The Object–Oriented Implementation of a Document Editor

Paul Calder
Flinders University

Mark Linton
Silicon Graphics

Abstract

Traditional document editors are large and complex. Using first-class objects to represent individual characters in a document, we have implemented an editor that is much smaller and simpler than editors of comparable power. This editor, named “Doc”, uses object sharing to reduce memory usage and an incremental update strategy to minimize screen redraw time.

Our measurements show the current Doc implementation uses 10 bytes for each character in a document and 30 bytes for each visible character. On a 10-MIPS workstation, Doc can draw over 10 full pages per second and can keep pace with interactive typing speeds. These measurements and our experience using Doc shows that a fully object-oriented approach to document editing is practical. We used the editor to prepare and publish this paper.

1 Introduction

WYSIWYG document editors support features such as text formatting, figure placement, page layout, and cross-referencing. The implementation of a document editor can be an expensive task, and the resulting software is often difficult to maintain and extend. Even a seemingly simple extension to a text editor can be unwieldy. For example, adding multiple fonts to the Motif text widget[6] or changing the formatting algorithm of the ATK text view[7] would require a substantial revision of existing code. More sophisticated document editors, such as FrameMaker, are much larger and therefore potentially more difficult to extend.

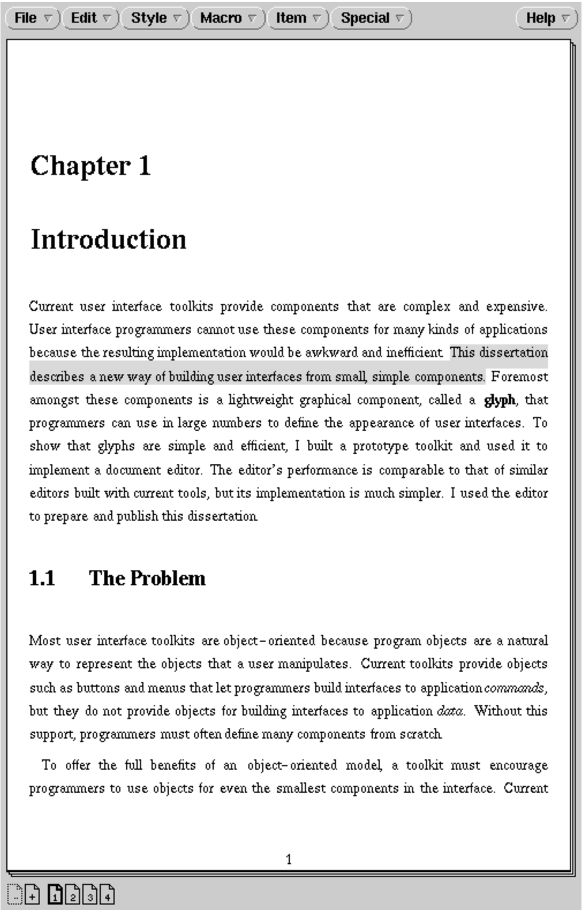
We have implemented a document editor, called “Doc”, that uses an object–oriented design all the way down to the level of individual characters. This approach leads to an implementation that is much smaller and simpler than comparable applications. In this paper, we describe the functionality of Doc and the classes that comprise the Doc implementation, and we present performance measurements of Doc running on a range of document sizes.

2 Editor functionality

The functionality of Doc is based on the LaTeX document preparation system[4]. The editor currently supports character styling, macro expansion, floating figures, automatic numbering and cross-referencing, embedded graphics, and simple tables. Doc does not currently support undo, an outline mode, or automatic generation of a table of contents or an index. Figure 1 shows a screen dump of Doc in use.

Character style primitives include font, color, and point size. Formatting parameters include margin widths, page shapes, and inter-line and inter-paragraph spacing. Style primitives are collected into user-defined character styles; changing the definition of a style will change the appearance of text that uses the style throughout the document.

Recurring document features such as section



Chapter 1

Introduction

Current user interface toolkits provide components that are complex and expensive. User interface programmers cannot use these components for many kinds of applications because the resulting implementation would be awkward and inefficient. This dissertation describes a new way of building user interfaces from small, simple components. Foremost amongst these components is a lightweight graphical component, called a **glyph**, that programmers can use in large numbers to define the appearance of user interfaces. To show that glyphs are simple and efficient, I built a prototype toolkit and used it to implement a document editor. The editor's performance is comparable to that of similar editors built with current tools, but its implementation is much simpler. I used the editor to prepare and publish this dissertation.

