that the surface is completely outside the underlying surface layers.

5.4.4. Activating Reaction Constraints

Next, the animator can turn on the reaction constraints and slowly reduce the pressure until the surface shrinks around the articulated figure. The reaction constraints push the skin surface outside the muscle layer, but leave it free to slide along it until it finds an energy minimum. The most important parameter of the reaction constraints is the spring constant, which controls how rapidly the surface fulfills its constraint. This is usually set as high as possible without causing the simulation to become unstable, and should have a time constant significantly smaller than any other time constants in the model. It is also important to make sure that the skin surface does not fall through its constraint surface so that the muscle comes completely outside the skin. This can be avoided by not moving the skeleton too rapidly while the simulation is progressing.

5.4.5. Binding Surface Points to the Skeleton

The effect of connective tissue is simulated by creating attractive spring constraints between individual points on the skin and the muscle surface [Witkin 87]. To add these "rubber-band" force constraints, the animator first places the skeleton in a neutral position, while the simulation progresses, so that the skin is well-distributed over the articulated figure. Then, some or all surface points can be bound to the underlying muscle layer, either manually on a point-by-point basis, or automatically, by selecting a group of points using the mouse and giving the *attach* command which, for each point, traces a ray perpendicular to the skin surface to determine the attach point on the muscle layer.

The spring and damping constants of the rubber-bands can then be adjusted, either globally or for individual points, to give the desired tightness or looseness of skin attachment. By locally varying these parameters, together with the local skin elasticity parameters, a variety of skin deformation and dynamic effects can be created such as sliding of skin over muscles, and squash and stretch and follow-through. Figure 4.8 shows an example of skin deformation for different skeleton postures.

5.4.6. Sculpting the Fat Layer

The fat layer is modeled simply as a thickness below the skin layer. It is therefore implemented by offsetting the skin surface points by their fat thickness perpendicular to the skin surface and using these points as inputs to the reaction constraints. The thickness of the fat layer can be adjusted, either as a global parameter or by selecting individual points on the mesh using the mouse and setting their fat thickness locally. This allows the animator to control the shape of the surface to some extent simply by sculpting the fat layer.

5.4.7. Animation

Once the character has been constructed and all of its physical parameters defined, it may be animated simply by animating the skeleton. The motion of the skeleton provides the input forces which drive the skin surface. There exists a variety of dynamic and kinematic techniques for animating articulated figures. We have chosen a key-frame animation technique in which a series of key postures is specified and a smooth skeletal motion is interpolated between them [Girard 87]. The key postures are specified interactively using the inverse kinematics manipulator described in the previous section.

Although the interpolated skeletal motion is a purely kinematic one, the resulting dynamic motion of the skin is physically simulated, resulting in a richer form of automatic inbetweening. For example, a perfectly cyclical skeletal motion such as a walk sequence will not necessarily result in a perfectly cyclical skin motion, but rather will vary somewhat from cycle to cycle, depending on the time constants of the elastic surface.

5.4.8. Interpolation of Key Postures

To animate the figure, the user positions the skeleton into a sequence of key postures, either without the elastic surface, or with the simulation running at a low surface resolution for interactive speed. A smooth motion can then be created by interpolating the joint angles using an interpolating spline [Kochanek 84]. The resulting skeleton motion may then be played back at any speed to check the animation, although if the simulation is running, this should not be too fast. To give an idea of the final animation, the motion sequence can be played back at full-speed (real-time). To get an accurate impression of the skin dynamics, it can be played back in simulation time. Key postures can be edited by selecting the particular key and repositioning the skeleton interactively.

5.4.9. Increasing the Surface Resolution

One of the advantages of using a rectangular mesh to represent the surface is that the mesh resolution can be changed quite easily. All the current values of the mesh (e.g. position, velocity, elasticity, spring constants) are bilinearly interpolated to determine the values of the higher resolution points. Although the character is usually designed and animated at a low surface resolution, once a motion sequence has been specified, the resolution can be increased and the same motion played back in simulation time to calculate a final motion sequence. This motion sequence is stored as a large array of successive elastic surface meshes and can be played back at interactive rates and viewed from different angles to check the final animation. Then the entire sequence can be rendered off-line using a standard rendering package.

5.4.10. Timing Statistics

Using unoptimized code on an SGI Crimson with VGX graphics, we have been able to construct and animate full characters such as the penguin in Figure 4.8 at a surface resolution of 16 x 16 mass points in one tenth real-time. With a redraw rate set to five frames per second, which is barely adequate for interactive work, about half the CPU time is spent redrawing the screen and half running the simulation. When scaled up to 32 x 32 mass points, the simulation slows down by a factor of eight to 1/80th real-time, which is still fast enough for calculating sequences for final rendering.

5.5. Input and Event Handling

Animated and interactive behavior are among the most confusing and poorly understood aspects of computer graphics design. These can actually be thought of together as the fundamental problem of dynamic graphics applications: how to modify graphical output in response to real-time input? Viewed in this way, input from the user results in interactive behavior, while input from other data sources or timers results in real-time animated behavior.

There are at least two reasons why dynamic behavior can be so difficult to design. Unlike graphical entities, which can be easily modeled with data structures, dynamic behavior is more difficult to visualize and tends to be buried within algorithms, not data structures. Secondly, there are at least three different software techniques for obtaining real-time input: asynchronous interrupts, polling and event queues. The challenge of dynamics in object-oriented design is: how to design and encode the behavior of graphics programs as easily as we can design and encode their visual appearance. To do this, we have to first try to construct a clear and understandable basic model of dynamic program behavior and then build a set of higher-level concepts on top of it.

5.5.1. Asynchronous Input

Interactive dynamic application programs receive real-time input from the world in the form of asynchronous events. The first task of the software system is to organize these events into a single event queue so that a single-threaded process can deal with them.

