Object-Oriented Design of Dynamic Graphics Applications

Enrico Gobbetti

Russell Turner

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

1. INTRODUCTION

The continued improvement and proliferation of graphics hardware for workstations and personal computers has brought increasing promenence 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 metaphores. 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. In the following paper, we use the term "dynamic graphics" to refer to this new style of program, which encompasses not only the now familiar mouse-based windowing applications but also real-time animation, interactive 3D, and virtual reality software.

Dynamic graphics techniques are now considered essential for making computers easier to use, and an 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.

Unfortunately, the creation and the implementation of such dynamic applications are a complex task: these systems have to manage an operational model of the real or virtual world, and simulate the evolution of this model in response to events that can occur in an order which is not predefined. They must handle multi-threaded dialogues with the user that are essential for direct manipulation interfaces, and make extensive use of many asynchronous input devices, ranging from the common keyboard and mouse to sophisticated 3D devices like Spaceball, Polhemus or DataGlove.

It was soon recognized that the classic approach of functional decomposition was not well suited for building these kinds of applications and that object-oriented techniques are more appropriate for this domain (Kay, 1977). In the following sections we will explain the reasons that lead to this point of view, and present an overview of some of the important object-oriented design priciples, issues and techniques that are involved in the construction of dynamic graphics systems.

2. OBJECT-ORIENTED GRAPHICS

2.1. Background

The relevance of object-oriented concepts for the purpose of building dynamic graphics applications has been recognized since computer graphics began, and many of these concepts were introduced by researchers working in the field of computer graphics. The beginning of modern interactive graphics are found in Sutherland's PhD work about the 2D interactive drawing system Sketchpad (Sutherland, 1963) which introduced concepts like creation of objects by replication of standard templates, hierarchical graphic structures with inheritance of attributes, and programming with constraints that can be considered as precursors of object-oriented technologies (Foley et al., 1990).

In the late 1960s, Alan Kay was working at the University of Utah on FLEX, a project for building "the first personal computer to directly support a graphics and simulation language" (Kay, 1977) based on the central ideas of the programming language Simula (Dahl and Nygaard, 1966) which can be considered the immediate ancestor of modern object-oriented languages. This work was continued later at the Xerox Palo Alto Research Center (PARC) where he helped to create a hardware and software system called *Dynabook*. The hardware part of Dynabook later became the Xerox STAR. The software part of Dynabook became the language Smalltalk (Goldberg and Robson, 1983). These have become the basis of modern graphical user interfaces.

The Smalltalk example showed that object-oriented techniques are ideally suited for dynamic graphics, but the performance problems raised by the fact that Smalltalk is an interpreted (rather than compiled) typeless language, limited for a while the main impact of this new methodology to the field of user interfaces and almost all workstations to appear since the early '80s come with some sort of object-oriented user interface toolkit.

2.2. Dynamic Graphics

As dynamic graphics applications become larger, more sophisticated, and move increasingly into the 3D realm, their software engineering problems have become acute. It is increasingly evident that the software design must make more use of assemblies containing standard components that can be reused and extended from project to project.

The recent availability of new compiled object-oriented languages such as C++ (Stroustrup, 1986), Objective C (Cox, 1986) and Eiffel (Meyer, 1987) make it possible for a new generation of dynamic graphics applications to emerge with fully object-oriented architectures. Such a development would be welcome because object-oriented design techniques seem to be the most promising solution to these problems.

2.3 Design Approach

In order to build any dynamic graphics application, from a simple two-dimensional user-interface manager to a complex three-dimensional animation system, three major problems have to be solved:

- How to describe and manage the model that the application is supposed to manipulate?

• How to render this model?

• How to obtain animation and interactive control?

A good solution to these questions should result in a system that is highly efficient, reusable and extensible. The object-oriented paradygm suggests that all relevent information should be encapsulated within objects. This leads to a major conceptual shift in design from a 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 new approach is actually a very intuitive one for most dynamic graphics applications, and it is quite natural to break an application up into objects representing subsystems and assemblies, each with its own appearance and behavior. There is, however, no unique solution to this problem, and the design of a good system structure is a problem that requires careful examination. In fact, as object-oriented graphics matures, and the basic problems such as languages and implementation are solved, larger design issues will increasingly dominate. The remaining sections will discuss some of these important issues and suggest some solutions for the problem of object-oriented design for dynamic graphics applications.

