1. BUILDING AN INTERACTIVE 3D ANIMATION SYSTEM

Enrico Gobbetti
Francis Balaguer
Angelo Mangili
Russell Turner

Computer Graphics Laboratory
Swiss Federal Institute of Technology
CH-1015, Lausanne, Switzerland

1.1. INTRODUCTION

The continued improvement and proliferation of graphics hardware for workstations and personal computers has brought increasing prominence to a newer style of software application program. This style relies on fast, high quality graphics displays coupled with expressive input devices to achieve real-time animation and direct-manipulation interaction metaphors. Such applications impose a rather different conceptual approach, on both the user and the programmer, than more traditional software. The application program can be thought of increasingly as a virtual machine, with a tangible two or three dimensional appearance, behavior and tactile response.

Dynamic graphics techniques are now considered essential for making computers easier to use, and interactive and graphical interfaces that allow the presentation and the direct manipulation of information in a pictorial form is now an important part of most of modern graphics software tools. The range of applications that benefit from this techniques is wide: from two-dimensional user interfaces popularized by desktop computers like Apple's Macintosh or the NeXT machine, to CAD and animation systems that allow the creation, manipulation and animation of complex three-dimensional models for purposes of scientific visualization or commercial animation. Future possibilities include the latest virtual environment research that permits an even more intuitive way of working with computers by including the user in a synthetic environment and letting him interact with autonomous entities, thanks to the use of the latest high-speed workstations and devices.

In this chapter we will present the design and implementation of an interactive key-frame animation system based on these dynamic graphics techniques. Key-frame animation is the standard technique used to generate most current commercial computer animation. In its simplest form, key-framing consists of placing the 3D objects to be animated in successive *key* positions and then interpolating some form of spline to generate a smooth motion. One major problem with building key-frame animation systems is that it is difficult to make them easy to interact with and control. We therefore set as a primary goal of this project the achievement of a direct manipulation system in which the user would at all times be able to see, move around within and interact with the animation environment in real time.

1.2. KEY-FRAME ANIMATION

Computer animation is closely related to movie making. A scenario describes the story, defines the actors, how they interact, and what emotions the film has to convey. The scenario is subdivided into a succession of scenes and a story board is realized giving the first general impression of the visual appearance of the film. Once these steps are accomplished, the animator can define the animation for each of the scenes of the movie. Then the animation is rendered and recorded frame by frame to video tape or film.

One way to define an animation is to use some kind of key-framing system, where the animator defines the animation of each actor independently by specifying key values over time of the parameters to animate and the system interpolates those keys to generate the in-between values of the parameters. This technique is inspired from the way traditional commercial animations are done and is therefore fairly close to the way animators are used to working. A large spectrum of other techniques for creating computer-animated movies exists: from scripting, where the animation is defined by a program in a specialized high-level language, to simulation, where animation results from the evolution of a world's model according to some predetermined laws. However, all these other techniques can be in some way be considered as different front ends to a key framing system.

The production of good quality animation by direct use of a key framing system requires that the animator be able to control all the finer details of the scene and the motion in a very interactive way, so it is possible for him to refine his work as many times as needed. The first step of the animator's work is the scene composition. Components of a scene are the animation camera, a set of lights of various types, and a set of geometric shapes. The animator must be able to specify the position, orientation and scaling of each component of the scene, to specify any other parameter of any type of object (such as the angle of view of the camera and the color of the lights) and to organize these components into a hierarchy.

Once a scene is assembled, the animator can start to work on the definition of the motion. The animator describes the animation on an object by object basis. For each object, a sequence of key values of its parameters is specified and the system generates the in-between values using some interpolation technique. Each animation parameter must be stored in a separate track to perform various kinds of interpolation. The animator must be able to choose the kind of interpolation he desires depending on whether he wants continuity of speed and acceleration or not. The interpolation of key values over time is, however, a rather ill-defined problem. The animator has to choose one type of curve among many, and the first choice is not generally the right one to generate the kinds of paths or timing desired by the animator.

Therefore an interactive key-frame animations system must provide real-time playback for the animator to be able to evaluate the animation from various points of view. The first animation is generally a first draft and the animator must be given very interactive tools to edit and refine through many iterations the curves which define the animation and the timing.

1.3. DESIGN APPROACH

The design and implementation of dynamic graphics applications such as the interactive key-frame system described here are complex tasks: these systems have to manage an operational model of the virtual world, and simulate its evolution in response to events that can occur in an order which is determined only at run time. In particular, this kind of application must be able to handle the multi-threaded style of man-machine dialogue that is essential for direct manipulation interfaces, and makes extensive use of many asynchronous input devices (from the ubiquitous keyboard and mouse to more sophisticated devices such as the SpaceBall or DataGlove).

The relevance of object-oriented concepts for solving this kind of problems has already been noted by several authors (see [Gobbetti et al, 1991] and [DeMay, 1991] for a discussion of this). From a software-engineering point of view, dynamic graphics applications have many aspects that can benefit from object-oriented techniques: for example, data abstraction can be used to support different internal data representations, several graphics drivers can be encapsulated in specific objects, different subclasses can offer the same interface for the manipulation of graphical objects, and information distribution can help to manage the inherent parallelism of dynamic graphics programs. From the point of view of a user, the direct manipulation metaphor of modern dynamic graphics programs is consistent with the fact that object-oriented applications are designed so to manipulate objects related in type hierarchies. All these considerations led us to the choice of an object-oriented technology for the design and implementation of our key-frame animation system.

Prior to this project, we already had another experience building object oriented software. This was a user interface toolkit developed on Silicon Graphics workstations using a custom-made object-oriented extension to C based on the concepts of Objective C [Cox, 1986]. This experience was successful but also showed us the limitations of using a hybrid language for the implementation of an object-oriented design. Because the language was hybrid, it was difficult to completely enforce the object-oriented paradigm, and programmers sometimes mixed procedural and object-oriented styles, obtaining in this way components that were hard to reuse. Furthermore, the language provided no multiple inheritance, no static typing and no genericity, limiting in this ways its expressiveness and sometimes influencing our design decisions. For example, code duplication was necessary in cases where multiple inheritance should have been used and many errors showed up only at run-time on the form of features being applied to objects that were not supposed to implement them. Another big problem with this system was its lack of garbage collection, which required us to spend inordinate amounts of time chasing memory bugs (dangling pointers, memory never being deallocated, etc.) and devising complex algorithms to destroy object structures.

All of this convinced us of the importance of using a pure object-oriented language for our work, and Eiffel [Meyer, 1988] was chosen for this new project because of its characteristics (multiple inheritance, static typing, dynamic binding, and garbage collection) that corresponded to our needs. Our previous experiences convinced us of the appropriateness of object-oriented technology for building libraries of reusable components. What was interesting for us when starting to work on the project of building the animation system was to know if it is possible to use this approach for creating *applications* and if reusable components can be created as a side-effect of using object-oriented techniques during the development of a program. Furthermore, we wanted to test if it was possible to use a pure object oriented language like Eiffel for the creation of an application with such large constraints on performance as an interactive animation system.

1.4. SYSTEM DESIGN

1.4.1 Identification of principal class clusters

To promote code-reuse and to allow several people to work on the project simultaneously, class libraries had to be designed so to provide efficient, reusable and extensible components for composing applications. At the beginning of the design process, a large amount of time was spent up front in group discussions about the design of the system. The problem was split into several principal clusters to be developed in parallel by different people, with each cluster carried through the design process to implementation.

The identification of these clusters was based on an analysis of the problem statement and on our previous experience designing other graphical systems. The practice of starting with the creation of the basic libraries instead of starting to directly analyze the functionalities required is typical of the bottom-up approach that object-oriented design favors [Meyer, 1987]. The clusters we selected for further design by this preliminary analysis were:

• a **mathematical cluster**, to contain useful basic mathematical abstract data types;

• a **spline cluster**, to provide means to animate these basic data types;

• a **modeling cluster**, to represent the various components of graphical scenes;

• a **rendering cluster**, to provide several rendering facilities;

• a **dynamic cluster**, to provide ways to encode interactive and animated behavior;

• a **user-interface cluster**, to provide standard interaction widgets and devices.

We find it very useful to describe the structure of our class clusters in term of object-relation diagrams, as in [Rumbaugh, 1991]. The definition of the static structure of our software is always the first step of our design process, and these diagrams allow us to give a schematic presentation of this structure that combines within the same diagram instance relations and inheritance relations. These diagrams have thus become for us a standard way for exchanging our ideas during the design meetings about the creation of our class clusters.

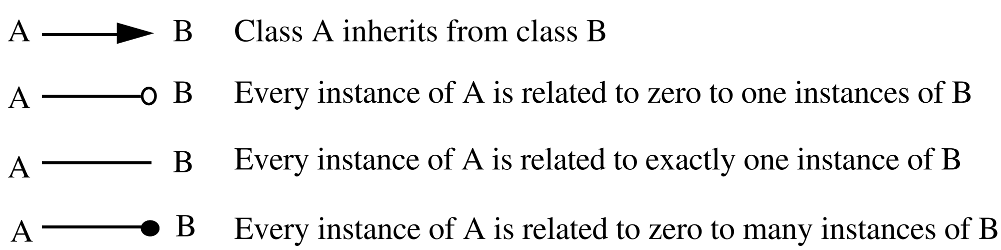

Figure 1. Object-Relation Diagrams

Object-relation diagrams are the only formal technique we use for describing our design process: no other types of diagrams or special notations were used during the design of the key-frame animation system. We find these diagrams useful but we do not insist too much in using other formalized schemes (data flow diagrams, for example) for representing other important aspects of our software. These other techniques could be of some use (especially if used in conjunction with some tool for keeping all these other representations up-to-date with respect to the code), and we will perhaps consider them in the future. Eiffel, with assertions and invariants, offers a way to represent the programming contracts between the different components of the system and their required behavior. Fragments of Eiffel code are therefore sometimes used by us quite early in the design in place of pseudo-codes or data-flow diagrams to define the functional model of some components.

