Object-Oriented Paradigms for Graphical-Object Modeling in Computer-Aided Design: A Survey and Analysis

Sandeep Kochhar
Computervision

Abstract

Making graphical-object modeling — the task of creating graphical objects — easier is one of the most important challenges facing the CAD and computer graphics community. To make the modeling task easier, many researchers have focused on object-oriented techniques, and over the last decade or so, several *object-oriented graphics* paradigms have been examined, with the goal of bringing established benefits of object technology — reusability, extensibility and maintainability — to graphical object modeling.

However, while the term "object-oriented graphics" is widely used, the paradigms developed by researchers differ widely in terms of the domains to which they are applicable, the tasks that they are meant to simplify, the amount of extensibility they offer, and the relationships they have to other subsystems in a large graphics application.

I provide a comparative description of the approaches used for object-oriented graphical modeling, especially in the context of CAD applications since these place the heaviest demands on graphical-object modeling capabilities. This work serves two purposes: 1.) to provide a conceptual framework for comparing existing paradigms, and 2.) provide an analysis of a few key systems, that can help the practitioner choose among the set of existing paradigms in specific application scenarios.

CR categories and subject descriptors: I.3.5 Computational Geometry and Object Modeling; I.3.6 Methodology and Techniques; I.3.m Object-oriented Graphics; D.1.5 Programming Techniques — Object-oriented Programming; J.6 Computer-Aided Design (CAD).

Additional Keywords: object-oriented graphics, graphics software architectures, graphical design and modeling, user interaction.

1 Introduction

The task of creating graphical depictions has traditionally been divided into two tasks: *graphical-object modeling* (creating graphical objects to describe the entity to be visualized) and *rendering* (creating an image from the model). Most of the research in computer graphics has focused on the rendering task, with the explicit goal of creating photo-realistic images. Recently, however, modeling has begun to receive more attention since most graphics applications do not provide powerful modeling paradigms, consequently placing severe demands on applications.

Author's address: Graphics Technology Group, Computervision R&D, MS 5-2, 14 Crosby Drive, Bedford, MA 01730, USA. Telephone: (617) 275-1800 x 4618. Electronic mail: kochhar@das.harvard.edu.

Modeling support in most traditional graphics systems (GS) is based on the conventional "structured display file (or list)" paradigm [1]: a basic structured display file (SDF) typically is a directed, acyclic graph, with the nodes representing geometric and attribute information and the edges representing modeling and camera transformations. Such systems include those based on the PHIGS [2] and GKS [3] ISO standards, and other widely-used systems such as Ithaca Software's HOOPS [4] and Pixar's RenderMan [5]. These systems are usually well-suited for general purpose graphics applications, especially those requiring photo-realism (ray-tracing, radiosity, etc.). However, CAD/CAM applications typically place a much higher demand on modeling capabilities and require higher-level abstractions. In addition, the development of new CAD applications in complex domains is often severely hampered by the the lack of common, reusable modeling components, which can result in several person-years of development being put into a new graphical modeling component, or in adapting an existing graphical modeling component.

To make the modeling task easier, many researchers have turned to object-oriented (OO) technology, with the goal of bring its proven benefits — reusability, extensibility and maintainability — to graphical-object modeling. Over the last decade or so, several *object-oriented graphics systems* (OOGS) and paradigms have been developed, and graphics class libraries or *toolkits* are now available. However, while the term "object-oriented graphics" is widely used to describe all such paradigms, these paradigms often differ widely in terms of the domains to which they are applicable, the tasks that they are meant to simplify, the amount of extensibility they offer, and the relationships they have to other subsystems in a large graphics application.

CAD applications — with their heavy demands on modeling capabilities — seem well-suited to benefit from object technology. Unfortunately, the varying meanings and capabilities of OOGS and paradigms makes it very difficult to choose the paradigm that best matches a particular application. For example, the requirements of a large mechanical CAD application are quite different from those of a CAD system for designing user-interfaces.

Thus, while it seems natural for a research and development group designing a new graphical-object modeling system to adopt object-oriented technology and paradigms, it is usually not obvious — without significant research and analysis — which aspects of the problem are amenable to (and likely to benefit from) such techniques. The goal of this paper is to survey several key systems from the recent literature and examine the approaches that they have used to apply object technology to their particular problem domain.

I provide a comparative description and classification of several approaches used for object-oriented graphical modeling,

with an emphasis on their suitability in the CAD application domain. The audience that I expect this paper to benefit consists principally of two groups: for the practising engineer, this paper provides a basis for choosing the appropriate complement of the available OOG approaches when developing new CAD applications; for the researcher, this study provides a useful framework for understanding and comparing OOGS.

Background and Relevant Work

My organization produces a large number of CAD/CAM products, across a wide spectrum of domains, and differing widely in complexity. One of the goals of my group was to design the common graphics modeling subsystem for the next generation of our CAD/CAM products. We began by analyzing the existing graphics capabilities and those needed in the future to formulate a comprehensive list of requirements for the graphics modeling subsystem (some of these will be discussed below). We then decided to use object-oriented technology for precisely the reasons listed earlier — to obtain the benefits of reusability, extensibility and maintainability. An analysis of our CAD/CAM requirements in comparison to the existing, published literature on OOGS led to the conceptual framework (described in this paper) for understanding OOGS. In addition, we formulated the concept of an object-oriented presentation system and developed a novel OOGS — the Unified Graphics Subsystem — that I describe later in this paper.

I expect that this work will benefit both researchers and practising engineers as I mentioned above. Two important themes in current research on OOGS are the specification of standardized APIs (application programming interfaces) for OOGS, and the specification of future graphics software architectures. For example, the SIGGRAPH '91 panel on "Object-Oriented Graphics" [6] focused on the need for standardization in the area of class libraries for graphics applications beyond user interface toolkits, while one of the major themes at the SIGGRAPH '92 panel on "Graphics Software Architecture for the Future" [7] was the nature of objects in "future" graphics subsystems. Some of the issues discussed by the panelists in both panels are similar to ones I use as a basis of classifying OOGS, including the need for a separation between the application modeling, graphics modeling and rendering levels, and the need to support multiple application paradigms.

Brief Review of Object-Oriented Terminology

While this paper assumes familiarity with object-oriented technology, I briefly review some basic terms here (details can be found in any text on object-oriented design,e.g. [8]). *Objects* are discrete, distinguishable entities that encapsulate data and behavior. A *class* is an abstraction that describes a collection of objects with the same data structure (attributes) and behavior (operations, also known as methods or messages). An object is said to be an *instance* of its class. *Inheritance* allows the sharing of attributes and operations amongs classes based on a hierarchical relationship: a *subclass* inherits attributes and behavior from a *superclass* and can selectively refine these, or add new ones. *Polymorphism* allows the same operation to behave differently on different classes, e.g. the triangle and circle classes in a drawing program may have different behaviors for the drawSelf operation. (In C++, such operations are known as *virtual methods*.)

One of the key issues in the design of an OOGS (or any OO system) is that of *extensibility*. Extensibility can refer to the capability to extend the object (class or instance) hierarchy, or to the capability to interface with new subsystems (e.g., new renderers or geometric modelers). The ability to add new objects or classes depends heavily on the implementation language. Three common approaches to extending a class hierarchy are:

- *inheritance:* an application can inherit (or, *subclass*) from the classes or objects in an OOGS and add application-specific data and methods, as well as override existing methods (to support polymorphic behavior). Multiple inheritance — inheriting from two or more classes — is often required by application objects that perform multiple roles, as is common at the higher-level subsystems in a CAD system (explained below).

- *layering* and *delegation:* an application can also define its own class hierarchy, in which a particular class might include a reference to an object from the OOGS. This is referred to as layering. In such cases, when an application object receives messages (method invocations) that are intended for the OOG subsystem, it simply forwards these messages to the OOGS object that it references. This is referred to as delegation.

- *callbacks*: by allowing the application to specify callbacks that are invoked when an object is acted upon in a certain context (e.g. selected or traversed), the OOGS can let the application extend the functionality provided by the OOGS (since the application can, at least theoretically, do "whatever it wants" at that stage).

