Supporting Graphics Using Delegation and Multi-Methods

D. Brookshire Conner

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-93-33
September 1993

Supporting Graphics Using Delegation and Multi-Methods

D. Brookshire Conner
Department of Computer Science
Brown University
Providence, RI 02912
dbc@cs.brown.edu

February 20, 1993

Abstract

We present an object-oriented design for a general-purpose interactive graphics system. By analysing features provided by many graphics systems, we identify object-oriented mechanisms best suited to implementing them, thus identifying mechanisms best suited to building graphics systems and applications. These mechanisms include dynamic, per object inheritance and multiple dispatch of methods. We claim the design is more general, simpler and more flexible than existing graphics systems since it uses fewer, simpler mechanisms to provide greater functionality.

1 Why Isn't Graphics Object-Oriented?

One of the first examples used in teaching object-oriented programming is a MacDraw-like drawing program. Each object on the screen, whether a rectangle, oval, or line, is derived from a base `graphicsObject` class supporting a virtual `draw` method (an example is shown in Figure 1. Thus, all objects support their own specific `draw` method, and new kinds of drawing primitives can be added easily. Right from the start, then, the field of graphics appears to be especially suited to object-oriented programming. In this paper, however, we show that this first impression is misleading, by examining real-world graphics systems and seeing how their needs are poorly supported by common object models. Instead, we propose a model designed specifically to address the needs of graphics.

Object orientation has much to offer graphics. It is widely acknowledged that graphics programming, especially interactive graphics programming, requires a lot of code [10]. Graphics problems are often complex, involving many special cases. Code to implement the interface of modern graphical applications is often half or more of the application's code. In addition, graphical applications are more common now than ever before. Since object-oriented programming is intended to address software engineering concerns, object-oriented programming may well help address the complexity of graphics programs.

Further, Graphics systems are often non-portable. Systems do exist on many different platforms typically sacrifice performance for speed. For example, the X Window System is available on many architectures, but cannot handle hardware for accelerating drawing operations. Different vendors provide various extensions to X to make it aware of particular hardware, but these extensions differ from vendor to vendor. A truly object-oriented system should be able to hide these differences behind an abstract interface.

This paper investigates better ways to apply object-oriented technology to the design of graphics systems. In examining existing graphics systems, we find many common mechanisms for information and code sharing. Unfortunately, these mechanisms are not directly supported by standard class-instance languages. Instead, graphics systems must implement these mechanisms from scratch, each time. Designing a new general-purpose mechanism is a difficult task, akin to programming language design, but satisfactory mechanisms do exist in current programming languages and can be used to provide support for a flexible, extensible, portable graphics system, as we shall see in the remainder of this paper.

Here, we present the design of a generic graphics system, not its implementation. Research indicates that the mechanisms we use can be implemented efficiently [1], but whether efficiently enough to satisfy the performance needs of the graphics community is a topic for future research.

Graphics applications, especially interactive applications, make use of editable objects that share information in editable ways. For example, in a

MacDraw-like editor, one can group objects together and dynamically add and remove objects from a group, changing colors of either individual objects or the entire group at once. Objects in groups thus share color information with the group in a dynamic, changing manner. However, inheritance (the fundamental sharing technique) in a class-instance system is an inherently static and unchanging relationship (since an object's inheritance pattern can typically not be changed after the object is created). For this reason, previous work indicated that a delegation-based object model, with its dynamic, changing patterns of sharing, was better suited to graphics than a class-based object model [4].

This paper, in addition to reaffirming that point, describes in more detail how delegation can implement various sharing mechanisms used in graphics systems. The method lookup required by graphics is also shown to be inherently multiply polymorphic. These observations lead us to a design for the object system of a generic graphics system.

2 The Generic Graphics System

Before describing our design, we summarize the common features of graphics systems this design is intended to support. Graphics systems seem to differ widely, especially in fine details, but we shall see that these differences are misleading: graphics systems provide much common functionality, even if the interfaces to that functionality are entirely different. At first glance, this functionality seems to map well into standard object-oriented concepts of class, data members, and methods. However, graphics systems use the corresponding graphics-specific concepts in ways that are not handled by the corresponding object-oriented mechanism. We begin with a brief overview of kinds of graphics systems, then present the common mechanisms these kinds of systems use, finally seeing some examples of real systems and their use of these mechanisms.

