Applying Object-Oriented Design to Structured Graphics

John M. Vlissides and Mark A. Linton
Stanford University

Abstract

Structured graphics is useful for building applications that use a direct manipulation metaphor. Object-oriented languages offer inheritance, encapsulation, and runtime binding of operations to objects. Unfortunately, standard structured graphics packages do not use an object-oriented model, and object-oriented systems do not provide general-purpose structured graphics, relying instead on low-level graphics primitives. An object-oriented approach to structured graphics can give application programmers the benefits of both paradigms.

We have implemented a two-dimensional structured graphics library in C++ that presents an object-oriented model to the programmer. The **graphic** class defines a general graphical object from which all others are derived. The **picture** subclass supports hierarchical composition of graphics. Programmers can define new graphical objects either statically by subclassing or dynamically by composing instances of existing classes. We have used both this library and an earlier, non-object-oriented library to implement a MacDraw-like drawing editor. We discuss the fundamentals of the object-oriented design and its advantages based on our experiences with both libraries.

1 Introduction

Many software packages have been developed that support device-independent interactive graphics [11, 3, 4, 6, 7]. These packages provide various ways to produce graphical output. In *immediate-mode*, a graphical element such as a line appears on the screen as soon as it is specified. Several packages provide procedures for adding graphical elements to a *display list*; the elements appear on the screen after an explicit call to draw the display list. Graphical elements in the list can be stored as data or as procedural specifications. *Structured graphics* packages allow elements in a display list to be lists themselves, making it possible to compose hierarchies of graphical elements.

Application programs designed for workstations make extensive use of graphics in their user interfaces. Many programs such as drawing and schematics editors let the user manipulate graphical representations of familiar objects. Structured graphics can simplify the implementation of such applications because much of the functionality required is already implemented in the graphics package. For example, drawing editor operations for translating and scaling geometric shapes, enlarging and reducing the drawing, and storing its representation are supported by most structured graphics packages. Graphical hierarchies could be used to compose and manipulate groups of notes on staves in a music editor. A project management system could define the elements of bubble charts using graphical primitives and allow structural changes to be made interactively using display list editing operations.

However, there are drawbacks to using structured graphics. The library of procedures that comprises such packages is often large and monolithic, rich in functionality but difficult for the programmer to extend. Extensibility usually requires access to and manipulation of internal data structures, but such access is dangerous and can compromise the reliability of the system. Also, it is often difficult to edit and manipulate the display list, particularly when its elements are represented procedurally, because there is no way to refer to graphic and geometric attributes directly. Editing the display list may be inefficient as well. For example, if the display list is compiled into a more quickly executed form, then the list must be recompiled following editing before it can be drawn. These deficiencies make it likely that the structure provided by the package will not map well to that required by the application, forcing the programmer to define data structures and procedures that parallel the library's.

An object-oriented design offers solutions to these problems. Intrinsic to object-oriented languages are facilities for data hiding and protection, extensibility and code sharing through inheritance, and flexibility through runtime binding of operations to objects. However, existing object-oriented programming environments [5, 9] rely on immediate-mode graphics, and object-oriented user interface packages [2, 1] do not support general-purpose structured graphics. Ida [15] uses an object-oriented framework that decomposes structured graphics into a set of building blocks that communicate via message passing. Ida supports high-

[0]To appear in the Proceedings of the USENIX C++ Conference, Denver, Colorado, October 1988.

level functionality such as scrolling, though it does not provide some graphical capabilities that structured graphics systems usually have, such as rotations and composite transformations.

We have developed a C++ [12] library of graphical objects that can be composed to form two-dimensional pictures. The library is a part of the InterViews graphical interface toolkit [8] and runs on top of the X window system [10]. Our aim was to learn how inheritance and encapsulation could be used in the design of a structured graphics library. A base class **graphic** is defined from which all other structured graphics objects are derived. We show how a hierarchy of these primitives can be composed to form more complex graphics and how features such as hit detection and incremental screen update are incorporated into the model. We also compare this library to an earlier, non-object-oriented structured graphics library implemented in Modula-2, relating experiences we had in using each library to implement a MacDraw-like drawing editor.

2 Class Organization

The graphic class and derived classes collectively form the Graphic library. The class hierarchy is shown in Figure 1. Its design was guided by the desire to share code as much as possible without compromising the logical relationships between the classes.