As I discuss several OOGS in Section 3, I will also indicate which of the above approaches have been used by those OOGS.

CAD System Architecture and Requirements

Since one of the goals of this paper is to examine OOG paradigms in the context of CAD systems, I review briefly the structure of a typical CAD system and the requirements it imposes on any OOG subsystems. CAD systems vary widely in terms of application domains and complexity. At one end of the complexity scale, a simple drawing editor might only have a small number of internal modules: rendering, graphical modeling, and application-specific modeling. At the other end, a large mechanical CAD system will often have over a dozen fairly complex subsystems. Figure 1 shows a typical layered architecture supported by large CAD systems. The modules comprising the bottom layer in the architecture consists of the core subsystems, such as Database, Geometry (modeling), User Interface, or Graphics (scene description and rendering). Upwards in the hierarchy are layers of applications built on top of the core subsystem and on other application modules. The lower levels of these — for example, the Selection, Animation, and Constraint-management modules use the core subsystems to provide other services for the next level up. As we move upwards in the layered architecture, the modules get more domain-specific: for example, a Piping module would be used for AEC (Architectural

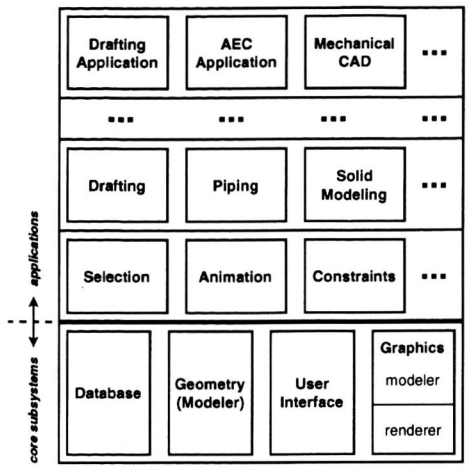

Figure 1: Architecture of a Large Mechanical CAD System

Engineering and Construction) applications; a fairly complex mechanical CAD package that supported Parametric History Editing, Feature-Based Design and Variational Geometry might include Drafting and Solid Modeling applications, along with the Constraint-management module.

One of the key concepts in CAD/CAM applications is that of a *presentation*. A CAD/CAM presentation is a graphical picture that is intended to convey information about a product or some part of a product. In the architecture shown in Figure 1, an application typically interacts with the user interface and geometry subsystems to create a mathematical model of a product or some part of a product. At each step in this process the application requests that the graphics subsystem update its visualization of the product or part being created. Additionally the application may want to add extra information (annotations, dimensions, etc.) to the presentation of a model by directly interacting with the graphics subsystem. The user can also *select* (or *pick*) various entities to perform operations on them as a set.

Most CAD/CAM presentations involve a set of common concepts, including: areas (e.g. views, drawings), groupings (e.g., layers), view-specific styles (e.g., fill areas and line fonts that depend on the viewing context), scale-invariant graphics (e.g., surface normal depictions whose size remains constant even when zooming), picking, style inheritance, persistence, entities, modeling and camera transforms, highlighting, zooming, and editing. Supporting these concepts requires that several decisions be made by the OOGS developers. For example:

- There is usually a mismatch between the capabilities listed above as requirements and those offered by the underlying rendering modules. A CAD/CAM graphics subsystem thus has to map the CAD/CAM-specific concepts onto the graphical primitives supported by the rendering modules. A OOGS designer needs to address the issue of how the application chooses the best mappings to optimize the display list structure and rendering.
- Should the geometric (modeling) and graphics (scene description and rendering) subsystems be separate? Combining them makes it difficult to integrate a different geometric modeler into the system; separating them completely can require duplication of product data in both subsystems, since there is no way to share representations. For a small graphics application, the choice may be a matter of expedience; however, both of the above choices are impractical for large CAD/CAM applications, and the developers need to design ways of sharing data between subsystems.

The answers to the above questions are difficult to formulate in the absence of a coherent framework for understanding the characteristics of the various OOG paradigms. One of the major goals of this paper is: to assist in this task by clearly delineating the important issues that make an OOG paradigm suitable or unsuitable for a variety of graphics applications — from small drawing editors to large CAD packages.

2 Conceptual Framework

In order to present the conceptual framework for organizing the OOG paradigms, I first consider the lower levels of Figure 1. Applications based on an OOGS range from simple systems that include only the Graphics Modeler and Renderer to complex applications that include not only the core subsystems from the figure, but also the Selection, Animation, and Constraint-management modules. The key difference between the OOG paradigms is based on the organization of the objects in the OOGS: how much behavior from each of subsystems from Figure 1 is incorporated into the objects in the OOGS? The two extreme cases are:

- to have completely standalone modules (the objects in the OOGS do not have any behavior or attributes related to the other modules), and,
- to have completely integrated modules (the objects in the system include not only graphical capabilities, but also the capability to interact with the database, respond to selection queries, satisfy constraints, and change their behavior over time in order to produce animations).

In Section 3, the OOG paradigms that I discuss fall at various points between these extremes.

For the purposes of comparing OOG paradigms within the above conceptual framework, I consider the following issues as important "dimensions" along which the paradigms differ:

- *modeling level*
- *selection handling*
- *temporal (time-dependent) behavior*
- *event handling and display model*
- *relationship to geometric modeler*
- *relationship to rendering subsystem*
- *relationship to database subsystem*
- *constraint handling*
- *extensibility*

These dimensions can be viewed as providing a recipe for choosing the appropriate complement of characteristics when designing an new OOGS. Let us explain each of them is some detail.

Modeling Level

One of the distinguishing factors between OOG paradigms is the level at which the paradigm operates: *graphical rendering*, *graphical modeling*, or *application modeling*:

- By *application modeling*, I refer to the process of describing the object model in terms of concepts appropriate for the application; for example, a mechanical CAD user would describe a part in terms of assemblies, features, etc.
- By *graphical modeling*, I refer to the process of describing the visualization of the object model in terms of traditional graphical concepts, such as structures (segments), lighting and appearance attributes, and transformations.[1]
- By *graphical rendering*, I refer to the process of converting the graphical model into a picture (image) suitable for display on some output medium (CRT, printer, etc.).

OOG paradigms have been developed to address each of the tasks of rendering, graphical modeling, and application modeling. While the emphasis of this paper is on modeling, for completeness I include a discussion of OOG paradigms for rendering in Section 3.

Selection Handling

The *selection* (or *picking*) operation allows a user to specify a criteria (for example, enclosure within a 2D region) for creating a set of subobjects that can then be subject to some common operation, such as scaling. Internally an OOGS can employ two distinct strategies to support selections:

- each object in the OOGS can have the capability to respond to selection queries, or,
- a separate selection subsystem can interface between the user-interface and graphics subsystems to support selection operations.

I consider the former an *input/output* OOGS, since it is capable of handling user-input (in addition to the output of graphical depictions); I consider the latter an *output-only* (display-only) OOGS.

In large CAD/CAM applications, a single graphics subsystem has to be useable by a variety of user-interface subsystems, each supporting its own user-interaction paradigm. In such scenarios, the graphics subsystem is used only to display visualizations of the mechanical parts, and thus is output-only. In other applications, for example drawing editors and user-interface builders, it is more natural for the objects (which often are user-interface entities anyway) to respond to user events, a task to which an input/output OOGS is better suited.

[1] In [9], Kochhar, Marks and Friedell view the graphical-modeling process itself as a combination of two different activities: *design* and *articulation*. Design is the more creative aspect of modeling. Articulation is the activity of providing a precise graphical description of an object conceptualization. However, in this paper, I treat graphical-object modeling as a whole when analyzing the OOG paradigms, since currently the design/articulation distinction has not sufficiently pervaded object-oriented graphics paradigms to be useful as an important comparative factor. I would like to investigate this issue in my future work.