3. THE GRAPHICAL MODEL

3.1. The need for a Graphical Model

Since interactive and dynamic graphics must respond to real-time input events as they happen, it is usually impossible to know when and in what order the events will come. Therefore, at any moment during the execution of a dynamic application the entire state has to be explicitely mantained in a global data structure.

A common example of this principle is the simple one of drawing a graphical figure on the screen. In traditional graphics libraries, the programmer draws a circle by calling one of the circle drawing routines, which has the immediate effect of making the circle appear on the screen. However, there is no record that the circle exists, so in an event-driven system, subsequent events can not know about the circle, and it is the application programmer that has to explicitely take care of maintaining this information.

In an object-oriented system, however, a circle is drawn by first creating a circle object and placing it in the graphical hierarchy, then issuing a redraw command. In this way, the graphical state of the system is always known from one event cycle to the next: every visible figure on the screen (windows, widgets, geometric figures, etc.) is an object fitted into a single graphical database. All details of the object's appearance (dimensions, color, position, etc.) are maintained as state variables within the object data structures themselves. If the screen needs to be refreshed, this can be done by traversing these data structures.

3.2. Two-dimensional models

The first object-oriented design for modeling two-dimensional graphics was Smalltalk's class libraries (Goldberg and Robson, 1983). Many later object-oriented graphics and user-interface toolkits were inspired by this example. For example, Macintosh's MacApp classes, using an object-oriented version of Pascal (Schmucker, 1986), NeXT Inc.'s NeXTstep written in Objective-C (Webster, 1986) and the University of Zürich's ET++

(Weinand et al, 1989), written in C++, were all strongly influenced by the original Smalltalk design.

Two dimensional geometric models are utilized by a wide range of applications such as editable drawing programs, desktop publishing and data display. Often, 2D models are closely integrated with windowing system models and interaction objects or widgets, so that 2D models form the basis for most user-interface toolkits. The basic libraries of the Eiffel language (Interactive Software Engineering, 1989) offer a simple example of object-oriented encapsulation of the concepts needed for representing two-dimensional models and will be presented here.

Graphics programming in Eiffel is based on four basic notions: *worlds*, *figures*, *devices* and *windows*.

A *world* is the description of a two-dimensional reality, part of which will be displayed, under some representation, on a graphical medium. It has an origin and a coordinate system and represents the entire two-dimensional plane.

Figures are components of the world. They are geometrical entities, such as circles, rectangles or strings of text, whose size and position are expressed in world coordinates.

Devices are portions of the computer screen that have their own coordinate system and are used to display partial representations of worlds.

Finally, *windows* serve to establish the correspondence between the components of a world and their graphical representation of the device. Windows are defined by two rectangles: one in world coordinates, that defines the portion of the world that is captured by the window, and one in device coordinate, that defines the part of the device that is used for display.

The class and instance relationships of Eiffel figures are shown in the following diagram. This type of diagram represents classes as boxes and relationships between them as lines. The cardinality of the relationship is indicated by circles at the end of the lines. A filled circle represents a cardinality of zero to n, an unfilled circle represents a cardinality of zero or one, and no dot represents a cardinality of one. Subclass relationships are represented with arrows directed from the subclass to the superclass.