The derived classes define the following graphical objects:

- **Point**, **Line**, **MultiLine**: a point, a line, and a number of connected lines
- **Rect**, **FillRect**: open and filled rectangles
- **Ellipse**, **FillEllipse**: open and filled ellipses
- **Circle**, **FillCircle**: open and filled circles
- **Polygon**, **FillPolygon**: open and filled polygons
- **BSpline**, **ClosedBSpline**, **FillBSpline**: open, closed and filled B-splines
- **Label**: a string of text
- **Picture**: a collection of graphics
- **Instance**: a reference to another graphic

All graphics maintain graphics state and geometry information. Graphics state parameters are defined in separate base classes. These include **transformer** (transformation matrix), **color**, **pattern** (for stippled area fills), **brush** (for line drawing), and **font**. Each graphics state class implements operations for defining and modifying its attributes. For example, transformers have translation, scaling, rotation, and matrix multiplication operations, and colors allow their component intensities to be varied.

A structured graphics package should be able to transfer its graphical representations to and from disk. GKS uses "metafiles" for this purpose. The files PHIGS uses are called "archives." Both packages provide procedures for saving and retrieving structures, for querying structures by name, and for deleting structures from the file.

The approach used by these packages requires the programmer to save and retrieve structures explicitly. The Graphic library uses persistent objects to automatically manage the storage of graphics. The graphic class and graphics state classes are derived from a **persistent** class that provides transparent access to objects whether they are in memory or on disk. Persistent objects are faulted in from disk when they are first referenced, and "dirty" objects are written to disk when the client program exits.

3 Graphic

The graphic base class contains a minimal set of graphics state including a transformer and foreground/background colors. Derived classes maintain additional graphics state according to their individual semantics. For example, the label class includes a font in addition to inherited state, filled objects maintain a pattern, and outline objects include a brush.

3.1 Operations

All graphics implement a set of operations defined in the base class. These include operations for

- drawing and erasing, optionally clipped to a rectangle,
- setting and retrieving graphics state values,
- translating, scaling, and rotating,
- obtaining a bounding box, and
- ascertaining whether the graphic contains a point or intersects a rectangle.

The `Contains` and `Intersects` operations are useful for hit detection. Their definitions are shown

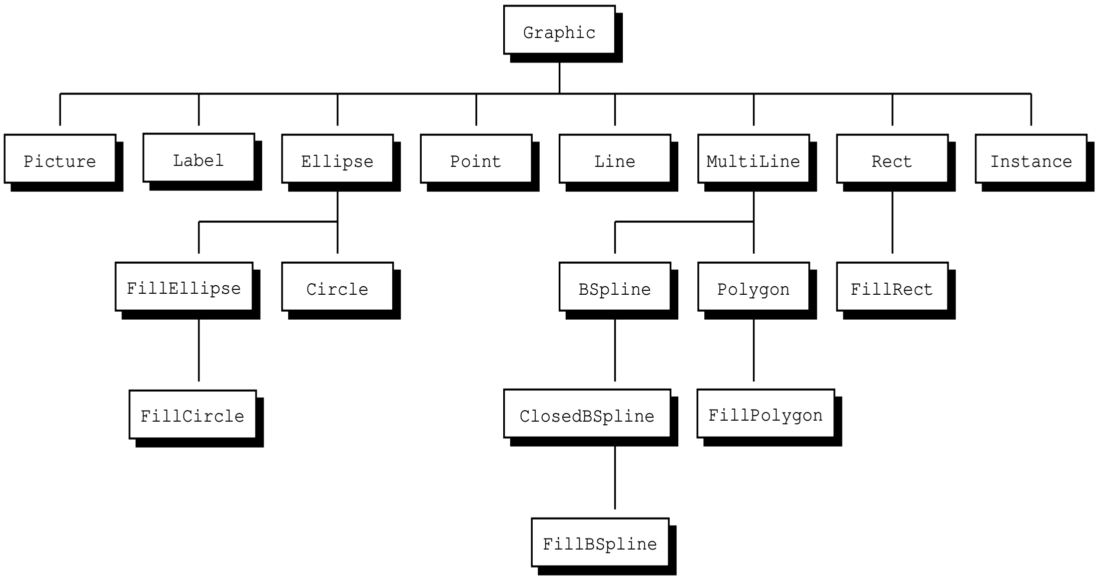

Figure 1: Graphic library class hierarchy

```
virtual boolean Contains(PointObj&);
virtual boolean Intersects(BoxObj&);
```

Figure 2: Interface to operations supporting hit detection

in Figure 2. `PointObj` and `BoxObj` are classes that serve as shorthand for specifying a point and a rectangular region, respectively. `Contains` can be used to detect an exact hit on a graphic; `Intersects` can be used to detect a hit within a certain tolerance.