In the next section we will give, for each of the clusters composing the animation system an overview of the reasoning that guided its design, analyze its structure and discuss the results obtained in terms of performance and reusability. The class libraries as presented here are the result of multiple iterations of our design process.

1.5. OVERVIEW OF THE PRINCIPAL CLUSTERS

1.5.1 Mathematical Cluster

1.5.1.1 Analysis

The mathematical foundation of the animation system, as for most non-trivial graphics applications, comes from linear algebra. The first step in the design was to provide an encapsulation of the basic concepts of this branch of mathematics.

Matrices and vectors are two of the most important abstractions found in graphics programming. Three-dimensional vectors, are needed to store and manipulate information such as the vertices of a faceted object or its normals, and four-dimensional matrices are the most common technique for representing the projective transformation defined by a virtual camera and the linear transformations used for positioning, orienting and scaling graphical objects. So, a MATRIX and a VECTOR are usually standard classes in any object-oriented graphics system, but most graphical libraries do not offer more than these two algebraic data types.

Four-dimensional matrices can be used to represent affine transformations (linear combinations of scaling, translation and rotation operations). However, not all instances of a four-dimensional matrix are affine transformations. Furthermore, this representation is not the only possible one and its exclusive use can make it difficult to perform some important operations like interpolating between transformations, a feature that is definitely needed for an animation system! What an animator usually wants to do when interpolating between two affine transformations is to separately interpolate their components (the rigid body transformations, translation and rotation, and the scale factors of the three axes). It is therefore important to be able, at any moment, to extract these parameters from a transformation.

For these reasons, we decided to include in our system a class of transformation objects able to consistently maintain and manipulate their four-dimensional matrix representation and their translation, orientation, and scale components through a well defined interface, thus ensuring, through a proper set of assertions, the affine nature of the transformation. Two of the transformation's components, translation and scaling, can be easily represented as three-dimensional vectors, and no new class needs to be added to the system. There are, however, several different possible solutions for representing rotations, the more frequently used being quaternions and Euler angles. Quaternions are, in fact, four-component homogeneous coordinates for orientation, and they are now becoming increasingly prevalent in animation systems because of their well-behaved mathematical properties ([Shoemake, 1985] discusses several issues related to this topic). A class for representing quaternions was therefore included in our design for a mathematical cluster.

The other type of basic mathematical entity that is usually found in graphical software is color, and a COLOR class is included in our system. Although colors are often represented as Red-Green-Blue triplets, there exist other representations and the color class has the ability to convert between these.

By making all these types of mathematical objects available to the designers and implementors of the key frame animation system, we aimed at providing a solid set of abstractions to be used together with the other basic Eiffel classes [ISE 1989] in order to simplify the task of writing large dynamic graphics applications.

1.5.1.2 Object Model

The following diagram shows the most important classes of our mathematical cluster and their relations (classes in parentheses are deferred):

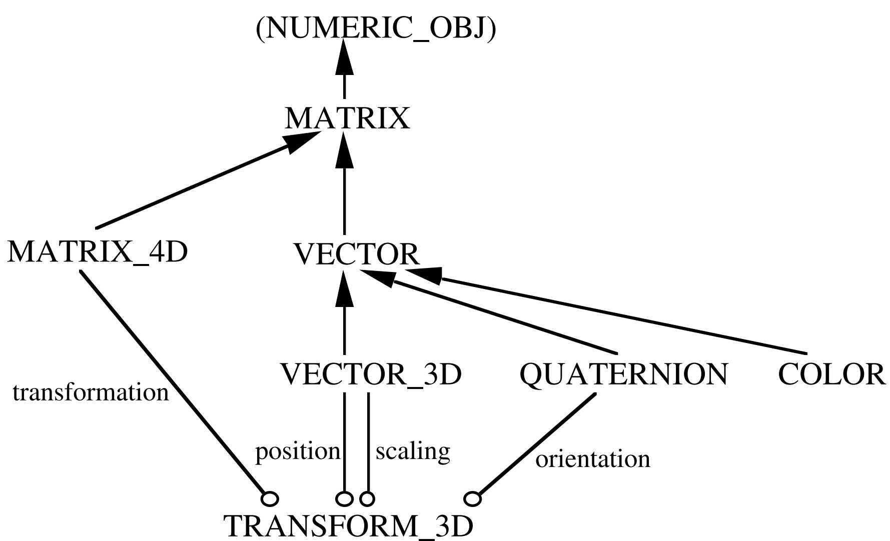

Figure 2: Mathematical cluster

The NUMERIC_OBJ deferred class defines objects on which the usual arithmetic operations such as "+", "-", "*", "/" are defined and provides a common interface for all these features. Most other classes of this cluster inherit from it, covariantly redefining features to ensure correct typing. A finer distinction between different types of numerical objects with respect to the behavior of the operations applied to them could be made, and NUMERIC_OBJ should perhaps be separated into different inheritance levels to represent fundamental mathematical types. However, we felt that the effort for designing this kind of more formal set of abstract classes was too great.

Instances of the MATRIX class can represent arbitrary rectangular matrices, and vectors of arbitrary dimension can be represented by instances of the MATRIX's heir VECTOR. Features for performing all sorts of computations, from basic arithmetic operations to solutions of arbitrary linear systems, are exported from these classes. In this design vectors are rectangular matrices with one of the two dimensions equal to one, consistently with the mathematical definition of the concepts they represent, and a proper class invariant of VECTOR ensures that this basic property is always verified. In this way, most of the operations on vectors are just inherited from matrices, and the system is able to distinguish between row vectors and column vectors.

Since four-dimensional matrices and three-dimensional vectors are widely used in graphical programming, the two classes MATRIX_4D and VECTOR_3D are provided with additional operations and more specific and faster implementations of ancestor's features. The concept of color, linear transformation, and quaternion, mentioned in the analysis, are represented by the COLOR, TRANSFORM_3D, and QUATERNION classes respectively.

1.5.1.3 Object Design

The object design of such basic classes as the ones present in this cluster is very important. These kinds of objects have to be used as basic building blocks for the rest of the system and it is therefore very important to have them both simple to use and highly efficient to improve the performance of the applications.

In order to provide regular interfaces, and therefore improve the usability of our classes, we follow some conventions for naming features. Some of these conventions are:

- features that return an alternate representation of an object always begin with *as*, as for *as_quaternion: QUATERNION* which can be applied to instances of MATRIX_4D;

- features that modify the value of *Current* as a result of some computations based on the parameters always begin with *to*, as in *to_sub(left, right:* **like** *Current)*, which can be applied to instances of NUMERIC_OBJ;

- features that store the results of their computations in one of their parameters always contain the indication *_in*, for example *row_in(i: INTEGER; v: VECTOR)* which can be applied to instances of MATRIX.

As a result of these conventions, the class features use as often as possible the style of modifying *Current* or one of the parameters instead of returning a new object as a result. In this way, it is the client's responsibility to allocate all the objects needed, therefore reducing the overhead due to allocation and collection of unnecessary temporary objects.

All the routines and the classes of our libraries are enriched with a set of preconditions, postconditions and invariants that specify their programming interface contract. We put some effort into defining these assertions, because we believe that they are a key element for promoting reusability. Assertions formally define the behavior of our components (and therefore we use them during the design phase), provide a certain form of documentation and help during debugging. Another interesting fact we noted about assertions is that their use helps to produce efficient software: by clearly defining the responsibilities of each component, we can avoid using defensive programming techniques, and therefore obtain more readable and at the same time more efficient code.

In order to implement the mathematical operations defined in the various classes of this cluster, and especially the MATRIX and VECTOR classes, we decided to use two standard Fortran libraries that are well known in the scientific programming community: BLAS [Lawson et al., 1979] and LAPACK [Anderson et al., 1990].

BLAS, or Basic Linear Algebra Subprograms, is a set of public domain routines able to perform basic numerical computations on one and two-dimensional arrays interpreted as vectors and matrices. This set of routines operates at a very low level and represents a general purpose kernel library. Highly optimized versions exists on specific machines (for example, they're now included in the Cray SCILIB).

LAPACK, or Linear Algebra Package, is designed as a portable and efficient linear algebra package implemented on top of BLAS. It provides routines for solving systems of simultaneous linear equations, finding least-squares solutions of overdetermined systems of equations, and solving various eigenvalue problems.

These two libraries offer high-quality implementations of numerical algorithms but are rather difficult to use in application programs because of the flat structure of these packages and the fact that the client of these routines has to take care of maintaining all the state information in auxiliary data structures between different calls (for this reason, some of the routines have tens of parameters!). These drawbacks are typical of libraries developed using standard procedural languages.

These two libraries are encapsulated as Eiffel classes whose routines are simple interfaces to the Fortran ones. All these routines have preconditions and postconditions that define their contract, ensuring that these external operations are properly used. For example, the Eiffel interface to BLAS routine *samax* is

```
samax(n: INTEGER;  sx: ARRAY[REAL];  lowx: INTEGER;  incx: INTEGER): REAL is
    -- REAL FUNCTION SAMAX(N,SX,INCX)
    -- returns the maximum absolute value of all entries in SX.
    -- N     Input, INTEGER N, number of elements in SX.
    -- SX    Input, REAL SX(N), vector to be searched.
    -- INCX        Input, INTEGER INCX, increment between elements of SX.
    --             For contiguous elements, INCX = 1.
    -- SAMAX  Output, REAL SAMAX, the maximum abs. value of all
    --             entries in SX.
    require
        n_non_negative: n>=0;
        sx_exists     : not sx.Void;
        sx_low_valid  : in_int_range(sx.lower+lowx, sx.lower,
                                     sx.upper);
        sx_high_valid : in_int_range(sx.lower+lowx+incx*(n-1),
                                     sx.lower, sx.upper);
    external
        C_samax: REAL name "samax_" language "C";
    do
        Result:= C_samax(@n, sx.to_c+Word_size*lowx, @incx);
    ensure
        -- Result is the maximum absolute value in the specified range of ofsx
    end; -- samax
```

These encapsulated routines are then used in the various classes of the mathematical cluster as internal implementations of various higher-level operations, Eiffel being used as a packaging tool that offers all the benefits of a pure object-oriented language. State information required by these routines is handled inside mathematical classes and these details are hidden to the client of the mathematical cluster.