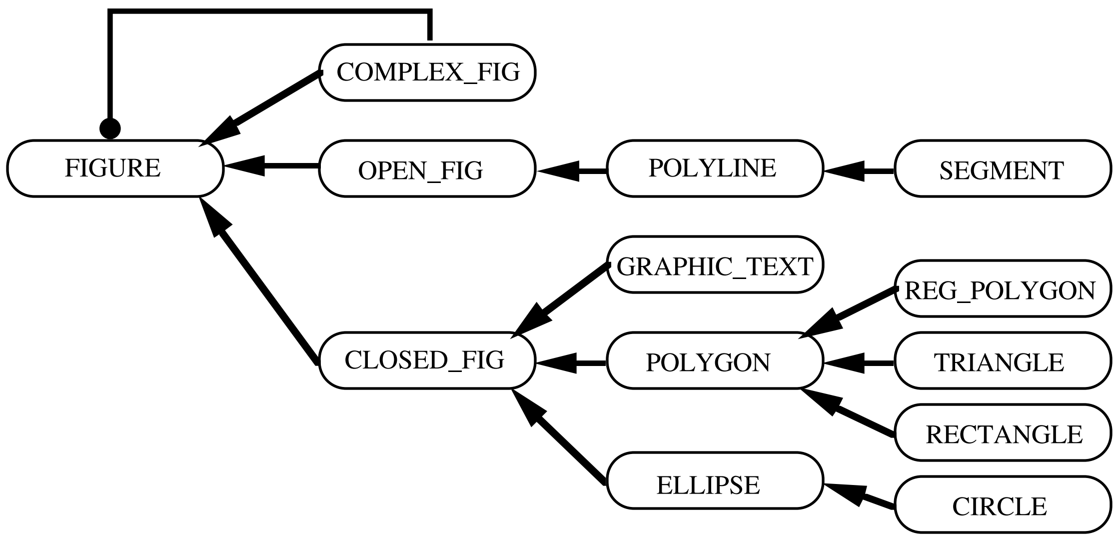

Fig 1: Two dimensional figures

Complex figures may contain any number of figures or other complex figures, forming an unlimited hierarchy. This hierarchy, which it should be stressed is an instance hierarchy and not a class hierarchy, allows complex figure objects to be arranged in a tree. Since each figure has its own coordinate system, the programmer can place graphical objects within the figure's local coordinate system and allow the default drawing methods to calculate the global coordinates.

Figures can therefore be said to inherit the coordinate systems of their parents, demonstrating a form of instance (as opposed to class) inheritance. Other characteristics can be inherited by figure's through the instance hierarchy such as foreground and background colors and other drawing characteristics.

3.3. Three-dimensional models

Several class hierarchies have been proposed for representing three-dimensional 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 by 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.

Three-dimensional dynamic graphics systems are typically concerned with the animation of models arranged in a hierarchical fashion (see Boulic at al., 1991). Such systems usually need to maintain the following kinds of information in their graphical data structures:

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

This kind of knowledge can be encapsulated in an object-oriented structure, with the responsibility of handling the different types of information decentralized among specialized classes.

An example of such a design, which used in the Fifth Dimension Toolkit developed at the Computer Graphics Laboratory of the Swiss Federal Institute of Technology in Lausanne (Turner et al., 1990), is represented in the following diagram.