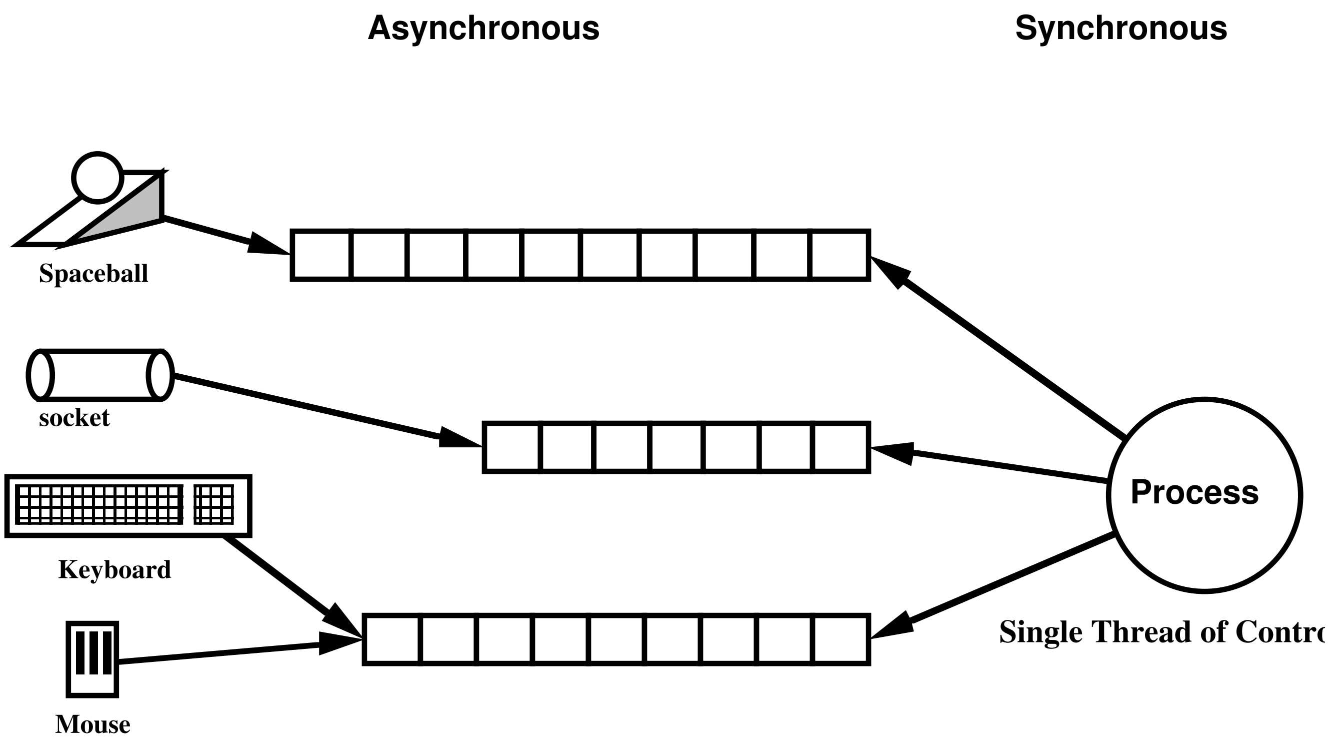

Figure 5.2: Asynchronous Input

5.5.2. Event-Driven Model

Most modern object-oriented graphics systems obtain their real-time input in the form of events in an event queue. Event queues are generally the preferred method, although some systems, such as the original Smalltalk, using polling and others, such as X windows [Nye 88], allow a mixture of the two.

Applications built using a purely event-driven input model usually consist of two sections: initialization and event loop. In the first section, the initial graphical data structures are built up. In the second section, events are responded to by changing the state of particular graphical objects, creating new objects, destroying existing ones, or by redisplaying the screen. This results in an application which is dynamically coherent, that is, after each cycle of the event loop, the entire data structure is up-to-date and consistent with itself.

Assuming a purely event-driven model, the basic application algorithm then takes the form of an event loop as follows:

```
loop forever
    Go into wait state;
    Wake up when event happens;
```

```
    Respond to event;
endloop
```

In such a structure, the dynamic behavior is implemented in the section `respond to event`. This is usually referred to as *event handling*.

5.5.3. Representing Events

How should this event driven model be represented in an object-oriented design? A common way is to represent each event as an instance of an event object which contains all the appropriate data associated with the event. Although this representation is useful for creating event queues, it is not always the most appropriate. Different types of event can contain completely different sorts of data, requiring a separate subclass for each type of event. More importantly, the event instances themselves are short-lived and if more than one instance exists at a time, there is a possibility that the data structures will become incoherent.

Once an event has been removed from the queue and needs to be handled, it can be represented as a message. This is quite natural because the act of sending a message is, like an event, a temporal, one-time occurrence. The parameters of the message contain the event data and the receiver of the message is some object which is able to handle the event. The entire event loop then consists of gathering the next event, converting it to a message and sending the message to an event handling object.

5.5.4. Event Handlers and Messages

Given the representation of events as messages, the problem of dynamic behavior becomes one of event handling. Good object-oriented design, however, suggests that event handling should be decentralized so that the task is split up among the various objects affected by the event. Each object implements its event handler in the form of event methods, one for each type of event recognized by the object. These methods effectively encapsulate the object's dynamic behavior, allowing it to be inherited and reused like its other characteristics. A more sophisticated approach involves moving the dynamic behavior into a separate object, called a controller, which implements behavior. This is analogous to the separation of rendering operation described in the previous section.

5.5.5. Distribution of Events

Just as the encapsulation of an object's graphical appearance allows higher-level graphical assemblies to be constructed from components, an assembly's dynamic behavior can be built up from the behavior of its component parts. To do this effectively, a mechanism must be used to distribute the events properly to the component objects. The standard solution to this, used for example in the Xt toolkit, is to distribute the events to the objects according to a predefined algorithm. In the Xt toolkit, which calls its user-interface objects *widgets*, the distribution is based on the widget's location on the screen and the position of the mouse. The widget has a certain amount of control over the distribution of events, by choosing whether to absorb the event or pass it on to an object underneath. It can also generate secondary "software" events by placing new events on the queue.

Although screen-based event distribution works quite well for 2D user-interface objects, there are several problems in generalizing the technique and applying it to 3D interactive graphics. The method of distributing the events is quite centralized and highly specialized for graphical user interface events. For user-interface objects, which usually occupy a rectangular region of the screen, the event can be distributed to whichever object the mouse is on top of. For 3D objects, less well-defined graphical objects, or non-graphical objects, there is no particular way of distributing the events. There are only a limited number of types of events and these carry information specific to traditional input devices such as the mouse and keyboard. Finally, it is difficult for objects to control the distribution of secondary events since they must be placed back on the central queue.