3.2 Drawing Operations

Figure 3 lists the set of drawing and erasing operations defined on graphics. InterViews defines **canvas** objects and the **coord** type. A canvas represents a region of the display in which to draw. Canvases are rectangular and may overlap. A coord is a integer coordinate.

The graphic base class implements each erasing operation in terms of the corresponding drawing operation. An erase operation first sets the foreground color to the background color, then calls the drawing operation, and finally resets the foreground color to its original value.

The operations taking a single parameter draw and erase the graphic in its entirety. The coordinate parameters are used to specify a rectangular region. *Bounded* `Draw` and `Erase` operations use the rectangular region as a hint to the graphic's visibility. Graphics may perform optimizations based on this information. For example, because canvases do not permit drawing outside their boundaries, bounded draw and erase operations can cull parts of the graphic that fall outside the canvas.[1]

`DrawClipped` and `EraseClipped` clip during drawing or erasing. They are useful when drawing must be strictly limited to a portion of the canvas. For example, `DrawClipped` is often used to redraw portions of a graphic that had been obscured by an overlapping canvas.

4 Composite Graphics

Picture and **instance** are composite graphics. A picture composes other graphics into a single object, while an instance is a reference to another graphic. Both rely on a notion of *graphics state concatenation* to define how they are drawn.

4.1 Graphics State Concatenation

Composite graphics are like other graphics in that they maintain their own graphics state information, but they do not have their own geometric information. Composition allows us to define how the composite's state information affects its components. The graphic base class implements a mechanism for combining, or *concatenating*, graphics state information. The default behavior for concatenation is described below. Derived classes redefine the concatenation operations as needed.

Given two graphics states A and B, we can write their concatenation as $A \oplus B = C$, where C is the resultant graphics state. Concatenation associates but is not commutative; B is considered "dominant." C receives attributes defined by B. Attributes that B does not define are obtained from A. An exception is the transformation matrix; C's transformer is defined by postmultiplying A's transformer by B's. B thus dominates A in that C inherits B's attributes over A's, and C's coordinate system is defined by A's transformation with respect to B's.

A graphic might not define a particular attribute either because it is not meaningful for the graphic to do so (a filled rectangle does not maintain a font, for instance) or because the value of the attribute has been set to nil explicitly. Defined attributes propagate through successive concatenations without being overridden or modified by undefined attributes. For example, suppose graphics state A defines a font but B does not. Moreover, C maintains a font but its value has been set to nil. Then $D = A \oplus B \oplus C$ will receive A's font attribute. If A's transformer is nil but B and C's are non-nil, then D will receive a transformer that is the product of B's and C's. If $D = C \oplus A \oplus B$, then D will receive a transformer that is the product of C's and B's.

The semantics for concatenation as defined in the base class are useful for describing how composite graphics are drawn, but derived graphics can implement their own concatenation mechanism. This creates the potential for concatenation semantics that are more powerful than the default precedence relationship. For example, the concatenation operation could be redefined so that concatenating two colors would yield a third that is the sum or difference of the two. Two patterns could combine to form a pattern corresponding to an overlay of the two. This behavior could be used to define how to draw overlapping parts of a VLSI layout.

The ability to redefine concatenation semantics demonstrates how inheritance lets the programmer extend the graphics library easily. Flexibility is thus achieved without complicating or changing the library.

[1] The bounded operation could obtain the rectangular region directly from the canvas. For generality, however, the region is specified explicitly.

```
virtual void Draw(Canvas*);
virtual void Draw(Canvas*, Coord, Coord, Coord, Coord);
virtual void DrawClipped(Canvas*, Coord, Coord, Coord, Coord);

virtual void Erase(Canvas*);
virtual void Erase(Canvas*, Coord, Coord, Coord, Coord);
virtual void EraseClipped(Canvas*, Coord, Coord, Coord, Coord);
```

Figure 3: Interface to drawing operations

4.2 Picture

Pictures are the basic mechanism for building hierarchies of graphics. Each picture maintains a list of component graphics. A picture draws itself by drawing each component with a graphics state formed by concatenating the component's state with its own. Thus, operations on a picture affect all of its components as a unit. `Contains`, `Intersects`, and bounding box operations are redefined to consider all the components relative to the picture's coordinate system. The picture class defines the operations shown in Figure 4 for editing and traversing its list of components. Pictures have a notion of a "current" component, which aids in the traversal by acting as a position marker in the list of components.

Pictures also define operations for selecting graphics they compose based on position. These operations are shown in Figure 5. The `...Containing` operations return the graphic(s) containing a point; `...Intersecting` operations return the graphic(s) intersecting a rectangle; `...Within` operations return the graphic(s) falling completely within a rectangle.