Temporal (Time-Dependent) Behavior

Another issue that differentiates among OOG paradigms is whether the paradigm supports the notion of temporal or time-dependent behavior in objects:

- Some OOGS allow an application to specify time-dependent behavior in objects; these objects can render themselves automatically as the time changes, thereby producing animations.
- In other OOGS, where objects do not include time-dependent behavior, animations are produced by allowing the application to vary the structure of the object instance-hierarchy or vary attributes of objects, and then re-rendering the scene.

Event Handling and Display Model

Another important issue in the OOG paradigms is

- whether objects in the OOGS should include user-interaction event handling, or,
- whether events should be handled by an external subsystem (such as the user-interface subsystem), which relies on the application to manipulate the graphics objects appropriately in response to user actions.

Important user-interaction events are those that relate to direct responses to graphical gestures performed by users (for example, mouse clicking, dragging, and resizing) — some of these might imply only a redisplay of portions of the object's image; others imply that an object change its appearance significantly (for example, highlighting itself when the mouse is clicked within its bounding box).

OOG paradigms that include event handling as part of the objects facilitate the creation of systems for drawing editors, user-interface builders, etc.; however, one limitation of this approach is that the "look and feel" of the user-interaction is controlled by the objects, and is not easily adapted to the look-and-feel supported by the overall application or windowing system. In a large suite of CAD packages that often have to support differing look-and-feel, this approach is not feasible if the OOGS is to be reusable by many applications. In this scenario, the preferable approach is to delegate the user-interaction handling and customization to the user-interface subsystem.

Relationship to Geometric Modeler

Consider a mechanical part being designed using a CAD system. The user needs to specify both the geometric representation and graphical visualization of this part:

- The geometric *representation* of a part refers to the application-specific geometric and product data needed for purposes of analysis and manufacturing.
- The graphical *visualization* refers to a description of the graphical structures needed to display the part (or to produce manufacturing drawings).

As an example, the product data might include: geometric descriptions of the subobjects and assemblies; topological relationships between faces, edges and vertices of subparts; materials-related information; and, manufacturing tolerances. The visualization of the same part, on the other hand, might require a hierarchy of graphical primitives, along with styling, modeling and viewing information. A key issue that an OOG paradigm needs to address is whether the objects in the OOGS incorporate both types of information or only the data needed for visualizing the product.

Large CAD/CAM applications normally have separate geometric (modeling) and graphics (scene description and rendering) subsystems. An OOGS that combines geometric and graphical modeling would usually be impractical for these applications, since such an OOGS often requires duplication of product data in the geometry and graphics subsystems, which results in severe performance overheads (in terms of time, memory and disk usage).

On the other hand, OOGS that allow the creation of objects containing both product (geometric) data and visualization data are useful for small applications, especially when a separate geometric-modeling subsystem is not easily available or implementable for pragmatic reasons. This is the case, for example, in drawing editors and simple drafting packages.

Relationship to Rendering Subsystem

OOG paradigms also need to be concerned with the issue of separation between the graphical modeling subsystem and the rendering system:

- Some OOG paradigms rely on a clear separation between the two subsystems. In this scenario, the rendering subsystem has a well-defined set of primitives that it is capable of displaying and the OOGS is responsible for converting objects to be displayed into those primitives. The main advantages of this approach are that new rendering techniques can be added without modifying the OOGS, and that the rendering system can make decisions about displaying the primitives in the most efficient manner. A disadvantage of this approach is that the OOGS can not easily take advantage of newly added rendering primitives, nor of context-specific knowledge about the efficiency of rendering primitives (for example, a certain primitive might be slow when rendered on one device and much faster on another). Some OOGS have addressed this issue by allowing closer communication between the two subsystems: the modeling system can query the rendering system for available drawing primitives and the efficiency of rendering them, and then decide on the best collection of primitives into which to transform the object to be visualized.

- Other OOG paradigms incorporate rendering methods into the objects themselves: thus, each object is capable of displaying itself in a given graphical context, without requiring the use of a separate rendering subsystem. One advantage of this approach is that objects can be copied (or sent via messages) from one application to another, without the latter knowing how to display the objects it receives (since these objects are capable of rendering themselves) — this allows complex objects to be "cut and pasted" just as easily as is possible with simple objects like Ascii text. The main disadvantage of this approach is that when different rendering techniques or primitives become available (for example, on a different display device or graphical accelerator), every object's implementation has to be modified.

Relationship to Database Subsystem

This issue is very similar to the previous one. An OOG paradigm can directly support persistent objects (using an OO database), can support objects that include archival methods, or can use the services of a database subsystem (with a well-defined interface) by converting the objects' attributes and relationships into a form suitable for archival. The retrieval of an object hierarchy follows the same principles: persistent objects may be retrieved automatically (by the OO database manager), objects may have methods for retrieving their attributes and relationships, or the OOGS may convert data retrieved by the database subsystem to recreate the object hierarchy.

Constraint Handling

An object model typically needs to satisfy a variety of constraints: graphical (for example, limits on available colors), geometrical (for example, minimum sizes of faces in a boundary representation), topological (for example, consistency between the numbers of faces, edges and vertices in an object), and product- or domain-specific (for example, minimum and maximum stress loads, tolerancing limits). Systems that simulate physical models also need to take into account constraints that describe the relationships between interactions of physical objects in the "real world." The manner in which constraints are described and satisfied is another distinguishing factor between the OOG paradigms. Some OOGS directly support "constraint objects" that incorporate methods to satisfy their conditions, an approach widely used in constraint-based drawing editors. A limitation of this approach is that incorporating all the types of constraints that an object might need to satisfy (as listed above) can be a difficult task. Moreover, adding new constraint types can result in significant modifications to the OOGS.

The alternative approach — suitable for large OOGS (e.g., a large mechanical CAD package that supported Parametric History Editing and Variational Geometry) — is to have separate constraint solvers (for the variety of types of constraints), which the application can invoke in response to user actions; these solvers obtain attribute-values for and manipulate the objects in the OOGS through the defined interfaces. Thus, new constraint solvers can be added without modification to the OOGS.

Extensibility

As I mentioned in Section 1, extensibility in an OOGS can refer to the capability to extend the object (class or instance) hierarchy, or to the capability to interface with new subsystems (e.g., new renderers or geometric modelers). While the mechanisms of inheritance, layering, and callbacks allow an OOGS to be extensible, several pragmatic issues must be addressed during the design of a large, extensible OOGS:

- One issue is that of knowing in advance which internal parts of the OOGS will need to be exposed in order for the extensions to be useful. For example, an OO renderer (from the next section) might have several useful objects and mechanisms to compute the intersections of a variety of objects; if these are not exposed, then extensions to the ray-tracing related objects might not be possible.

- If an OOGS itself relies on the support of an underlying subsystem, then capabilities of that subsystem might need to be exposed in order for the application to benefit from extensibility. For example, an OO presentation system (from the next section) typically uses the services of a low-level rendering system. Suppose that this low-level rendering system supports a "triangle mesh" primitive, but the OO presentation system does not initially include a class that directly corresponds to a triangle mesh. If, however, an application is expected to extend this OOGS by adding NURBS and complex mesh-based surfaces, then the OOGS needs to allow the application to use the services of the underlying rendering subsystem, which might require exposing the "triangle mesh" primitive.

- In a callback that an application provides, it is often difficult to know in advance what context (current state) information to provide to the application callback routine so that it can perform meaningful operations. As an example, if a callback is invoked during display-list traversal, the callback would need to be handed the current graphics context (and possibly methods for modifying it), output device, etc.

3 Object-Oriented Graphics Paradigms

I now present several OOG paradigms in the context of the conceptual framework described in Section 2:

- *Object-Oriented Rendering*
- *Object-Oriented Structed Display File (SDF)*
- *Object-Oriented Animation*
- *Object-Oriented User-Interface Builder (UIB)*
- *Object-Oriented Drawing and Document Editing (DDE)*
- *Object-Oriented CAD/CAM Presentation*

The issues from Section 2 will be examined with respect to the paradigms. Figure 2 shows the issues that I believe play an important role when a developer has to decide on the appropriateness of a particular paradigm for some application system. Thus, for example, the interaction between the Selection subsystem and the OOGS is an important factor in the cases of the OO Animation, UIB and Presentation paradigms.