2.1 Kinds of graphics systems

Graphics systems are tailored to a variety of applications. Most modern applications have graphical user interfaces, typically two-dimensional. Thus, there are a variety of graphics libraries supporting 2D drawing and interaction. These include the X Window System and toolkits written for it, such as Motif [13] and Interviews [19], the PostScript programming language, Garnet [11], and the Macintosh Toolbox. Some graphics libraries are tailored to more specific tasks, such as image composition and three-dimensional rendering, as provided by PHIGS [7], GL [14], and Doré [9]. These packages are currently targeted at relatively limited application domains, such as CAD/CAM. However, much as 2D graphical interfaces became widespread when 2D screen technology became sufficiently advanced, we believe that 3D graphical interfaces will become widespread as 3D rendering hardware becomes a commodity feature. Anticipating this development, recent research has presented object-oriented systems for three-dimensional interaction, including Brown University's UGA system [20], SGI's IRIS Inventor [16], and Sun's TBAG [5].

2.2 What do graphics systems provide?

All graphics systems provide three basic entities:

- primitives
- attributes
- actions to apply to the primitives

Primitives are the obvious objects in a graphics system. They include polygons, lines, points, and other geometric shapes. As *visible* objects, they make obvious *programming* objects. Attributes in graphics systems are used to describe the appearance of the primitives, including such features as color, position, and shape. Finally, most graphics systems provide a number of operations that can be performed on sets of graphics primitives. Some of the common operations include drawing a set of primitives and their attributes, identifying the object selected (picked) by an interactive user, and determining what objects overlap or intersect.

These three basic entities seem to provide mechanisms for classes in the form of primitives, data members in the form of attributes, and methods in the form of operations. Once again, graphics appears to be a natural domain in which to apply object-oriented programming. However, a closer examination of primitives, attributes, and operations, and of how they work together shows us that this is not the case. To see this, consider drawing an object. Exactly what code to call depends on the real type of at least *two* parameters — what kind of object it is (i.e., polygon, line, text) and where it is being drawn (i.e., on a high-resolution printer, on an X window, on a hardware-accelerated window). This is not handled by standard virtual functions, which determine what method to call (dispatch) on the basis the real type of *only one* parameter, the receiver. Thus, we survey some well-known graphics systems to determine how the mechanisms supporting primitives, attributes and operations differ from common class-instance models.

2.3 Some examples of graphics systems

In examining mechanisms supporting primitives, attributes, and operations, we focus on systems that are first and foremost well-known and in extensive use. These include the X Toolkit, Garnet, and PHIGS. Most of these are not object-oriented, but they provide much the same level of functionality, minus extensibility, as their object-oriented cousins. The other systems we consider are not as well-known. Instead, these systems are examples of the latest research in graphics systems. These systems are SGI's Inventor, Sun's TBAG, and Brown University's UGA. They all make use of object-oriented technology and they all support interactive 3D graphics, which, as we mentioned before, we expect to become more widely available in the near future.

2.3.1 X Toolkit

The X Toolkit library provides object orientation in C for interactive 2D graphics. Additional toolkits built on top of the X Toolkit, such as Motif, provide classes for actual user interface objects such as menus, scrollbars, and pushbuttons. User interface objects, called *widgets*, are hierarchically organized, providing such facilities as multiple windows with controls clustered according to intended use. While most attributes in the X Toolkit receive default values, it is also possible (and encouraged) to specify attributes according to the widgets' hierarchical composition. For example, all buttons on a single control panel can be colored identically by coloring the panel itself. Notice that this is sharing of attributes along a *run-time* structure, the hierarchical composition of the widgets, not along the *compile-time* structure of the widget classes of the interface toolkit.

While it is possible (although clumsy) to add new widget classes to an X Toolkit widget set like Motif, it is impossible to extend the widget set in another important dimension: the kinds of operations that can be performed. For example, you can't extend the Motif toolkit with PostScript drawing methods for each widget (e.g., to get pretty illustrations of your interface) without rewriting Motif from the ground up. Alternatively, when custom hardware is available, it would be desirable to modify Motif so that drawing routines took advantage of it. Again, this is impossible without a total rewrite. Adding new operations to be performed on graphical objects is so desirable that some of the other systems discussed below expend great effort in providing this extensibility.

