AN INTERACTIVE 3D GRAPHICS CLASS LIBRARY IN EIFFEL

Russell Turner
Enrico Gobbetti
Jean-Francis Balaguer

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

Angelo Mangili

Scientific Computation Center
Swiss Federal Institute of Technology
CH-6900 Manno, Switzerland

ABSTRACT

An object-oriented design is presented for building interactive 3D graphics applications. The design takes the form of a library of classes written in Eiffel, an object oriented language with multiple inheritance, static typing, dynamic binding, garbage collection, and assertion checking. The classes form a set of reusable components from which a variety of other interactive 3D graphics applications could easily be constructed. A discussion of the overall design goals and philosophy is given. This is followed by a summary description of the purpose and implementation of each of the component class clusters. Finally, the issues are discussed of applying object-oriented techniques to interactive 3D graphics, including encapsulation of existing software and the implementation on a Silicon Graphics Iris workstation.

1. INTRODUCTION

A new three-dimensional style of interacting with computers is emerging. This style relies on fast, high-quality graphics displays coupled with expressive, multi-degree-of-freedom input devices to achieve real-time animation and direct-manipulation interaction metaphors. 3D interactive techniques are already being used in systems that require the creation and manipulation of complex three-dimensional models in such application domains as engineering, scientific visualization or commercial animation. Other possibilities include the newest virtual environment research which strives for a more intuitive way of working with computers by including the user in a synthetic environment.

The design and implementation of an interactive 3D application are extremely complex tasks. The application program has to manage an model of the virtual world depicted on the screen, and to simulate its evolution in response to events from the user. These events can occur in an order which is determined only at run time. In particular, an interactive, event -driven application must be able to handle the multi-threaded style of man-machine dialogue associated with direct manipulation interfaces, and it must be able to make extensive use of the various asynchronous input devices at the disposal of the user. These can vary from the keyboard and mouse to more sophisticated devices such as the spaceball or DataGlove.

Seen from a software-engineering viewpoint, the design of interactive 3D applications can benefit from object-oriented techniques in several ways. For example, data abstraction can be used to support different internal data representations, multiple graphics drivers can be encapsulated in specific objects, a variety of subclasses can offer the same interface for the manipulation of graphical objects, and the distribution of information can be used to manage the parallelism inherent in direct manipulation programs. From the point of view of a user, the direct manipulation metaphor allows the intuitive behavior and relationships of the objects on the screen to mirror the class and instance hierarchies of the data objects. Object-oriented construction is therefore a natural approach for the design and implementation of an interactive 3D graphics system.

We had already gained some experience building object oriented software with the development of a research user-interface toolkit for the Silicon Graphics Iris workstations [Turner, 90] using a custom-made object-oriented extension to C based on the concepts of Objective C (Cox, 1986). This experience showed us the limitations of using a hybrid language for the implementation of an object-oriented design. Because the language was an extension of a traditional language, it was difficult to completely enforce the object-oriented paradigm, often resulting in a mixture of procedural and object-oriented styles. In addition, the language provided no multiple inheritance, and no static typing, which limited its expressiveness and sometimes influencing design decisions. For example, code duplication was often necessary in cases where multiple inheritance should have been used instead. Another big problem with our system was its lack of garbage collection, which required us to spend too much time chasing memory bugs and devising complex algorithms to destroy object structures.

As a result, we became convinced of the importance of using a pure object-oriented language for our work, and Eiffel was chosen for the 3D library because of its characteristics which corresponded closely to our needs. In particular, Eiffel supports multiple inheritance, static typing, dynamic binding, garbage collection, assertion checking and the ability to call other languages easily. We were interested to find out if these powerful object-oriented features would help in the process of 3D design and we also wanted to test if it was possible to use a pure object oriented language like Eiffel for an area with such large constraints on performance as interactive 3D graphics.

In this chapter we will present the design and implementaaion of a set of classes in Eiffel which can be assembled to create these types of applications. Section 2 discusses our design goals and methods, section 3 presents a detailed description of the various clusters of classes making up the library, section 4 describes how these classes are assembled into applications and section 5 discusses some of the issues we have encountered.

2. SYSTEM DESIGN

2.1. Identification of principal class clusters

We initially spent most of our time in group discussions about the design of the system. The software development tools for Eiffel encourage the grouping of related classes into what are called *clusters*. We therefore decided to split the problem 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 3D graphics application domain and on our previous experience designing other graphical systems. This process is typical of the bottom-up approach that object-oriented design tends to promote. The clusters we identified were:

- 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 also developed some lower-level clusters for providing data structure and mathematical functionality. One of these, the mathematics cluster, provides standard mathematical objects such as VECTOR and MATRIX, as well as classes more specific to computer graphics such as TRANSFORM_3D and QUATERNION.

For the purposes of describing the class structure of the clusters, we have made use of object-relation diagrams, as in (Rumbaugh, 1991). The definition of the static structure of the design is usually the first step in the design process, and these diagrams allow us to give a schematic presentation of this structure which combines both instance relations and inheritance relations. These diagrams have become a standard way for us to exchange ideas during the design meetings.