We believe that this ability to reuse components written in other languages is a key feature of Eiffel. After all, isn't one of our main goals reusability? Many years of programming efforts have been spent in developing and testing large software libraries in various languages, and some of these libraries, like BLAS and LAPACK, provide a set of functionalities that is really worth reusing. Languages such as C++ and Objective-C try to encourage this reuse by allowing the programmer to continue developing their software in hybrid languages. However, we feel that this approach does not enforce a clean separation between the object-oriented and non-object-oriented portions of the software and therefore perpetuates the software engineering problems of the traditional languages.

The Eiffel approach, on the other hand, is to define clean and localized interfaces with external components, without making any compromise with the object-oriented paradigm the language is based on, obtaining in this way the best of these different worlds. In our case, the ability to reuse LAPACK and BLAS was an important aspect in our development. A lot of functionality was added in a relatively small amount of time and high performance was immediately obtained thanks to the use of these routines, without being forced to change the object-oriented nature of our mathematical components. Our mathematical library is just a starting point, and we hope that in the future, more work will be done for providing these kinds of tools to Eiffel developers.

1.5.2 Representing Animation

1.5.2.1 Analysis

Animation can be defined as the description of the evolution of some parameters over time. We decided to build an interactive key framing system where this evolution is described by specifying key points at certain times, and an interpolator computes the in between values. Having described how to represent the basic mathematical entities needed by the system using our mathematical cluster of classes, the next problem that we need to attack is the definition of their variation over time using interpolation techniques.

The simplest interpolation technique is linear interpolation. However, this presents the major disadvantage of discontinuities of speed and acceleration at the key points. Another technique is spline interpolation. Splines are curves described as a succession of polynomial segments defined over some parameter. These polynomials can be of any degree, but generally in animation, polynomials of degree three are used. We speak, in that case of cubic splines.

Various families of spline curves exist. The most commonly used in the field of computer animation are Cardinal Splines and the Kochanek-Bartels superset . They both interpolate through the control points defining the segments. The major interest of the Kochanek-Bartels splines is that segments are defined by starting and ending points and starting and ending tangents. This allows easy definition of the curve, and permits the creation of local discontinuities by specifying different ending and starting tangents. Kochanek-Bartels and Cardinal splines are only C1 continuous, meaning they don't offer continuity of acceleration, which limits the control you have on the timing.

Another important family of splines are Basis splines, commonly called B-splines. B-splines offer global parameterization on the whole curve and their cubic form offers C2 continuity, i.e. continuity of speed and acceleration (more generally a B-spline of order k is a curve defined by polynomial segments of degree (k-1) and that presents C(k-2) continuity). Hence we can defined very smooth paths but also create discontinuities at a point. Their major drawback is that B-splines do not interpolate through the control points and therefor an interpolation layer has to be built on top of them.

A B-spline combined with an interpolation layer is a very versatile tool. Its mathematical formulation is very general, allowing us to use genericity to build parametric curves of anything as a function of time or of any other parameter. These curves can be of any degree offering from linear interpolation to high continuity interpolation. The interpolation layer and the capability to go back and forth between the approximation and interpolation representation provide a very flexible tool for the programmer and designer.

In the case of computer animation, the spline curve represents the history of one parameter. As the timing is generally different for each parameter (translational parameters of a camera won't generally evolve over time in the same way as the field of view, for example) it is necessary to provide history for each parameter independently. We use in this case the term *track*.

For these reasons, we decided to create the B_SPLINE class and a subclass which offers the interpolation interface to the curve for the programmer. As pointed out earlier, the animation system has to deal with the evolution over time of several types of objects (affine transformations, colors, etc.) In order to be able to represent this fact, we decided to design subclasses of our mathematical objects which are able to store a history of their value. We call these objects animated variables. An animated variable is defined as being a variable of some type having some history information storage mechanism, which we decided to build on top of B-splines curves. Animated variables can be of any type as long as interpolation has some meaning on this type.

1.5.2.2 Object Model

The animation cluster is composed of a set of classes that are used to represent the animated variables and the history storage information.

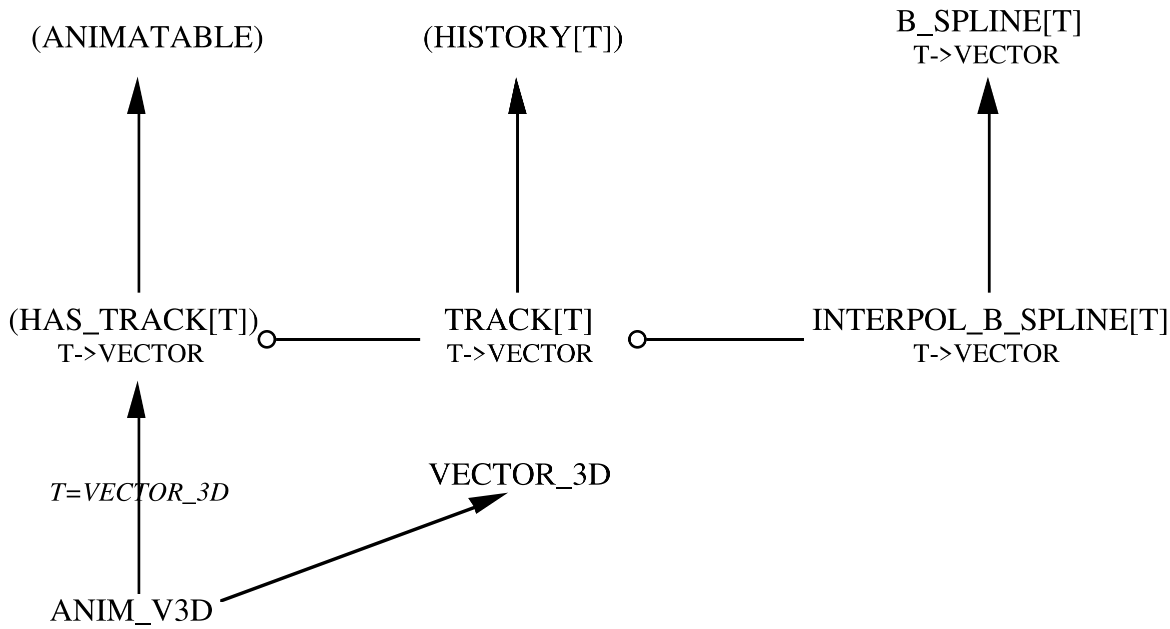

Figure 3: Basic animation classes

Three basic sorts of class can be identified:

- **B-splines** (subclasses of B_SPLINE) represent parametric B-splines of any dimension and any order defined over a knot vector giving the distribution of the parameter along the curve and a control polygon. This is an approximation curve. An important subclass is the INTERPOL_B_SPLINE class which offers the interpolation interface to the approximation curve. The basic mechanisms to go back and forth from approximation and interpolation representations are implemented here.

- **histories** (subclasses of HISTORY) represent the history storage mechanism. A history is defined as a time ordered list of key values which is traversable over time. An important subclass is the TRACK class where B-spline curves are used to interpolate between the key values.

- **animatable variables** (subclasses of ANIMATABLE) are variables having a time dependent value. They have a current value and a current time and mechanisms to update and retrieve information from their history. An important subclass is HAS_TRACK which implements the history using a TRACK.

In this architecture, history mechanisms and history storage are decoupled. A B-spline curve should not make any assumptions that the parameter used is time. On the other hand, a history should not make assumptions about how the history is stored. Intermediary classes such as TRACK makes the link between history mechanisms and information storage. On the other hand, for animated variables, the type of the variable and the type of history storage used is also decoupled. This allows us to define, for example, an animated quaternion whose history is defined over a track of three dimensional vectors or an animated transformation defined as an aggregate of other animated variables.

1.5.2.3 Object design

The use of interpolation to compute the in between values imposes two constraints on us. The first is that any animated variable should be interpolable. The second constraint is that , as we use parametric B-spline interpolation, a multi-dimensional variable should be interpolable on each dimension independently.

Object oriented languages such as Eiffel offer a way to express these constraints in the classes by constraining the generic parameter. In order to perform animation, we must record a history and then play it back. History is recorded by specifying a list of key values which are interpolated for playback. However, some manipulation may be necessary before yielding the desired value at a given time. Hence, animated variables behave as windows on their history: setting the current time moves the window and changing the value of the variable changes the content of the window.

These general mechanisms are implemented in the deferred class ANIMATABLE. A *store* feature transmits the value of the window to the history, while an *update* feature sets the value of the window to the value of the history at the current time. A *set_time* feature makes the window move along the history.

When an *update* feature is applied to an animated variable, it applies the feature *value_in* to its history. In the case where the history is an instance of the TRACK class, the *value_in* feature is applied to instances of the B_SPLINE class, causing the evaluation of the polynomial function defining the segment for the current time. This choice of implementation led to the introduction of a POLYNOM class (a subclass of NUMERIC_OBJ) which implements symbolic polynomial calculus and evaluation. The problem of evaluating the history at a given time is then reduced to the search for the proper segment of the curve and the evaluation of the polynomial defining the segment.