2.3.2 Garnet

Garnet is a 2D user interface system written in Lisp, intended to help with all phases of user interface design and construction. It uses a delegation-based object system with one-way constraints between attributes of objects [12]. Delegation was used in Garnet because standard class-instance models (such as CLOS) were found to be insufficiently flexible. However, delegation is used primarily to support primitives as classes or exemplars. It could also be used to provide the hierarchical composition of interface objects (like that found in the X Toolkit); however, hierarchical composition is instead provided by an additional mechanism, *aggregates*.

One-way constraints are another mechanism provided by Garnet. However, one-way constraints can be handled trivially in a delegation system like the Self programming language [17, 18] by simply resending a message.

Garnet places a great deal of emphasis on declarative programming, which solves a recurrent problem in graphics programming: the order of operations. Most graphics systems use the order of attributes to determine which attribute to use for a primitive. This produces code that is hard to debug and predict, especially in the presence of complex flow of control. Garnet's constraints make possible a declarative approach to specifying object attributes. However, we have already seen that Garnet's constraints could be adequately handled by the kinds of mechanisms already in the system. Thus, rather than using one simple mechanism (delegation) to provide the variety of functionality necessary, Garnet provides a complex (and thus harder to learn and understand) set of features. Our design for an object system will attempt to remedy this.

2.3.3 PHIGS

The PHIGS system is a standard system for high-performance 3D drawing. It is not object-oriented. We discuss it because it is a well-known system that provides what is often considered a minimum level of functionality for 3D graphics. PHIGS groups its attributes and primitives into hierarchies of *structures*. Order of attributes within a structure (as well as all previous attributes encountered in the hierarchy) determines which attributes apply to a primitive. PHIGS offers a strictly closed set of primitives, attributes, and operations. Some improvements on PHIGS, notably Doré [9], provide limited forms of extensibility, allowing additional primitives, attributes and operations. However, a new mechanism must be provided to make possible each set of extensibility.

2.3.4 Inventor

This is one of the most recent object-oriented graphics libraries and is the first one widely available for 3D graphics implemented in C++. Like PHIGS, it supports two ways to share attributes between primitives: the order attributes are specified in, and the hierarchical composition of attributes and primitives. Operations are supported as separate classes (rather than actual C++ member functions) to let programmers provide new operations. In order to determine what piece of code to call when an operation is applied to a particular primitive, runtime type information indexes into a 2D array of function pointers, in which rows represent primitive types, and columns represent operation types. Because of the way in which it is implemented, inheritance of operation implementations works along rows, but not along columns. Thus, extending the set of operations is supported, but only in a clumsy and limited way. As in Garnet, we see a variety of complex mechanisms being implemented, where simpler, more powerful mechanisms would have sufficed to achieve the same functionality.

2.3.5 TBAG

TBAG, a graphics system from Sun Microsystems, provides stateless data types representing primitives and attributes. By contrast, primitives and attributes in systems like PHIGS, Doré and Inventor make use of stateful data types. Values of these types are produced by function objects related to one another by multi-way constraints. The function objects can be evaluated with different parameters to provide time-varying or user-controlled geometry. This is a rather different approach to graphics programming, deriving more from functional programming than object-oriented or procedural programming. Despite this difference, though, it still has some of the drawbacks of other graphics systems. For example, like Inventor, TBAG must index into a 2D array of function pointers to determine what code to call when an operation is applied to a stateless graphics value.

2.3.6 UGA

UGA is an interactive graphics system from Brown University that uses a delegation system to provide attribute sharing between objects. Its object system is singly polymorphic. Recent work describes an interactive way to build richly interactive three-dimensional objects [21] by snapping and constraining simpler objects to one another. However, determining the behavior of two objects snapped to each other depends on the types of both objects. For example, snapping a point to a plane asserts different constraints from snapping a point to a line. Ideally, both operations should be performed by an abstract `snap` method on two generic objects. Clearly, this cannot be performed by a singly polymorphic object system.

2.3.7 Future systems

As more and more applications provide more dynamic, interactive, graphical interfaces, the demands on future graphics systems will increase [8] [3]. They will have to support objects that in one interactive session may change appearance and behavior, and change internal representation to perform at optimal speeds. Complex graphics systems that provide several mechanisms where one simple mechanism would suffice will unnecessarily complicate the implementation of such applications. Graphics systems today are already too complex to learn easily. To make these highly interactive applications possible, we must make graphics systems at once simpler and more flexible.

