Glyphs: Flyweight Objects for User Interfaces

Paul R. Calder and Mark A. Linton
Center for Integrated Systems
Stanford University
Stanford, California 94305

Abstract

Current user interface toolkits provide components that are complex and expensive. Programmers cannot use these components for many kinds of application data because the resulting implementation would be awkward and inefficient. We have defined a set of small, simple components, called glyphs, that programmers can use in large numbers to build user interfaces. To show that glyphs are simple and efficient, we have implemented a WYSIWYG document editor. The editor's performance is comparable to that of similar editors built with current tools, but its implementation is much simpler. We used the editor to create and print this paper.

1 Introduction

Most user interface toolkits are object-oriented because program objects are a natural way to represent the objects that users manipulate. Current toolkits provide objects such as buttons and menus that let programmers build interfaces to application commands, but they do not provide objects for building interfaces to application data. Without this support, programmers must often define many components from scratch.

To offer the full benefits of an object-oriented model, a toolkit must encourage programmers to use objects for even the smallest components in the interface. Current toolkits let programmers build interfaces that contain hundreds of objects. But to reflect the fine-grained structure in application data, programmers may need hundreds of thousands of objects. For example, text is logically composed of characters, so an application that displays text should therefore build its views from objects that display individual characters. Using this approach, a moderate-sized document, such as a technical paper or a chapter in a book, will need at least 50,000 character objects.

This approach can dramatically simplify an implementation, but it is practical only if objects are simple and cheap. The combination of faster hardware and more efficient object-oriented languages has now made it possible to use objects at this finer level of granularity without sacrificing interactive response.

We have designed a set of 'flyweight' components, called glyphs, that are simple and efficient. The Glyph base class defines a protocol for drawing; subclasses define specific appearances such as graphic primitives (lines and circles), textual primitives (characters and spaces), and composite objects (tilings and overlays). Applications define their appearance by building hierarchies of glyphs.

We have implemented glyphs as an extension to the InterViews toolkit [2], which is written in C++. To show that glyphs are practical, we have implemented a WYSIWYG document editor that creates a glyph for each character in the text. Although its implementation is simple, this editor is powerful enough and efficient enough for many common document preparation tasks. We used our editor to create, format, and print this paper.

In section 2 we show how to build a simple text view using glyphs. We start with a simple ASCII file viewer and evolve it to a view that supports multiple fonts, embedded graphics, hyphenation, and multiple column formatting. Section 3 discusses the glyph class design in more detail, and section 4 describes how it can be implemented efficiently. In section 5 we compare glyphs with similar components provided by other toolkits. Finally, section 6 examines the implementation and performance of our glyph-based editor.

Research supported by a gift from Digital Equipment Corporation and a grant from the Charles Lee Powell Foundation.

```
void TextView::draw(
  Canvas* canvas;
  const Painter& p,
  const Allocation& a
) {
  Font* f = p.font();
  Coord x0 = a.x();
  Coord x = x0;
  Coord y = a.top() - f->ascent();
  Coord line_height =
    f->ascent() + f->descent();
  rewind(file);
  int c;
  while ((c = getc(file)) != EOF) {
    if (c == '\n') {
      x = x0;
      y -= line_height;
    } else {
      p.character(canvas, c, x, y);
      x += f->width(c);
    }
  }
}
```

Figure 1: A simple TextView

2 Building a text view

We chose to illustrate the use of glyphs in a text-based application because this example illustrates many of the shortcomings of current toolkits. Despite the central role text plays in many applications, most current toolkits limit the variety of text views that programmers can build. These toolkits provide only large, complex components that offer packaged solutions to specific text viewing problems; they do not provide simple components that let programmers define their own solutions. If a programmer needs more than the predefined components provide, then he must revert to low-level programming to define a new view.

We start by considering an application that will display ASCII-encoded text from a file. We define a glyph subclass, TextView, whose appearance depends on the file. In this example, we focus on the component that displays the text; we are not concerned with the application scaffolding that surrounds the text view because existing toolkits adequately handle this part of the interface.

A glyph's appearance is determined by its draw operation. To define TextView's appearance, we must redefine this operation. Draw will be called with parameters specifying the surface on which to draw (called a Canvas), the graphics attributes to use when drawing (a Painter), and geometrical information about the size and position of the view (an Allocation).