1.5.3 The Graphical Model

1.5.3.1 Analysis

Since dynamic graphics applications must be able to respond to asynchronous input events as they happen, designers have to build their programs with little knowledge about when and in what order the events will occur. Therefore, at any moment during the execution of a dynamic application the entire state has to be explicitly maintained in a global data structure.

So, one of the first steps in building a graphics system is to design this data structure which is called the graphical model. This model consists of the set of classes used to represent the virtual objects to be manipulated. It is worth noting that rendering is not the only operation that needs to be done on these objects. In an interactive key-frame animation system, for example, graphical objects need to be animated and dynamically controlled in their shape and appearance. Thus, many different aspects have to be considered when designing the graphics model, which is implemented in the graphical cluster.

Several class structures have been proposed in the literature for representing three-dimensional hierarchical scenes. These designs strive to provide good encapsulations of the concepts useful for modeling, rendering and animating the types of complex objects that are necessary for dynamic graphics applications. Examples are proposed in [Fleischer et al, 1988], which describes an object-oriented modeling testbed, [Grant et al, 1986], which presents a hierarchy of classes for rendering three-dimensional scenes, and [Hedelman, 1984] which proposes an object-oriented design for procedural modeling.

Key-frame animation systems are typically concerned with the animation of models arranged in a hierarchical fashion: a transformation tree defines the position, orientation, and scaling of a set of reference frames that transform the space in which graphical primitives are defined. This kind of hierarchical structure is very handy for many of the operations that are needed for modeling and animating a graphical tree (see [Boulic at al, 1991]): objects can be easily placed one relative to another and the simulation of articulated figures, which is essential for any type of animation work, can be done in a natural way. Figure 4 shows an example of hierarchical scene representation.

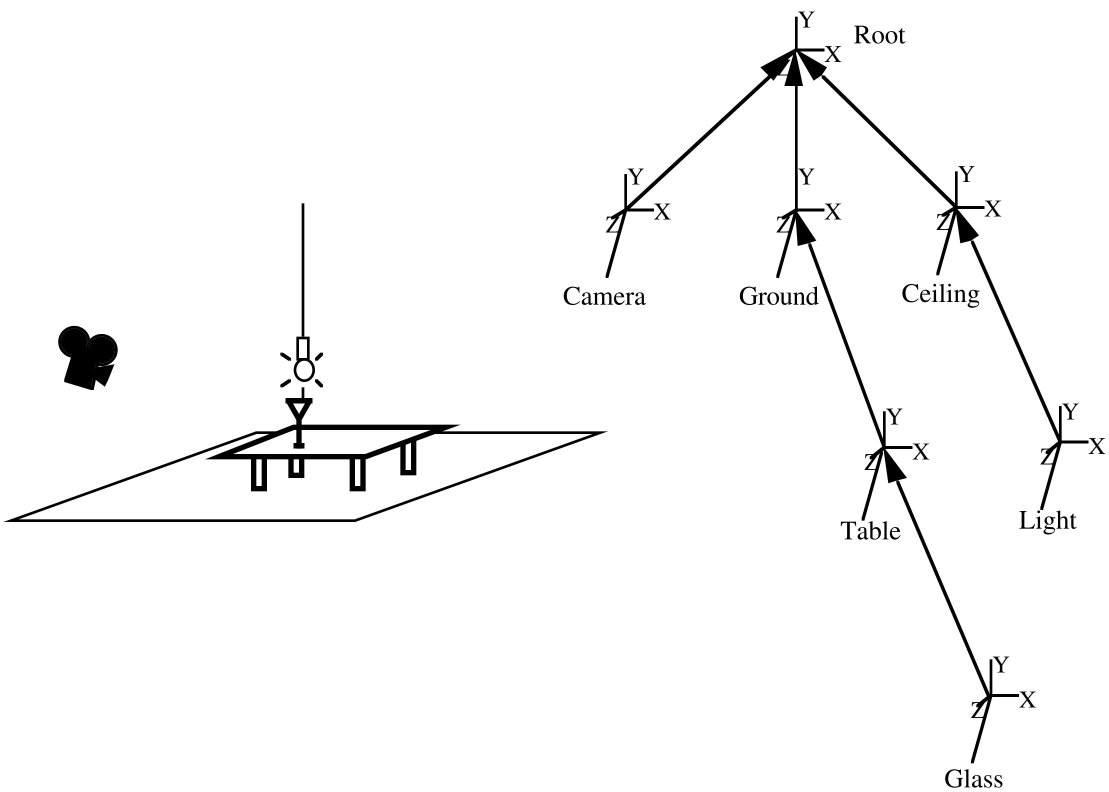

Figure 4: A hierarchical scene

1.5.3.2 Object Model of the Hierarchical World

The basic modeling classes of our system are modeled as an object-oriented representation of classical animation hierarchies. Its structure is presented in the following diagram:

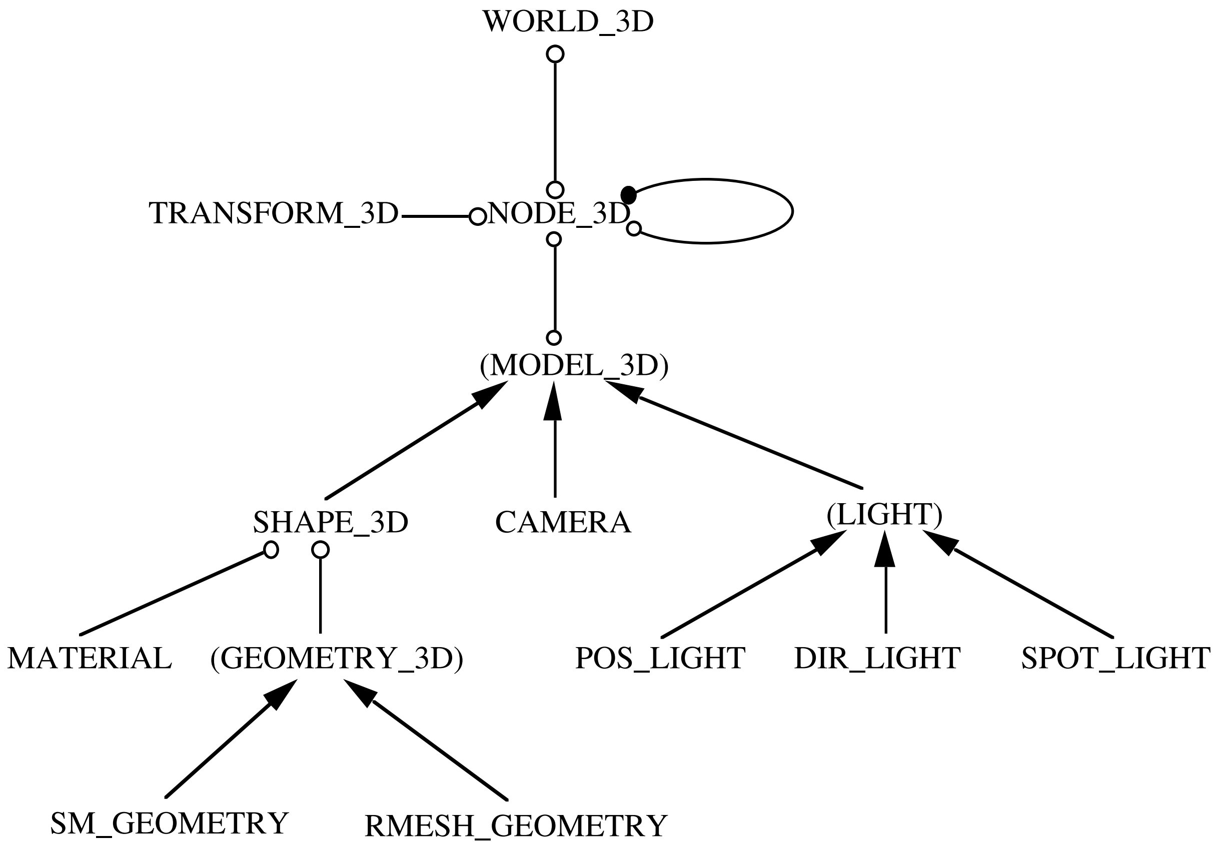

Figure 5: Basic modeling classes

The WORLD_3D class represents the three-dimensional scene that the animation system manipulates and contains all the global information: the environmental illumination parameters (packaged in instances of AMBIENT), and the geometric hierarchy being manipulated.

The transformation hierarchy is represented by instances of NODE_3D related in a hierarchical fashion, and is used to specify the position, orientation, and scaling of the reference frames to which the models are attached. This geometric information is packaged in TRANSFORM_3D objects, and animation paths are stored by replacing plain TRANSFORM_3D instances with instances of the subclass ANIM_TRANSFORM_3D, which are able to store a transformation history and to play it back.

To NODE_3D instances it is possible to attach instances of MODEL_3D, which defines the concept of three-dimensional models in the reference frame of their node. A scene can be composed of three basic types of MODEL_3D: lights, cameras, and shapes. It is worth noting that this design clearly separates the hierarchical structure of the scene from the models attached to them and that models themselves are not hierarchical. This comes from the fact that it is common practice during the creation of an animation sequence to start by designing the hierarchical structure of the scene, and to attach to this structure some simple models to test the animation. These simple models can easily be replaced by more elaborate ones as work progresses. In this way, complex hierarchical structures like skeletons can be designed and reused several times with different shapes attached to them to change their look. The clear division in our design between nodes and models makes this work easier.

An instance of LIGHT represents a source of light, and maintains information about the color and intensity of the emitted rays. Subclasses of LIGHT, such as DIR_LIGHT, POS_LIGHT, and SPOT_LIGHT define various types of sources and maintain more specific information.