Pictures draw their components starting from the first component in the list. The `Last...` operations can be used to select the "topmost" graphic in the picture, while `First...` operations select the "bottommost." The `Graphics...` operations return as a side-effect an array of all the graphics that satisfy the hit criterion. These operations also return the size of the array.

The following example demonstrates how concatenation can be used and extended using pictures. Consider a what-you-see-is-what-you get text editor that implements paragraphs using a subclass of picture called **paragraph** and words using a subclass of label called **word**. Both pictures and labels maintain a font attribute. Thus, each word can define its own appearance, and the paragraph can override the appearance of all the words through concatenation. For instance, defining a font attribute on the paragraph would cause all words to appear in that font independent of their individual attributes.

By deriving paragraph from picture, we can change the concatenation semantics; for example, the concatenation of an italic font with a bold font could yield a bold italic font. Defining an italic font attribute on the paragraph would thus italicize the paragraph without ignoring the font of individual words. Alternatively, paragraphs could rely on words to define the concatenation semantics. Thus, instances of different word subclasses could respond differently to formatting changes within the same paragraph.

4.3 Instance

An instance is a reference to another graphic (the *target*). Graphic library instances are functionally equivalent to instances in Sketchpad [14]. The concatenation of the instance's and target's graphics states is used when the instance is drawn or erased. An instance can thus redefine any aspect of the target's graphics state, but it cannot change the target's geometric information.

Instances are useful for replicating "prototype" graphics. Once the prototype is defined, it can appear at several places in a drawing without copying. Also, structural and graphics state modifications made to the prototype will affect its instances, thus avoiding the need to change instances individually.

5 Incremental Update

Structured graphics can be used to represent and draw arbitrarily complicated images. Many images (and most interesting ones) cannot be drawn instantaneously. Incremental techniques can be used to speed the process of keeping the screen image consistent with

```
void Append(Graphic*);
void Prepend(Graphic*);
void Remove(Graphic*);

void InsertAfterCur(Graphic*);
void InsertBeforeCur(Graphic*);
void RemoveCur();
void SetCurrent(Graphic*);
Graphic* GetCurrent();

Graphic* First();
Graphic* Last();
Graphic* Next();
Graphic* Prev();
boolean IsEmpty();
boolean AtEnd()
```

Figure 4: Picture editing operations

```
Graphic* FirstGraphicContaining(PointObj&);
Graphic* FirstGraphicIntersecting(BoxObj&);
Graphic* FirstGraphicWithin(BoxObj&);

Graphic* LastGraphicContaining(PointObj&);
Graphic* LastGraphicIntersecting(BoxObj&);
Graphic* LastGraphicWithin(BoxObj&);

int GraphicsContaining(PointObj&, Graphic**&);
int GraphicsIntersecting(BoxObj&, Graphic**&);
int GraphicsWithin(BoxObj&, Graphic**&);
```

Figure 5: Picture operations for selection

changes in the underlying graphical structure. Such techniques will be effective if the user makes small changes most of the time, and experience with interactive graphics editors shows this to be the case.

To support incremental update, the Graphic library includes a **damage** base class. A damage object is used to keep the appearance of graphics consistent with their representation. Damage objects try to minimize the work required to redraw corrupted parts of a graphic. The base class implements a simple incremental algorithm that is effective for many applications. The algorithm can be replaced with a more sophisticated one by deriving from the base class.

5.1 Interface

The interface to the damage class appears in Figure 6. When a damage object is created it is passed a graphic (usually a picture) for which it is responsible. The `Incur` operation is called by the client program whenever the graphic is "damaged." The graphic is incrementally updated when `Repair` is called. `Reset` discards accumulated damage without updating the graphic. Clients can determine whether any damage has been incurred using the `Incurred` operation.

5.2 Implementation

The damage class implements a simple algorithm for incremental update. Each damage object maintains zero, one, or two non-overlapping rectangles. A damage object must be notified whenever the graphic's appearance changes by calling the `Incur` operation with either a region of the canvas or a graphic as a parameter. If a graphic is supplied, its bounding box determines the extent of the damaged region.

`Incur` either stores the new rectangle representing the damaged region or *merges* it with one or both of the rectangles it has stored. Merging replaces a stored rectangle with the smallest rectangle circumscribing the rectangles being merged. `Repair` calls `DrawClipped` on the graphic for each stored rectangle.

The number of rectangles maintained by damage objects is limited to two because successive increases in the number of rectangles bring diminishing returns. This is a result of the overhead associated with drawing a graphic clipped; for complicated graphics this involves significant computation. We found that the limiting value of two yielded subjectively the quickest screen update on average in an object-oriented drawing editor based on the Graphic library. Typically the user either transforms an object in place (producing a single damaged rectangle) or moves an object (producing one or two rectangles). Assuming that drawing editors represent a fair benchmark for interactive graphics applications, the two-rectangle limitation offers advantages in both performance and implementation simplicity.