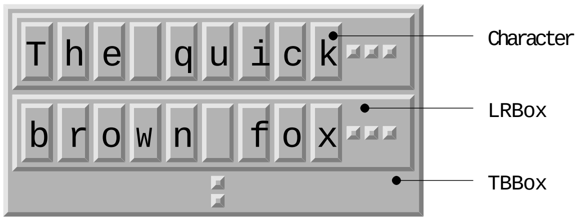


Figure 2: TextView object structure

The straightforward approach is to iterate through the file, drawing a representation of each printable character and beginning a new line for each newline character. Figure 1 shows what the code would look like. However, if the view is redrawn frequently, this simple implementation will be inefficient because it reads and draws the entire file on each call, even if only a small part of the view is damaged.

We can make the view more efficient by storing in the view a representation of the text and the coordinates at which it is drawn. Then we need draw only those lines (or parts of lines) that lie within the damaged region. However, this approach can considerably complicate the view's implementation. For example, the Athena Text Widget [4], which uses a similar scheme, contains about 10,000 lines of code spread over 20 source files. Of course, the Text Widget does more than just display text. But even if we count only the text display code, the implementation contains several thousand lines.

We can retain the simplicity of the original code by defining TextView as a composite glyph that contains other glyphs. We will use three predefined glyph subclasses: a Character draws a single character, an LRBox is a composite that tiles its components left-to-right, and a TBBox is a composite that tiles top-to-bottom. Figure 2 shows the resulting object structure, and figure 3 shows the code that builds the

```
TextView::TextView (FILE* file) {
  TBBox* page = new TBBox();
  LRBox* line = new LRBox();
  int c;
  while ((c = getc(file)) != EOF) {
    int (c == '\n') {
      page->append(line);
      line = new LRBox();
    } else {
      line->append(new Character(c));
    }
  }
  body(page);
}
```

Figure 3: TextView built from glyphs

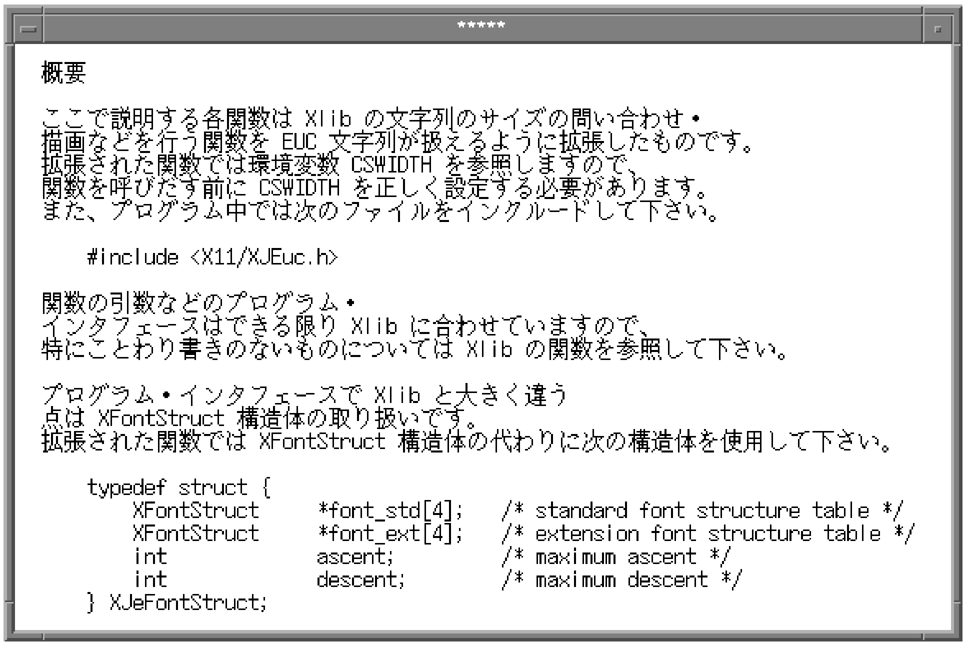


Figure 4: TextView displaying EUC-encoded text

view. We iterate through the file, creating a Character for each printable character, an LRBox to contain all the characters in each line, and a TBBox to contain all the lines in the file. The body function installs the TBBox as the internal representation of the TextView. The draw operation (which is not shown) simply calls draw on the TBBox.