1.1 The Problem

Most user interface toolkits are object-oriented because program objects are a natural way to represent the objects that a user manipulates. 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

1

Figure 1: Doc screen dump

headings, figure captions, labels, and citations are specified by macros. These macros can contain fixed fragments of text or graphics, counters that generate section or figure numbering, or text parameters that customize the expansion. For example, a macro for chapter headings might contain the fixed text "Chapter", a counter for the chapter number, and a text parameter for the chapter name. Like styles, macros are user-defined; changing the macro definition changes the appearance of all its expansions in the document.

Document floats represent material that is associated with a specific point in the document but not embedded within the text flow. For example, figures or tables that accompany a document are often presented in a float if they are so large as to interrupt the smooth reading of the text. The document editor implements floats as separate text flows; they can contain anything that can appear in the main flow. The user can place the float at the desired position interactively while the editor dynamically flows the surrounding text around the float.

Doc can import an image from a file and embed it in the document as if the image were a single character. Similarly, Doc can read a file containing a structured graphics description and represent the graphics hierarchy as a single character. Typically, images or graphics are placed in a floating figure.

Doc provides a rudimentary table editor in which each table cell is a separate text flow. The width and height of the table will change as text is edited within its cells. Currently, the table editor can only insert or remove whole rows and columns in the table; it does not support advanced formatting features such as spanning cells. The table feature does, however, provide a simple demonstration of non-hierarchical layout. For layout purposes, a table cell is contained in both a row and a column; Doc uses a "dual hierarchy" layout to represent this structure.

3 Implementation overview

Doc is written in C++ using the InterViews class library[1,5]. The basic InterViews building block is a *glyph*, which is an object that defines a geometric shape. Glyphs are responsible for drawing an appearance in a given area, though they may be partly or wholly transparent. Because hit detection is intimately tied to rendering, glyphs also define picking. A glyph is hit if the glyph's appearance would intersect the pick coordinates. Glyph subclasses provide operations for input handling and focus management.

InterViews provides a mechanism for creating glyphs that directly support the TeX composition model of boxes and glue[2,3]. In particular, an *hbox* glyph is an aggregate that lays out its components left-to-right, stretching or shrinking the components as necessary to fit the available space. A *glue* glyph is transparent space of a given natural size, stretchability, and shrinkability in one dimension. A *discretionary* glyph defines a formatting penalty for a composite object that "breaks" at the glyph, and it defines the appearance of the break. For example, inter-word spacing is a reasonable place to place a line break, so a glyph that represents the space will have a lower penalty than a glyphs that represents a printable character. When the space glyph is selected as a break point, its appearance has zero-width.

Glyphs are intended to be as cheap as possible. The minimum storage for a glyph is eight bytes. Glyphs also can be shared; that is, a glyph instance may be referenced by more than one aggregate. The glyph methods pass enough context for an object to draw itself correctly wherever it is used.

The InterViews glyph design is described in more detail elsewhere[1]. Our focus here is on how we use glyphs in the implementation of Doc, and the effect of this approach on performance.

4 Document items

Like most editors, Doc uses a model-view separation: the application data is structured as a hierarchy of items; the view is a separate hierarchy of item views. An item can have more than one view; when it changes, the item updates all its views.

An abstract base class, *Item*, defines the interface to the data classes, and an abstract base class, *ItemView*, defines the interface to the view classes. Each type of document item is implemented as a subclass of Item; each Item subclass has a corresponding ItemView subclass. Table 1 lists the classes defined in the current implementation.

Item	View	Function
TextItem	TextView	text flow
TabularItem	TabularView	simple table
PSFigItem	PSFigView	graphic
FloatItem	FloatView	float
LabelItem	LabelView	reference target
RefItem	RefView	cross-reference
CounterItem	CounterView	counter
PagenumberItem	PagenumberView	page number

Table 1: Item subclasses and their views

Figure 2 shows a partial listing of the Item C++ class definition. The read and write methods define the item's persistent storage format. In addition to saving the item on disk, these methods are used for cut-and-paste and other operations that involve item copying. The label method supports the editor's referencing scheme. The method implements a pre-order traversal of the item (and any embedded items); the parameter defines the referencing context for the traversal. The change method tells the item that an embedded item has been modified. Most items notify their views about the change, then propagate the change to their parent. The view method creates a new view of the item. The notify method tells the item to update its views. Separating the notify method from the operations that change items allows us to make several changes to an item before propagating the effects to the views. This batching improves performance for many common editing operations.