6 Experience

The design of the Graphic library was based on experience with an earlier structured graphics library we implemented in Modula-2. The Modula-2 design emphasized high drawing speed over low latency. It also tried to handle incremental update completely automatically; that is, it had no operation comparable to `Incur`. The extent of damage was inferred from the operations performed on each graphical object. Though the package attempted to provide an object-oriented interface, the implementation language's lack of inheritance resulted in a monolithic library that could not be extended easily.

We have developed two versions of an object-oriented drawing editor called `idraw`, shown in Figures 7 and 8. The first version uses the Modula-2 graphic library, while the second version uses the C++ Graphic library. This gives us a good opportunity for comparing the two libraries based on actual usage.

6.1 Graphics State Propagation versus Concatenation

A difference between the Graphic and Modula-2 libraries is in the way they manage graphics state. Modula-2 graphical elements propagate their graphics state to the leaves of the graphics hierarchy as part of the modification operation. Graphic library objects defer the propagation until they are drawn, relying on the concatenation mechanism to do the job. The rationale behind propagation was to make drawing as fast as possible. It was believed that on-the-fly concatenation would slow drawing unnecessarily. Thus, as much work as possible was done before the drawing routine was called.

We realized that propagation was a mistake as we used the Modula-2 library to implement `idraw`. Propagating graphics state each time an operation is called precludes amortizing many changes over a few draws. That is, if several state-modifying operations are made before the graphic is drawn, we can avoid traversing the structure if we defer propagation to draw time, when we must traverse it anyway.

Having made propagation an integral part of the Modula-2 library, there was no practical way for users

```
void Incur(Graphic*);
void Incur(BoxObj&);
void Repair();
void Reset();
boolean Incurred();
```