The code that builds a TextView is similar to the original draw code, except that instead of calling functions to draw the characters, we build objects that will draw themselves whenever necessary. Using objects solves the redraw problem because only those objects that lie within the damaged region will get draw calls. The programmer does not have to write the code that decides what objects to redraw—that code is in the toolkit (in this example, in the implementation of the Box draw operation). Indeed, the glyph-based implementation of TextView is even simpler than the original code because the programmer need only declare what objects he wants—he does not need to specify how the objects should interact.

```
while ((c = getc(file)) != EOF) {
  if (c == '\n') {
     line = new LRBox();
+ } else if (!isascii(c)) {
+    line->append(
       new Character(
          tojis(c, getc(file)), k14
       )
     );
  } else {
     line->append(new Character(c, a14));
  }
}
```

Figure 5: Modified TextView that displays Japanese text

2.1 Multiple fonts

Because we built TextView with glyphs, we can easily extend it to add functionality that might otherwise be difficult to implement. For example, Figure 4 shows a screen dump of a version of TextView that displays EUC-encoded Japanese text. Adding this feature to a text view such as the Athena Text Widget would require a complete rewrite. Here we only add two lines of code. Figure 5 shows the change.

Character glyphs take an optional second constructor parameter that specifies the font to use when drawing. For ASCII-encoded text we create Characters that use the 8-bit ASCII-encoded “a14” font; for JIS-encoded text (kanji and kana characters) we create Characters that use the 16-bit JIS-encoded “k14” font.

2.2 Mixing text and graphics

We can put any glyph inside a composite glyph; thus it is straightforward to extend TextView to display embedded graphics. Figure 6 shows a screen dump of a view that makes the whitespace characters in a file visible by drawing graphical representations of spaces, newlines, and formfeeds. Figure 7 shows the modified code that builds the view.

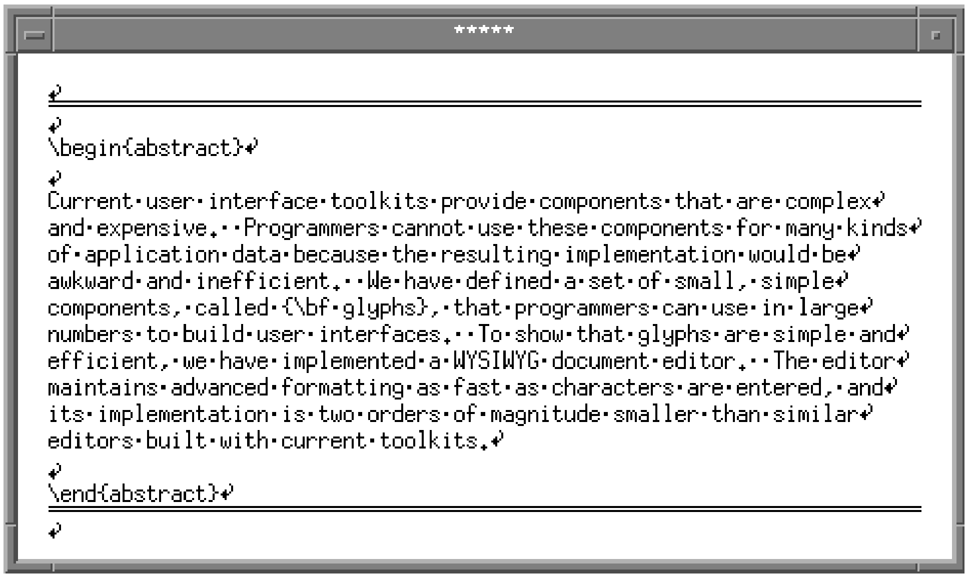


Figure 6: TextView displaying space characters

A Stencil is a glyph that displays a bitmap, an HRule draws a horizontal line, and VGlue represents vertical blank space. The constructor parameters for Rule and Glue specify the object's size in printer's points.