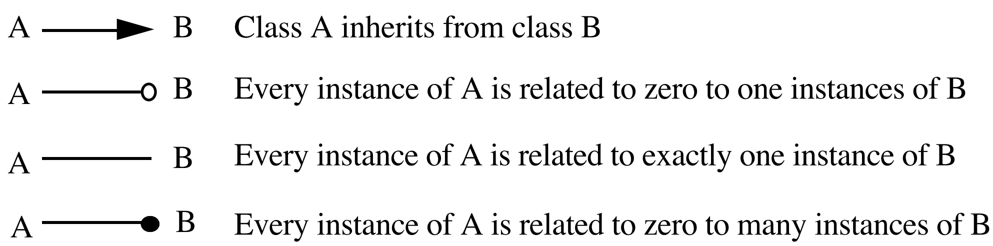

Figure 1. Object-Relation Diagrams

These diagrams are the only formal notation we have used during design process. No other types of diagrams or special notations were used. These diagrams are useful but we tend to resist more formalized schemes (data flow diagrams, for example). These other techniques could be of some use, particularly if used together with some case tools for maintaining diagrams up-to-date with respect to the code, and we may consider them in the future. Fortunately, Eiffel's assertions and invariants offer a way to represent the programming contracts between different components of the system. We therefore sometimes use fragments of Eiffel code early in the design instead of pseudo-code or data-flow diagrams.

2.2. Conventions

To encourage more regular interfaces, and to improve the reusability of our classes, we adopted some conventions for naming Eiffel features. Some of these conventions are:

- features that return an alternate representation of an object always begin with *as*. For example, *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*. For example, *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 suffix *_in*. For example, *row_in(i: INTEGER; v: VECTOR),* which can be applied to instances of MATRIX.

One result of these conventions is that features tend to follow the style of modifying *Current* or one of the parameters instead of creating a new object. Therefore, it is the client's responsibility to allocate all the necessary objects, and this can reduce the overhead due to allocation and collection of unnecessary temporary objects.

We put some effort into defining preconditions, postconditions and invariants for all the routines and classes because, by specifying the programming interface contract, they are a key element for promoting reusability. Assertions formally define the behavior of the classes and therefore we use them early on in the design phase. They also provide a certain form of documentation and help during debugging. Assertions also help to produce efficient software. By clearly defining the responsibilities of each component, defensive programming techniques can be avoided.

2.3. Encapsulation

One of the important features of Eiffel, in our opinion, is the ability to encapsulate routines written in other languages. Because this can be done cleanly, it helps to promote pure object-oriented design. It is also essential if we are to acheive one of the main goals in object-oriented programming which is reusability. Decades of programming effort have been spent in developing and testing large software libraries in various languages. Some of these libraries, for example the FORTRAN BLAS and LAPACK libraries, provide a functionality that could not be particularly improved upon by reimplementing them in an object-oriented language. Hybrid languages such as C++ and Objective-C try to encourage this reuse by allowing the programmer to continue developing their software using object-oriented extensions. Unfortunately, this approach does not enforce a clean separation between the object-oriented and non-object-oriented portions of the software. By putting an object-oriented gloss on traditional languages, we perpetuate the software engineering problems of the traditional languages into the object-oriented languages.

On the contrary, the Eiffel approach is to define a clean and localized interface with the non-Eiffel language components. This does not compromise the object-oriented paradigm upon which the language is based. Our VECTOR and MATRIX classes are implemented on top of the BLAS and LAPACK standard FORTRAN libraries, and for us the ability to reuse this functionality was an important aspect of the development. A great deal of functionality at a high performance was added in a small amount of time, without sacrificing the object-oriented design.

3. OVERVIEW OF THE PRINCIPAL CLUSTERS

3.1. The Graphical Model

3.1.1. Analysis

Interactive 3D graphics applications must be able to respond to asynchronous input events as they happen, so designers must build their programs to behave properly no matter when and in what order the events will occur. This is usually done by maintaining a global data model which represents the current state of the application program at any moment during its execution. In an object-oriented system this global data model, called the graphical model, consists of a hierarchy or directed acyclic graph of instances representing the virtual objects to be manipulated. Many different aspects have to be considered when designing the graphics model, such as rendering, interactive behavior, inheritance of attributes and maintaining internal consistency. Several class structures have been proposed in the literature for representing three-dimensional hierarchical scenes. Examples are found in the modeling and rendering library Doré [Kubota, 1992], [Fleischer, 1988], which describes an object-oriented modeling testbed, [Grant et al, 1986], which presents a hierarchy of classes for rendering three-dimensional scenes, [Hedelman, 1984] which proposes an object-oriented design for procedural modeling, and the object-oriented 3D interaction toolkit UGA [Conner, 1992].

Interactive 3D graphics systems are typically concerned with manipulating models arranged in a hierarchical fashion. This hierarchy can be represented as a tree of homogeneous transformations which define the position, orientation, and scaling of the reference frames in which graphical primitives are defined.(see [Boulic, 1991]).

3.1.2. Object Model of the Hierarchical World

The classes from which the graphical model is built are contained in the modeling cluster, which is presented in the following diagram:

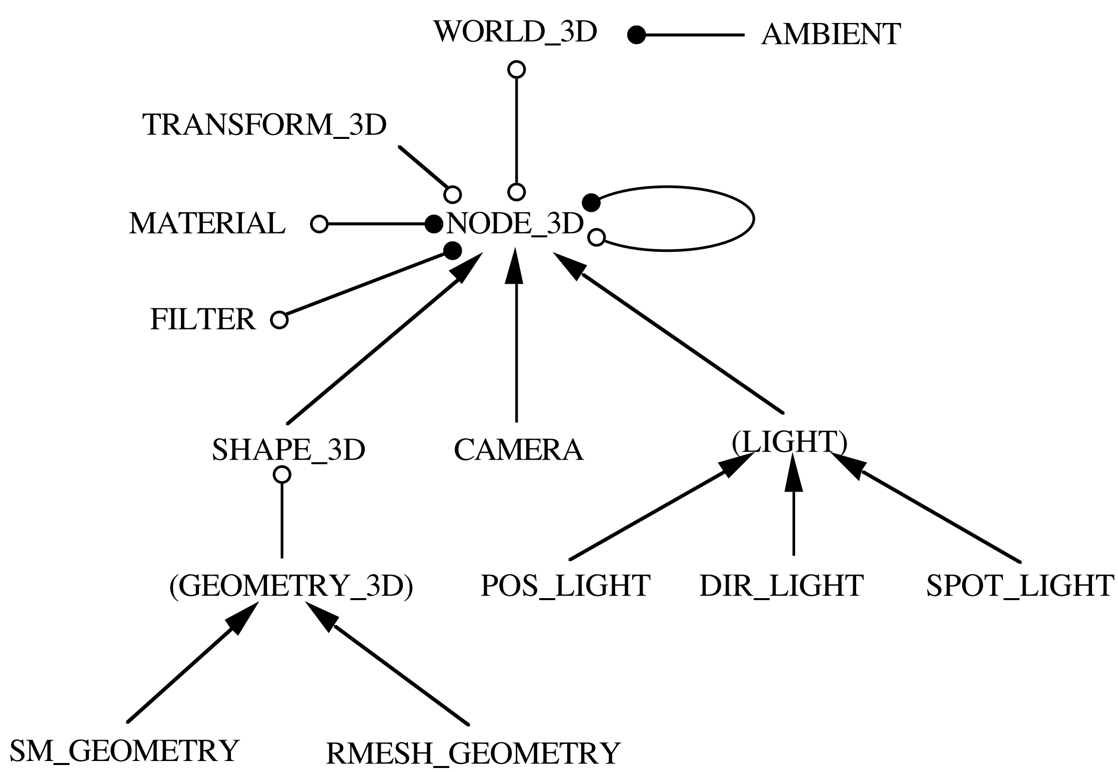

Figure 2: Basic modeling classes

The WORLD_3D class represents a three-dimensional scene and contains the top level of the graphical modeling hierarchy and all the other global information that the application manipulates such as the environmental illumination parameters (packaged in instances of AMBIENT).

The transformation hierarchy is represented by instances of NODE_3D which maintain a transformation object and also may contain child nodes so as to form a tree. It is used to specify the position, orientation, and scaling of the reference frames to which the models are attached. In addition, the NODE_3D contains pointers to other objects which maintain information about the node such as MATERIAL and FILTER. These objects may be multiply referenced and their attributes inherited through the instance hierarchy by delegation. The NODE_3D itself can be subclassed into three varieties: lights, cameras, and shapes, which represent the three basic types of objects in the hierarchy.

Instances of MATERIAL are used to define the reflectance properties of physical surfaces. They contain information such as the color and intensity of the emission, diffuse, and specular components as well as shininess and transparency factors for the specular reflection. Instances of FILTER represent a two-dimensional image which can be projected onto the contents of the node in various ways to achieve texture mapping.

Instance of LIGHT represent a light source and maintain information about its color and intensity. Subclasses of LIGHT, such as DIR_LIGHT, POS_LIGHT, and SPOT_LIGHT define various types of light sources and maintain more specific information such as direction, location in space and angle of projection.

An instance of CAMERA represents a virtual camera positioned in the scene, and maintains information about its viewing frustum and its perspective projection. It is through virtual cameras that the 3D world is rendered on a 2D screen.

The SHAPE_3D class represents the concept of a physical object having a geometric shape in Cartesian space. Geometries are defined in a separate class, to make them more general and reusable. By implementing SHAPE_3D so as to reference an instance of a GOEMETRY_3D, it is possible, for example, to define operations on abstract geometry, independant of the other shape characteristics. Also, since the geometries can be multiply referenced, a single geometry may be used in multiple locations in the hierarchy, with different node characteristics. In this way, the hierarchical structure of the scene can be designed independently of the geometries. Initially, simple geometries can be used which then are easily replaced by more elaborate ones as work progresses. Complex hierarchical structures like skeletons can be designed and reused several times with different geometries attached to them to change their appearance.

The GEOMETRY_3D class can be subclassed to provide various types of geometries. Examples are: SM_GEOMETRY, which defines a geometric object by specifying its surface as a mesh of triangular facets, and RMESH_GEOMETRY, which represents objects as rectangular meshes of three-dimensional points.

3.2. Multiple Inheritance

One notable aspect of Eiffel is its support for multiple inheritance. In fact, the language and the standard clusters that come with it tend to support a very fine-grained approach to multiple inheritance, which we have adopted in our design philosophy. The result is a large number of relatively small classes, many of them deferred, which encapsulate a single concept or piece of functionality. The following diagram illustrates this type of inheritance for the NODE_3D class, although we usually do not show it in so much detail in our other object-relation diagrams.