Figure 3 shows the ItemView class definition. Most view operations are initiated through the command method. Views interpret the command (which typically comes from a menu or a keystroke binding) in a view-specific way. Embedded views pass commands that they do not understand to their parent views. The repair method initiates an update operation by calling notify on the view's subject; the subject, in turn, will call the update method of all its views.

```
class Item {
public:
    Item(
        Document*, Item* parent, int style, int source
    );

    virtual void read(istream&);
    virtual void write(ostream&);
    virtual void label(const char*);
    virtual void change(Item*);
    virtual Glyph* view(
        ItemView* parent, DocumentViewer*
    );
    virtual void notify();
};
```

Figure 2: Item class interface

```
class ItemView :
  public MonoGlyph, public Handler
{
public:
  ItemView(DocumentViewer*, ItemView*);

  virtual boolean command(const char*);
  virtual void repair();
  virtual void update();
};
```

Figure 3: ItemView class interface

4.1 TextItem

A TextItem, which represents a text flow, is a list of embedded items. Associated with each item is a character code, which defines the role that the item plays in the text, a style code, which specifies its graphical attributes, and a source code, which tells where the item originated.

An item's character code identifies the type of data that the item represents. For instance, an item might represent the letter "a" or an end-of-paragraph marker. Most items have character codes that correspond to character encodings in a standard encoding vector (the editor uses the Adobe standard font encoding for graphical characters). Other codes identify non-graphic items such as discretionary hyphenation points, spacers and fillers, and anchor points for floating items. Views use item codes for operations that relate to the content of the text. For example, searching and cursor motion operations are based on item codes, as are higher-level functions like automatic hyphenation and pairwise letter kerning.

An item's style is an index into a document-specific table of style properties. Style attributes include text point size, font face, color, and formatting metrics. Views use style information for operations that relate to the appearance of the text. For example, changing the point size of a text selection involves changing the style codes of each item in the selection.

An item's source code tells how the item was generated. Most items are part of the document body (they constitute the text that the author typed), but others arise as a result of macro expansion or file inclusion. Source codes are primarily used to correctly update the document following a change in a macro or style definition and when saving the document to a file. Views also use source codes to control their behavior. For example, the editor will prevent the user from editing text that resulted from a macro expansion.

```
class TextItem : public Item {
public:
  TextItem(
    Document*, Item* parent, int style, int source,
    const char* parameters, int size_hint = 0
  );

  virtual int item_count();
  virtual Item* item(int index);
  virtual int item_code(int index);
  virtual int item_style(int index);
  virtual int item_source(int index);
  virtual int insert(
    int index, int code, int style,
    int source, Item*
  );
  virtual int remove(int index, int count);
  virtual void replace(
    int index, int count, int style
  );
};
```

Figure 4: TextItem class interface

Finally, each item in a text flow can contain a pointer to another item. The embedded item can be another text flow, or it can be of any other item type. However, most items represent simple characters.

Figure 4 shows the new methods that the TextItem class defines. The item_count method reports the number of items in the flow; items are referenced by their index into the flow. The item, item_code, item_style, and item_source methods return the parameters of the specified item. The insert method adds an item at the specified position; the remove method deletes items. Finally, the replace method changes the style codes of a range of items.

4.2 Referencing and Numbering Items

The editor uses three types of items to implement its numbering and referencing capabilities: a CounterItem specifies the location within the document where a

Item	No break	Pre break	Post break	Penalty
letter ‘a’	Character(‘a’)			
hfil	HGlue(fil)			
negthinspace	HGlue(-1 pt)			
nobreak space	HGlue(1 en)			
par break		HGlue(fil)	Strut()	good
visible hyphen	Character(’-’)	Character(’-’)	Strut()	hyphen penalty
discretionary hyphen	Strut()	Character(’-’)	Strut()	hyphen penalty
word space	HGlue(1 en)	Strut()	Strut()	0
sentence space	HGlue(1.5 en)	Strut()	Strut()	0

Table 2: Selected character item glyphs

section or figure count should be incremented, a LabelItem provides a named cross-reference point, and a RefItem specifies where a cross reference should appear in the text. These items redefine the label method to manipulate a set of named counter values and a set of named label texts.