An instance of CAMERA represents a camera positioned in the scene, and maintains information about its viewing frustrum and its perspective projection.

An instance of SHAPE_3D is an encapsulation of the concept of a physical object having a shape and a material in the Cartesian space. Geometries and materials are defined in completely separate classes, to make them more reusable and to be able, for example, to manipulate a purely geometric object in a program that does not have to know about its possible material properties.

Subclasses of GEOMETRY_3D are provided to offer various ways to define the geometric properties of a physical object. Examples are: SM_GEOMETRY, which defines a geometric object by specifying a surface model as a set of triangular facets, and RMESH_GEOMETRY, which represents objects as rectangular grids of three-dimensional points.

Instances of MATERIAL are used to define the behavior of physical objects with respect to light. They contain information such as the color and intensity of emission, diffuse, and specular components as well as the shininess and transparency factors.

All the state information about the model being manipulated can be maintained within this graphical hierarchy. Inheritance and polymorphism are used to handle in a simple and efficient way the different types of graphical shapes. The addition of new types of geometries, for example, is done by defining new subclasses of GEOMETRY_3D, and specifying the relevant operations.

1.5.3.3 The Icon world

In a three dimensional scene, some models are not directly visible. A light, for example just emits some light into the scene, and a camera simply defines a viewpoint and a perspective projection. However, in a three-dimensional application based on direct manipulation, models such as cameras, lights, and purely user-interface constructs such as control points, may have to be made visible for the user to select and manipulate them.

These models exist only for means of user interface, hence they should not be part of the scene hierarchy. Therefore we decided to introduce the concept of three dimensional icons, which are used to make visible, by some recognizable representation, models which are usually invisible when making the final rendering of the scene.

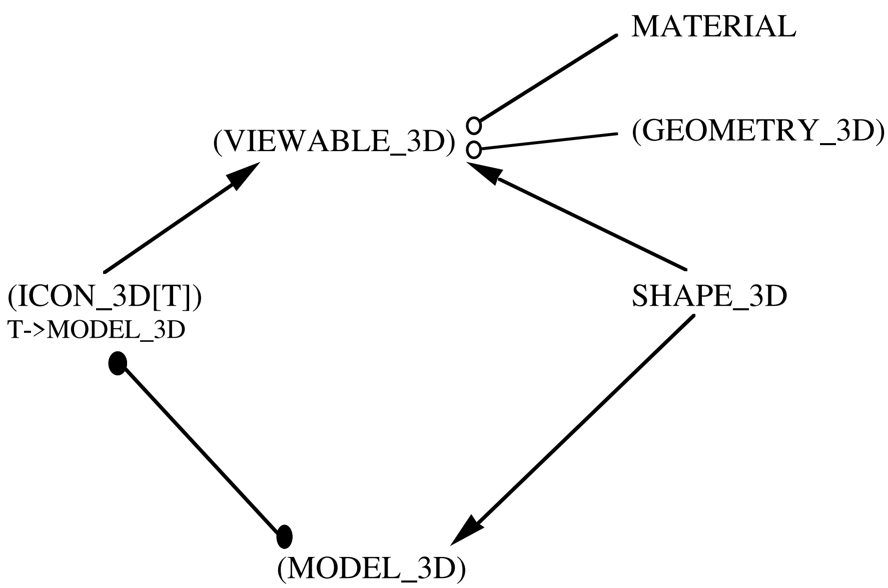

Figure 6: the icon world.

The introduction of the ICON_3D class led to the creation of VIEWABLE_3D, which represents the capability of maintaining a geometric representation and a material. This abstract class defines characteristics that are common to its descendants SHAPE_3D and ICON_3D.

1.5.3.3 Object Design

The implementation of the different classes composing the modeling cluster did not pose particular problems. Most of the responsibilities of these classes consist of handling some kind of data structure (like an n-ary tree of transformations for NODE_3D and a bi-dimensional array of points and normals for RMESH_GEOMETRY). Since we were able to directly use the basic data structures classes provided by Eiffel's library: this allowed us to concentrate most of our time on the purely mathematical side of geometric objects.

1.5.4 Rendering

1.5.4.1 Analysis

The modeling hierarchy presented in the previous section defines the way we organize and maintain the data structures representing a graphical scene for our dynamic applications. The next step in the object-oriented design process is to organize the types of operations defining how the data structures should be manipulated.

For an application such as key-frame animation, two of the most important types of operations are implementing the visual appearance and dynamic behavior of the different graphical objects. An important design question that arises is: where should the graphical appearance and dynamic behavior be encoded? Two possible solutions are: to encode these features directly in the model (for example by adding a specific *render* feature to the various graphical objects), or to design other sets of classes that are able to perform these operations.

In simple two-dimensional architectures, such as user-interface toolkits, these kinds of operations are usually encoded directly in the model. For more sophisticated applications, this kind of approach is not feasible because there is no simple and unique way for a graphical object to perform complex operations like drawing itself, or responding to events.

If we analyze the problem of rendering a three-dimensional scene, several reasons suggest the creation of auxiliary classes:

- many different algorithms for drawing graphical scenes may coexist in the same system. Examples are: ray-tracing, radiosity, or z-buffering techniques. The details about these techniques should not be known by every graphical object.

- rendering may be done using several different types of output units, such as a portion of the frame buffer or a file, and it is not necessary for all this knowledge to be spread out among all the graphical objects.

- several rendering representations, such as wire-frame or solid, may be selectable on a per graphical object basis. The same object may be viewed by several different windows at the same time, each view using a different representation.

We can see that the rendering feature needs much more information than the type of the object to be rendered, and that the same graphical instance can be rendered in several different ways depending on the type of renderer and the type of representation. Therefore, we decided to design and implement new types of objects to maintain this additional information and to implement the various rendering algorithms.

1.5.4.2 Object Model

The rendering cluster is composed of a set of classes that are used to display a scene. The following diagram shows the basic design of this cluster:

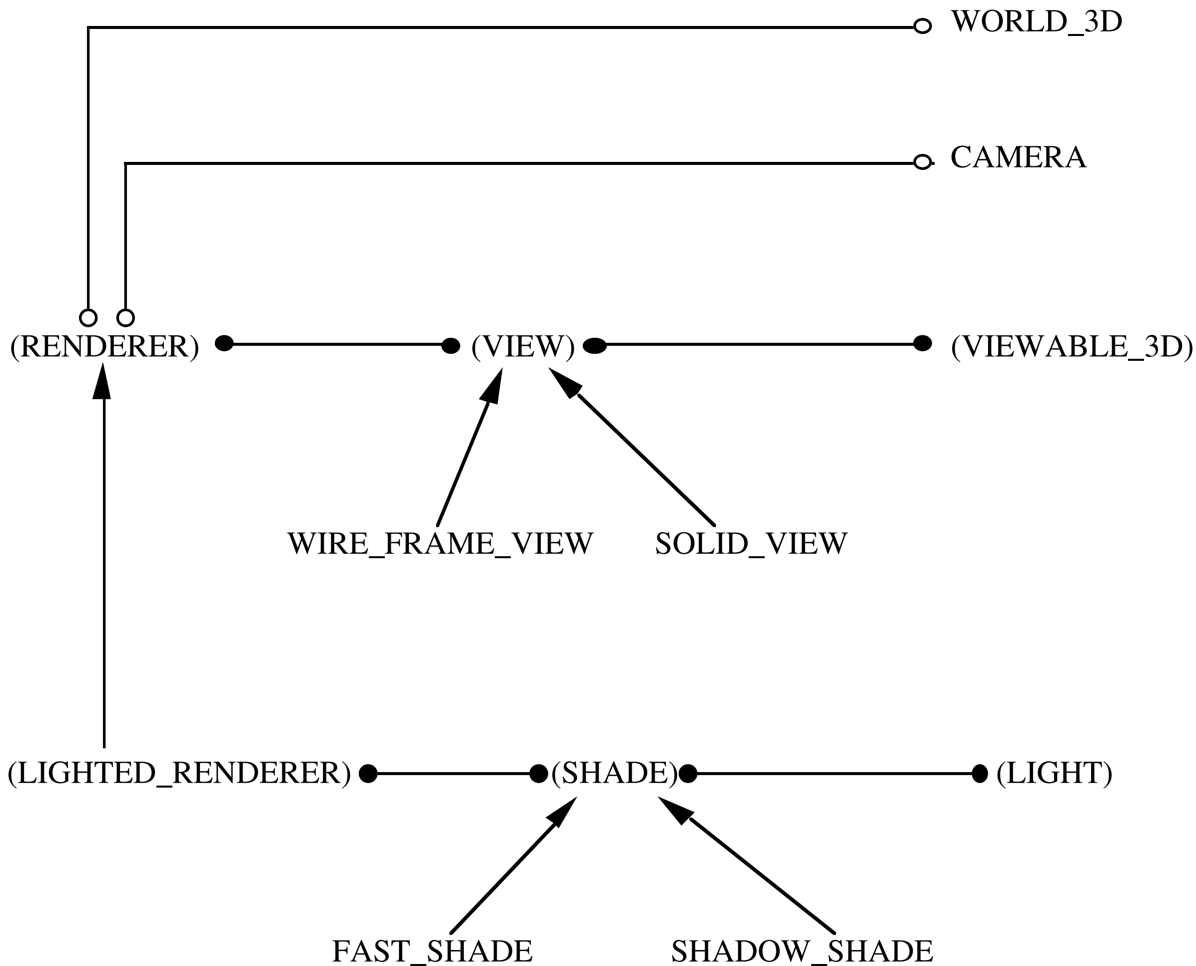

Figure 7: Basic rendering classes

Five basic sorts of classes can be identified:

- **renderers** (subclasses of the RENDERER abstract class), which represent a way to render entire scenes. The code for actually rendering three-dimensional scenes is implemented here. An important subclass of RENDERER is LIGHTED_RENDERER, which describes renderers that can take into account illumination parameters when computing a visual representation of the scene.