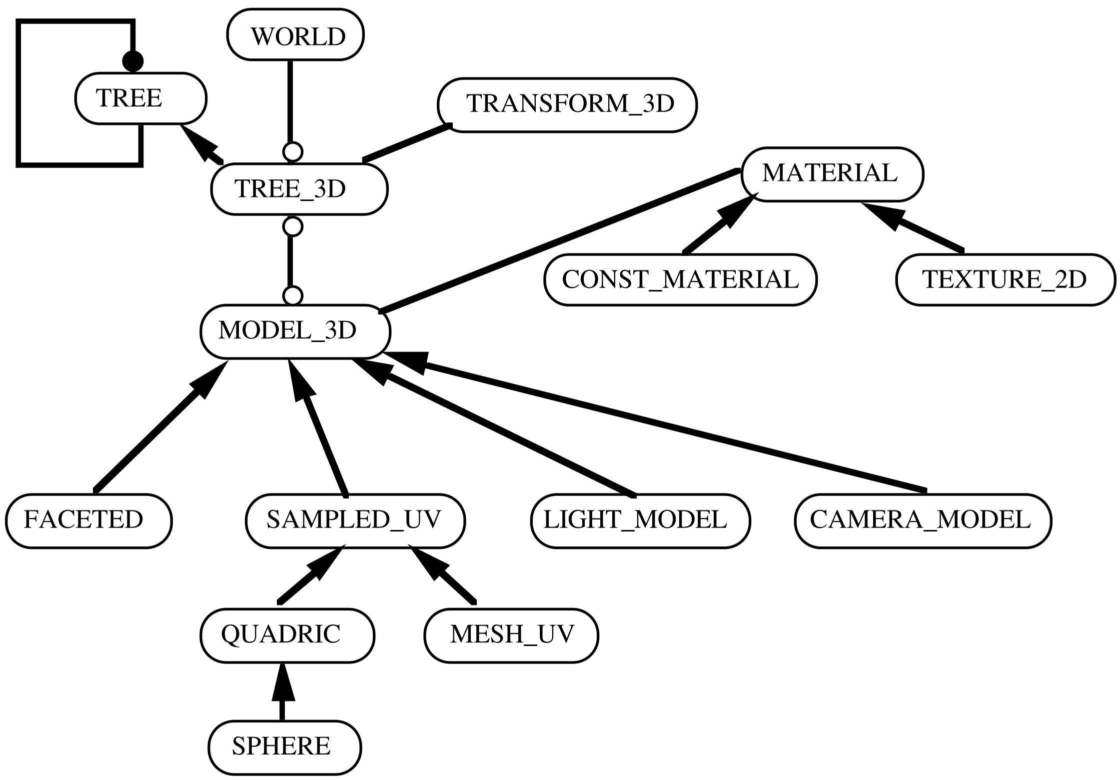

Fig. 2: Basic modeling classes

Four basic concepts can be identified in this diagram:

- the *world* represents the three-dimensional scene that applications manipulate and contains all the information that is global to a scene such as the global illumination parameters and the geometric hierarchies that are being manipulated.

- the *three-dimensional models* are encapsulations of the concept of a physical object having a shape in the Cartesian space. The different subclasses of MODEL_3D define different ways to describe and manipulate this shape, such as FACETED, whose instances representent geometric objects composed by triangular facets and SAMPLED_UV, whose instances are parametric objects sampled on two local coordinates.

- *materials* represent the way to give an optical behavior to the models. Examples are CONST_MATERIAL, which allows an object to be rendered with constant illumination parameters over the whole surface, and TEXTURE_2D, which allows the specification of surface properties that vary according to two texture coordinates. Default mappings exist for every kind of shape.

- the *transformation hierarchy* is used to specify the position of the objects in the world

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 objects and materials. The addition of new types of graphical objects, for example, is done by defining new subclasses of MODEL_3D and specifying the relevant operations. Programs that were able to manipulate graphical objects before this extension will be able, without any need for recompilation, to also manipulate scenes containing the new type of object.

4. RENDERING

4.1. Should Graphical Objects Draw Themselves?

The modeling hierarchies presented in the previous section is an example of how to organize and maintain the data structures in two and three-dimensional dynamic graphics applications. The next step in the object-oriented design process is to package this information together with operations defining how the data structures should be manipulated. These packages define classes that represent the operational model of the simulated world.

For dynamic graphics applications, 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? In simple 2D architectures they are usually encoded directly in the model For example, a slider object class will contain methods to redraw the slider in its current position and to update the position of the slider according to the position of the mouse. For more sophisticated applications, particularly in 3D, it usually becomes necessary to move this functionality out into separate classes. This is because one of the main goals in designing good component graphical objects is that they be reusable and extensable. To do this most effectively, they need to be general purpose, small and uncomplicated, encapsulating enough functionality to be useful without being difficult to reuse. Complex behavior requires complex design, and often more classes have to be added to the system in order to package some more functionalities. The decentralization principle that underlies object-orientation has to be applied at any level.

4.2 Rendering 2D Graphics

An example of how the rendering operation should be separated from the graphical object itself can be taken from the domain of device independant 2D graphics. Suppose there are a number of different geometrical object classes (e.g circles, rectangles, lines) which need to be rendered on different types of devices. However, some devices have hardware support for certain types of geometries, while others require implementation in software. If the rendering operation is implemented within the object, then each object has to be modified when a new device is added.

Where should this information be maintained and how can it be made accessible for the drawing function? The simple polymorphism obtained by definining a rendering function for every graphical object is not sufficient for handling this case, where the action to be performed depends on so many other factors. A better solution is to create new classes to handle the information related to rendering, such as type of algorithm and type of representations, and to use inheritance to create specific versions, such as ray-tracing, wireframe, and Gouraud shading. Dynamic binding can be used to choose the right implementation among all the possibilities. This method was introduced in the two-dimensional graphic classes of Smalltalk-80 (Ingalls, 1986).