3 The Design

In Section 2.3 and previous work [4], we saw that many graphics systems attempt to provide attributes that can be shared among primitives in a changeable, object-by-object basis.[1] We further saw that graphics systems try to provide extensible sets of operations on primitives, and that these operations often act on several primitives at once.

These characteristics show us why graphics is not as readily implemented in an object-oriented fashion as one might expect. Most class-instance models, which comprise the vast majority of object-oriented languages, support none of these characteristics. Despite the fact that graphics systems appear to provide typical object-oriented facilities, the classes (primitives), data members (attributes), and methods (operations) of graphics do not function in the same way as in standard class-instance models.

Sharing between classes and their data members (primitives and attributes) is static in class-instance systems and dynamic in graphics systems. Further, sharing between classes and data (primitives and attributes) in class-instance systems is restricted to occur across large groups (i.e., per class), and not on an object-by-object basis. As we have also seen, graphics systems have different requirements on how to determine the code to call in response to a message (i.e., what implementation of an operation to use with a particular primitive).

Thus, we believe that an object system to support object-oriented graphics must have three primary traits:

- Sharing between objects that is dynamic, to support the changing relationships of attributes and primitives in interactive graphics
- Sharing between objects that is per object, since sharing between attributes and primitives is not fully limited by primitive type
- Sharing between sets of objects, to support the multiple dispatch of operations performed on primitives

We now consider the nature of these traits in graphics systems, and then go through the potential advantages of these traits gained for graphics systems, both separately and in combination.

[1] In terms of the "Treaty of Orlando", this is *dynamic, implicit, per-object* sharing [15].

3.1 The basic object model

We begin by examining the nature of the object model without inheritance. In a traditional object-oriented system, a method (that is, a specific response to a message) is associated with one and only one object, whether that object is a class, as in class-instance systems, or a prototype, as in delegation systems like Self [17, 18]. As we have seen, this is insufficient to support the operations applied to primitives in graphics systems, which choose code based on several objects, not just one.

Thus, in our design, methods are associated with not one object but with a set of objects. Every method is associated with a list of objects that corresponds to the message's parameters. Conceptually, messages are sent, not to one object but to a set of them. In response to a message, the method associated with the set the message was sent to is evaluated. This enables objects to cooperate in determining what method to use. For example, a `draw` message could be sent to both a graphical primitive and a drawing area. Different methods could be used depending on either the kind of primitive (e.g., polygon or line) or the kind of drawing area (e.g., printer or window). Evaluating a method could perform some code or simply return a constant value, allowing groups of objects to share either data (attributes) or code (operations) in a uniform manner.

This model provides the first two traits we require for graphics. Methods and data are shared in a per-object way, since methods are associated with objects (not with classes) and methods associated with only a single object are equivalent to standard methods in per-object systems under this design.

This form of multiple-dispatch is derived from that presented for the Cecil programming language, which supports multi-methods [2]. However, Cecil is a complete programming language, whereas we propose an object model that could be implemented in numerous programming languages. Cecil thus addresses a broader spectrum of applications, including an incremental typing facility that seems to be unnecessary for graphics (and is not included in this design). We will point out other differences from Cecil as they arise.

3.2 Inheritance

The multiple dispatch mechanism described above in Section 3.1 supports two of the desired traits of our graphics system: per-object sharing, and sharing between sets of objects. We now add an inheritance mechanism to our model that supports more extensive sharing. For instance, the basic object model present above requires that a method be specified for *all* sets of objects that a message might be sent to. In order to support graphics, however, this sharing must be dynamic.

3.2.1 Parent slots

We identify particular methods as *parent methods*. To be a parent method, a method must be called for single objects *only*, reference a single object *only*, and finally be specially annotated that it is a parent method. However, any one object can have any number of parent methods. Further, parent methods are assigned priorities, so that different parent methods can have a greater or lesser importance to the object. Clearly, this is a form of per-object inheritance, since parent methods are associated with individual objects.