A general solution to these shortcomings can be found in NeXTstep's target/action metaphor [Webster 89]. In NeXTstep's InterfaceBuilder, user-interface objects communicate via *action* messages which have a single parameter, the source. This source is the object which sent the action message and can be queried by the receiver of the message, or *target* object, for any associated data. User interface objects can be *bound* together so that when, for instance, a slider object is moved, it sends an action message to a second slider so that the two move in tandem.

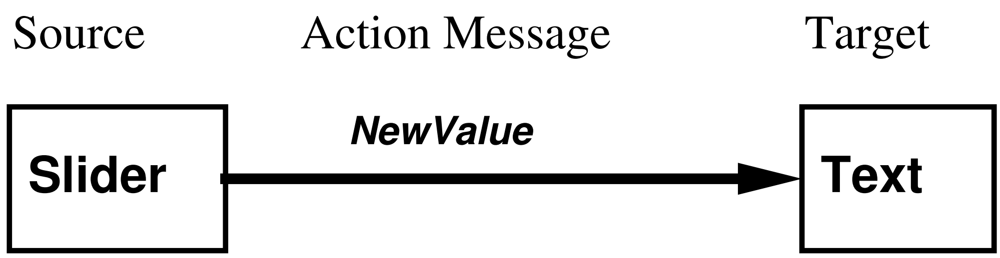

Figure 5.3: Action Messages

This representation suggests the concept of an event being a signal between two connected objects, a source and a target, much as two IC chips communicate via a signal on a connecting wire. The only information transmitted by the event itself is its type, represented by the selector name. Any other data must be explicitly queried from the source by the handler of the event. This eliminates the need for various different data structures for each type of event. The data is contained in the source object and it is up to the handler to decide which type of information to look for.

By extending action messages to include all types of events, a decentralized event distribution mechanism can be created in which every event has a source and a target. The fact that events have to come from somewhere suggests a software architecture in which every input device or source of real-time data is represented by an object. Rather than having a *Mouse Moved* event and a *Spaceball Moved* event, we instead can have a single *New Value* event which can come from either the Mouse or the Spaceball object. This helps to support reusability, because device objects can be interchanged easily, and decentralization, because the event generating code is distributed among the various device objects. It also supports virtual device objects such as graphical widgets because there is no syntactic difference to the handler between *software* events coming from a virtual device and *real* events coming from real device.

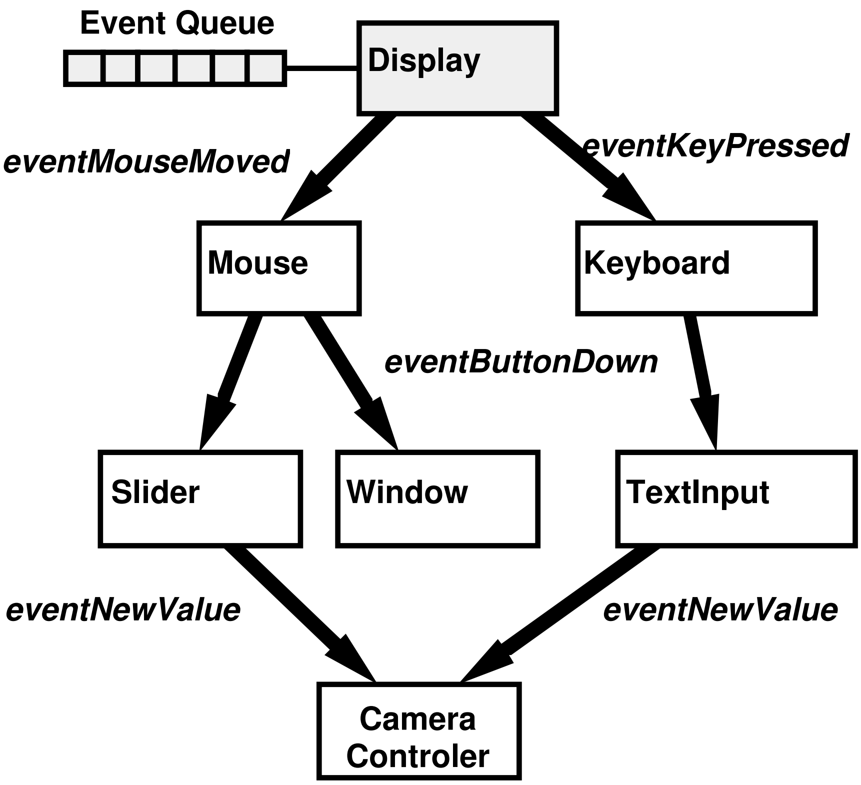

Figure 5.4: Event Networks

5.6. 3D Interaction Techniques

5.6.1. 3D Graphical Model

3D graphics systems are typically concerned with the animation of models arranged in a hierarchical fashion [Boulic 91]. Such systems usually need to maintain the following kinds of information in their graphical data structures:

- the shapes of the models, described in a local reference frame;
- their position, orientation and scale in Cartesian space;
- the hierarchical relation between the different reference frames;
- the rendering attributes of the different models.

This kind of information can be encapsulated in an object-oriented structure, with the responsibility of handling the different types of information decentralized among specialized classes. The issue in 3D interaction research is how, using limited user-input devices, to interact with such a complex three-dimensional virtual world.