4.3. Rendering a Three-Dimensional Scene

If we analyze the problem of rendering a three-dimensional scene, it is clear that, as in the 2D device-independant graphics example, implementing the rendering operation within graphical objects is not entirely satisfactory. Several reasons discourage this solution:

- lots of 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 object.

- rendering may be done using several output units, such as a portion of the frame buffer or a file, and all this knowledge has not 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.

An example of a design that uses separate modeling and rendering classes can be found in the Fifth Dimension Toolkit. Figure 3 shows the basic structure of its rendering classes.

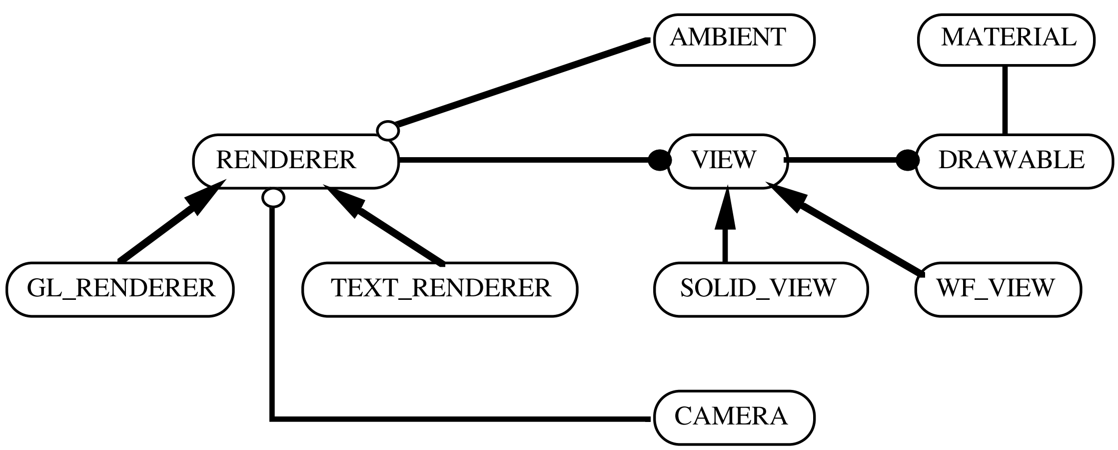

Fig 3: Basic rendering classes

Five basic sorts of classes can be identified:

• *renderers* (instances of subclasses of the RENDERER abstract class), that represent a way to render entire scenes. The code for actually rendering three-dimensional scenes is implemented here.

• *cameras* (instances of subclasses of CAMERA) that are objects able to return viewing information, such as the CAMERA_MODEL class.

• *ambients* (instances of subclasses of AMBIENT, such as the WORLD) that are objects able to return global illumination informations.

• *drawable models* (instances of subclasses of DRAWABLE, such as MODEL_3D in the modeling cluster) that are visible objects having position and shape.

• *representations* (instances of subclasses of VIEW), that define how a drawable object should be represented.

In this architecture, the representation objects act as intermediaries between the drawable models and the renderer, telling the renderer what technique (e.g. wireframe, solid) should be used to render each graphical object. The drawable models maintain only geometric information about the shape and position of the graphical object. The material objects attached to the drawable models maintain information about the optical properties of its surface. The camera object maintains geometrical viewing information used to project the drawable models into screen coordinates.

In order for a renderer object to be able to display a single graphical object, it must consult all of these other types of objects to determine the necessary drawing algorithm. This is done not through conditional statements but rather by using a dispatch method inherent to the object-oriented programming mechanism of dynamic binding called multiple polymorphism (Ingalls, 1986). The following diagram shows an example of application of this method.