The objects that these parent methods reference shall be assignable, making possible the kind of dynamic inheritance required by graphics. Dynamic inheritance is a questionable feature to some. In fact, Cecil does not appear to provide dynamic inheritance. However, in interactive graphical applications, graphics objects change sharing relationships frequently, and using a static form of inheritance would make it necessary to destroy and recreate an object whenever it wished to share information with a new object.

3.2.2 Method lookup

To perform method lookup in the presence of inheritance, a semantics equivalent to Cecil's is sufficient. We describe the mechanics using a recursive procedure.

1. Take the set of objects.
2. Build every set consisting of one parent of each of the objects in the passed set. If a set contains an object that has been considered before, do not consider it again.
3. If there are no such sets, lookup fails.
4. Otherwise, find methods with the same name associated with the same set.

5. If exactly one method is found, evaluate it.
6. If more than one method is found, the message is ambiguous.
7. If no methods are found, recurse on every set produced in 2.

3.3 Resulting improvements

Many features implemented by several ad hoc mechanisms in other graphics systems are supported by the design described above. Here, we give examples of how equivalent or improved functionality can be provided in a simple way by our design.

3.3.1 A single uniform sharing mechanism

In our design, only one mechanism supports attribute sharing among primitives: inheritance. PHIGS, Inventor, Garnet and others all have two or more, typically using hierarchy, order, and occasionally some form of constraints. With only a single sharing mechanism, graphics code is easier to understand and debug.

The uniformity of this model has the further advantage of making graphics code more declarative. Rather than stating a recipe (i.e., an order in which to apply attributes), this model allows statements of relationships (inheritance). Thus, the graphics programmer tends to state what is intended, rather than how to do it (cf. `openStructure` and `closeStructure` from PHIGS).

The equivalents of particular sharing mechanisms are easily written with this model. For example, one-way constraints are simple: if, for example, the color of object A is constrained to be the same as the color of object B, A can simply send color messages to B. Since no new mechanisms are needed, programmers get additional power simply by rearranging what they already know. Individual multi-way constraints, such as three views of a color using three different representations that are all consistent, are also simple: the shared value is simply a multi-method. If a programmer wishes to access such a shared value as if it were an unshared value, a suitable method can be written that simply resends the unary message as a message to all objects sharing the value. Interrelated multi-way constraints can be handled similarly.

3.3.2 Cyclic dependencies

Typically, only specialized graphics systems allow cycles — e.g., in systems intended to produce fractal images, cycles show up as recursion. Usually graphics systems simply do not handle cycles, a limitation the programmer is expected to work around. In fact, only a very few systems even detect the cycles and disallow them. Most simply fail in some inelegant way upon encountering a cycle (e.g., by entering an infinte loop).

Why are cycles important? Consider this example: A's color depends on B's, but B's position depends on A's. Such a relationship is usually expressed with either constraints or reference nodes (i.e., a new linguistic mechanism). The kind of method lookup we have described handles cycles without problems: either a search of the inheritance graph finds a method to call, or it doesn't (an error). When it doesn't find a method, the search always terminates.

3.3.3 Full extensibility

This model makes extensibility possible for primitives, attributes and operations. Double-dispatch systems can sometimes make it easy to extend the primitive set, but extending the operation set becomes hard. Some systems have the contrary problem: easily extensible operation sets, but limited primitive sets. Those systems that do provide extensibility for both primitives and operations are typically limited in some way (e.g., Inventor's limitation on inheriting methods for operations) and clumsy to use. This will enable users of this design to add customizations supporting specialized hardware *and* specialized, application-specific primitive types in an easy way.

4 Conclusions

Present object systems do not provide the kinds of information sharing and method lookup necessary for graphics programming. We have presented a design for an object system using known technologies that *does* sufficiently support what today's state-of-the-art graphics systems are trying to do.

This solution is not a magic bullet: someone still has to write the particular implementations of primitives and operations on them, although this task is in the domain of well-understood problems [6]. More generic problems, such as information sharing and method lookup, should not be implemented by application-specific components (e.g., graphics systems).

5 Future work

Preliminary work is under way on a graphics library using this object model that should be usable on a variety of hardware platforms in a variety of programming environments. Extensive testing will be necessary once the implementation is completed to determine if the implementation lives up to the goals of the design.