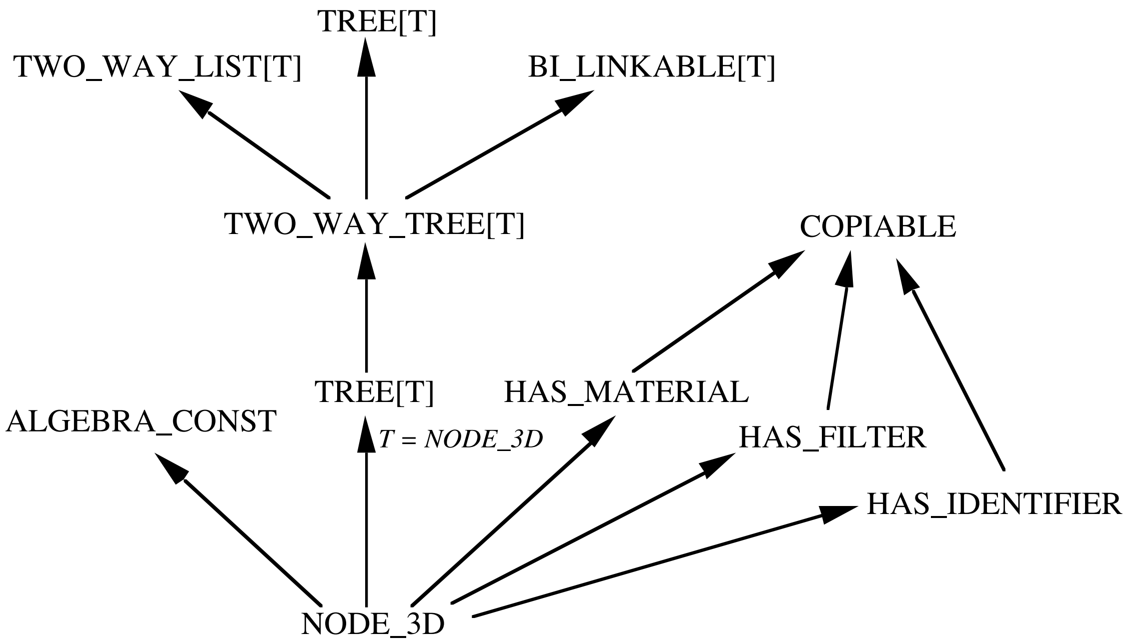

Figure 3: Multiple Ancestors of the NODE_3D Class

In this way, multiple inheritance allows us to compose the NODE_3D's behavior as a collection of partial views, each one defining a particular programming contract and functionality.

3.3. Rendering

3.3.1. Analysis

For 3D interactive graphics, two of the most important types of operations are rendering the visual appearance and implementing the 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 them directly in the model (e.g. by adding a specific *render* feature to the various graphical objects), or to design a new set of classes that are able to perform these operations.

When designing simple two-dimensional class libraries, such as user-interface toolkits, these kinds of operations are usually encoded directly in the model. For more sophisticated applications, however, this kind of approach is usually not feasible because there may be no simple way for a graphical object to perform these operations based on its own internal data.

Taking as an example the operation of rendering a three-dimensional scene, several arguments suggest the creation of auxiliary classes:

- there are potentially many different algorithms for drawing graphical scenes which can coexist in the same system. For example: ray-tracing, radiosity, or z-buffering techniques. The details of these techniques should not have to be known by every graphical object.

- rendering may occur on several different types of output devices such as the frame buffer, a texture map, or an output 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 selectible 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.

- some rendering algorithms may require access to global modeling data simultaneously. For example, a hidden surface algorithm may need to depth sort a polygon display list.

Obviously, the rendering operation needs much more information than just the type of object to be rendered. Also, a single graphical instance can be rendered in several different ways depending on the type of renderer and the type of representation. We therefore decided to design and implement a new set of classes to maintain this additional information and to implement the various rendering algorithms.

3.3.2. Object Model

The rendering cluster is composed of a set of classes that are used to render a three-dimensional scene. The following diagram illustrates the basic design of this cluster:

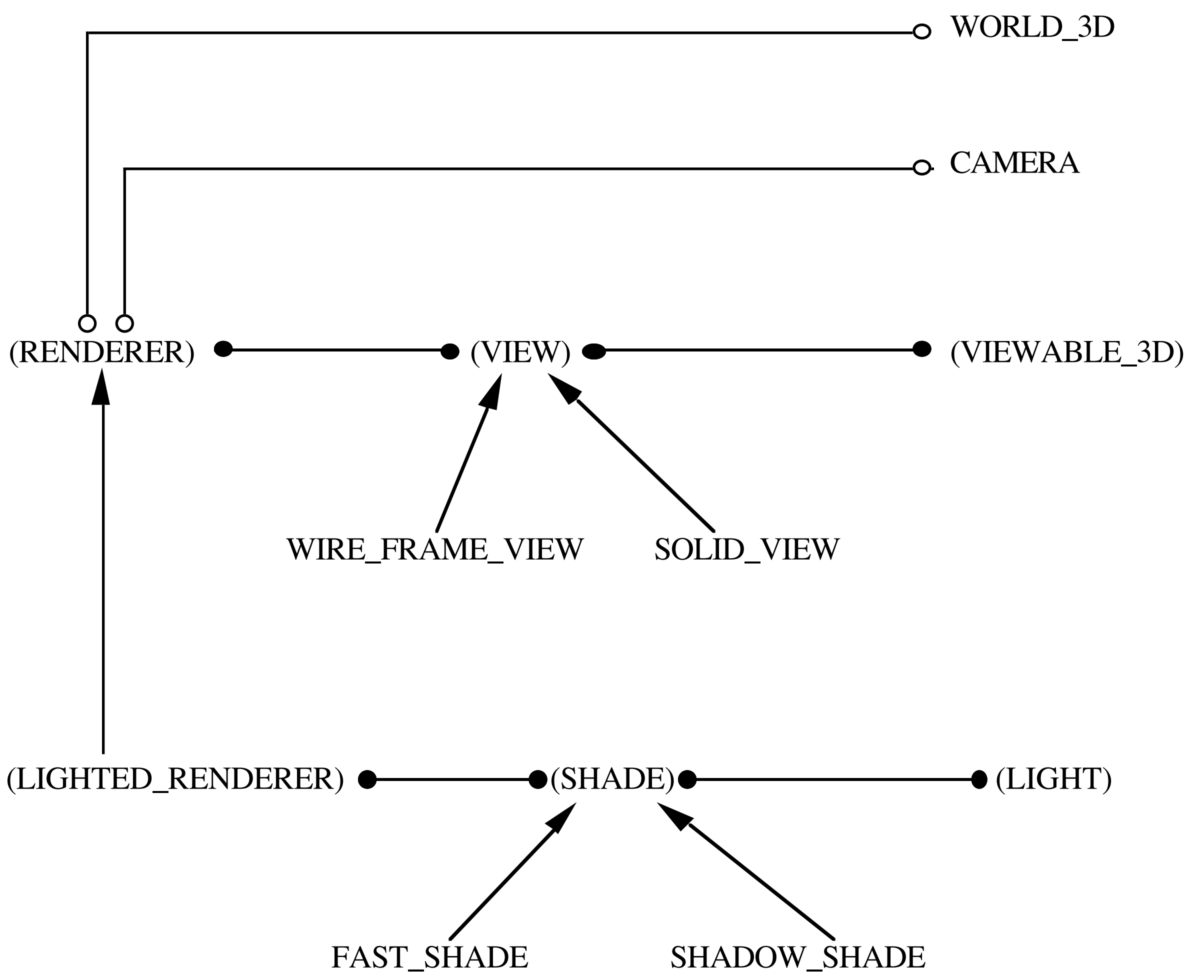

Figure 4: Basic rendering classes

Five basic sorts of classes can be identified:

- **renderers** (subclasses of RENDERER). These represent particular techniques for rendering entire scenes. The actual rendering algorithm is implemented in these classes. An important subclass of RENDERER is LIGHTED_RENDERER, which describes renderers that use illumination parameters to compute a visual representation of the scene.

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

- **worlds** (subclasses of WORLD). These are objects which contain a modeling hierarchy and other information such as global illumination parameters.

- **viewable models** (subclasses of VIEWABLE_3D such as SHAPE_3D in the modeling cluster). These represent visible objects which have position, orientation and scale in Cartesian space, as well as a geometry, and a material.

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

In this architecture, the view objects act as intermediaries between the models and the renderer, telling the renderer what technique (e.g. wireframe, solid, highlighted) should be used to display each graphical object. This provides a clean separation between a model and how it is viewed. Since multiple views may reference the same model, an application can have, for example, more than one window onto a world, each representing the objects in different ways.

3.3.3. Object Design

When a renderer displays a particular graphical object, it first consults its views and their attached viewable objects to determine the necessary drawing algorithm. This type of rendering operation, the binding of which depends on more than one target, can be described as polymorphic on more than one type.

Object oriented languages with dynamic binding like Eiffel, which dispatch 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 conditional statements. This ability to have the data determine the algorithm 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. We have done this for the rendering operation, which is implemented by applying a feature to each polymorphic variable we want to discriminate and letting the dynamic binding make all the choices (Ingalls, 1986).

To illustrate how this method works, we will look at the various classes that form the rendering cluster. To render a scene, a *render* feature is applied to a renderer instance, which has the task of displaying all the objects that are attached to its views. To do this the renderer, after some initializations, sets up the camera and applies a *render* feature to all its views with itself as a parameter.

Each time the *render* feature is applied to a view, it communicates back to the renderer information about what kind of geometries are attached to it through its viewable objects. 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 perform a similar kind of operation by applying the *portray* feature to the model's geometry and respond to the *view* feature call with a more specific viewing feature depending on the type of view object. For example, 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 must implement a new *view_...* feature for each of the types of geometries that need to be distinguished. The specific view features themselves are implemented in VIEW subclasses by calling back to the current renderer with the specific render feature (e.g. *render_wf_rmesh*) to display the object in the desired representation. It is only at this point that the graphical object is actually displayed on the screen. Figure 5 shows a typical example of the sequence of feature invocations resulting from the *render* feature being applied to a renderer.

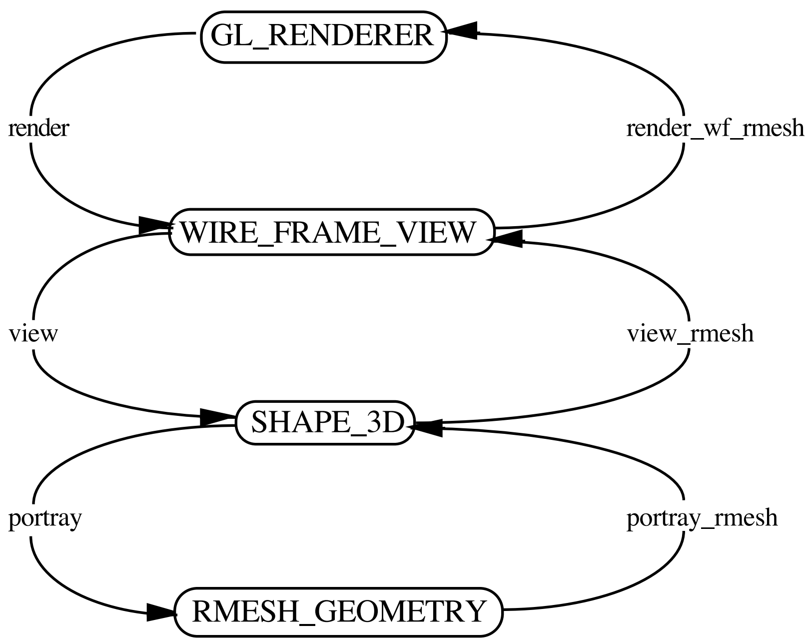

Figure 5: Multiple dispatching for a *render* feature call

As the diagram shows, rendering a single object sets off a chain of feature calls which pass through the view, the viewable model and its geometry and back again ultimately resolving to the appropriate rendering feature. In this way, the composition of the instance data structure alone automatically determines which specific rendering algorithm is invoked.