Figure 6: Interface to damage class

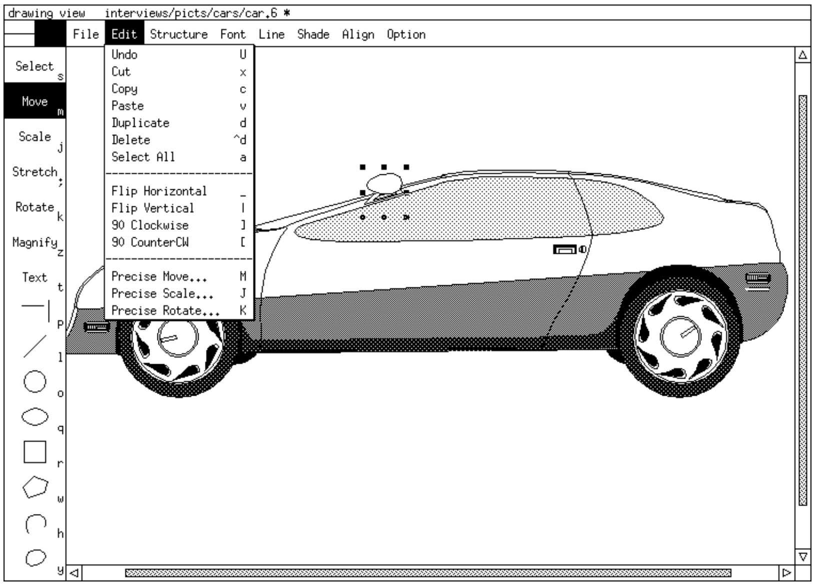

Figure 7: The `idraw` drawing editor, Modula-2 version

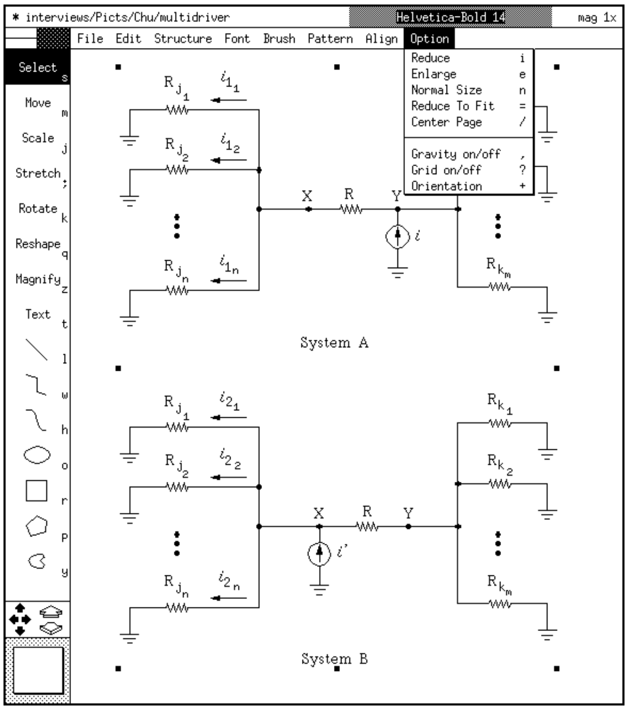

Figure 8: The `idraw` drawing editor, C++ version

to modify the library to use concatenation. An object-oriented design would have used inheritance to facilitate the modification of the library to use concatenation. In comparison, it would be straightforward to derive a new sort of picture and redefine its graphic state modification operations to propagate attributes immediately.

6.2 Incremental Update

The Modula-2 graphics library implemented an automatic incremental update feature. The library kept track of changes to objects by storing lists of rectangles with each object. Newly-added rectangles were merged with any rectangles in the list they intersected. The list of rectangles was ultimately limited by the object's bounding box; when a rectangle in the list became large enough to subsume the bounding box, the incremental update mechanism was disabled and the object would be drawn in its entirety.

The `Redraw` procedure was used to initiate incremental redraw of a graphical object. `Redraw` erased the regions defined by the rectangles in the object and redrew the object clipped to each rectangle. Any nested objects would be redrawn recursively.

This approach worked—the screen was never left in an inconsistent state following incremental redraw—but it did not always perform the update in an efficient way. The generality of the algorithm coupled with the lack of a way for the programmer to influence the redraw mechanism often rendered the facility useless; the programmer would bypass the mechanism and redraw damaged objects explicitly.

To illustrate, consider the case where a drawing is restructured so that an object obscured by other objects is brought to the top. A simple way to update the screen is simply to draw the object; nothing else need be redrawn. However, the incremental algorithm did not consider this optimization, and `Redraw` proceeded to redraw all the obscured objects as well.

The more serious problem arose because damaged rectangle information was *always* accumulated, since `Redraw` could be called at any time. This added overhead to every appearance-modifying operation. The overhead remained even if the programmer decided to bypass automatic redraw and perform the update manually. The addition of a `Disable` procedure that turned off rectangle accumulation complicated the use of the package and presented problems of its own: What should happen when automatic redraw is enabled again? Should old damage information be eliminated? How do we know the screen is still consistent?

The lesson we learned was that it is important not to exclude the programmer from the update process. Damage objects do not in any way interfere with the normal operation of graphics. They incur no overhead unless they are used, and they encapsulate the incremental update algorithm, making it easy to enhance or replace. In contrast, the update mechanism pervaded the older library. Damage objects give programmers the option of performing tricks of their own when updating the screen without paying for mechanisms they do not use.

6.3 Persistence

We have mixed feelings about having used persistent objects in the Graphic library. On one hand they are convenient because they free the programmer from worrying about storage. On the other hand, objects created by a program live in their own world analogous to the address space in which they were created. Thus, objects cannot communicate across program or machine boundaries easily, nor is there provision for moving objects from one world to another.

Persistent objects are useful for preserving the state of a program transparently across executions, but they are not suitable for communicating the state between processes. We expect that a later version of the Graphic library will incorporate a more conventional storage mechanism.

6.4 Cached Bounding Boxes

To improve performance, the more complex graphics such as multilines, polygons, splines, and pictures cache their bounding box once it is calculated. Caching can save substantial time, especially for large pictures, because the bounding box is needed whenever a graphic is drawn clipped or bounded.

The object-oriented approach makes it easy to add this optimization to classes that can use it without penalizing other classes. The graphic base class declares operations for caching, invalidating, and retrieving a bounding box. These are null operations by default; derived classes can redefine them if they use caching. Thus, individual graphics can define their own caching and invalidation criteria. Furthermore, since the base class does not allocate storage for the bounding box, no overhead is incurred on subclasses that do not require caching.

6.5 Quantitative and Qualitative Comparisons

This section presents quantitative and qualitative comparisons of the Modula-2 and C++ structured graphics libraries and versions of `idraw`. Note that any direct comparisons are necessarily crude because of differences in design criteria, in our experience level at the start of each library's implementation, and in the implementation languages themselves. Nevertheless, we offer these comparisons to add insight into the relative merits of the Modula-2 and C++ implementations.

Table 1 shows the source code sizes for both libraries and both versions of `idraw`. The library code is divided into five components: common code (that is, code that implements the same functionality in both libraries), code for incremental update, code for storing graphical objects on disk, code for hit detection, and comments. The `idraw` code is divided into common code, user interface code, and comments.