I also describe some recent OOGS that illustrate the principles of each paradigm.[2] Note that any particular system that one might examine is certain to be based on more than one of the paradigms listed above. For example, an object-oriented rendering system is likely to have at least an object-oriented structured display list as well. However, I discuss the paradigms separately in order to explain the salient issues.

Object-Oriented Rendering

At one end of the graphical image-generation process is rendering. I refer to a GS as based on the object-oriented rendering paradigm if the main goal of that system is to attempt to improve the quality of rendering software — in terms of better maintenance, extensibility, and reuse — by applying object-oriented techniques. Such systems typically use objects and relationships to represent the entities and stages involved in the rendering process, for example transformations, clippers, colors, lights, scan lines, and ray intersectors. These systems also support extensibility via subclassing and polymorphism; for example, in a ray tracing system that relies on intersections of a **ray** object with an abstract **shape**, a new graphical primitive, say **tetrahedron** might be added as long as it incorporates methods to respond to **compute-intersection** messages. Thus, object-oriented rendering is characterized by the following motivations:

- to use object-oriented technology to improve the rendering software

- to allow extensibility via subclassing and polymorphism

An example of an object-oriented rendering system is Melcher and Owen's RTCPP system for ray tracing [11]. RTCPP contains geometric objects that derive from an abstract base object **geo_object** that contains virtual methods for calculating intersections with rays, performing euclidean transformations, and computing shading properties. The core of the ray tracing process consists of computing intersections of rays and **geo_objects**. As mentioned above, a new primitive can be added as long as it defines the appropriate virtual methods; polymorphism allows the rest of the ray tracing code to remain unchanged. As another example of object-oriented rendering, the GOll system [12] incorporates several object-based rendering algorithms — ray tracing, Z-buffer, and painter's algorithm — as well as a completely object-based rendering pipeline. (This system will be further discussed below.) Other examples of object-oriented rendering are presented in [13] and [14].

Object-Oriented Structed Display File

An object-oriented structured display file (OOSDF) provides modeling support at a higher level than the object-oriented rendering paradigm. This paradigm is based on mapping the traditional structured display file (that formed the basis of early computer graphics modelers) into a class hierarchy, where classes represent both: nodes that contain geometric information, and edges that contain modeling and viewing transformations. The OOSDF paradigm is characterized by the following properties:

- the entities that comprise the traditional computer graphics SDF are mapped (almost directly) into classes in the OOSDF

- the major motivation is to bring benefits of OO technology to to graphical modeling software

[2] In each section, besides examining some OOGS, I also list other references that readers interested in those paradigms might wish to pursue. Even though work in object-oriented graphical modeling has been going on since the beginning of computer graphics (e.g, in Ivan Sutherland's Sketchpad system [10]), I focus on recent systems and references since these often illustrate the most mature ideas and research in the field, and form a good starting point for obtaining other references.

Paradigm	Issue							
	Modeling Level	Selection Handling	Temporal Behavior	Event Model	Geometric Modeler Relationship	Database Subsystem Relationship	Constraint Handling	Extensibility
OO Rendering	√							√
OO SDF	√							√
OO Animation		√	√	√			√	√
OO UIB	√	√		√			√	√
OO DDE	√	√		√			√	√
OO Presentation	√	√		√	√	√	√	√

Figure 2: Salient Issues for the OOG Paradigms

Early examples of OOSDFs can be found in systems based closely on the GKS and PHIGS standards [15, 16]. Other examples of OOSDFs include Egbert and Kubitz's Grams system [17], and Bahrs *et. al.*'s GOII system [12]. The Grams system attempts to raise the level at which an application performs modeling by separating application modeling, graphical modeling, and rendering into subsystems with formally-defined interfaces. While these goals are similar to those of the Unified Graphics Subsystem described below (under Object-Oriented Presentation), Grams does not explicitly contain any CAD-specific application modeling.

In the GOII system, a 3D graphical scene can be represented by a Scene object that provides the capability to compose graphical objects (3D shapes), lights, and transformations. Views of the scene are obtained by specifying appropriate Camera objects. GOII supports a variety of object-based rendering algorithms, as was discussed earlier. Figure 3 shows the class hierarchy and rendering pipeline (Boxes 1 to 6) of GOII. Objects are employed throughout the rendering and graphical modeling pipeline (annotations below the boxes in the figure list the objects involved at each stage). The process begins (in Box 1) with the creation of SDF objects that are then are composed into hierarchical scenes (Box 2) and rendering begins once viewing information is chosen (Box 3). The specified rendering algorithm is initiated (Box 4), resulting in a 2D hierarchical picture (Box 5) consisting of objects that can "draw themselves" in response to user interactions (Box 6).

The OOSDF paradigm is suitable for applications that are used to create graphics "scenes" without the need for interaction with external geometric modeling systems. The disadvantage of this paradigm is that the modeling constructs supported are too low-level for complex CAD applications.

Object-Oriented Animation

The object-oriented animation paradigm moves beyond 3D graphical modeling and adds a fourth dimension — time — by allowing the specification of temporal behavior in graphical objects. Animations are produced by sending the objects messages to render themselves; the objects modify their appearance (thereby changing the graphical image) as a function of time. Important concerns in any OO animation system (especially those for simulating physical systems) are constraint-satisfaction (since without constraints subparts of the scene can not be modified in a coherent manner) and constraint-representation (since a constraint typically involves more than one graphical object and cannot be part of a single object). Usually constraints are specified as classes that form relationships between graphical objects, along with methods to satisfy the constraint relationship. Thus, the following properties characterize the object-oriented animation paradigm:

- objects incorporate temporal behavior
- constraints are represented as relationships between objects
- efficient constraint-satisfaction has to be incorporated

Any of the other OOG paradigms discussed in this section is amenable to object-oriented animation. For example, an OO rendering system might support simple animations based on color- and light-manipulation, while an OO CAD/CAM presentation system (discussed below) might support sophisticated animations based on the satisfaction of kinematic constraints, or on the evolution of a design created using variational-geometry techniques (Section 1).

Several object-oriented animation systems have been developed by researchers [18, 19, 20, 21, 22, 23, 24, 25]. One recent example is an animation system presented by Zeleznik *et. al.* [26, 27]. This system provides objects that encapsulate behavioral properties — such as gestural controls and spring constants — as well as a variety of constraint solvers (based on inverse kinematics, dynamics and finite-element techniques). The animation is described through message passing between objects. The messages that an object receives describe how the object should change over time; these messages can cause objects to be transformed, deformed, colored, shaded and dynamically moved. As an example, a message might specify the location in 3D of an object as a function of time; as the time changes, rerendering the scene creates a new depiction.

The strength of an OO animation systems lies in the fact that animations can be produced more simply and elegantly than with conventional animation systems, since the temporal behavior is encapsulated in objects, (thereby simplifying the structure of the animation controller). The main disadvantages of this approach are that fairly sophisticated controllers and constraint solvers need to be tightly integrated with the objects in the system, and that predicting the effect of a user action can be difficult since since the temporal behavior is spread over each object.