2.3 Formatting the text view

Glyphs encapsulate the task of drawing; we can define other components that handle higher-level operations such as text formatting. These components manipulate arbitrary glyphs—they are not restricted to text. Toolkits that define only large-grained objects cannot provide these kinds of components because they do not define the primitive components that the operations manipulate.

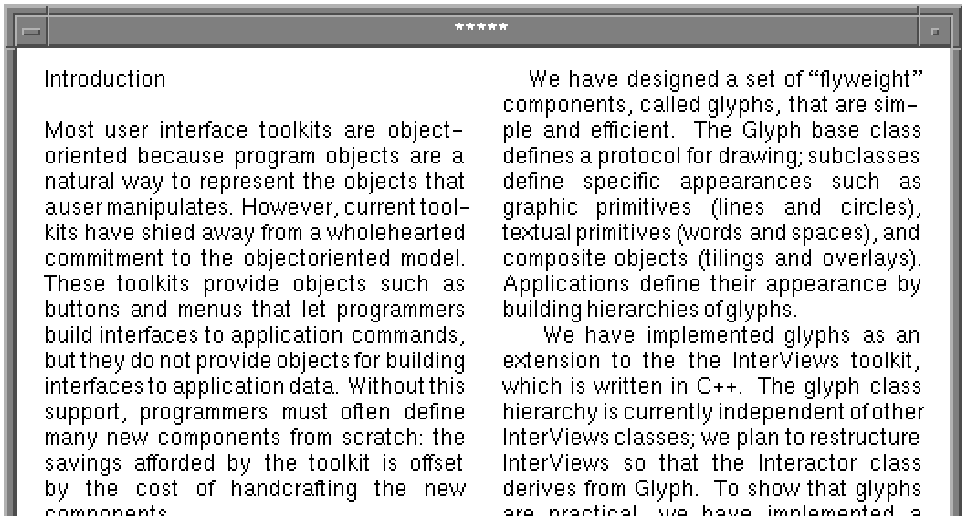
```
Bitmap* nl = new Bitmap("newline.bm");
Bitmap* sp = new Bitmap("space.bm");

while ((c = getc(file)) != EOF) {
  if (c == '\n') {
    line->append(new Stencil(nl));
    page->append(line);
    line = new LRBox();
  } else if (c == ' ') {
    line->append(new Stencil(sp));
  } else if (c == '\f') {
    page->append(line);
    page->append(new HRule(1));
    page->append(new VGlue(1));
    page->append(new HRule(1));
    line = new LRBox();
  } else {
    line->append(new Character(c));
  }
}
```

Figure 7: Code for displaying space characters

Adding text formatting capabilities to TextView is straightforward. Figure 8 shows a version that formats text (or any other glyphs) into lines of a specified width. The code introduces two more predefined glyphs: a Composition is a composite glyph that formats its components into groups, and a Discretionary represents

```
TextView::TextView (FILE* f, Coord w) {
  Composition* page = new LRComposition(
    new TBBox(),
    new SimpleCompositor(), nil, w
  );
  int c;
  while ((C = getc(file)) != EOF) {
    if (c == '\n') {
      page->append(
        new Discretionary(
          PenaltyGood,
          new HGlue(fil),
          new HGlue(), nil, nil
        )
      );
    } else if (c == ' ') {
      page->append(
        new Discretionary(
          0,
          new Character(' '),
          new HGlue(), nil, nil
        )
      );
    } else {
      page->append(new Character(c));
    }
  }
  body(page);
}
```

Figure 8: Formatting TextView


Introduction

Most user interface toolkits are object-oriented because program objects are a natural way to represent the objects that a user manipulates. However, current toolkits have shied away from a wholehearted commitment to the objectoriented model. These toolkits provide objects such as buttons and menus that let programmers build interfaces to application commands, but they do not provide objects for building interfaces to application data. Without this support, programmers must often define many new components from scratch: the savings afforded by the toolkit is offset by the cost of handcrafting the new

We have designed a set of "flyweight" components, called glyphs, that are simple and efficient. The Glyph base class defines a protocol for drawing; subclasses define specific appearances such as graphic primitives (lines and circles), textual primitives (words and spaces), and composite objects (tilings and overlays). Applications define their appearance by building hierarchies of glyphs.