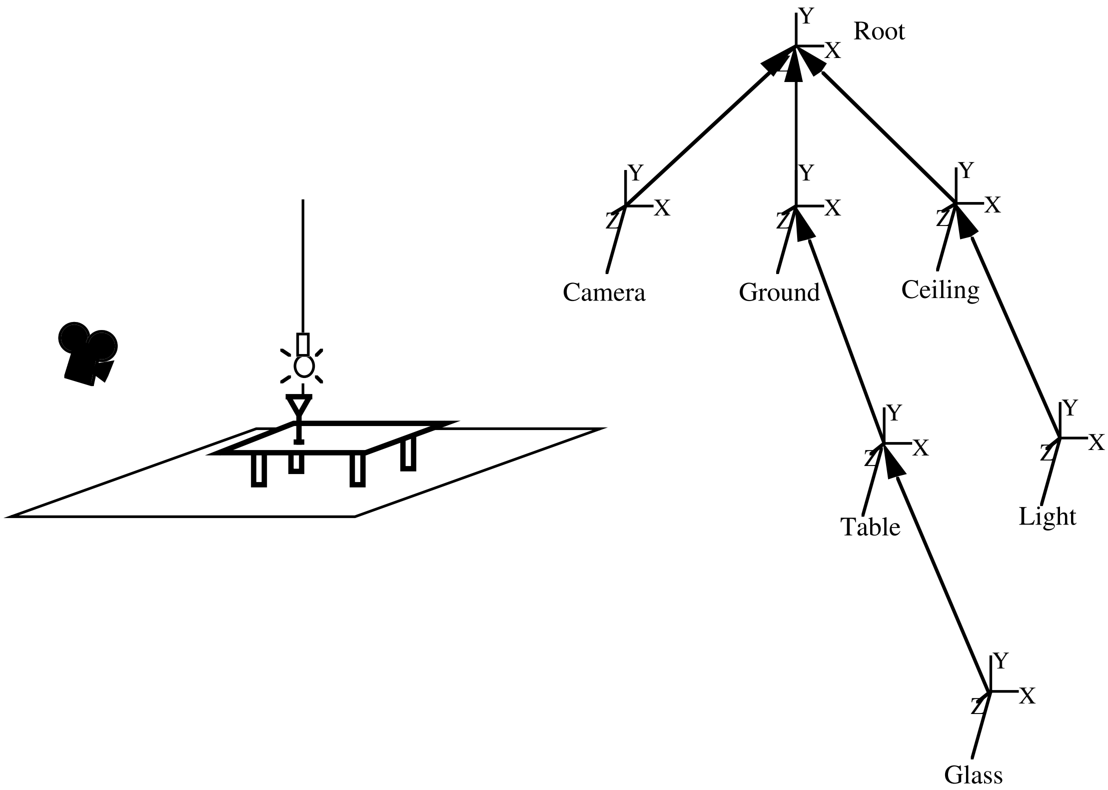

Figure 5.5: A 3D Hierarchical Scene

5.6.2. Widgets

A standard technique for interacting in 2D user interface toolkits is the mouse-controlled user-interface widget. This can be thought of as a dynamic two-dimensional icon representing such typical real-world control panel items as buttons, sliders, toggles, dials, etc. Widgets can also be used to type in text, and to display two-dimensional data. The simplest approach to implementing 3D interaction is to use 2D widgets. For example, one can bind a slider to the x position of one node in the hierarchy so that by moving the slider, that node and all its decedents are moved. This is the standard approach we have taken in the LEMAN system, since it is very simple and straightforward using the 5D Toolkit tools. However, relying exclusively on this technique can rapidly become impractical, since there are so many variables in a 3D scene to control, the amount of screen real-estate devoted to 2D widgets soon exceeds that for the 3D scene itself and finding the right widget can become a problem.

A second type of approach is to extend the concept of a widget into three dimensions, building 3D widgets. These can be thought of as virtual mechanical devices which the user can grab and manipulate using the mouse as well as multi-dimensional input devices such as the spaceball or dataglove. 3D widgets can be thought of as virtual tools which can be summoned at will and attached to 3D objects in order to manipulate them. Since they are part of the 3D virtual world, they do not take up as much screen real-estate as 2D widgets and they can have a much larger variety and complexity of functions. 3D widgets, however, can be difficult to control and very difficult to implement, although new constraint-maintenance techniques show promise in this area.

5.6.3. Direct Manipulation

A different approach to 3D interaction is direct manipulation. This concept is important in 2D interfaces where the user can select, grab, drag and rotate two-dimensional objects or icons using the mouse. This extends naturally to three dimensions because we humans work every day in a three-dimensional world and therefore have developed skill in manipulating it. Direct-manipulation techniques try to adapt these skills for controlling 3D objects in virtual worlds. The challenge of direct manipulation interface design is how to find good interaction metaphors for manipulating 3D graphical data. Interaction metaphors are basically a mapping between the raw input data coming from a (often multi-dimensional) input device and the actual model parameters that need to be controlled. To be comprehensible to the user, the metaphor should mimic some type of real-world manipulation method. An obvious example of a 3D interaction metaphor with a dataglove, for example, is to be able to simply grab an object and move it around in space. There are other possibilities, however, using the same input devices. For example, one could move the camera, or point and fly in certain directions. Unfortunately, not many workstations are yet equipped with data gloves. However, there are now a number of six degree-of-freedom input devices available and our laboratory has a spaceball device connected to many of the workstations.

5.6.4. Mouse-Based Events

Many 3D workstation installations still only use a mouse for user interaction. Surprisingly, the mouse can be quite useful for interacting in three dimensions, despite its obvious two-dimensional deficiencies. In particular, it can be used for picking objects. The position of the cursor on the screen can be used to calculate a ray from the eye which can then be ray-casted into the 3D scene. Whichever object is intersected first is then "picked", and can be designated as a selected object for further operations. The mouse can also be used to move 3D objects in ways constrained to two dimensions. For example, dragging a 3D object within a two dimensional plane can be accomplished easily. Selecting a different mouse button or shift key can change the direction of motion from left-right-up-down to the in-out direction. The mouse can also be used to rotate an object about a fixed center, using a virtual trackball metaphor. This can be thought of as grabbing (with the mouse) the surface of an imaginary glass sphere in which the 3D object is embedded and dragging, causing the sphere to rotate.

5.6.5. Transform Events

The virtual trackball, the Spaceball, the 3D mouse and the dataglove all have something in common: they all generate 4x4 homogeneous transform information. Since the dataglove and 3D mouse are positional devices, they generate absolute transform event values, while the Spaceball and trackball generate relative transforms which we call delta transforms. This allows us to create generic transform events and program our event handlers to respond to these kinds of events without regard for the particular input device.

5.6.6. Transform Event Response

Exactly how to respond to these transform events is a more complicated matter. Depending on the type of interaction metaphor being used, it may be necessary to convert the transform matrix into a particular coordinate system before applying it. Simply manipulating a rigid body in space, for example, can be quite subtle. The input device (say, the Spaceball) is in the coordinate system of the display screen, so it is necessary to convert the transform matrix into viewing coordinates. However, if we want to rotate it, we probably want to do so about the object's own local coordinate origin, not the camera's, so the origin needs to be in local coordinates. Other more sophisticated ways of responding to transform events involve, for example, maintaining some kind of constraints such as through inverse kinematics.