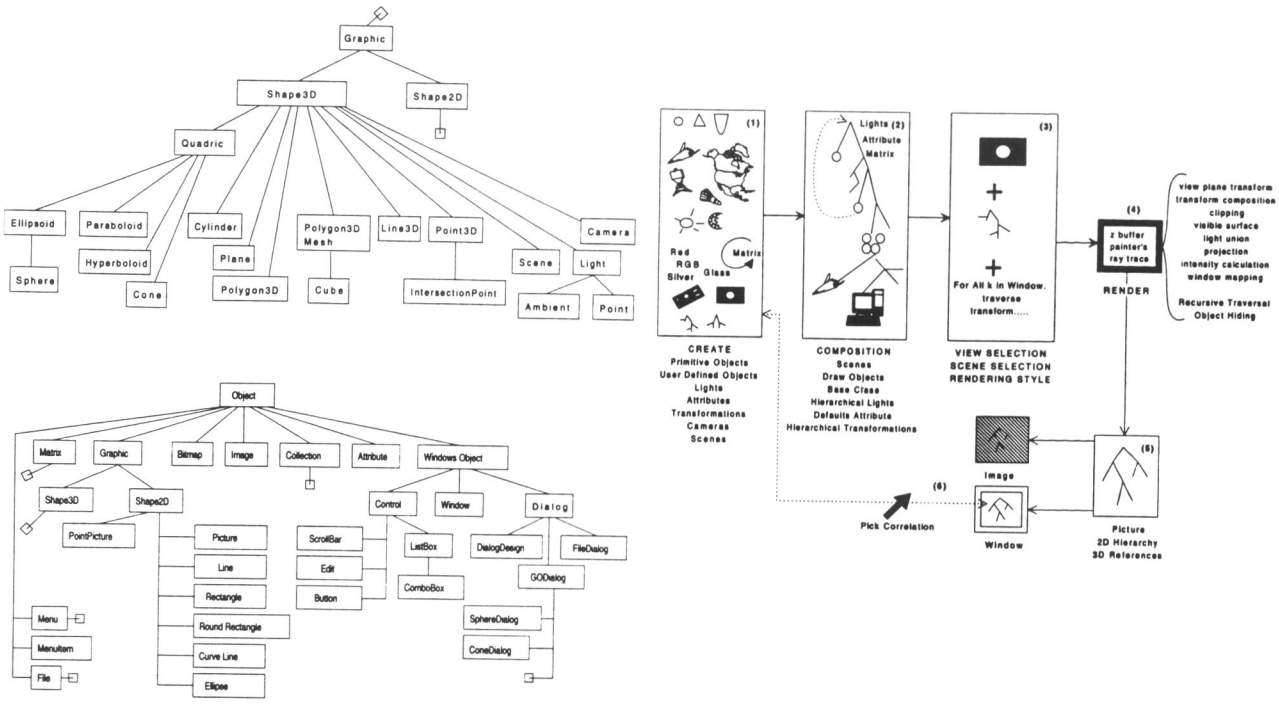

Figure 3: GOII Class Hierarchy and Rendering Pipeline (reproduced with permission from [12])

Object-Oriented User-Interface Builder

An object-oriented user-interface builder (OOUIB) allows a designer to construct a user-interface, which can then be linked into an application. In an OOUIB, the graphical objects are the user-interaction components, for example buttons, sliders and dialog boxes. The process of constructing a user-interface typically involves choosing components from a palette, placing them on application windows, establishing constraints between the layouts (for example, alignments and spacing between components), establishing communication paths between them (for example, passing the result computed by one object to another), and establishing connections ("hooks") between the objects and the application modules (for example, setting callbacks). Moreover, all OOUIBs allow the designer to switch between "design" and "test" modes; in the latter, the objects become "live" and respond to user-interaction events as they would in the completed application. As the above description shows, the graphical objects in an OOUIB need to be fairly autonomous in their behavior and interaction with other objects and the application. In addition, the objects need to be selectable and displayable in a variety of contexts. Thus, the following properties characterize the OOUIB paradigm:

- objects incorporate event-handling, rendering and selection handling
- geometric (layout) constraint handling is an important requirement

Examples of OOUIBs include the NeXTSTEP InterfaceBuilder [28], the Interviews ibuild application [29], and the Andrews toolkit [30].

InterfaceBuilder is a tightly-integrated OOUIB that forms part of the NeXTSTEP development environment (on NeXT workstations and IBM-PC compatibles). InterfaceBuilder relies heavily on layering and delegation to support extensibility. Figure 4 shows an example of using InterfaceBuilder to build the user-interface for a simple temperature converter and square root calculator. The window on the top left (titled "Palettes") displays the available user-interface components; these can be selected and dragged onto the application window (titled "Universal Calculator") and menu (top right, titled "Calculator/Calculations"). As menu items are added to the application menu, they can be bound to methods on specified objects. The bottom left window allows the user to graphically browse through the available objects (which can include Sound and Image objects) and instance them in the application, as well as create new object types (for example, CalculatorInstance). The bottom right window allows object attributes and methods to be modified. Currently, the user has just added the "Calculate" button on the application window, added sound and an image to it, and has just bound it to the calculate method on the CalculatorInstance object (as shown by the dimple next to the calculate method on the bottom right win-

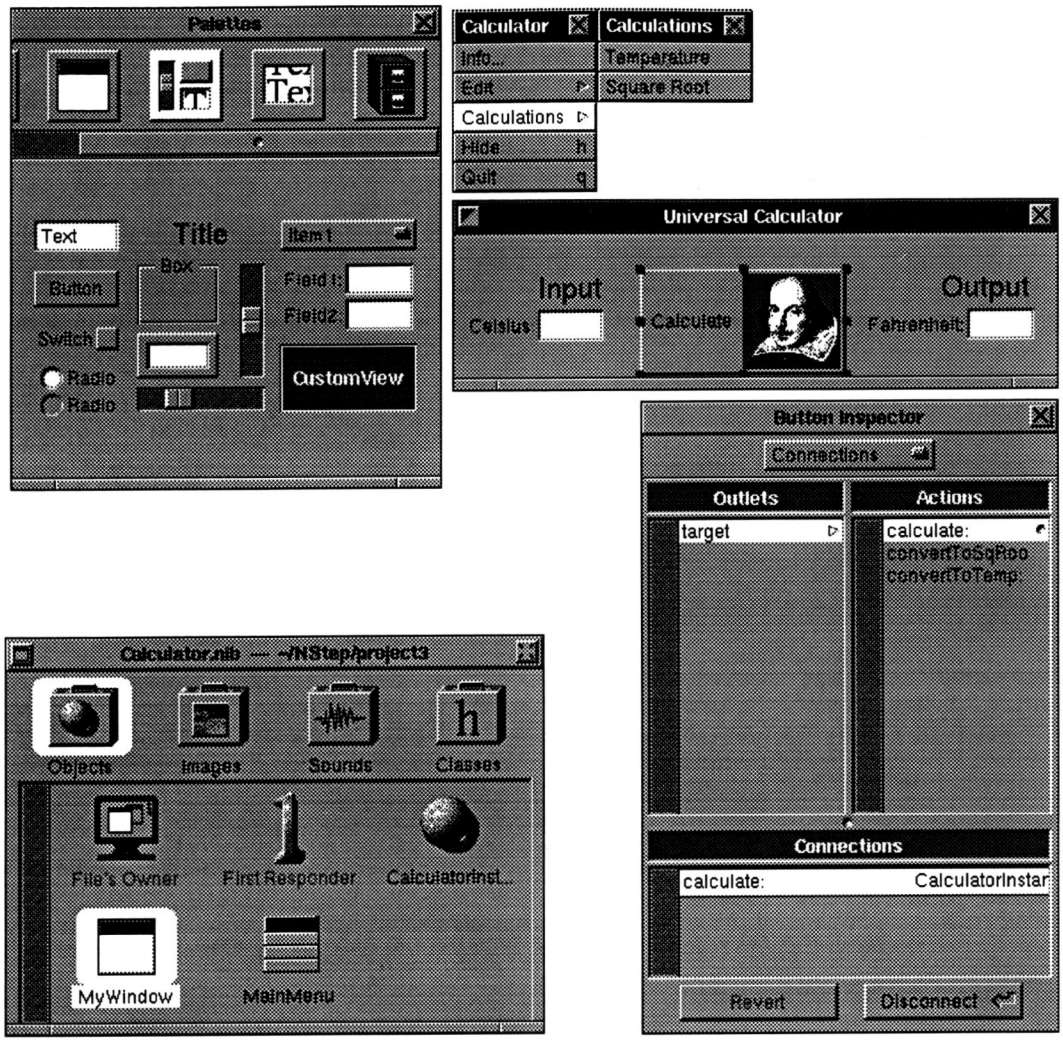

Figure 4: Example of Using the NeXTSTEP InterfaceBuilder

dow). Once the interface is completed, the InterfaceBuilder will save the source files on disk, to which one can add the actual temperature/square-root conversion code.