We have implemented glyphs as an extension to the the InterViews toolkit, which is written in C++. The glyph class hierarchy is currently independent of other InterViews classes; we plan to restructure InterViews so that the Interactor class derives from Glyph. To show that glyphs

Figure 9: TextView with multi-column formatting and hyphenation

possible place for a formatting break.

In this example, we use an LRComposition to break the text into lines. The LRComposition creates LRBoxes that contain the glyphs in each line and then it inserts the LRBoxes into the TBBox that was passed to its constructor. The resulting object structure is similar to the previous one we built explicitly; the difference is that the composition is now computing the line breaks. We tell the composition how to find formatting breaks by passing a Compositor to its constructor. Here we use a SimpleCompositor, which does simple line filling. Other Compositors implement different algorithms. For example, a TeXCompositor implements the TeX line breaking algorithm, which finds the set of line breaks that minimizes total demerits for a complete paragraph.

The first constructor parameter for Discretionary defines the formatting penalty that would be incurred if the break was taken; the remaining parameters define the appearance of the discretionary in its unbroken and broken states. In this example, we represent space characters as discretionaries with zero penalty (suitable places for a break). When unbroken, the Discretionary looks like a normal space character; when the break is taken, it looks like a piece of Glue. The constant PenaltyGood in the constructor for the newline Discretionary represents the best possible place to break. This break, which marks the end of a paragraph, will always be taken.

We can continue to extend TextView by adding support for more advanced formatting. For example, in addition to line breaking, we can use Compositions to format lines into columns and columns into pages. We can also specify possible hyphenation points by inserting Discretionaries, and we can justify the right margin by using stretchable Glue for space characters. Even with these additional features, the implementation contains less that 50 lines of code. Figure 9 shows a screen dump of this version of TextView. We used a TeXCompositor for line breaks, which produces high-quality formatting. The compositor takes few hyphenation points, and the ''color'' of the text is even because the TeX algorithm is usually able to avoid bad breaks.

3 Glyph protocol

Building views with glyphs is like building a model with Lego blocks: in both cases the builder chooses components and snaps them together. The bumps and holes on Lego blocks specify how they fit together. Similarly, the protocol that glyphs obey specifies how glyphs fit together. Lego is a successful building set because it offers components that are easy to understand and easy to use. It's easy to build complex structures, even those composed of large numbers of blocks, because it's easy to fit individual blocks together. In designing the Glyph protocol we sought this same simplicity.

The Glyph protocol consists of three operations:

- Request defines a glyph's preferred screen space allocation. Composite glyphs calculate their own requests from the requests of their components.

- Allocate tells the glyph how much space it actually got. Composite glyphs apportion their allocation among their components according to the components' requests.

```
void Character::request(const Painter&, Requisition& requisition) {
  requisition.require(
    Dimension_X,
    Requirement(_font->width(_c), 0, 0)
  );
  requisition.require(
    Dimension_Y,
    Requirement(_font->ascent() + _font->descent(), 0, 0)
  );
}

void Character::draw(
  Canvas* canvas, const Painter& painter, const Allocation& allocation
) {
  Painter p(painter);
  p.font(_font);
  p.character(canvas, _c, allocation.x(), allocation.y());
}
```

Figure 10: Character class implementation

- Draw specifies the glyph's appearance. Composite glyphs recursively draw their components.

Glyph subclasses redefine these three operations to specify particular behavior. For example, figure 10 shows the implementation of request and draw for the Character class.

Glyphs request enough space to display themselves at their preferred size; they also specify their willingness to stretch or shrink if the preferred size cannot be allocated. In figure 10, Character requests a preferred width equal to the width of the displayed character in the selected font, and it requests a preferred height equal to the sum of the font's ascent and descent. This glyph is unwilling to stretch or shrink in either the X or Y dimension, but other glyphs, such as glyphs that represent the spaces between words, specify non-zero stretchabilities or shrinkabilities.

The draw operation creates a new painter (graphics context) so that it can override the default font in the painter it was passed. Then it draws its character at the specified (and) position on the canvas (drawing surface).