5.7. LEMAN System Design

LEMAN was designed to allow rapid prototyping and experimenting with different types of user-interface style. In practice, when we add new functionality to the system, we usually first add a mouse-based interface to control it, using a slider widget or trackball metaphor, for instance. This can be done quite quickly using the interactive interface building tools of the Fifth Dimension Toolkit. Once this has been tested, we can then easily add other multi-dimensional input devices such as the spaceball, valuators, MIDI keyboard, or dataglove, simply by changing a run-time configuration file. This allows us to quickly change user-interface metaphors for experimentation.

5.7.1. Design Philosophy

Ideally, the entire system would have been written in an object-oriented language. For several reasons, however, the C language was chosen. First, a large number of mathematical, and other types of routine already existed, some public domain, some developed in our lab, which I wanted to reuse in the LEMAN system. Secondly, this software was intended to become part of the overall research software for the lab, and in all probability would be extended and modified by future research assistants, so it was important to work in a language that would be comprehensible to all. Last, a large part of the system code is pure numerical routines, for which object-oriented programming was not particularly advantageous. I had already developed a technique for doing object-oriented programming in C, called MOOC, for the Fifth Dimension Toolkit project, and this worked quite well for the 5D Toolkit, in which most of the application programmers did not create new classes, but simply used the existing classes. Actually creating new classes, and subclassing existing ones, however, is somewhat cumbersome in MOOC, and I didn't want to be too tied down to this particular style of programming, so I decided to build the LEMAN system in straight C.

The LEMAN system code therefore consists of three major components, each with its own style of programming. The first component implements the two-dimensional widgets and input devices and calls the Fifth Dimension Toolkit using the MOOC system. The second component is a set of purely numerical routines to implement all of the low level geometric and physically-based modeling calculations. These numerical routines were written in a traditional, FORTRAN-like style, with large numbers of parameters consisting of simple data structures such as arrays, and not allocating any dynamic memory.

The third system component is made up of a set of C modules organized in a manner similar to classes in an object-oriented language. These "classes" have constructor and destructor routines defined for them so that they can allocate memory to create "instances" which can be arranged in a typically object-oriented run-time data model. A limited form of attribute inheritance is permitted through the means of the macro preprocessor, although there is no message dispatching. This "object-oriented" portion of the code can be considered a higher level layer which is implemented on top of the 5D Toolkit and numerical routine libraries. The LEMAN classes themselves can be grouped into three categories, more or less along the lines of the standard MVC (Model-View-Controller) paradigm.

5.7.2. System Overview

From a run-time point of view, the LEMAN system can be viewed as a collection of distributed processes communicating through an interprocess communication protocol. For reasons of bandwidth limitation over the ethernet local area network, most of the numerical simulation and graphical rendering routines are contained in a single process which runs on a high-performance Silicon Graphics workstation. The other processes, which can be running on other machines, are used to implement much of the user interface portion of the software. This includes the 5D Toolkit widget panels, which provide the two-dimensional user interface component of the system. These panels are constructed interactively using FirstStep, the user interface building tool which comes as a standard application in the 5D Toolkit. Remote processes are also used to collect events from and send events to external input and output devices such as the MIDI keyboard, video recorder (using SMPTE time code) and audio synthesizer.