The ibuild OOUIB is part of the Interviews toolkit. One interesting aspect of ibuild is the support for layout constraints in the style of the Knuth's text formatter TEX[31]. Andrew — a toolkit written in the Class language — provides a set of basic components that can be combined easily to build user interfaces for multi-media applications. Other examples of the OOUIB paradigm can be found in [32, 33, 34, 35].

An OOUIB has the advantages that adding a new object (e.g., a new type of slider) is simplified since most of the behavior of the new object is specified within the object's methods and does not affect other objects. The major disadvantage is that the objects often need to incorporate a large amount of platform-specific and window-system-specific information, since they need to "understand" how to handle user interactions and how to communicate with the windowing system and other objects.

Object-Oriented Drawing and Document Editing

Similar to an OOUIB, object-oriented drawing and document editing (OODDE) systems employ autonomous objects that are capable of displaying themselves in different contexts and of interacting with other objects. However, there is one important difference between the two: while objects in an OOUIB need to include methods to respond to user-interaction events, objects in an OODDE do not have this requirement. In an OODDE system, usually the application is responsible for handling user-interface events and manipulating the graphical objects appropriately. For example, the application might intercept a mouse click, make selection queries to find out which object were selected, and then highlight these objects by changing their color. Thus, the OODDE paradigm is characterized by these properties:

- objects include rendering methods

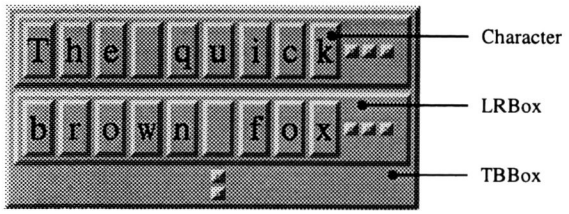

Figure 5: Text Object in Unidraw (reproduced with permission from [38])

- objects need methods to handle select queries

Examples of the OODDE paradigm include Unidraw [36] and IRIS Inventor [37]. Unidraw is a framework for creating 2D graphical, object-oriented editors that can be tailored to specific application domains. An example of an OODDE systems constructed using the Unidraw distribution is the **doc** document editor [38, 39]. **Doc** uses objects ("glyphs") to represent individual characters in a document; thus, every character in a document is capable of responding to select queries and display requests. Figure 5 shows a simple text string containing glyphs. The text object contains a **TBBox**, which tiles its subcomponents top-to-bottom. The **TBBox** contains **LRBox**es, which tile their components left-ro-right. Each **LRBox** contains a **Character** glyph object. Once the text object is created, it is rendered by invoking the **draw** method on the top **TBBox**, which recursively invokes the draw method on its subcomponents. Both **LRBox** and **TBBox** respectively insert horizontal and vertical "glue" (stretchable space) between their subcomponents and use algorithms similar to TEX to arrange and align components (e.g., text justification).

IRIS Inventor is an object-oriented, 3D toolkit that is useful for describing 3D scenes and interactive applications. Inventor's objects include an event model and selection handling, which allows them to be used to build 3D drawing editors. Other examples of the OODDE paradigm can be found in [40, 41]. The advantages and disadvantages of the OODDE paradigm are similar to those shared by the OOUIB paradigm.

Object-Oriented CAD/CAM Presentation

In Section 1, I discussed the requirements for an OOGS suitable for large CAD/CADM applications, and used the term *presentation* to refer to a graphical picture that is intended to convey information about a product or some part of a product. The object-oriented presentation (OOPR) paradigm is intended to support exactly those requirements. The OOPR paradigm moves beyond general graphical modeling to CAD-specific application modeling. Moreover, it avoids duplication of product data between the graphics and geometry subsystems by defining a clear separation and communication protocol between them. The OOPR paradigm is characterized by the following properties:

- there is a clear separation between the OOGS and the geometric modeling systems

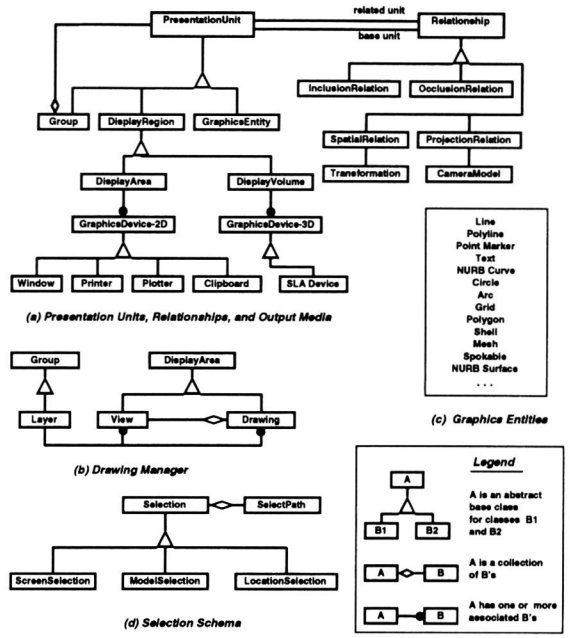

Figure 6: Subset of UGS Schemas (portions reproduced with permission from [42])

- the classes in an OOPR system are tuned to CAD-specific requirements
- display list management and optimization is performed separately from the task of describing a presentation

An example of an OOPR system is the Unified Graphics System (UGS) [42], which is intended to be the next generation graphics system for all CAD/CAM applications developed at my organization. An application describes the picture to be presented to the user in terms of a *presentation hierarchy* — a collection of instances of UGS objects (such as views, drawings, layers, and styles) and relationships (such as modeling transformations and viewing projections) between them. The application manipulates the presentation hierarchy in response to user actions, such as requests for selection, geometric manipulation, or animation. UGS performs several display list and rendering optimizations (transparently to the application), for example spatially organizing entities in an octree for fast rejection during clipping and selection.

The more than 200 classes available in UGS can be organized into five sections or *schemas*:[3]

1. The *presentation organization schema* (Figure 6(a)) defines the classes required to describe the basic structure of a CAD presentation — in terms of *presentation units* and their *relationships*. Presentation units can be graphical

[3]Some of these schemas were originally derived from similar schemas in the PDES/STEP [43] presentation model.