- **cameras** (subclasses of CAMERA), which are objects able to return geometric viewing information such as perspective and viewpoint.

- **worlds** (subclasses of WORLD), which are objects able to return global illumination information.

- **viewable models** (subclasses of VIEWABLE_3D, such as MODEL_3D and ICON_3D in the modeling cluster) that are visible objects having position, orientation and scale in the Cartesian space, a geometry, and a material.

- **views** (subclasses of VIEW), which define how a viewable object should be represented.

In this architecture, view objects act as intermediaries between the viewable models and the renderer, telling the renderer what technique (e.g. wireframe, solid) should be used to display each graphical object.

1.5.4.3 Object Design

In order for a renderer object to be able to display a single graphical object, it must consult its views and the attached viewable object to determine the necessary drawing algorithm. This shows that the rendering feature is polymorphic on more than one type.

Object oriented languages like Eiffel, by means of its dispatching which is done on the basis of the target type at the moment of feature application, offer a way to select between different implementations of the same operation without using complicated conditional statements which are difficult to maintain and extend. This is one of the major advantages of object-oriented programming and can be used even when the dispatching has to be done on more than one type. This is true for the rendering operation, which is implemented by applying a feature to each polymorphic variable we want to discriminate and let the dynamic binding make all the choices [Ingalls, 1986].

To show how this method works, let's look at the various classes that form the rendering cluster. In order to render a scene, a *render* feature has to be applied to a renderer, which has the task of displaying all the objects that are attached to its views. To do this the renderer, after some initializations, has to set up the camera and apply a *render* feature to all its views with itself as a parameter. This algorithm is completely general and can be written in the abstract RENDERER class:

```
render is
        -- Render the current scene
        require
                not is_rendering
        do
                if not camera.Void then
                        begin_rendering;
                        define_camera(camera);
                        from views.start until views.offright loop
                                views.item.render(Current);
                                views.forth;
                        end;
                        end_rendering;
                end;
        ensure
                not is_rendering
        end; -- render
```

When a *render* feature is applied to a view, its task is to know what kind of geometries are attached to it through its viewable objects, and to communicate back to the renderer all this specific information. This is done by storing the current renderer and applying a *view* feature to all the viewable objects known by the view. The viewable objects will do the same kind of operation to know about their geometry and respond to the *view* feature call from the view with a more specific feature call indicating all the type information needed: a *view_rmesh* feature will be applied by objects conformant to RMESH, a *view_sm* feature by objects conformant to SM_GEOMETRY, and so on. So, every subclass of VIEW will have to implement a new *view_...* feature for each of the types of geometries that need to be distinguished. The Eiffel code of the abstract VIEW class is as follows:

```
deferred class VIEW export ... inherit
    GAP_LIST[VIEWABLE_3D]
        rename
            Create as gap_Create;
feature

    current_renderer: RENDERER;
        -- The renderer that is currently asking information to the view

    render(r: like current_renderer) is
        -- Applies back to r  a more specific render_... feature with all
        -- the information needed for rendering each item of the list.
        require
            R_exists: not r.Void;
            Not_rendering: current_renderer.Void;
        do
            current_renderer:= r;
            from start until offright loop
                item.view(Current);
                forth;
            end;
            current_renderer.Forget;
        ensure
            Not_rendering: current_renderer.Void;
        end; -- render

    view_sm(m: MATERIAL; t: TRANSFORM_3D;  o: SM_GEOMETRY) is
        -- Send back a render_.._sm(Current, m, t, o) to current_renderer
        require
            Rendering : not current_renderer.Void;
            Material_exists: not m.Void;
            Transf_exists : not t.Void;
            Item_exists : not o.Void;
        deferred
        end; -- render_sm

    view_rmesh(m: MATERIAL;t: TRANSFORM_3D;o: RMESH_GEOMETRY) is
        -- Send back a render_.._rmesh(Current, m, t, o) to current_renderer
        require
            Rendering : not current_renderer.Void;
            Material_exists: not m.Void;
            Transf_exists : not t.Void;
            Item_exists : not o.Void;
        deferred
```

```
        end; -- render_rmesh
...

end -- class VIEW
```

Each of the specific view features is implemented in VIEW subclasses by applying back to the current renderer a specific render feature with all the information needed to display the object using the right representation, thus telling the renderer to actually display the object. So, for example, the WIRE_FRAME_VIEW class defines the *view_sm* feature as follows:

```
view_sm(m: MATERIAL; t: TRANSFORM_3D;  o: SM_GEOMETRY) is
        require
                Rendering : not current_renderer.Void;
                Material_exists: not m.Void;
                Transf_exists : not t.Void;
                Item_exists : not o.Void;
        do
                current_renderer.render_wf_sm(Current, m, t, o);
        end; -- render_sm
```

Figure 8 shows a typical example of feature invocation when the *render* feature is applied to a renderer.

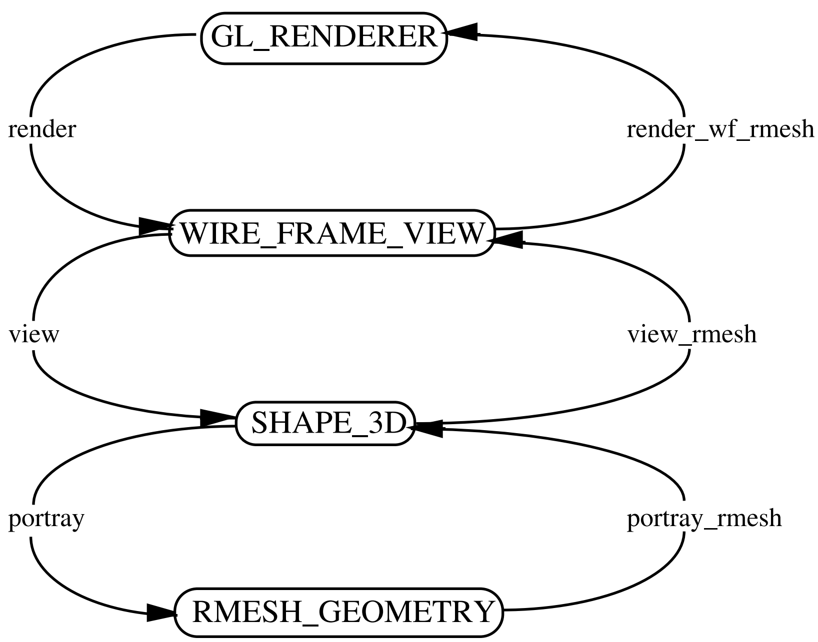

Figure 8: Multiple dispatching for a *render* feature call

As the diagram shows, rendering a single object involves setting off a chain of feature calls, passing through the view, the viewable model and its geometry, ultimately resolving to the appropriate rendering feature. In this way, the composition of the instance data structure automatically determines the rendering algorithm.

1.5.5 Dynamics and Input

1.5.5.1 Analysis

Animated and interactive behavior are among the most confusing aspects of computer graphics design. These can actually be thought of together as the fundamental problem of dynamic graphics: 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 real-time clocks results in animated behavior.

The first problem a sequential application (i.e. a single process with one thread of control) has to solve is the multiplexing between different asynchronous input sources and the handling of those various inputs in a consistent temporal order. The most common way to do this is to have a central input-gathering algorithm responsible for selecting between several input queues (such as the windowing system, various devices, and inter-process communication channels) and extracting each input event in the proper time order, resulting in a single time-ordered queue of input events which can be handled sequentially.

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

Initialize the application and select input channels

from start until over loop

> *Go into wait state;*
>
> *Wake up when input arrives;*
>
> *Respond to input;*

end

In such a structure, the dynamic behavior is implemented in the section *Respond to input.* The most natural object-oriented way to model a workstation with multiple input devices is for each input device (e.g. Mouse, Spaceball) to be represented as a separate instance of a particular input device class. Given such a model, the first thing that has to be done in responding to an input event is to interpret the type of input data received, and update accordingly the object representing the source of this data. If, for example, when the application receives an input indicating that the user moved the mouse, the state of the mouse object has to be updated.

Once the input device objects are updated, the state of the application has been changed and some action has to be performed to respond to this input event, implementing in this way the dynamic behavior of the program. So, for example, if we want the virtual camera in our scene to move when the user moves his SpaceBall, some mechanism must be used to implement this dynamic behavior An obvious way to do this is to implement some feature in the CAMERA class that could be called every time the state of the SpaceBall changes. However, for the same reasons that led to the separation of rendering operation described in the previous section, it is better to move the code implementing the dynamic behavior into a separate object, called a *controller*. In this case, in order to change the behavior of our virtual camera, we simply have to call the appropriate feature of its associated controller object. We call these controller objects dynamic objects because they change their state in response to external input.

Just as the encapsulation of an object's graphical appearance allows higher-level graphical assemblies to be constructed from graphical components, an assembly's dynamic behavior can be built up from the behavior of its dynamic components. To do this effectively, a mechanism must be used to represent the changes of state of dynamic objects in response to input and to represent the dependencies among these objects so they can be updated. We call these changes of state *events*, and consider the problem of updating dependent objects to be the problem of proper distribution of events between objects.

1.5.5.2 Object Model

The dynamic behavior of an application, as described in the previous section, requires the creation of two sets of classes: a first cluster for maintaining and multiplexing between multiple input channels and a second cluster for representing and distributing events.