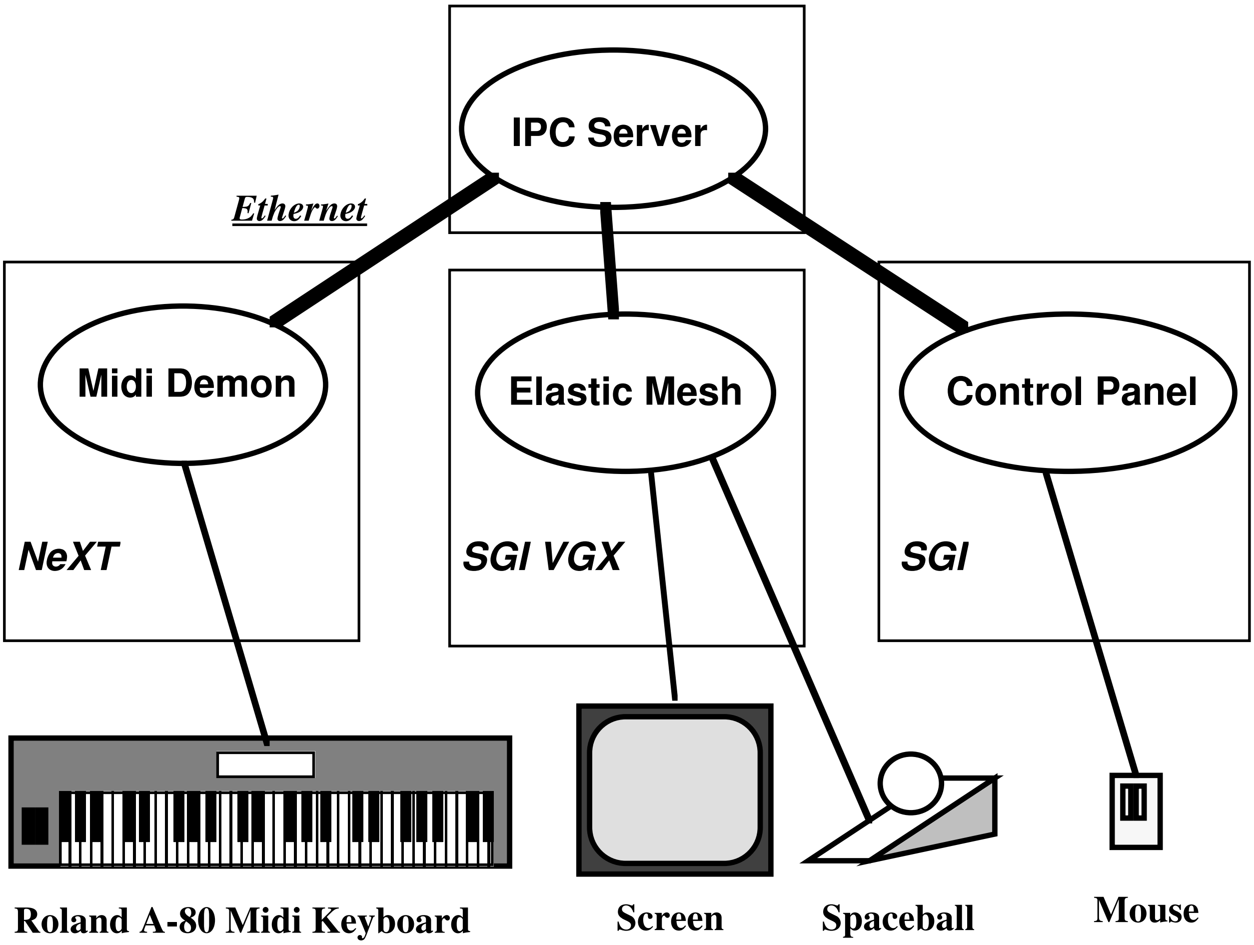

Figure 5.5: Cooperating Processes

5.7.3. IPC Event Message Protocol

All of these processes communicate event information via simple byte streams according to an ascii (i.e. human readable) event protocol. Although this is rather inefficient, it allows easy monitoring of events as they pass between processes. It also allows certain events to be filtered out or altered according to a set of rules specified as a list of regular expression substitutions. These event filters, which are stored as files and loaded into the processes at run time, allow various input devices and widget panels to be bound to certain commands in the main application process without changing the source code. For example, a button with a certain name can be bound to the "quit" event or a particular key on the MIDI keyboard can be bound to a certain control parameter. Events can also be stored as command lines in a startup command file which is loaded in and executed at run time. Both event filter specification and event commands can be stored in the same file, called a command file (.cmd extension) which can actually be executed like a Unix shell script.

5.7.4. File Format

The event protocol can therefore be thought of as a type of (human readable) language that the LEMAN system uses to communicate *events* or control information between *concurrent* processes. There also exists a second type of language, again human readable, for communicating *data* between *successive* processes run at different times, in other words, a save file format. All LEMAN classes which represent some portion of the data model (the M part of the MVC structure) know how to store themselves to an ascii file along with pointers to their referenced instances, so a simple "save" routine called on the top-level object in the data model will recursively store the entire data model. The language is therefore a "flat" listing of each instance, with pointers providing the graph of instance relations. This is more general than a recursive type of language, which can normally only encode simple hierarchy, and allows for any type of general graph of relations to be represented. Within each object instance, attributes are encoded as keyword-value pairs, with the values having any number of parameters. Order of keywords is unimportant and unrecognized keywords are ignored. In this way a certain amount of forward and backward compatibility can usually be afforded between different versions of the file format.

5.7.5. Memory Management

Lack of proper memory management is a fundamental problem of the C language and virtually impossible to solve without redesigning the language itself. Schemes such as reference counts only work for limited types of data model (in particular, directed acyclic graphs or DAGs) and rapidly become more trouble than they are worth for more complicated data models. My approach to the problem, since I am normally working on machines with quite a bit of memory, has normally been to simply not destroy any object that has the possibility of being multiply referenced. I also maintain, for each class, a table of pointers to each allocated instance, so as an extreme measure, I can destroy every instance of a given class. This can normally only be done safely at limited times such as when loading in a completely new data model.

5.7.6. Modeling Classes

The modeling classes represent the actual data model upon which the application program acts, similar to the document concept on the Macintosh. This, in practice, means that the modeling classes are distinguished by alone having load and save routines defined on them. In certain situations, modeling instances may be used to implement portions of the user interface, for example highlight materials or 3D icons, and these normally are not saved. Ideally, in keeping with the fine-grained MVC philosophy, the modeling classes should be completely passive repositories of data, having neither the ability to draw themselves nor the ability to respond to events. This would require separate viewers and controllers for each modeling class. For practical reasons, I have only made this separation of function in a few of the highest-level and most complicated modeling classes.

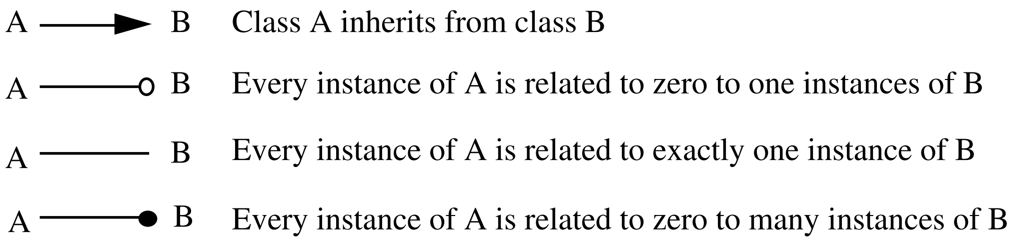

Figure 5.6: Object-Relation Diagrams

The diagram of the modeling hierarchy classes is shown in figure 5.7, using a style of object-relation diagram, described in figure 5.6. The heart of the modeling hierarchy is the node class. This object maintains local and global transformation matrices, front and back surface materials, and a texture, as well as viewing attribute information and joint angle information if it is not a fixed node. The node maintains a list of children, for implementing node hierarchy, and a pointer to a subnode, for maintaining subhierarchies. This is how links are represented, for example. The node can also contain a pointer to a model object. Like the material and texture objects, the models can be multiply referenced, but nodes normally cannot. The material objects maintain information about the node's surface material such as diffuse and specular reflectance. The texture object maintains a pointer to a texture bitmap file. The model object can be one of a variety of shapes, or a general rectangular or triangular mesh. The model class also maintains global deformation parameters.