4 Making glyphs cheap

Massive use of objects is practical only if the objects themselves are cheap and efficient. These concerns drove the design and implementation of the glyph protocol.

We made glyphs small by relying on dynamic state: we pass information about a glyph's context to each glyph operation. This approach has two important consequences:

1. Glyphs need not store all the information that specifies their appearance. For example, Character glyphs do not store their position or a full complement of their graphics context because position and painter are passed to the draw operation.
2. Glyphs can be shared—the same glyph can appear in more than one composite glyph. Sharing reduces the effective size of a glyph because the actual size is spread over the shared uses. For example, TextView need build only a single instance of Character to represent the letter ''a''; it can use that same instance wherever the letter appears in the file.

Some glyphs do store information about their context. For example, it is sometimes necessary to know the actual geometry allocation or graphics context for a particular glyph. The protocol allows glyphs to store information that they need; however, it does not require them to do so.

We made glyphs fast by tuning the hot-spots we observed in running applications. Often, we were faced with the familiar trade-off between space and time; a fast implementation will store information that is needed often but costly to calculate, but too much stored information will make an application unacceptably large.

An important example is the implementation of the Box composite glyph. Boxes calculate the allocations of their components based on the component's individual geometry requests. This information is required whenever the Box is drawn because the information must be passed to the components' draw operations. Boxes store the results of the allocation calculation because recalculating it on each draw would be too slow. However, they do not store the requisition information that they need to calculate the allocations because re-allocating a box occurs much less often than re-drawing.

5 Comparing Toolkits

Current toolkits provide primitive components that are more expensive than glyphs. Table 1 shows the sizes of the simplest viewable components in the X Toolkit Athena Widgets [4], the Andrew Toolkit [3], ET++ [6], and our extended version of InterViews.

Glyphs are smaller than any of the other objects because they do not store graphical state and because their role is limited only to drawing. The primitive components in other toolkits are also responsible for handling other aspects of the interface, such as user input or view updates, that are often inappropriate at the lowest level of the view structure.

Like Extended InterViews, both the Andrew Toolkit and ET++ allow programmers to build hierarchical views containing embedded views of arbitrary types. These toolkits define components that can display documents containing mixed text and graphics. In practice, however, the size of the basic embeddable components and the lack of the ability to share components limits their use in large numbers—the components are used at the word or paragraph level instead of the character level.

Table 2 estimates the number of lines needed to implement two components using each of the toolkits. We counted the additional lines of code needed to implement the components on top of the primitive toolkits objects in table 1. ''Label'' refers to a component that displays one line of static text, often as a label or heading in a view. Usually, this component is one of the simplest predefined views in the toolkit. ''TextView'' refers to the component in the toolkit that is most similar to the TextView we built in section 2. All the toolkits provide components that do more than simply display text. We estimated the number of lines related specifically to the display task: for the Athena Widgets, we based our estimate on the AsciiSink component of AsciiText; for the Andrew Toolkit, we used the textview class; for ET++ we used the StaticTextView class.

The numbers for glyph-based InterViews do not include code for classes such as TeXCompositor, but the other toolkits do not provide the functionality that TeXCompositor does. The key observation is that glyphs allow us to extend the toolkit with higher-level components that programmers can use in their own views.

Toolkit	Object	Size
Athena Widgets	Simple	132
Andrew Toolkit	view	44
ET++	VObject	32
Extended InterViews	Glyph	8

Table 1: Sizes of base objects in bytes

Toolkit	Label	TextView
Athena Widgets	500	2000
Andrew Toolkit	300	3000
ET++	500	2000
Extended InterViews	10	20

Table 2: Lines of code for views

Several other systems define lightweight graphical objects in the spirit of Glyphs. However, none attempts to use these objects at as fine a grain as we use Glyphs. In particular, no system uses objects to represent components as small as individual characters in text views.

Smalltalk-80 defines a View object as part of the Model-View-Controller (MVC) user interface framework [1]. Views are similar to Glyphs in that they are responsible only for the appearance of the interface (input is handled by Controller). However, Views are more expensive because they store graphical context and position information.

Ida [7], an object-oriented graphics framework built on Impulse-86, defined data source objects that produce a visible representation of data. Ida applications define their appearance by composing data displays, which display data sources. However, Ida provides no mechanism for building hierarchies of data sources.