The design of the input cluster is represented in the following diagram:

Figure 9. Classes for Input Multiplexing and Inter-Process Communication

The design of this set of classes is adapted from the set of inter-process communication classes developed by Matt Hillman [Hillman, 1991] and partially reuses most of its components.

Objects of class NET_NODE can form and accept socket connections with other processes, and multiplex between all of these when waiting for input. It is through these connections, represented by NET_CONNECTION objects, that asynchronous inputs from various input devices, from the window system, and from remote processes arrive to the application. These inputs are represented by objects of type NET_MESSAGE which are able to both read from and write to a network connection. On top of these basic classes, several extensions are implemented by means of specialized connection objects and messages: one of these is the user interface toolkit presented in the next section.

Events and their distribution are modeled using a cluster of classes whose principal components are shown in the following diagram:

Figure 10. The Dynamic Cluster

Three types of objects are used in modeling the dynamic behavior of our applications: these are events, handlers, and dynamic objects.

Dynamic objects maintain a list of possible types of events it can sent out to other objects, that is, a list of instances of a specific subclass of EVENT for each type of event they can transmit. An event is transmitted every time a dynamic object has to communicate a change of state, the type of event indicating what kind of state change occurred in the dynamic object. Each event instance maintains a list of handler objects, called subscribers, which are objects that want to be informed whenever the event is transmitted. These subscribers then handle the event to implement the dynamic behavior of the application.

Dynamic objects generate events by applying to the event instances they own a feature called *transmit*.. This feature is implemented in each subclass of EVENT by applying to all the subscribers a specific handling feature having the source of the event (i.e. the dynamic object which transmitted the event) as its only parameter. For example, an event of class BUTTON_DOWN transmits the event by the application of the feature *handle_button_down*, and an event of class BUTTON_UP transmits by applying *handle_button_up*. Again, as in the rendering cluster, this distinction is done completely through the dispatching mechanism inherent to the dynamic binding and not through conditional statements.

This representation of dynamic behavior introduces 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, however, is its type, indicated by the feature called, and any other data must be explicitly queried from the source by the handler of the event. The handler can then update its internal state and perform actions accordingly to the changes of state in the source objects and its programmed behavior. Secondary events can be transmitted by handlers that are themselves dynamic objects, the complex behavior of an application being in this way encoded in the graph of connections traversed by the events.

Dynamic objects are very important concept in our system design: graphics applications written inside our framework can in fact be thought of as big networks of dynamic objects that transmit events and handle them in real time.

1.5.5.3 Object Design

Objects that can handle events are subclasses of the HANDLER class. Several deferred subclasses of HANDLER exist, each one defining an general object that can handle a certain type or collection of types of events. Specific implementations of handlers are obtained by writing classes that inherit from the general handlers and redefine the specific features associated with the desired events. One example is the FLIGHT_C controller, which is used to move three-dimensional objects using a flying vehicle metaphor:

```
class FLIGHT_C export
    set_target, repeat H_VALUE

inherit

    H_VALUE
            redefine
                    handle_new_transform;

feature

    target: NODE_3D;

    set_target(other: like target ) is
            do
                    target:= other;
            end;

    handle_new_transform(source: T_TRANSFORM) is
            -- Handles the event by moving `target' using a flying vehicle metaphor
            -- No action if no target.
            require
                    not source.Void; not target.Void;
            do
                    if not target.Void then
                            target.set_global_transf(
                                    source.value * source.delta_local * source.value.inverse *
                                    arget.global_transf);
                    end;
            end; -- handle_new_transform

end; -- class FLIGHT_C
```

This class inherits from H_VALUE which is a subclass of HANDLER capable of handling events that indicate changes in some values, and redefines the feature *handle_new_transform* to perform the required actions when an event is transmitted indicating that a new transformation object is available.

1.5.6 User Interface

The goal of the user interface cluster is to provide the typical interactive capabilities associated with modern graphics workstations, namely a mouse, a windowing system, and standard types of interaction widgets, like text input, sliders, and buttons. We also wanted to encapsulate in an object-oriented way some of the newer 3D input devices such as the SpaceBall or the DataGlove.

This was, in fact, the goal of an earlier development effort by our group to create a user-interface toolkit, called the Fifth Dimension Toolkit, for use in our laboratory [Turner et al., 1990]. When that project started, we had no access to an object-oriented language, so we developed a technique for doing object-oriented programming in C, mentioned earlier. In designing this toolkit, which was inspired to a large extent by the NextStep AppKit [Thompson, 1989], we consciously tried to make as purely object-oriented a design as possible.

The modeling of events, in particular, is rather innovative. NextStep's Target/Action paradigm, in which two objects communicate via an action message, with the message's source as a parameter, was extended to model all events. The concept of a centralized event queue was abandoned, replaced by a decentralized collection of event-generating objects such as a mouse object, keyboard object or window object. Therefore there is no conceptual or syntactic difference between a mouse object sending an event when it is moved, and a button widget sending an event when it is pressed.

One particularly powerful feature of the toolkit is an interface builder application, which allows panels of widgets to be arranged and their attributes edited interactively. The resulting user-interface panels can then be stored in a human-readable (and editable) ASCII file and loaded in by an application program at run-time.

Given this functionality required by the animation system, and given the fact that numerous other application programs had already been developed using the 5D Toolkit, we were reluctant to start over from scratch in Eiffel. We therefore considered encapsulating the toolkit in a Eiffel class. This presented some problems, however, since encapsulating an object-oriented software library is considerably more difficult that a non-object-oriented one. Unlike the BLAS and LAPACK routines, which do not maintain their own data structures or state, an object-oriented software library allocates memory and sets up a network of interrelated data structures. The encapsulating Eiffel code, therefore, must either duplicate all of an objects internal data structures itself, with its attendant problems of consistency, or it must separately encapsulate each of the objects maintained by the encapsulated object. The problem was further complicated for us by the event distribution mechanism, in which any dynamic object can send an event to any other object. Since we wanted our Eiffel objects to receive events from the 5D Toolkit objects, an event translating mechanism had to be build.

Our solution was to associate a parallel Eiffel instance of single class, UI, for every instance of every type of toolkit object. Since the 5D toolkit objects are dynamically typed, the single UI class encapsulating all the toolkit functionality is appropriate. Most of the toolkit objects are lower level, however, and do not need to be directly accessed by Eiffel. Therefore a mechanism was devised so that the parallel Eiffel UI object for each toolkit object is created only on demand when it is needed in the Eiffel application. Later when it is no longer used on the Eiffel side, it is garbage-collected by Eiffel even though the toolkit object still remains. An interface was also build so that toolkit events are translated into Eiffel events as they occur. Because the Eiffel types of events were inspired by the toolkit, this was not too difficult, but the problem of mapping one system of user-interface event types onto another is in general not trivial. Finally, we integrated the toolkit Display class, which is the main source of toolkit events, into the Eiffel inter-process communication classes so that toolkit events could be interleaved in with events from other processes.

1.6. BUILDING THE APPLICATION

Our key framing application is implemented by constructing a root class which assembles the classes making up the different clusters described in the previous section.

The first part of the application is devoted to the initialization of its data structures, and all the panels of user interface objects are loaded from files created by the user-interface builder. Windows are created to contain these panels and to display three-dimensional scenes. The visibility state of the different windows is then set according to the initial state of the program. A top-level handler class implements the commands which, in response to user input, instantiate the various graphical and dynamic objects that make up the animated scene.

To the user, the animation system appears as a collection of windows displaying either a three-dimensional scene or a control panel. Any desired combination of windows may be made visible at any given time. The principal windows are as follows:

- The two **three-dimensional viewing windows** display images of the scene as perceived by virtual cameras. A first window shows the view from a working camera, and is used for interaction: in this window, the animator builds his scene by creating a hierarchy of nodes and attaching them together. Nodes can be selected directly by clicking on them with the mouse. The second window is viewed from the animation camera: it is used for real time play-back and for defining camera motion.

- The **transit window** is the three-dimensional equivalent of the Macintosh clipboard and is used when loading models from files and for cut and paste operations like moving sub-hierarchy in the tree.

- The **node window** permits control of various parameters of the currently selected node like the rendering type, the visibility of the node, etc.

- The **spline window** controls various properties of the animation path of the currently selected node. The animator may change the start time of a spline, or edit the spline through the use of a control panel to add, insert, delete or replace key values.

- The **playback window** is used for controlling the real-time playback of animations. It is resembles a control panel very similar to that of a VCR, offering operations such as moving frame by frame, controlling the animation speed and so on.

Once the interface objects have been created, controllers are then attached to the different widgets and devices using the subscription mechanism presented earlier in the chapter. Specific widgets are identified inside the panels created by the interface builder by using their names.

The interactive behavior of the application is encoded inside the controllers that respond to events generated by input devices, by interaction widgets or by other controllers. Some of the controllers that were written are general purpose and reusable in other applications (an example being the flying vehicle controller previously presented). Some others, like controllers that change the event subscription bindings between widgets and other controllers in order to change the behavior of the application, are quite application specific and hard to reuse. We are currently working on refining our model for dynamic behavior by adding more general event routing classes in order to further simplify the programming of such controllers.

Once the static structure and the initial binding of controllers to widgets and devices is set, the event loop is started. Events generated by the various input devices propagate through the bindings and controllers perform the required actions.

The input devices currently used by the application are a SpaceBall, a mouse and a dial box. The mouse is used to select nodes and to specify position and orientation in the 3D space using a trackball metaphor, the SpaceBall is used to manipulate cameras, lights and nodes using various metaphors, while the dial box is used to control some continuous parameters such as scaling, color, etc..

When the animator selects a graphical object, appropriate controllers are bound to it and the various interaction panels display the relevant information concerning the current selected node.

For the dial box, the action of selecting an object may also change the controller tied to the device as the parameter controlled by each dial depends on the type of selected object. If a shape is selected the dial box acts on the scaling parameters of its node, while in the case of a camera the dials control the field of view and the clipping planes.