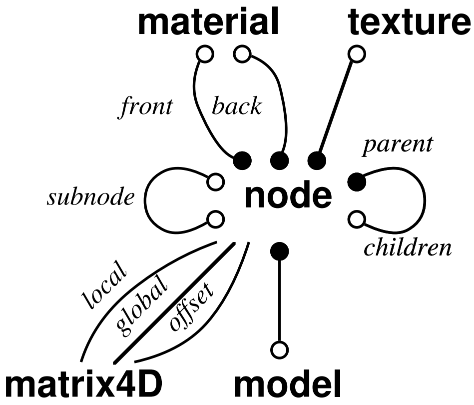

Figure 5.7: Modeling Classes

5.7.7. Maintaining Internal Consistency

Roughly speaking, the entire run-time data model for LEMAN character consists of the following components: For representing the skeleton, a strict DAG containing a hierarchy of node3D instances which may multiply reference lower level instances such as models, materials, and textures. Sitting next to the skeleton DAG, with multiple pointers into it at many levels, are the multitrack instance, one side, and the elastic surface instance on the other. Taken together these constitute a single animated layered elastic character, which may be saved and loaded at will. On top of the modeling data structure rest the controller and view instances, above which lies the single commander instance.

Maintaining internal consistency of such a complicated data model is obviously a difficult task. Ideally, I would like to use some sort of constraint maintenance system such that constraint relationships between objects could be declared, and then maintained using a predefined constraint maintenance algorithm. However, this was not available to me at the time. Therefore, the techniques used for maintaining the internal consistency of the LEMAN data model are by necessary ad hoc. They are not always reliable, and often inefficient, but they get the job done and are reasonably straightforward and simple.

The basic cycle of dynamic behavior in the system is the event loop. Events are removed from the queue and then handled. As each event is handled, data structures are updated as much as is determined necessary for subsequent events to be able to act. At periodic intervals, (e.g. five times per second) a clock tick event occurs which triggers a redraw routine. The draw operation first performs a more thorough updating operation on the entire data model so that it will be prepared for the subsequent rendering operation. When there are no events in the queue, an *idle_loop* operation is called continually as long as no event is found in the queue. This *idle_loop* operation performs a single iteration of the real-time motion control and physical simulation operations, which themselves require the data model to be updated to a certain degree, although not as thoroughly as for rendering.

The most important update operation is on the skeleton modeling DAG. This consists of a tree of node instances with multiple references to model, light, camera, material and texture instances. These latter classes maintain an updated flag which is only set to false when some internal parameter is changed by, for example, some interactive editing operation. It can be very difficult to tell when a node3D instance needs to be updated, however, so no updated flag is maintained. At times when it is necessary to be sure that the hierarchy is up-to-date, the entire node3D hierarchy (or subportions thereof) is updated in a recursive operation. This involves concatenating the path of local transformations to determine the global transformation. The inverse global transformations, which are normally used only in special operations such as inverse kinematics, are only updated on a "need-to-know" basis.

5.7.8. Event Handling

The 5D Toolkit can provide event information in two forms. The first form, which works for locally generated events only, returns the event as a MOOC event object containing an integer flag representing the event type and a pointer to the source object. The second form, available with the IPC enhancements, returns both local and remotely generated events in the form of an event protocol string containing the event information (consisting of type string, source name string, and data string). Since it is string based, this is less efficient, but by programming all event handling according to the IPC event protocol strings, an application program can be made to respond to both local and remote events transparently, and the full power of the IPC protocol and its event filtering are made available.

Event handling in LEMAN is therefore performed by examining these event protocol strings. At the top level, the commander object examines the source string of the event and distributes it to the various view and control instances according to the second level name. Event source strings take the form of hierarchical names, separated by dots, and by convention, the second level name of an event's source object is identified with its destination object's class name. In this way, if I want to send events to the currently selected node3D instance, I can simply build a widget panel with the name "node3D" and any events coming from widgets in the panel will be directed to the node controller object which will distribute them to its current node pointer. In this way, the user interface panels can be designed completely interactively and can be located either locally or on a remote machine.

Once the event has passed down the hierarchy to its target instance, it is handled by the object's *handle_event* routine. At this point, appropriate action can be taken by examining the first level source name, which by convention is the name of the individual widget which sent the event. The can be done either by writing a large compound "if" statement for each possible first level source name or, if the event changes the value of a numerical parameter (in LEMAN the majority of events fit this category), it can be handled automatically by the parameter handling facility of the handler class, from which most LEMAN classes inherit. This facility allows any numerical parameter attribute of a class to be declared at run-time along with an identifying name string. The parameter will then be updated automatically, according to the parameter and event types, when an event bearing its name is handled. This facility makes the job of putting a new control parameter under interactive control a simple two-step matter of declaring the parameter in the object's create routine and interactively placing an appropriate widget in the control panel.

5.7.9. Drawing

Before the rendering operation can be carried out, the entire graphical model must be updated completely. First the skeleton hierarchy is updated, starting from the root node. At each node, the current joint value is clamped to be within the bounds of its minimum and maximum values. Then the offset transform is determined from this value, depending on whether it is a rotational or translational node. The local transform is then premultiplied by the offset transform to determine the final transform. This is then postmultiplied by the global transform of the parent node to determine the global transform. The node then updates any modeling instances it may contain, which usually consists of updating the vertices and normals of its polygonal representation if any model parameters have changed. Next the elastic surface object is updated, which consists simply of recalculating its normal vectors, since any change to the elastic surface vertices is implemented by the evolve simulation routine, and is assumed up-to-date. The *draw* routine is then called on the node viewer and elastic surface viewer, which render the skeleton and skin surface respectively according to the current viewing parameters such as wireframe, highlighting selected nodes, etc.

5.7.10. Motion Control

Simple forward motion control of the skeleton can be performed by changing the angle of a single joint using a slider in the node controller window. The next update routine with therefore automatically do the forward kinematics calculations. Inverse kinematic motion control is implemented by the node controller object, using the delta_ik routine. This routine takes an end-effector node as an input parameter and constructs a kinematic chain from it to the node controller's current kinematic root node. It then allocates memory for and fills in the components of the Jacobian matrix. Each time the end effector is moved by a small amount, the Jacobian is recalculated and a differential joint motion is computed.

5.7.11. Attribute Inheritance

Node attribute inheritance is determined at draw-time, rather than update time. The reason for this was to allow multiple views of the same hierarchy with different viewing attributes. When the draw routine is called by the node view object, it passes along a pointer to an inheritance callback routine. The inherited attributes of each node are then determined by walking up the hierarchy until all undefined attributes have been resolved. At each level in the hierarchy, the inheritance callback routine is called which can override the inheritance for special nodes such as the currently selected node. This mechanism is used for highlighting special nodes and changing the colors of subhierarchies accordingly. It is also used interactively for selectively changing the visual attributes of portions of the hierarchy. This could probably be better implemented using a separate attributes object which would be used to determine the attribute inheritance at update time, if a suitable way could be found to allow multiple views.