The Coral user interface toolkit [5] defines graphical objects that represent primitive graphical objects such as lines and polygons. Hierarchies of graphical objects are built with composite objects called aggregates. The sizes and positions (or other attributes) of graphical objects are specified using constraints. However, the overhead of the constraint system, and the size of the objects, makes them considerable more expensive than glyphs.

6 A Practical Example

To test our ideas and our implementation, we built a document editor that uses a glyph to display each character in a document. Our goal was to show that this approach can lead to dramatic savings in implementation effort yet is practical using current technology. We styled our editor as a technical paper preparation tool—we used the editor to produce this paper.

The editor has an extensive set of text formatting capabilities: character formats include bold, italic, typewriter, and sans-serif styles in a range of text sizes; page formats allow multiple columns and variable page sizes; and document formats support various text sizes and line spacings. These features are easy to provide because they are well supported by the glyph model—they are a simple extension of the ideas we discussed in Section 3. For instance, characters are styled by creating glyphs with the desired fonts, and formatting breaks are calculated by a compositor (we used a TeX compositor for both line and column breaks).

The editor also has several features that are not commonly found in editors of this type. For example, document floats (figures and tables) are anchored to the text, but their position can be overridden by the user. Final figure placement is specified by simply "plowing" the figure to the desired position—the editor dynamically flows the text around the figure as it is moved. We have found that this interactive approach is convenient for fine-tuning the final appearance of the document. Of course, this technique is only practical because the editor is able to reformat the document at interactive speeds.

In addition to text formatting, our editor has a rudimentary table editor and can import graphics generated by idraw (a drawing editor) or from screen dumps. Tables and graphics are represented simply as glyphs embedded in the text; idraw drawings are hierarchies of graphical objects, screen dumps are scanned images, and tables are row-and-column composites containing editable text cells. Tables will grow in width and height to accommodate text typed into the cells.

In designing our editor, we have emphasized those features that best show the value of the glyph model. Currently, our editor does not support higher-level document features such as automatic numbering and referencing—we inserted the numbering and referencing in this paper by hand. We plan to add these features in the future.

6.1 Lines of Code

We estimated the implementation effort for our editor by counting lines of code. Our figures do not include toolkit code such as the implementation of the various glyphs and compositors, since that code is not specific to the editor.

Table 3 gives a code breakdown of the editor's implementation by component. Not surprisingly, the bulk of the code is contained in the text-related components. Table 4 gives an alternate breakdown by

Component	Lines
Text	800
TextView	1800
Graphics	700
GraphicsView	200
Table	400
TableView	500
Other	300
total	4700

Table 3: Lines of code by component

Function	Lines
Reading and writing	1150
Data representation	850
Views	2500
Other	200
Total	4700

Table 4: Lines of code by function

function, which shows that a significant proportion of the code is devoted to reading and writing the editor's external data files. The 1800 lines in the TextView component are about evenly divided between formatting the text, handling user input, and supporting floating figures and plowing.

6.2 Execution Speed

We measured the editor's performance by instrumenting the code and timing critical operations. In particular, we timed the drawing and formatting operations, since these are crucial for acceptable performance. We measured user cpu time for 10 repetitions of each operation, using unoptimized code on a DECstation 3100. Then we divided the results by 10 to get a single-repetition average.

Table 5 lists the single-page drawing times for several different pages of this paper, each containing different amounts of text. The page labeled "small" corresponds to a page covered mostly by non-textual figures. We used a page that had about 20 percent of its area covered by text. A "large" page is almost entirely text; we used a page with about 90 percent text. For the "average" page, we chose a page with about 60 percent text.

Page type	Draw	Format
small	0.10	0.13
average	0.28	0.28
large	0.38	0.37

Table 5: Seconds to draw and format one page

We did not measure the drawing times for graphics in the text. Intuitively, the text-drawing times will dominate because there are usually many more character objects on a page than graphical elements.

Table 5 also lists the format times for the same pages. We measured the time taken to create and format the Boxes and Characters that represent a visible page. Since the editor creates a fully-formatted glyph structure for only the currently-visible page, the time needed to build the whole document structure is only slightly longer.