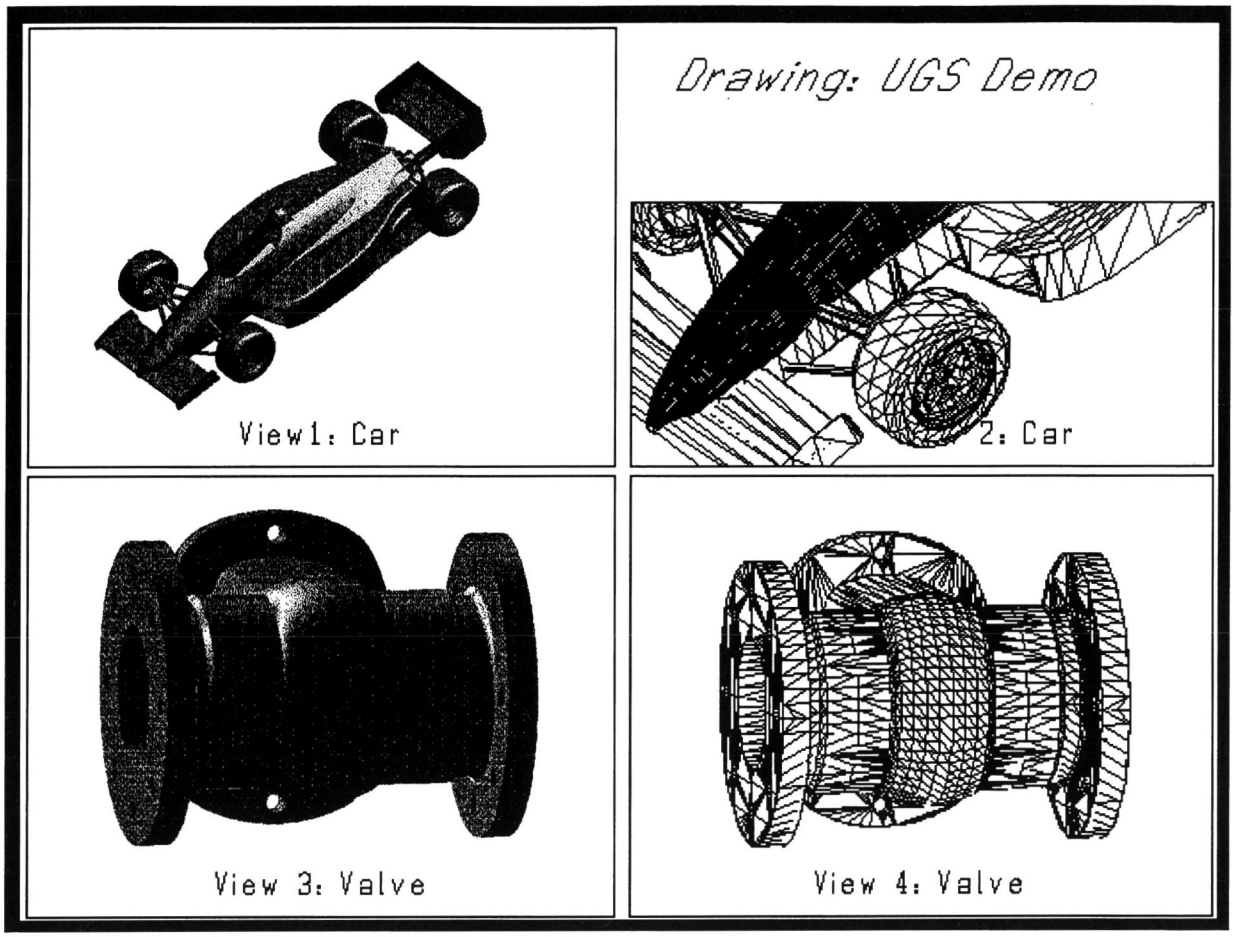

Figure 7: Example of Using UGS

primitives, display regions, or groups of other units. Relationships between units can specify modeling transformations, projections (viewing operations), occlusions (e.g., "in-front-of"), and inclusions (used in assembly modeling). Portions of a presentation hierarchy can be output to various 2D and 3D graphics devices. An example of CAD-specific support in UGS is the Drawing Manager, shown in Figure 6(b). Thus, application developers can think in terms most natural to many CAD/CAM systems, for example, *drawings*, *views* and *layers*.

2. The *presentation entity schema* contains classes used in the visual depiction of geometric entities or annotations. Figure 6(c) lists some of the presentation entities available in UGS. An application constructs a model of a part (entity) using a geometric modeler. The UGS *entity-level interface* [44] extracts the relevant information from the part database to create a visualization of the part using presentation entities. This mechanism allows UGS to be used with a wide variety of geometry subsystems, without having to "know" their specific representation formats and without having to duplicate the data representing the entities. In addition, applications may add annotations (for example, title boxes and labels) to the presentation of a model by directly using UGS presentation entities.

3. The *presentation appearance schema* contains classes required to control entity appearance — for example, line weights and dash patterns, hatch patterns, surface reflectance parameters, text font, etc. UGS objects to control appearance can be associated to any presentation unit in a hierarchy; appearance attributes inherit down the presentation hierarchy. In addition, the application of appearance objects may be restricted to particular contexts — for example, a line may be dashed only in a particular view or layer, or a coordinate triad might be shown in different colors depending on whether the axes are going into the screen or out towards the viewer, or text labels on dimensions may only be shown within a particular range of view angles. I am aware of no other OOGS that provides such extensive context-sensitive styling capabilities that are required for any large CAD/CAM application.

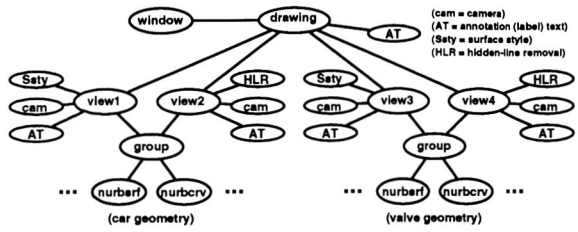

Figure 8: UGS Instance Hierarchy Corresponding to Example in Figure 7

4. The *presentation resource schema* contains general utility classes, including color definitions, light source tables, character and symbol font libraries and the camera model definitions.

5. Finally, the *selection schema* (Figure 6(d)) defines an interface to support three types of user interaction: 1) *Screen selection* (selecting entities based on user-specified 2D locations on a window), 2) *Model selection* (selecting entities based on 3D locations), and, 3) *Location selection* (projecting a user-specified 2D location on a window onto a construction plane). To unambiguously specify the set of selected entities, complete paths are made available to the application (e.g., a line may have been selected in a particular layer or view). The selection mechanism also supports a variety of operations — including filtering, boolean operations and application callbacks — to limit and combine select queries in ways that are needed in most large CAD/CAM applications.

Figure 7 shows an example of a presentation created using UGS; Figure 8 shows the UGS instance hierarchy that was used to describe the presentation. The presentation consists of a drawing that contains four views and some annotation text (the label "Drawing: UGS Demo"). The drawing is associated to (and hence displayed on) a UGS X-window object. (Note that by associating the drawing to a UGS Clipboard object, one could also "paste" its image directly into a document processing application). Each view contains annotation text (the view labels) and some part geometry. *View1* and *view2* depict the model of a racing car model (which contains over 550 NURB curves and 3300 NURB surfaces), while *view3* and *view4* depict the model of a valve (which also contains several hundred NURB curves and surfaces). Each view has associated cameras that allow the application to adjust the view of the part geometry and annotations based on user actions; for example, the user is being shown a zoomed-in view of the car in *view2*. By setting surface shading style attributes on *view1* and *view3*, shaded images of the car and valve, respectively, are obtained in those views; by setting hidden line removal style attributes on *view2* and *view4*, hidden-line-removed images of the car and valve, respectively, are obtained in those views.

4 Summary and Concluding Remarks

This survey of object-oriented paradigms for modeling graphical objects shows that each has its own very different characteristics, strengths and weaknesses. As we have seen, an OOG paradigm can exist between the two extremes:

- objects (in the OOGS) include only the graphical attributes necessary to render them, or,
- objects (in the OOGS) include not only the graphical attributes necessary to render them, but also the complete geometric modeling information, user-interaction handling, display methods, constraint satisfaction and animation methods.

Both of the above are, in general, too extreme to be useful in CAD applications. A realistic OOGS has to be designed by choosing the appropriate combination of properties to incorporate into objects, the best combination being dependent on the specific domain and application. In this paper, I have attempted to examine the issues that need to be understood in order to make such decisions, as well as give examples of the particular choices made by some recent OOGS.

As examples, consider two scenarios:

- suppose one is designing a MacDraw-like drawing (or text) editor, and would like to employ object technology. From the discussion of Section 3, one would conclude that the separation of object behavior across several subsystems might not be worth the extra development and code maintenance; rather, it would be expedient to incorporate graphical, database, and geometric behavior within objects.
- suppose one is designing a large mechanical CAD/CAM system, and would like to employ object technology. From the discussion of Section 3, one would conclude that the geometric, graphical, database, and other behavior of objects in the system need to be split across appropriate subsystems, both to ease the development of these subsystems and to allow these subsystems to be used in a variety of application settings.

In addition, this paper would also provide pointers to the appropriate systems that one might examine for comparison in each case.

In this paper, I have analyzed and compared what I consider to be mature object-oriented graphics paradigms. Researchers are pursuing several other directions in object-oriented graphics: time-critical computing, distributed objects, "live links" between objects, and multi-media. Currently, many of these are the subject of debate in the graphics research community, while others are being standardized through consortiums (e.g. the Object-Management group's CORBA standard, or ISO's PREMO standard for multi-media). I expect that in a few years, these techniques will become mature enough to be subject to an analysis similar to the one presented in this paper.

References

[1] Foley, J., Van Dam, A., Feiner, S., and Hughes, J. 1990. *Computer Graphics Principles and Practice*. Addison-Wesley, Reading, MA.