5.7.12. Inside-Outside Tests

Calculation of the reaction constraints is broken up into two stages. First, the entire elastic surface rectangular mesh is inside-tested against the skeleton model shapes to see if any of the points are within the muscle layer surfaces. If any of these points are found to be inside, a rectangular mesh of gradient vectors is returned, pointing towards the muscle surface. Then this gradient array is passed on to the reaction constraint routine. This routine takes the gradient array, together with the array of external forces, and calculates a new array of forces which enforce the reaction constraint.

The inside/outside test is performed by the `inside_` routines which are defined on the node and model classes. When called on the root node of the character skeleton, this routine takes the array of elastic surface points and recursively inside-tests it with each of the models in the skeleton hierarchy. Within the model inside routine, the surface array is first transformed into local coordinates, then (if there are any deformations present) into the undeformed coordinate system.

At this time, the actual intersection test is performed for each point in the array against the surface shape. For spheres and cylinders, this is a simple radius test. For superellipses, this involves one invocation of the inside/outside function for each point. If this function is less than one, the point is inside the superellipse. The constraint gradient is then estimated by calculating the gradient value of the i-o function and multiplying by the difference of the i/o function from one. This gradient is then transformed back into local deformed and then global coordinates (making sure to use a covariant transformation) to yield the final constraint gradient vector array. This array, which essentially identifies each point which doesn't meet the constraint along with a direction in which to move to attempt to meet the constraint, is then ready to be passed on to the reaction constraint calculating routine.

5.7.13. Picking

The selection operation is usually done with the right mouse button and is implemented using the picking routines. Picking, that is finding out which object the cursor is over, is performed exclusively by ray-tracing. First the x and y position of the cursor in pixels is determined and from it a ray in screen coordinates is constructed. This ray is then converted to global coordinates based on the current viewing matrix and perspective projection. The ray is then passed down the skeleton node hierarchy, being transformed into local coordinates at each node, and intersection tested against each model primitive. The closest intersection point, i.e. with the smallest ray parametric coordinate, is returned along with a pointer to the node and any local model information about the point such as its u-v coordinates. The global position of the intersection point can be determined from the global ray. The skin rectangular mesh is also tested for intersection by exhaustively testing each polygon in the mesh. The same ray-tracing routines are also used by the automatic binding algorithm for finding near-by attach points on the muscle surface.

5.8. Analysis

The LEMAN system was designed not only to test new 3D character models, but also to see if the types of direct manipulation interaction metaphor used in 2D could be generalized to a complex 3D domain such as character animation. Despite the greater sophistication of three-dimensional application domains, and the resulting complications involved in designing as well as using such software systems, the results of these experiments are definitely positive. Almost every type of 2D direct-manipulation technique can work as easily in the three dimensional domain. By using the two-handed "ball-and-mouse" metaphor, the user can use the left hand (for example) to position the character in any desired manner for selection or manipulation by the right hand. This effectively extends the dimensionality of the mouse as an input device, since it can select any exterior point on the character or move in two-dimensions within any arbitrary plane selected with the spaceball. Since much more information can be packed into a three-dimensional volume than onto a two dimensional plane, we can see that direct manipulation metaphors in three dimensions have the potential to make more efficient use of available screen space and increase the bandwidth of the human computer interface.

Direct manipulation of a 3D object can be contrasted, in terms of information content, to using widget-based interaction with many overlapping windows. In both cases, the information is stored in three dimensions and the screen simply shows a two-dimensional slice through the data. In the overlapping windows case, one must search the three-dimensional space by laboriously popping up different windows until the desired one is found. In the 3D direct-manipulation case, we are able to view a perspective rendering of the three-dimensional information space and move about freely within it, observing it from any angle. Since this is closer to what we do in the real world, we are better equipped to interact with a computer in this way.

A second, and somewhat surprising conclusion arrived at from building and using the LEMAN system is the interactive modeling capabilities presented by direct manipulation of a physical simulation in real time. The original attraction to physically-based techniques was because of its potential to enhance the quality of animation by making it more natural. It was therefore surprising to find out that it also made interaction more natural, and often made what would have been complex interactive tasks quite simple. For example, one important interactive task in building a layered elastic character is binding the skin to the muscle layer. This requires finding a mapping from every point on the elastic surface to a point on the muscle layer, a layer which has not topological similarity at all with a rectangular mesh surface. Using the LEMAN system, however, this turned out to be quite a simple task, as long as the physical simulation is running. By wrapping the elastic surface around the muscle layer initially and shrinking it with the reaction constraints turned on, the skin surface eventually finds an energy minimum where it is more-or-less equally distributed around the muscle layer. Binding therefore simply consists of attaching each skin mass point to its nearest perpendicular muscle surface point. By adjusting the position of the articulated figure when the binding operation is done, different types of skin attachments can be made. This kind of physically-based CAD technique may have considerable potential not only in animation, but in other fields as well, for example in general surface modeling.

These experiences reinforced the belief that layered elastic models, such as the elastic surface layer model, are a promising approach to constructing animated three-dimensional characters. By using 3D interactive techniques to manipulate the physical model as the simulation progresses, an animator can rapidly build and animate complex characters. Making the computer a genuinely useful and creative tool in the hands of a character animator requires a variety of modeling techniques combined in the right way and manipulated using the proper interaction metaphors. How to do this can only be determined by experimenting with the various possibilities. By building test systems such as LEMAN, in which various types of interactive construction and animation techniques can be explored, practical software tools for creating expressive character animation can be built.

The LEMAN system is a prototype, however, and does not yet present a practical system for 3D character animation. One of the main limitations, in particular, is the finite difference mesh, which has topological restrictions making it difficult to create surfaces with thin appendages (like arms and legs). Moving to a finite element discretization should remove these restrictions. Addition of self-collision detection to the skin would allow greater deformations at the joints and more pronounced wrinkling. Adding dynamic properties to other layers such as the fat and muscle layers would also enhance realism, as well as using some more advanced skeleton animation methods.