3.4. Dynamics and Input

3.4.1. Analysis

Implementing interactive and animated behavior is among the most problematic aspects of computer graphics design. These can actually be thought of together as the problem of dynamic graphics. How does the application change its graphical output in response to asynchronous 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 (a single process with one thread of control) must solve is multiplexing between different asynchronous input sources and handling the various input data in a consistent time order. The standard way to do this is in object-oriented toolkits is to have a central input-gathering algorithm responsible for selecting between the various input queues (e.g. 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. For an application which assumes this 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 this event loop structure, the dynamic behavior of the application is implemented in the section *Respond to input*. A natural object-oriented way to model a workstation with multiple input devices is for each input device (e.g. mouse, spaceball, keyboard) to be represented by a separate instance of a particular input device class. To implement such a model, the first action taken in response to an input event is to update the input device object representing the source of this data. For example, when the application receives an input event indicating that the user moved the mouse, the state of the mouse object has to be updated.

Once an input device object has been updated, the state of the application has changed and some action must be performed to respond to this input event and implement the dynamic behavior of the program. For example, if we want the virtual camera viewing the scene to move when the user moves his spaceball, a mechanism must be devised to implement this. An obvious way to do this is to implement a feature in the CAMERA class that is called every time the state of the spaceball changes. However, for similar reasons that led to the separate rendering and modeling classes described in the previous section, it is often better to move the code implementing the dynamic behavior into a separate object, which we call a *controller*. In the case of the virtual camera, we would implement a camera controller object which can be attached to it. The controller knows how to update the camera parameters in response to spaceball events. We call these controller objects dynamic objects because they change their state in response to external input.

The encapsulation of an object's graphical appearance allows more complex graphical assemblies to be constructed from graphical components. We would like to be able to build more complex dynamic behavior in a similar way by assembling dynamic objects together. To do this effectively, a mechanism must be developed to represent the changes of state of dynamic objects in response to input and to maintain the dependencies dynamic objects so they can be updated. We call these changes of state *events*, and implement the updating of dependent objects through the proper distribution of events.

3.4.2. Object Model

To model the dynamic behavior of an application, as described in the previous section, we have created two clusters of classes: an input cluster for maintaining and multiplexing between multiple input channels and a dynamic cluster for representing and distributing events.

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

Figure 6. 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.

The NET_NODE class represents objects that can form and accept socket connections with other processes, and merge input from all of these sources into a single event stream. It is through these connections, which are represented by instances of NET_CONNECTION objects, that asynchronous input from the various input devices, from the window system, and from remote processes arrives to the application. The input data is 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, we have implemented several extensions by means of specialized connection objects and features. One of these is the user interface toolkit presented in the next section.

The distribution of events are modeled in the dynamic cluster whose principal components are shown in the following diagram:

Figure 7. The Dynamic Cluster

Three types of objects are used in modeling the dynamic behavior of our applications: events, handlers, and dynamic objects. A dynamic object maintains a list of the different 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 it can transmit. A dynamic object transmits an event every time it 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 need to be informed whenever the event is transmitted. These subscriber objects then handle the event to implement their dynamic behavior.

3.4.3. Object Design

A dynamic object generates an event by applying to its event instances a feature called *transmit*. This feature is implemented in the EVENT class by applying a specific handling feature to all of its subscribers. This handler feature, which is implemented separately by each subclass of EVENT, has 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 button down event by application of the feature *handle_button_down*, and an event of class BUTTON_UP transmits by applying *handle_button_up*. As in the rendering cluster, this distinction is made completely through the dispatching mechanism inherent to the dynamic binding and not through conditional statements.

This particular representation of dynamic object behavior follows the concept of an event being a signal between two connected objects, a source and a target. The only information transmitted by the event itself, however, is its type, which is indicated by the name of the feature called. 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 according to the changes of state in the source objects and its own internal behavior. In a similar manner, secondary events can be transmitted by handlers that are themselves dynamic objects. In this way, the overall behavior of an application can be encoded to a large degree in the graph of connections traversed by the events.

The ability to handle events is represented by the HANDLER class, which implements the details of event dispatching. Several deferred subclasses of HANDLER exist, each one defining a general dynamic object that can handle certain types of events. Handler classes can be implemented by inheriting from these general handlers and redefining the handler features to respond to specific events. In this way, the dynamic behavior of an object is encoded entirely within its event handler features.

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.

3.5. User Interface

The user interface cluster provides the sorts of interactive capabilities associated with modern graphics workstations, in particular, a mouse, a windowing system, and standard types of 2D interaction widgets, like text input, sliders, and buttons. It also encapsulates in an object-oriented way some of the newer 3D input devices such as the Spaceball or the DataGlove.

This functionality was already available to us from an earlier development effort by our group to create a user-interface toolkit, called the Fifth Dimension Toolkit, for use in our laboratory. 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 based on [Cox, 86]. In designing this toolkit, which was inspired to a large extent by the NextStep AppKit [Thompson, 89], we consciously tried to make as purely object-oriented a design as possible.