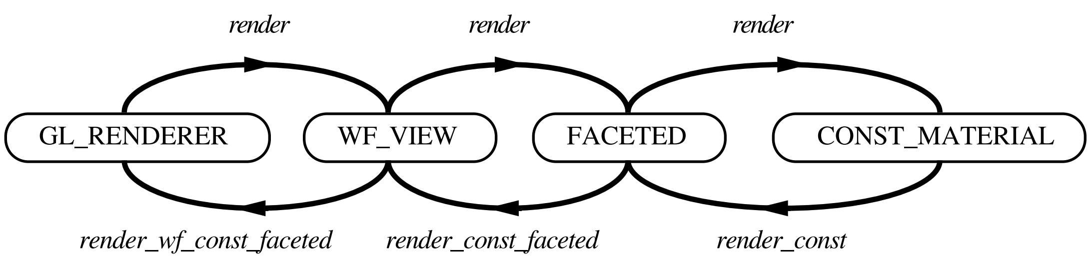

Fig. 4: Multiple dispatching

As the diagram shows, rendering a single object involves setting off a chain of message invocations, passing through the representation, drawable model and constant material objects, ultimately resolving to the appropriate rendering method. In this way, the composition of the instance data structure automatically determines the rendering algorithm.

5. DYNAMICS AND INPUT

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

There are at least two reasons that dynamic behavior can be so difficult to design. Unlike graphical entities, which can be easily modeled with data structures, dynamic behavior is more difficult to visualize and tends to be buried within algorithms. Secondly, there are at least three different software techniques for obtaining real-time input: asynchronous interrupts, polling and event queues. The challenge of dynamics in object-oriented design is: how to design and encode the behavior of graphics programs as easily as we can design and encode their visual appearance. To do this, we have to first try to construct a clear and understandable basic model of dynamic program behavior and then build a set of higher-level concepts on top of it.

5.1. Event-Driven Model

Most modern object-oriented graphics systems obtain their real-time input in the form of events in an event queue. Event queues are generally the preferred method, although some systems, such as the original Smalltalk, using polling and others, such as X windows (Nye, 1988), allow a mixture of the two.

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

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

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

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

5.2. Representing Events

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

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

5.3. Event Handlers and Messages

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

5.4. Distribution of Events

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

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

5.5. Decentralized Event Distribution

A general solution to these shortcomings can be found in NeXTstep's target/action metaphore (Webster, 1989). In NeXTstep's InterfaceBuilder, user-interface objects communcate via "action" messages which have a single parameter, the source. This source is the object which sent the action message and can be queried by the receiver of the message, or "target" object, for any associated data. User interface objects can be "bound" together so that when, for instance, a slider object is moved, it sends an action message to a second slider so that the two move in tandem.

This representation 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 is its type, represented by the selector name. Any other data must be explicitly queried from the source by the handler of the event. This eliminates the need for various different data structures for each type of event. The data is contained in the source object and it is up to the handler to decide which type of information to look for.

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

6. BUILDING APPLICATIONS

6.1. Object-Oriented Toolkits

In the previous sections we outlined some of the object-oriented principles that form a basis for the design of dynamic graphical software and presented how these techniques help producing software components that are more extensible and reusable. Providing large libraries of such components is a common solution for helping application programmers in their work. Object-oriented component libraries, often called *toolkits* or *toolboxes,* have been proposed in several fields, user interface software being perhaps the most influenced by this kind of approach.

User interface toolkits, whose design and look and feel is greatly influenced by the seminal example of Smalltalk's system, remain one of the major commercial successes of the application of the techniques presented in this paper. Some examples are Apple's Macintosh's Toolbox (Apple Computer, 1985) and Xt for the X windowing system *(Ref)*. and Sun's SunView (Sun Microsystems, 1986) These kinds of libraries offer to application programmers a collection of reusable user interface components, such as windows, buttons, sliders, and can be easily used from not object-oriented application programs. Much of the code of typical applications built on top of such toolkits is merely concerned with the creation and the assembly of instances of predefined components, and with the handling of the relevant events.

One consequence of this approach is that the object-oriented design process tends to shift from the traditional top-down to a bottom up strategy. In fact, the major effort in object-oriented graphics programming often is put into designing a good set of general-purpose graphical objects, such as a user interface toolkit, which are only later combined to form applications. This is conceptually similar to modern digital electronic design where more effort is put into designing the modular IC chip components than into the finished circuits they contain. (Cox, 1986) has emphasized this analogy.

However, simply calling a lower-level toolkit is not an entirely satisfactory solution to the problem of software reuse. Because of the similarity of the overall structure of dynamic graphics applications and the similarities between subsystems within the same domain, it should be possible to simplify the task of building up applications from scratch.

6.2. Application Frameworks

It is possible to exploit this similarity of structure 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 dynamic 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.