[2] PHIGS: Programmer's hierarchical interactive graphics system. 1985. ISO TC97/SC21/N819.

[3] GKS: Graphics kernel system. 1985. ISO 7942.

[4] HOOPS Graphics software. 1991. Ithaca Software, Alameda, CA.

[5] Upstill, S. 1990. *The RenderMan Companion*. Addison-Wesley, Reading, MA.

[6] Object-oriented graphics. 1991. SIGGRAPH '91 Panel (Computer Graphics 25 (4)).

[7] Graphics software architecture for the future. 1992. SIGGRAPH '92 Panel (Computer Graphics 26 (2)).

[8] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. 1991. *Object-Oriented Modeling and Design*. Prentice Hall, New Jersey.

[9] Kochhar, S., Marks, J., and Friedell, M. 1991. "Interaction paradigms for human-computer cooperation in graphical-object modeling," in *Proceedings of Graphics Interface '91*, pages 180–191, Calgary, Canada, June 3-7 1991. Morgan Kaufmann Publishers, Palo Alto, CA.

[10] Sutherland, I. E. 1963. Sketchpad: A man-machine graphical communication system. In *Proceedings of the Spring Joint Computer Conference*. Spartan Books, Baltimore, MD.

[11] Melcher, K. and Own, G. 1992. "Object-oriented ray tracing: A comparison of C++ versus C implementations," in Cunningham et. al. (Eds.), *Computer Graphics Using Object-Oriented Programming*. Wiley, New York.

[12] Bahrs, P., Dominick, W., and Moreau, D. 1992. "GOII: An object-oriented framework for computer graphics," in Cunningham et. al. (Eds.), *Computer Graphics Using Object-Oriented Programming*. Wiley, New York.

[13] Youssef, S. 1986. "A new algorithm for object oriented ray tracing," *Computer Vision, Graphics and Image Processing*, **34(2)**:125–137, May.

[14] Schneider, B. 1988. "A processor for an object-oriented rendering system," *Computer Graphics Forum*, **7(4)**:301–310, December.

[15] Wisskirchen, P. 1990. *Object-Oriented Graphics: From GKS and PHIGS to Object-Oriented Systems*. Springer-Verlag, New York.

[16] Czech, M. 1990. "GKS in an object-oriented environment," *Computers and Graphics*, **14(3/4)**:373–375.

[17] Egbert, P. and Kubitz, W. 1992. "Application graphics modeling support through object orientation," *IEEE Computer*, **25(10)**:84–91, October.

[18] Breen, D. E., Getto, P. H., Apodaca, A. A., Schmidt, D. G., and Sarachan, B. D. 1987. "The clockworks: An object-oriented computer animation system," in Marechal, G., editor, *Eurographics '87*, pages 275–282. North-Holland, August.

[19] Chmilar, M. and Wyvill, B. 1989. "A software architecture for integrated modelling and animation," in Earnshaw, R. and Wyvill, B., editors, *New Advances in Computer Graphics. Proceedings of CG International '89*, pages 257–276. Springer-Verlag.

[20] Fiume, E., Tsichritzis, D., and Dami, L. 1987. "A temporal scripting language for object-oriented animation," in Marechal, G., editor, *Eurographics '87*, pages 283–294. North-Holland, August.

[21] Kuehn, V. and Muller, W. 1991. "Advanced object-oriented methods and concepts for simulation of multi-body systems," in *Eurographics Workshop on Animation and Simulation*.

[22] Lorensen, W. and Yamron, B. 1989. "Object-oriented computer animation," in *Proceedings of the IEEE 1989 National Aerospace and Electronics Conference NAECON 1989*, pages 588–595.

[23] Maiocchi, R. and Pernici, B. 1990. "Directing an animated scene with autonomous actors," in Magnenat-Thalmann, N. and Thalmann, D., editors, *Computer Animation '90 (Second workshop on Computer Animation)*, pages 41–60, April. Springer-Verlag.

[24] Mahieddine, M. and Lafon, J. C. 1990. "An object-oriented approach for modelling animated entities," in Magnenat-Thalmann, N. and Thalmann, D., editors, *Computer Animation '90 (Second workshop on Computer Animation)*, pages 177–187, April. Springer-Verlag.

[25] Baker, M. 1992. "An object-oriented approach to animation control," in Cunningham et. al. (Eds.), *Computer Graphics Using Object-Oriented Programming*. Wiley, New York.

[26] Zeleznik, R. C., Conner, D. B., Wloka, M. M., Aliaga, D. G., Huang, N. T., Hubbard, P. M., Knep, B., Kaufman, H., Hughes, J. F., and van Dam, A. 1991. "An object-oriented framework for the integration of interactive animation techniques," *Computer Graphics (Proceedings of ACM SIGGRAPH '91)*, **25(4)**:105–112, July.

[27] Zeleznik, R. C., Herndon, K. P., Robbins, D. C., Huang, N., Meyer, T., Parker, N., and Hughes, J. F. 1993. "An interactive 3D toolkit for constructing 3D widgets," *Computer Graphics (Proceedings of ACM SIGGRAPH '93)*, **27(4)**:81–84, August.

[28] Garfinkel, S. L. and Mahoney, M. K. 1993. *NeXTSTEP Programming*. Springer-Verlag, New York, New York.

[29] Linton, M., Vlissides, J., and Calder, P. 1989. "Composing user interfaces with Interviews," *IEEE Computer*, **22(2)**, February.

[30] Howard, J. 1988. "An overview of the Andrew File System," *Proceedings of the 1988 USENIX Winter Conference*, Dallas, Texas, February 9-12.

[31] Knuth, D. E. 1984. *The TEXbook*. Addison-Wesley, Reading, Massachusetts.

[32] Barth, P. S. 1986. "An object-oriented approach to graphical interfaces," *ACM Transactions on Graphics*, **5(2)**:142–172, April.

[33] Coutaz, J. 1987. "The construction of user interfaces and the object paradigm," in *Proceedings of ECOOP '87*, pages 135–144, June.

[34] Szekely, A. and Myers, B. 1988. "A user interface toolkit based on graphical objects and constraints," in *Proceedings of ACM OOPSLA '88*, pages 36–45.

[35] Hubner, W. and de Lancastre, M. 1989. "Towards an object-oriented interaction model for graphics user interfaces," *Computer Graphics Forum*, **8(3)**:207–217, September.

[36] Vlissides, J. and Linton, M. 1989. "Unidraw: A framework for building domain-specific graphical editors," *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, October.

[37] Strauss, P. S. and Carey, R. 1992. "An object-oriented 3D graphics toolkit," *Computer Graphics (Proceedings of ACM SIGGRAPH '92)*, **26(2)**:341–349, Chicago, July 26-31.

[38] Calder, P. and Linton, M. 1990. "Glyphs: Flyweight objects for user interfaces," *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, October.

[39] Calder, P. and Linton, M. 1992. "The object-oriented implementation of a document editor," *Proceedings of OOPSLA '92 (ACM SIGPLAN Notices 27 (10))*, pages 154–165, October, Vancouver, Canada.

[40] Fellner, W. and Kappe, F. 1990. "EDEN — An editor environment for object-oriented graphics editing," in Vandoni, C. and Duce, D., editors, *Eurographics '90*, pages 425–437, September. North-Holland.

[41] Roseman, D. 1992. "Design of a mathematicians' drawing program," in Cunningham et. al. (Eds.), *Computer Graphics Using Object-Oriented Programming*. Wiley, New York.

[42] Kochhar, S. and Hall, J. 1992. "An object-oriented CAD/CAM presentation system," *Proceedings of the Third Eurographics Workshop on Object-Oriented Graphics*, October 28-30, Champery, Switzerland.

[43] PDES: Product Definition Exchange using STEP. 1990. ISO CD 10303-42. (STEP = Standard for Exchange of Product Model Data).

[44] Koifman, M., Hall, J., and Lindgren, T. 1992. "A framework for object oriented CAD/CAM graphics: An application view," *Proceedings of the Third Eurographics Workshop on Object-Oriented Graphics*, October 28-30, Champery, Switzerland.