A particularly useful feature of the 5D toolkit is an interface builder tool, modeled after the NextStep's interface builder, which allows panels of widgets to be arranged and their attributes edited interactively. The resulting user-interface widget 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 and the fact that numerous other application programs had already been developed using the 5D Toolkit, we decided not to start over from scratch in Eiffel but rather to encapsulate the 5D toolkit in a Eiffel class. This presented some problems, however, because encapsulating an object-oriented software library is considerably more difficult that a non-object-oriented one. Unlike traditional subroutine libraries, which usually 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. Any encapsulating Eiffel code, therefore, must either duplicate all of the underlying object's internal data structures itself, raising problems of consistency, or it must separately encapsulate each of the objects maintained by the encapsulated object. For us, this problem was further complicated by the fact that the toolkit event distribution mechanism, in which any dynamic object can send an event to any other object, would require events to be sent in both directions across the language boundary. 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 5D toolkit object. Since the toolkit objects are dynamically typed, the single UI class encapsulating all the toolkit functionality is reasonable. Most of the 5D toolkit objects are lower level and do not need to be directly accessed by Eiffel, so 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. When the instance is no longer referenced on the Eiffel side, it is garbage-collected by Eiffel even though the 5D toolkit object still remains. An interface was also built to translate 5D toolkit events into Eiffel event types as they occur. Because the Eiffel types of events were largely 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.

4. BUILDING APPLICATIONS

The following diagram gives a typical example of how various classes from the modeling, rendering and dynamic clusters are combined to form an application structure. In this case, a flight controller object is attached to a particular node in the modeling hierarchy. This controller subscribes to *new_transform* events from the trackball object. When the trackball receives a *mouse_moved* event, it responds by transmitting a *new_transform* event. When this is received by the flight controller, it responds by updating the new position of the node.

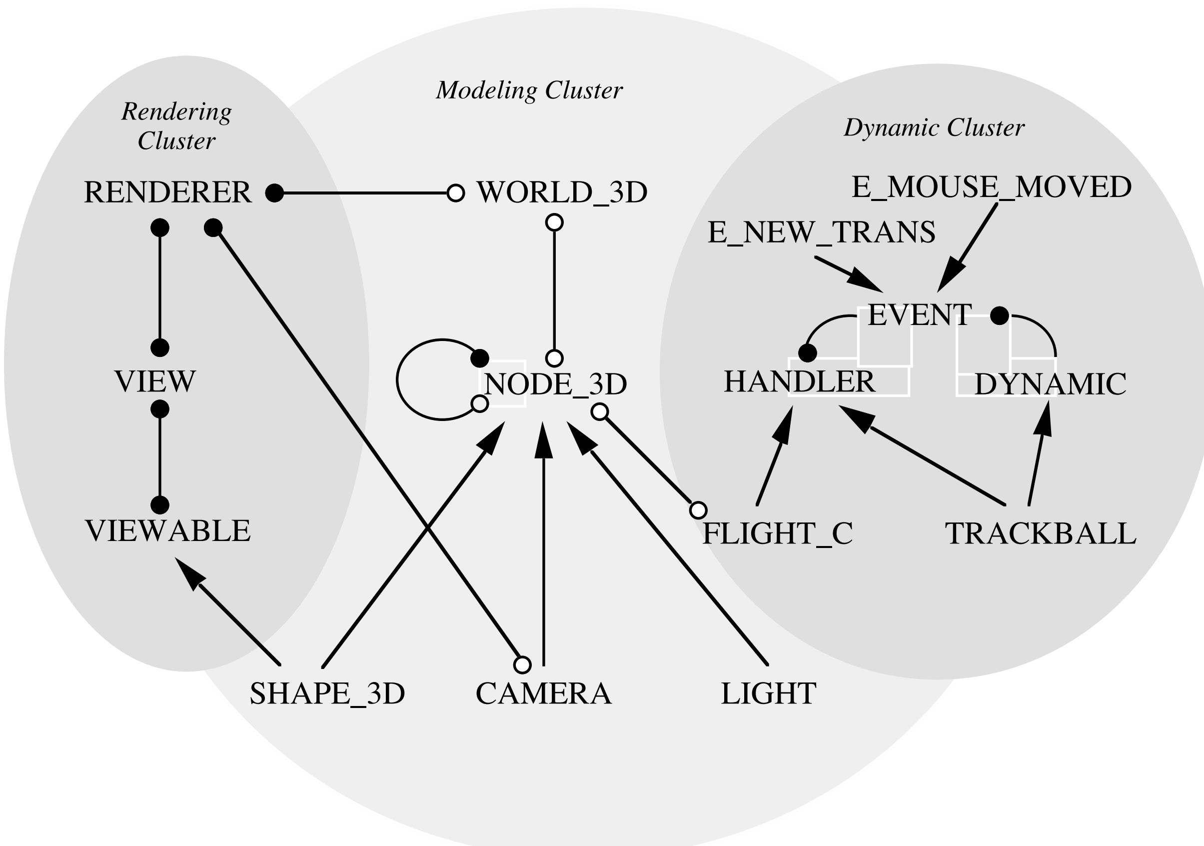

Figure 8: Example of Application Program Structure

Using the Eiffel 3D class library, an interactive 3D application program can be implemented by constructing a top-level class. When the program is started, it invokes the *Create* feature of this class, which instantiates the various objects, assembles them, and starts the event handling loop. The first part of the *Create* algorithm initializes the data structures, and loads in all the panels of user interface objects from files created by the user-interface builder. It then creates windows to contain these panels and to display the three-dimensional scenes. The visibility state of the different windows are set to reflect the initial state of the program. The event handling loop is then started up and managed by a top-level handler class, which in response to user input, implements the various application program commands

The application appears to the user as a collection of windows displaying either three-dimensional views of the world, or widget control panels. Combinations of windows may be made visible at any given time as desired. After the interface objects have been instantiated, controllers are then bound to the different widgets and devices using the subscription mechanism presented earlier. Specific instances of user-interface widgets can be identified by their names, created using the interface builder .