The idea of frameworks was developed at Xerox PARC by the Smalltalk group, and the first widely used framework was based on the model-view-controller (MVC) concept of design found in Smalltalk-80 (Krasner et al., 1988).

6.2.1. The Model-View-Controller Framework

The MVC framework is based on a uniform model of representing interactive graphical objects. To construct such an interactive object, three specific components, named *view*, *controller*, and *model* are required:

- the *view* object is concerned with rendering: and must know how to convert the important aspects of the model to a visible form.
- the *controller* object implements dynamics and provides the mechanisms that interpret input events as commands and updates the model accordingly.
- the *model* object maintains the information of the appication domain and provides the interface that allow controllers and views to acess it.

A model can be associated with many view-controller pairs and generic utilities are provided by the different classes of the framework to establish the connections between components and to propagate changes to maintain coherence. Figure 5 illustrates the behavior of MVC classes.

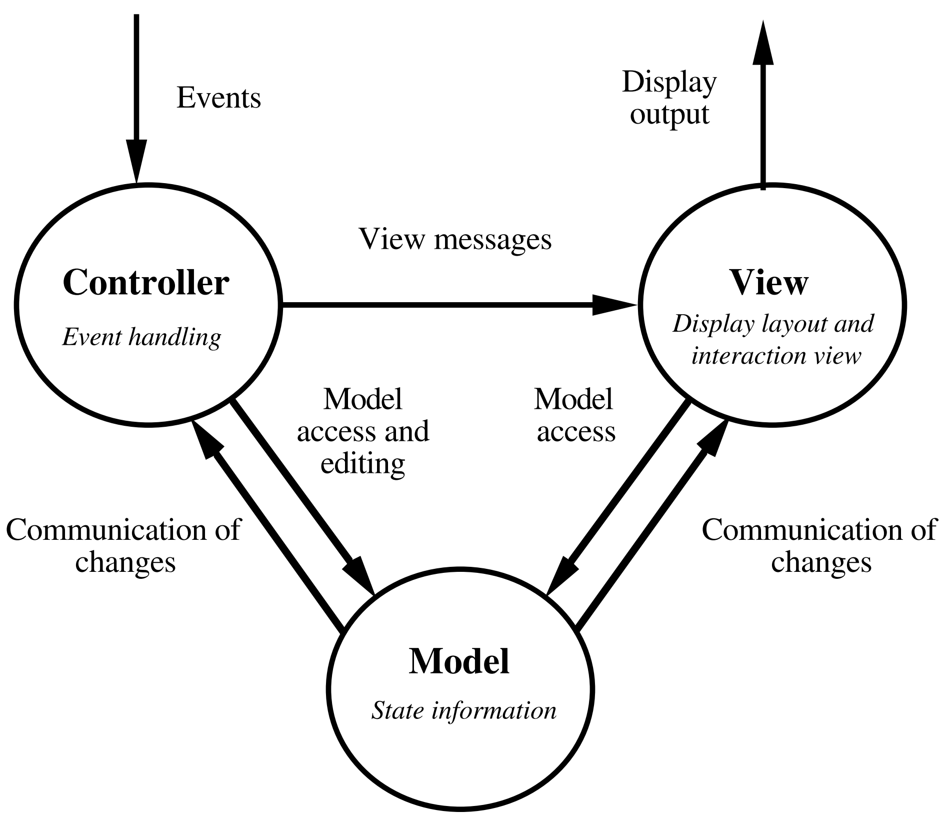

Fig. 5: Model-View-controller behavior.

An application program using the MVC paradigm is made by providing concrete subclasses of the Model, View, and Controller abstract classes to implement the behaviors specific to the application. These concrete classes usually must provide the implementation of a limited number of deferred methods, the rest being already supplied by the framework.

The separation between modeling, rendering and dynamics that has been described in the previous sections of this paper is enforced by the MVC paradigm. This fundamental division of powers may become a particularly useful concept in the design of future dynamic graphics application frameworks.

6.2.2. Other Frameworks

A number of user interface frameworks have been implemented on the basis of the MVC design, such as MacApp (Schmucker, 1986) that handles all aspects of Macintosh applications, and ET++ (Weinand et al., 1989) that offers the same features for Unix workstations.