The times in table 5 represent worst-case reformat times. Most editing operations will reformat only a small part of the page (often just a single line); only when a line spill causes a change in column breaks will significant reformatting occur.

The combination of draw and format times is a good indication of the longest delay that a user sees when using the editor. For example, adding a blank line near the beginning of a page might cause the whole page to be reformatted and redrawn, which usually results in about a half-second delay.

6.3 Memory usage

Table 6 lists the total memory that the editor dynamically allocated for documents of three different sizes. The ''small'' document is the first half of this paper (5 pages of text and about 15000 characters). The ''medium'' document is the whole of this paper (10 pages and about 30000 characters). The ''large'' document is two copies of this paper pasted end-to-end (20 pages and about 60000 characters). Table 7 shows the breakdown of memory usage in the editor.

The largest single component is used by the hierarchy of Boxes, Glue, and Characters that display the current page of the document. Most of this memory is used by the LRBoxes that display each line—boxes store geometry allocations for each of their components. This stored information makes redraws efficient, and it enables the editor to efficiently determine positions in the document corresponding to mouse coordinates.

The editor stores less information for non-visible pages. However, it must store enough information so that it can correctly determine column breaks. We use a TeXCompositor for column composition, which ensures that column breaks are always calculated to minimize demerits for the whole document. To do this calculation, the compositor must know the sizes of all the lines in the document. Our strategy is simply to replace the LRBoxes that represent lines on non-visible pages with equivalent-sized Glue objects. This strategy implies that the editor must re-create the LRBoxes when lines become visible.

Document	Memory usage
small	1500
medium	2300
large	4000

Table 6: Total memory usage (Kbytes)

Component	Memory Usage
Visible page	500K
Non-visible page	25K
Sub-item	9K
Character	15
Overhead	200K

Table 7: Memory usage breakdown (bytes)

The editor represents the structure of the document by building a hierarchy of embedded text editors. Sub-items such as section titles, enumeration items, table cells, and bibliography entries appear as simple glyphs to their surrounding context. This strategy allows us to build documents of arbitrary complexity without complicating the editor's implementation, but it makes the editor's memory usage dependent on the document's complexity.

Finally, the editor allocates memory to store the actual text of the document. Our current implementation uses three buffers: the first buffer stores the character codes, the second stores character formats, and the third stores pointers to the (shared) Character glyphs. Currently, we use simple hole-in-the-middle buffers so that the editor can efficiently insert and delete characters. This strategy works well for small to medium documents, but it might waste memory for large documents because of fragmentation in the memory allocator.

7 Conclusion

Glyphs are flyweight objects that greatly simplify the construction of user interface applications. The reduction in implementation complexity outweighs any extra processing overhead that results from using large numbers of objects.

Although we have focussed here on the use of glyphs in text applications, glyphs are also useful for other kinds of data. We envision using glyphs for anything from spreadsheet cells to video clips. In the future, applications will routinely include video, audio, and animations in addition to text and graphics. The simplicity offered by lightweight objects such as glyphs will provide substantial leverage for building these applications.

References

[1] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. Journal of Object-Oriented Programming, 1(3):26–49, August/September 1988.

[2] Mark A. Linton, John M. Vlissides, and Paul R. Calder. Composing user interfaces with InterViews. Computer, 22(2):8–22, February 1989.

[3] Andrew J. Palay et al. The Andrew Toolkit: An overview. In Proceedings of the 1988 Winter USENIX Technical Conference, pages 9–21, Dallas, Texas, February 1988.

[4] Ralph R. Swick and Terry Weissman. X Toolkit Widgets—C Language Interface. Digital Equipment Corporation, March 1988. Part of the documentation provided with the X Window System.

[5] Pedro A. Szekely and Brad A. Myers. A user interface toolkit based on graphical objects and constraints. In ACM OOPSLA '88 Conference Proceedings, pages 36–45, San Diego, CA, September 1988.

[6] Andre Weinand, Erich Gamma, and Rudolf Marty. ET++—An object-oriented application framework in C++. In ACM OOPSLA '88 Conference Proceedings, pages 46–57, San Diego, CA, September 1988.

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