Acknowledgements

This work was supported in part by NSF/DARPA, IBM, Sun Microsystems, NCR, Hewlett Packard and Digital Equipment Corporation. The author would like to thank Andries van Dam, Peter Wegner, and Pascal Van Hentenryck for enlightening discussions.

References

[1] Craig Chambers. *The Design and Implementation of the* SELF *Compiler, an Optimizing Compiler for Object-Oriented Progamming Languages*. PhD thesis, Stanford University, 1992.

[2] Craig Chambers. Object-oriented multi-methods in Cecil. In *Proceedings of ECOOP*, 1992.

[3] D. Brookshire Conner, Scott S. Snibbe, Kenneth P. Herndon, Daniel C. Robbins, Robert C. Zeleznik, and Andries van Dam. Three-dimensional widgets. In Marc Levoy and Edwin E. Catmull, editors, *Proceedings of the 1992 Symposium on Interactive Three-Dimensional Graphics*, pages 183–188. ACM SIGGRAPH, March 1992.

[4] D. Brookshire Conner and Andries van Dam. Sharing between graphical objects using delegation. In *Proceedings of the 1992 Workshop on Object-Oriented Graphics*, 1992. To be published.

[5] Conal Elliott, Greg Schechter, Salim Abi-Ezzi, and Michael Deering. *TBAG: Time, Behavior, and Geometry*. Unpublished, 1991. Sun Microsystems internal document.

[6] James Foley, Andries van Dam, Steven Feiner, and John Forbes Hughes. *Computer Graphics: Principles and Practice*. Addison Wesley, second edition, 1990.

[7] Tom Gaskins. *PHIGS Programming Manual*. O'Reilly & Associates, Inc., 1992.

[8] Mark Green and Robert Jacob. SIGGRAPH '90 workshop report: Software architectures and metaphors for non-WIMP user interfaces. *Computer Graphics*, 25(3):229–235, July 1991.

[9] Kubota Pacific, Inc. *Doré Programmer's Manual*, 1992.

[10] Brad A. Myers. User-interface tools: Introduction and survey. *IEEE Software*, pages 15–23, January 1989.

[11] Brad A. Myers, Dario A. Guise, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Edward Pervin, Andrew Mickish, and Philippe Marchal. Garnet: Comprehensive support for graphical, highly interactive user interfaces. *IEEE Computer*, pages 71–85, November 1990.

[12] Brad A. Myers, Dario A. Guise, and Brad Vander Zanden. Declarative programming in a prototype-instance system: Object-oriented programming without writing methods. In *Proceedings of OOPSLA*, pages 184–200, 1992.

[13] Open Software Foundation. *OSF/Motif Reference Guide*.

[14] Silicon Graphics, Inc. *GL Programmer's Manual*.

[15] Lynn Andrea Stein, Henry Lieberman, and David Ungar. A shared view of sharing: The treaty of Orlando. In Won Kim and Frederick H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, ACM Press Frontier Series. ACM Press, 1989.

[16] Paul S. Strauss and Rikk Carey. An object-oriented 3D graphics toolkit. In Edwin E. Catmull, editor, *SIGGRAPH '92 Conference Proceedings*, pages 341–349. ACM SIGGRAPH, Addison-Wesley, July 1992.

[17] David Ungar and Randall B. Smith. SELF: The power of simplicity. In *OOPSLA '87 Conference Proceedings*, pages 227–241, 1987. Published as *SIGPLAN Notices*, 22, 12 (1987).

[18] David Ungar and Randall B. Smith. SELF: The power of simplicity. *Lisp and Symbolic Computation*, 4(3), 1991.

[19] John M. Vlissides and Mark A. Linton. Unidraw: a framework for building domain-specific graphical editors. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 158–167, 1989.

[20] Robert C. Zeleznik, D. Brookshire Conner, Matthias W. Wloka, Daniel G. Aliaga, Nate Huang, Phillip M. Hubbard, Brian Knep, Henry Kaufman, John F. Hughes, and Andries van Dam. An object-oriented framework for the integration of interactive animation techniques. In Thomas W. Sederberg, editor, *SIGGRAPH '91 Conference Proceedings*, pages 105–112. ACM SIGGRAPH, Addison-Wesley, July 1991.