By default the SpaceBall controls the camera of the window, while the mouse through its trackball controller object controls the currently selected node. Using a menu, the animator can modify these default bindings and change the control metaphor for the selected node or the current camera.

An animation is defined by specifying key points. To define an animation, the user has to attach a spline to the selected node. Then he moves the node with the SpaceBall or the trackball and presses on the SpaceBall button (or on the new key button of the spline window), to record the current state of the selected node as the position of the new key.

1.7. DISCUSSION

In undertaking the project of building a key-frame animation system, we were interested in answering several important questions about the use of object-oriented technology:

- is it possible to use an object-oriented approach to build an application, or does this methodology only lead to building tools for building tools for building tools?

- is it possible to develop reusable components as a side effect of using an object-oriented methodology when building applications?

- is a pure object-oriented approach usable for such an high-performance system as an interactive animation application?

These questions aim at knowing if object-oriented techniques are really adequate to face with the problems of building real world applications.

1.7.1 Building object-oriented applications

The experience we had with the key-frame animation project showed us the feasibility of building large applications using a complete object-oriented methodology for all of its components, at least in the field of dynamic graphics.

Our approach for building the application was to start by defining the problem we wanted to solve, in this case the creation of a key framing application, and to analyze this informal problem statement in order to identify its principal components. Before starting to build this application, we already had some experience in the field of computer graphics and object-oriented design, and this clearly guided us in the identification of the relevant class clusters that would make compose the application.

These components were later analyzed, and clusters of classes were created by defining at first their static structure (inheritance and instance relations) and later their functional behavior. This process was reiterated several times until a satisfactory solution was found. Once the cluster of classes was completed, the application was built by assembling in a proper way its components. This phase also led to several passes of refinement on the design process and to the modification of the basic clusters design.

Even if the approach of decomposing a software project around the objects it manipulates is more "natural", software design still remains a difficult task and lots of time has to be spent in discussing the conception of applications.

We feel that most of the problem in using object-oriented techniques is to conceptually shift the design from the functional decomposition approach, in which the basic unit of modularization is the algorithm, to a data decomposition approach, in which the basic unit of modularization is the data structure. This shift usually requires some effort in people trained to use traditional methods.

From our point of view object-oriented techniques are often incompatible with more traditional ones. It is most of the time confusing to mix object-oriented and traditional functional decomposition techniques, and we therefore prefer to use a pure language like Eiffel instead of mixed paradigm languages such as C++ or Objective-C. However, the ability to call traditional languages from Eiffel was extremely useful for reusing existing libraries.

1.7.2 Building reusable components

The bottom-up approach that object-oriented design favors for building applications leads to the creation of software systems that are large assemblies of basic components. The question is: are the components designed during the creation of an application reusable?

From our experience, we can give a quite affirmative answer. Most of the components created for the key framing application are currently being reused in such different applications as a neural network simulator, a distributed virtual reality system, and a system for simulating the physical behavior of elastically deformable surfaces.

However, we have to admit that components are not usually reusable from the beginning, and some effort is always needed to make them general enough to be able to exploit them in different applications.

The first step for making a component reusable is to have it usable. To test this fact, client applications need to be written and feedback obtained so as to ameliorate the design. For this reason, we have the impression that the approach of building test applications in order to obtain a first draft of class libraries is necessary, and for most non-trivial class libraries preferable to directly building components from first principles.

Another advantage of object-oriented techniques is that it is possible to exploit the similarity of structure of all application in a field by creating frameworks that define and implement the object-oriented design of an entire system such that its major components are modeled by abstract classes. High level classes of these frameworks define the general protocol and handle the default behavior, which is usually appropriate for most of the cases. Only application-specific differences have to be implemented by the designer through the use of subclassing and redefinition to customize the application. The reuse of abstract design which is offered by this solution is even more important than the obvious reuse of code.

Several well known application frameworks exist, especially in the field of user interfaces. Examples are: Smalltalk's MVC ([Krasner et al. 1988]), Apple Computer's MacApp ([Schmucker 1986]), and the University of Zurich's ET++ ([Weinand et al., 1989]).

Our dynamic, modeling, and rendering clusters are a first step towards building a framework for our interactive three-dimensional graphics applications, but much work still remain to be done to make this framework general enough for the creation of future applications.

1.7.3 About performance

Performance is an important issue in building dynamic graphics program. Optimization reasons are often used as an excuse for not using pure object- oriented techniques for the development of such applications, and choosing languages that offer the possibility of freely mixing the procedural and the object-oriented paradigm such as C++ ([Stroustrup 1986]) or Objective-C ([Cox 1986]).

The development of the key-frame system showed us that high performance applications can be obtained using a pure object-oriented language such as Eiffel without compromising the design. Our animation system is able to render fully shaded scenes containing several thousands of polygons at interactive speed (more than ten refreshes per second) on a Silicon Graphics Iris, and allows the user to edit and animate three-dimensional shapes using a complete direct manipulation metaphor.

The fact that the language is purely object-oriented and statically typed allows the compiler to perform important optimizations (such as inlining, unneeded code removal, and simplification of routine calls), so to obtain a high-performance code without having to compromise with the purity of the design. Optimization of several important aspects of our software system was often obtained by creating specialized subclasses that handle special cases.

The availability of a garbage collector for Eiffel allowed us to simplify algorithms and data structures obtaining therefore a more compact and efficient code. The collector is incremental and was therefore possible to be used in our interactive programs without disturbing the user. By carefully designing our components so as to minimize the creation of temporary objects, we are able to contain the cost of object allocation and deallocation in our software system to under 10%. Our previous experience with the development of a user interface toolkit using an object-oriented extension of C showed us the importance of these memory issues: in this previous system a great deal of design effort was spent in defining and maintaining appropriate data structures and storage schemes in order to properly destroy unreferenced objects.

A key factor for the success of our project was the ability to interface our Eiffel classes with pre-existing libraries written in other languages. This was done by defining a clear interface between these external libraries and the Eiffel world and did not disturb the object-oriented nature of the system.

1.8 CONCLUSIONS

The challenge of building dynamic graphics applications that realize the full potential of modern computer graphics hardware remains immense. Object-oriented design techniques, however, provide a significant advance toward the creation of reusable and extensible software components and assemblies for dynamic graphics construction.

Our experience with using a pure object-oriented approach and implementation language for building a key-frame animation system was very satisfying and showed us that these techniques are well suited for creating high-performance applications made of assemblies of reusable components in the field of dynamic graphics.

Most of the components that were created during this project are still being reused and extended for our current work, making it possible for us to concentrate our efforts in solving the specific problems of new application domains.

We are therefore continuing to use Eiffel and object-oriented techniques for our current research work, which focuses on the fields of neural networks, cooperative work for animation, and physically-based simulation of deformable models.

1.9 BIBLIOGRAPHY

Anderson E, Bai Z, Bischof C, Demmel J, Dongarra J, DuCroz J, Greenbaum A, Hammarling S, McKenney A, Sorensen D (1990): Lapack Working Note 20: A Portable Linear Algebra Library for High-Performance Computers, Computer Science Dept., University of Tennessee, Knoxville.

Boulic R, Renault O (1991): 3D Hierarchies for Animation, in New Trends in Animation and Visualization, John Wiley.

Cox BJ (1986): Object Oriented Programming: An Evolutionary Approach, Addison-Wesley, Reading, Massachusetts.

DeMay V (1991): Flexible Graphic Design System, Object Composition, Université de Genève, 1991: 145-155.

Fleischer K, Witkin A (1988) A Modeling Testbed, Proc. Graphics Interface '88: 127-137.

Gobbetti E, Turner R (1991): Object-Oriented Design of Dynamic Graphics Applications, in New Trends in Animation and Visualization, John Wiley.

Grant E, Amburn P, Whitted T (1986) Exploiting classes in Modeling and Display Software, IEEE Computer Graphics and Applications 6(11).

Hedelman H (1984) A Data Flow Approach to Procedural Modeling, IEEE Computer Graphics and Applications 4(1).

Hillman MF (1990) A Network programming package in Eiffel, Proc. TOOLS 2, Paris: 541-551.

Ingalls DHH (1986) A Simple Technique for Handling Multiple Polymorphism, Proc. ACM Object Oriented Programming Systems and Applications ‘86.

ISE (1989) Eiffel: The Libraries. Interactive Software Engineering, TR-EI-7/LI.

Lawson, Hanson, Kincaid, Krogh (1979): Basic Linear Algebra Subprograms for FORTRAN usage, ACM Transactions on Mathematical Software, Volume 5, pp. 308-323.

Meyer B (1987): Reusability: The Case for Object-Oriented Design, IEEE Software, Vol. 4, No. 2, March 1987, pp. 50-64.

Meyer B (1988): Object-Oriented Software Construction, Prentice-Hall, Englewood Cliffs, New Jersey.

Schmucker K (1986): Object-Oriented Programming for the Macintosh. Hayden.

Shoemake K (1985): Animating Rotation with Quaternion Curves, Computer Graphics 19(3): 245-254.

Stroustrup B (1986): An Overview of C++, SIGPLAN Notices 21(10): 7-18.

Thompson T (1989): The Next Step, Byte 14(3): 365-369.

Turner R, Gobbetti E, Balaguer F, Mangili A, Thalmann D, Magnenat-Thalmann N (1990) An Object Oriented Methodology Using Dynamic Variables for Animation and Scientific Visualization Proceedings Computer Graphics International 90 Springer-Verlag: 317-328.

Weinand A, Gamma E, Marty R (1989): Design and Implementation of ET++, a Seamless Object-Oriented Application Framework, Structured Programming 10(2): 63-87.