This partitioning lets us take into account different capabilities and levels of commenting when comparing code sizes. For example, the Graphic library has a general persistent object facility, whereas the Modula-2 library supports only manual read/write of graphical objects. Graphic subclasses implement fine-grain hit detection, while the Modula-2 library can detect hits only within an object's bounding box. The Modula-2 library uses a more complicated incremental update mechanism and is commented more heavily than the Graphic library. Modula-2 `idraw` implements scroll bars, pull-down menus, and rubberbands explicitly, while InterViews provides this functionality in the C++ version.

From the information in Table 1, we conclude only that the C++ and Modula-2 code is comparable in size. The amount of common code in the structured graphics libraries is about the same, and the C++ version has proportionally more code to implement added functionality. The Modula-2 `idraw` is somewhat smaller than the C++ version, taking into account that C++ `idraw` relies on InterViews to implement its user interface. However, C++ `idraw` provides more functionality, including arbitrary-level undo (versus single-level for Modula-2 `idraw`), more sophisticated text editing, and user customizability.

A possible disadvantage of an object-oriented implementation is a runtime performance penality because of overhead such as method lookup. In the implementation of C++ we used, the overhead amounts to three or four extra memory references per virtual function call [13]. To see whether this overhead has a significant impact on the performance of `idraw`, we measured how long it took each version of `idraw` to do three different operations on two different drawings, `car.6` (shown in Figure 7) and `multidriver` (shown in Figure 8). These are representative of two common types of drawings: artistic drawings with

		Modula-2	*C++*
structured graphics library	common code	3600	3500
	incremental update	500	100
	hit detection	400	1500
	persistence	600	1300
	comments	700	300
	total lines	5800	6700
`idraw`	common code	13000	14000
	user interface	2000	0
	comments	1000	2000
	total lines	16000	16000

Table 1: Comparison of Modula-2 and C++ source code (in lines)

many complex, overlapping polygons and splines, and technical drawings consisting mainly of rectangles, lines, and text with little or no overlap. We timed the following operations:

1. In the "zoom #1" test, the drawing is zoomed from half size to quarter size and back. The drawing is fully visible throughout the test.

2. In "zoom #2," the drawing is zoomed from half size to full size and back. The drawing is clipped when drawn at full size so that only half is visible.

3. In "rotation," the (top-level) object in the drawing is rotated 90°.

Table 2 shows the average of ten trials for each test. The C++ version outperforms the Modula-2 version in every test. The difference in speed is greatest for the rotation test on `car.6`, but this difference is exaggerated because of a bug in the Modula-2 library's incremental update routine that caused redundant redraws of two subcomponents. In general, the Modula-2 library is handicapped by the extra traversals associated with graphic state propagation and incremental update. The results would be more comparable if the Modula-2 library were modified to use concatenation and the simpler incremental update algorithm of the damage class.

The last quantitative comparison involves the object code sizes for each library and `idraw` version. These values are shown in Table 3. The C++ sizes are larger mainly because of the added functionality of both the Graphic library and C++ `idraw`, constructor, destructor, and inline code, and the overhead associated with virtual pointer tables.

From a qualitative standpoint, the Graphic library and the corresponding version of `idraw` are both significantly better structured, more understandable, and "cleaner" overall than their Modula-2 counterparts. One could argue that the lessons learned in the Modula-2 implementation efforts led to superior C++ versions. However, the versions of `idraw` were developed by two different people. In fact, the Modula-2 version was its author's second attempt at a drawing editor, while the C++ version was its author's first attempt. The object-oriented paradigm simply invites good program structuring through inheritance, encapsulation, and late binding, all of which promote modularity and flexibility.

7 Conclusion

A striking aspect of graphics packages such as CORE, GKS, and PHIGS is their size and complexity. These packages are intended as standards that provide machine independence, extensive functionality, and generality, and they largely succeed in these respects. However, all reflect their procedural implementation in their interface. Programmers cannot extend primitives through inheritance to modify their semantics. The result is a substantial complexity penalty for every increase in flexibility.

For example, some packages bind graphics state attributes statically to graphical objects when the objects are created. Others provide a simple form of state inheritance by allowing graphics to reference other graphics in a manner similar to instances in the Graphic library. These facilities are significantly less flexible than the graphics state concatenation mechanism, the semantics of which can be changed on a per-class basis. In an object-oriented package, generality can be achieved through class inheritance instead of supporting a broad range of behaviors explicitly.