One major advantage of this approach is that the guidelines for the user interface can be implemented in software, so to guarantee an uniform interface for all the applications and to increase in this way their user-friendliness. Such guidelines can be developed by teams composed of specialists from different domains (designers, psychologists, programmers) and an improvement in these guidelines can be incorporated into the framework with a great benefit to all of the applications.

Although most of the frameworks focus on user interfaces, this design technique, which is perhaps the most impressive realization of object-oriented ideas, can potentially be used in other application domains. A relatively unexplored research area is the development of application frameworks for 3D dynamic graphics applications. As 3D and interaction metaphores become better understood, the form that such application frameworks would take may become more obvious.

6.3. Interactive Construction of Applications

The use of frameworks provides a tangible basis for realizing a long sought dream of software engineering: Interactive creation of application programs. Although completely interactive software construction is at very best a long way off in the future, some promising attempts have been made in the real of specifying aspects of 2D user interfaces. These software construction tools are usually referred to as user interface design systems.

Perhaps the most promising of these to be marketed commercially is the Interfadce Builder application, which is part of the NeXTstep environment on the NeXT computer. The Interface Builder is particularly interesting because not only does it allow standard user-interface objects to be created and edited interactively (in a manner similar to an interactive drawing program) but it allows certain aspects of their assembled behavior to be specified as well. Using the target/action metaphore described in the previous section, the InterfaceBuilder allows connections to be specified graphically (by interactively drawing a line) between objects so that the data output of one object is automatically sent to the input of another. This allows a major part of the larger assembled behavior, if not the detailed individual behavior, of a group of objects to be designed and constructed without programming a single line of code.

7. CONCLUSIONS

The challenge of building dynamic graphics applications that realize the full potental 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. As object-oriented techniques become more accepted and as language and implementation issues are resolved, more attention can be focussed on the larger design issues. Application frameworks provide a structure into which software components can be assembled. General design principles, in particular the MVC metaphore, provide a basis for a more rigorous and well understood dynamic graphics design methodology. These principles can themselves form the basis for increasingly automated interactive software construction tools for building the next generation of dynamic graphics applications.

ACKNOWLEDGMENTS

We wish to thank David Breen and Kurt Fleischer for their suggestions in the research and our colleagues Angelo Mangili and Francis Balaguer for reviewing the text.

REFERENCES

Apple Computer (1985), Inc., Inside Macintosh, Addison-Weslez, Reading MA.

Berard E (1989): *What is object-oriented...*, Comp.object, UseNet, World.

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

Dahl OJ, Nygaard K (1966): *SIMULA -- an ALGOL-Based Simulation Language*, Communications of the ACM, 9(9): 671-678.

Dodani, Hughes, Moshell (1989): *Separation of Powers*, Byte 14(3): 255-262.

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

Foley J, van Dam A, Feiner S Hughes J (1990): Computer Graphics: Principles and Practice (2nd ed), Addison Wesley, Reading Massachusetts, 1990.

Goldberg A, Robson D (1983): Smalltalk-80: The Language and Its Implementation, Addison-Wesley, Reading, Massachusetts.

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).

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.

Korson, McGregor (1990): *Understanting Object-Oriented: A Unifying Paradigm*, Communications of the ACM 33(9): 40-60.

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.

Nye A (1988): Xlib Reference Manual, O'Reilly & Associates, Inc.

Sun Microsystems (1986): SunView Programmer's Guide, Sun Microsystems, Mountain View, CA.
Sutherland I (1963): *Sketchpad, A Man-Machine Graphical Communication System*, Ph. D. Thesis, Massachusetts Institute of Technology, January 1963.
Thompson T (1989): *The Next Step*, Byte 14(3): 265-269.
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.
Webster BF (1989) The NeXT Book. Addison Wesley.
Weinand A, Gamma E, Marty R (1989): *Design and Implementation of ET++, a Seamless Object-Oriented Application Framework*, Structured Programming 10(2): 63-87.
Stroustrup B (1986): *An Overview of C++,* SIGPLAN Notices 21(10): 7-18.
Boulic R, Renault 0 (1991): *3D Hierarchies for Animation*, in New Trends in Animation and Visualization, John Wiley.