[21] Robert C. Zeleznik, Kenneth P. Herndon, Daniel C. Robbins, Nate Huang, Tom Meyer, Noah Parker, and John F. Hughes. A toolkit for interactive construction of 3D interfaces. In *SIGGRAPH '93 Conference Proceedings*. ACM SIGGRAPH, Addison-Wesley, July 1993. Video paper. Submitted for publication.

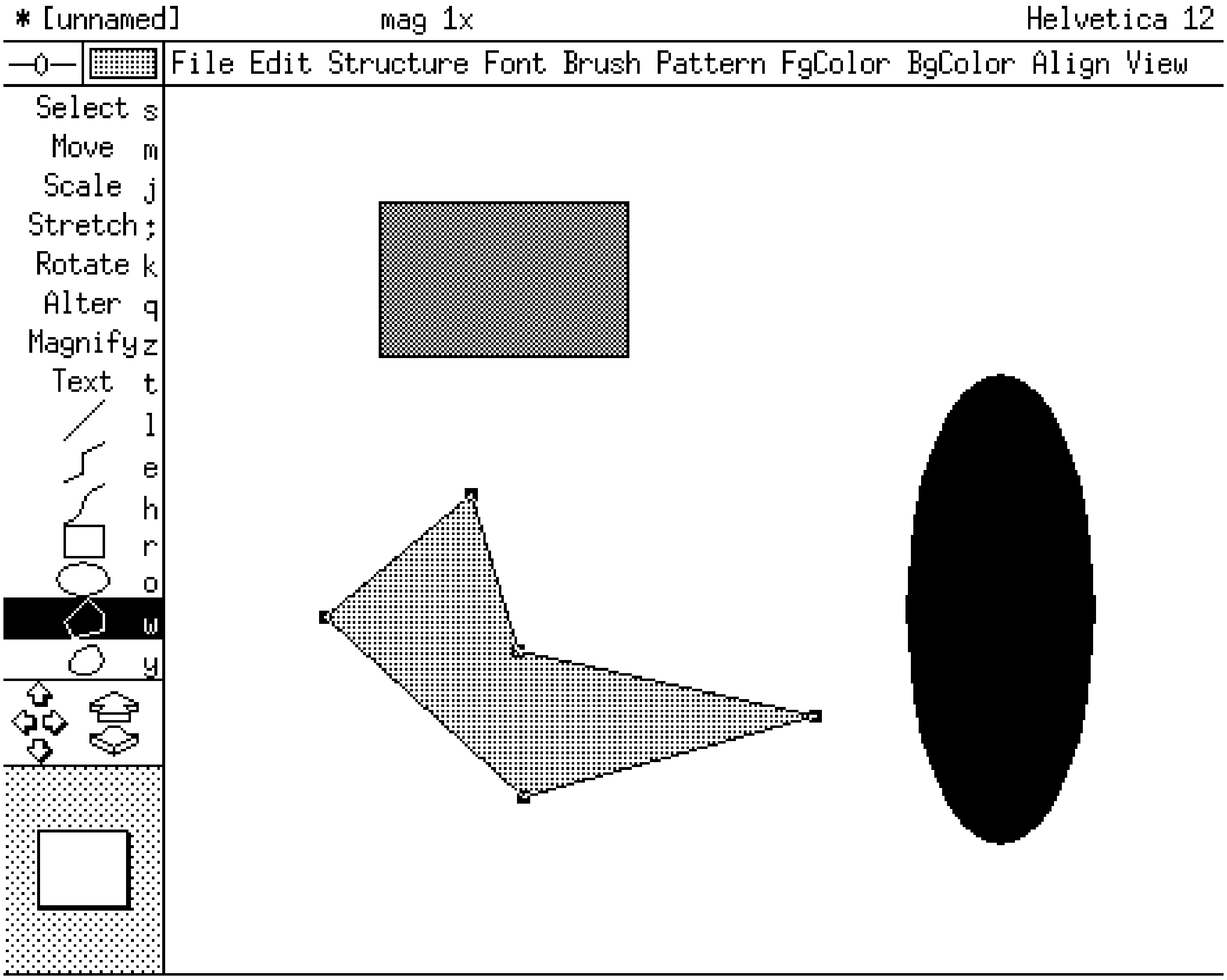

Figure 1: A typical drawing program. Each kind of primitive would be implemented as a subclass of a generic `drawableObject` class.