Drawing	*Test*	*Modula-2*	*C++*
`car.6`	zoom #1	18	8.3
(82 objects)	zoom #2	12	6.3
	rotation	15	4.5
`multidriver`	zoom #1	24	12
(361 objects)	zoom #2	18	8.3
	rotation	11	6.7

Table 2: Comparison of Modula-2 and C++ `idraw` drawing performance (in seconds)

	Modula-2	*C++*
structured graphics library	40	110
`idraw`	130	280

Table 3: Comparison of Modula-2 and C++ object code sizes (in kilobytes)

Another advantage of the object-oriented approach is the ability to treat graphical objects generically, relying on the runtime system to determine the correct method for a particular object. The `virtual` mechanism accomplishes this in C++. Thus, functionality such as hit detection can be implemented in a simple way without identifying objects with element pointers and labels. Furthermore, escape mechanisms for exploiting special hardware facilities are unnecessary; subclasses can be derived that reimplement key operations such as `Draw` to take advantage of unique capabilities.

In our experience, structured graphics is useful for applications that allow the user to manipulate graphical objects interactively. Structured graphics is less useful for implementing the appearance of the user interface. It is unnecessary to define scroll bars, menus, and buttons using structured graphics because they are simple to draw procedurally and their structure rarely changes. Thus, structured graphics is not a replacement for immediate-mode graphics.

We are interested in using the Graphic library for animating graphics. Structured graphics is appropriate for animation if the hardware is fast enough to support it. Also, the current implementation does not provide three-dimensional capabilities. Extending the library to support three dimensional graphics would require significant additions to base class functionality, for example, to incorporate operations governing lighting models and point of view, three-dimensional analogs of `Contains` and `Intersects`, and additional information when clipping.

Of more immediate interest is the introduction of version 2.0 of C++ [13] with multiple inheritance, among other enhancements. Though single inheritance is very useful, it often forces the programmer to derive from one of two equally attractive classes. This limits the applicability of predefined classes, often making it necessary to duplicate code. For example, there is no way to derive a graphic that is both a circle and a picture; one must derive from one or the other and reimplement the functionality of the class that was excluded.

The availability of multiple inheritance will undoubtedly change the class hierarchy shown in Figure 1. Classes such as **filled** and **open** could be defined to simplify the relationships between filled and non-filled graphics, which are currently derived as they are to maximize code sharing. Persistence could be implemented as a separate class from which to inherit. Thus, non-persistent classes can avoid the small space overhead caused by deriving graphic from a persistent class.

Acknowledgments

This work was supported by the Quantum project through a gift from Digital Equipment Corporation. John Interrante implemented the C++ version of `idraw`. Paul Calder and Craig Dunwoody provided helpful comments on earlier drafts of this paper.

References

[1] Apple Programmer's & Developer's Association. *MacApp: The Expandable Macintosh Application*, 1987.

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

[3] P. Bono et al. GKS: The first graphics standard. *IEEE Computer Graphics & Applications*, 2(5):9–23, July 1982.

[4] Status report of the graphics standards planning committee of ACM/SIGGRAPH. *Computer Graphics*, 13(3), Fall 1979.

[5] Adele J. Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, Reading, MA, 1984.

[6] W.T. Hewitt. Programmers Hierarchical Interactive Graphics System (PHIGS). In G. Enderle et al., editors, *Advances in Computer Graphics I*. Springer-Verlag, 1986.

[7] Keith A. Lantz and William Nowicki. Structured graphics for distributed systems. *ACM Transactions on Graphics*, 3(1):23–51, January 1984.

[8] Mark A. Linton, Paul R. Calder, and John M. Vlissides. InterViews: A C++ graphical interface toolkit. Technical Report CSL–TR–88–358, Stanford University, July 1988.

[9] Patrick D. O'Brien, D.C. Halbert, and Mike F. Kilian. The Trellis programming environment. In *ACM OOPSLA '87 Conference Proceedings*, pages 91–102, Orlando, FL, October 1987.

[10] Robert W. Scheifler and Jim Gettys. The X window system. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.

[11] Silicon Graphics, Inc. *IRIS User's Guide*, 1984.

[12] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 1986.

[13] Bjarne Stroustrup. The evolution of C++: 1985 to 1987. In *Proceedings of the USENIX C++ Workshop*, pages 1–21, Santa Fe, NM, November 1987.

[14] I.E. Sutherland. *Sketchpad: A Man-Machine Graphical Communication System*. PhD thesis, MIT, 1963.

[15] Robert L. Young. An object-oriented framework for interactive data graphics. In *ACM OOPSLA '87 Conference Proceedings*, pages 78–90, Orlando, FL, October 1987.