The interactive behavior of the application is determined by both the internal behavior of the controllers, as implemented in their event handler features, and by their connections to input devices, interaction widgets and other controllers. Some of the controllers that were written are general purpose and reusable in other applications. Others, like controllers that change the event subscription bindings to alter application behavior, are more application specific and difficult to reuse.

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 and panel widgets propagate through the network of bindings to the controllers which 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 virtual trackball metaphor. The spaceball is used to manipulate cameras, lights and nodes through differential transformations which can be interpreted according to various interaction metaphors, while the dial box is used to control other continuous parameters of the selected objects such as scaling, color, and camera viewing angle.

When the user selects a graphical object, using the mouse, appropriate controllers are bound to it and the affected interaction panels are updated to display the relevant information for the selected node. For the dial box, the action of selecting an object may also change which controller is tied to the device since the parameter controlled by each dial depends on the type of selected object. For example, if a shape is selected the dial box acts on the scaling parameters of its node, while if a camera is selected the dials control the field of view and the clipping planes. Usually, the spaceball is set up to controls the camera, while the mouse through its trackball controller object controls the currently selected node. Using a menu, the user can modify these default bindings and change the control metaphor for the selected node or the current camera.

5. DISCUSSION

5.1. Building reusable components

The bottom-up approach, which object-oriented design tends to promote for building applications, leads to the creation of software systems that are large assemblies of reusable basic components. We have to admit, however, that components are not usually reusable from the beginning, and some effort is usually required to make them general enough for exploitation in different applications.

The first step for making a component reusable is to make it usable in the first place. To ensure that this is true, client applications need to be written and the resulting feedback used to improve the design. For this reason, we believe that building test applications 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. One such test application that we have built is a key-frame animation system, which allows objects to be animated by interactively placing them in key positions and then interpolating a spline curve to obtain a smooth motion [Gobbetti, 1993].

Object-oriented techniques make it possible to exploit the similarity of structure of all applications in a particular domain 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 simply the reuse of code.

Several well known application frameworks exist, particularly in the area 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]).

We believe that our dynamic, modeling, and rendering clusters are a first step towards developing a framework for our interactive three-dimensional graphics applications. However, much work still remains to be done to make this framework general enough for the creation of future applications.

5.2. Performance

Performance is an important concern when building interactive 3D graphics program. Poor performance is often used as a criticism of using pure object-oriented techniques for the development of such applications, and as an argument in favor of using languages that freely mix the procedural and the object-oriented paradigm such as C++ ([Stroustrup 1986]) or Objective-C ([Cox 1986]).

The development of the key-frame animation system showed us that high performance applications can be obtained using a pure object-oriented language such as Eiffel without compromising the design. This particular application, built using the Eiffel class library 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, allowing the user to edit and animate three-dimensional shapes using a direct manipulation metaphor.

Several factors permitted this kind of high performance. The fact that the Eiffel 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), resulting in a high-performance code without compromising 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 ability to encapsulate highly-optimized FORTRAN numerical routines also contributed.

The availability of garbage collection in Eiffel often allowed us to simplify algorithms and data structures, resulting in more compact and efficient code. Since the garbage collector is incremental, it is possible to use it in interactive programs only at those times when it does not disturb the user. By carefully designing the components to minimize creation of temporary objects, we have been able to limit the CPU cost of object allocation and deallocation to under 10%. Previous experience developing a user interface toolkit using an object-oriented extension of C showed us the importance of these memory issues. In this 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.

6. CONCLUSIONS

Interactive 3D graphics is still in its infancy as a user interface paradigm. The challenge of building applications that realize the full potential of modern 3D computer graphics hardware remains immense. However, the development of object-oriented design techniques represents a significant advance toward the goal of creating reusable and extensible software components and assemblies for interactive 3D graphics software construction.

Our experience using a pure object-oriented design strategy and implementation language for building a general-purpose interactive 3D software library was very positive and showed us that these techniques are well suited for creating high-performance applications made of assemblies of reusable components in the field of interactive 3D graphics. In practice, it is almost impossible to build up a complete software system from scratch, and therefore even a pure object-oriented design needs some way to interface with existing software in traditional languages. We feel that the best way to do this is to have a well-defined separation between the object-oriented and traditional components of the software, in which the object-oriented components can be thought of as a higher-level language layer on top of the traditional language layer. This paradigm could be extended to a multi-layered approach with a declarative layer, based on constraint or logic programming, built on top of the object-oriented layer.

Most of the components that were created during this project are still being used 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.

BIBLIOGRAPHY

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

Conner DB, Snibbe SS, Herndon KP, Robbins DC, Zeleznik RC, Van Dam A (1992) Three-Dimensional Widgets. SIGGRAPH Symposium on Interactive Graphics: 183-188.

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

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.

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

Kay AC (1977): Microelectronics and the Personal Computer, Scientific American, 237(3): 230 - 244.

Krasner GE, Pope ST (1988): A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. Journal of Object-Oriented Programming 1(3): 26-49.

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.

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.

Sutherland I (1963): Sketchpad, A Man-Machine Graphical Communication System, Ph. D. Thesis, Massachusetts Institute of Technology, January 1963.

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.

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

Kubota Pacific, Inc. (1992) Doré Programmer's Manual.

Gobbetti E, Balaguer JF, Mangili A, Turner R (1993) Building an Interactive 3D Animation System, in Object-Oriented Applications, Meyer B, Nevson JM, editors, Prentice-Hall