Generating cross reference information and section numbers for a large document is potentially expensive because it requires a complete traversal of the document. However, references change only when a counter, label, or reference item is inserted or deleted. The editor detects these rare events and initiates the reference traversal automatically.

5 Document views

The DocumentViewer class represents the top-level window for editing a document, providing menus to issue commands, buttons to change the current page, and dialog boxes for entering data and specifying properties. DocumentViewer also maintains the views of the document’s text flows and coordinates operations that transcend the view hierarchy. For example, the viewer controls reading, writing, and printing of the document, and it controls the distribution of keyboard and mouse events to the views.

DocumentViewer maintains a view of the document body and views of the document floats. The sizes and positions of document floats determine the size and shape of the space into which the document body is formatted. When a float changes in size or position, the document is reformatted to fit the new space.

5.1 TextView

The most important view class, TextView, is a view of a TextItem. Doc uses two types of TextView. The simpler version, MinipageView, can format text into a single rectangular block of lines. The more complex version, PagingView, formats text into multiple columns and pages of various sizes and shapes. The document editor uses a PagingView for the body of the document and MinipageViews for each secondary text flow and float. The two types of TextView are similar in most respects; they differ primarily in the way they use compositions to build the view structure.

Most components in text are simple character items; for these items, TextView builds glyphs that model the item’s role in the formatted text. For example, for printable characters, the view builds character glyphs, for inter-word spaces it builds discretionaries that contain glue, and for paragraph breaks it builds discretionaries that force formatting breaks.

Table 2 lists the glyphs that the view builds for selected character items. The type of glyphs that the view builds determines its formatting behavior. The first four entries in the table are simple glyphs; they will never be selected as formatting break points. The remaining items are discretionaries. If a discretionary is selected as a break point, it will contribute its pre-break appearance to the end of the line before the break and its post-break appearance to the line after the break; if it is not selected as a break point, it will contribute its no-break appearance to the current line.

A *strut* is a glyph with the same height as a character but zero width. The view uses struts to ensure that otherwise-empty lines will have the correct height.

The table also lists the formatting penalties incurred for each discretionary break. The constant *good* specifies a break that will always be taken. A penalty of 0 neither encourages nor discourages the break. A value of *hyphen penalty* reflects the aesthetic cost of hyphenation; a large value discourages hyphenation. The default hyphen penalty ensures that hyphenation will rarely occur unless the document has unusually narrow columns.

Character item glyphs are shared. The editor builds one glyph instance for each character item glyph in a particular style; views that need these items fetch the shared glyph from the editor.

5.2 MinipageView

Structurally, a MinipageView is a top-to-bottom tiling box containing left-to-right boxes. Instead of building the boxes directly, a MinipageView uses a *composition* object for this task. Compositions encapsulate much of the complexity of formatting and ensuring that changes to the items are correctly reflected in the view. To the MinipageView, a Composition is a single composite glyph; the view simply inserts its component glyphs into the Composition.

MinipageView uses an LRMarker glyph to highlight the currently selected text range. LRMarkers highlight the extent of the marked text by drawing a colored region either behind or in front of the text. Figure 5 shows the arrangement diagrammatically; the variables dot and mark record the character indices that mark the selection. The figure assumes that the workstation can display more than two colors; it draws a distinguished color behind the text. On single-plane displays, the editor uses an LRMarker that draws an XOR region in front of the text.

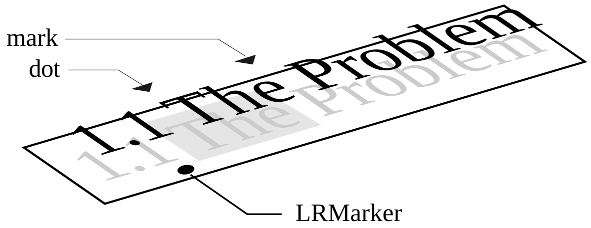


Figure 5: Highlighting with an LRMarker

5.3 PagingView

PagingView is similar to MinipageView. However, whereas MinipageView has a single level composition composing character-level glyphs into lines, Figure 6 shows how PagingView adds extra compositions that format lines into columns and columns into pages. The extra levels of composition do not complicate the code that builds the view since only the top-level (characters) composition is explicitly maintained by the view. The other levels assemble glyphs that were made by their enclosing composition.

Compositions store information that allows them to update their internal structure efficiently, and they try to confine their changes to a small part of the total structure. The effect of this behavior is to limit the amount of recalculation that occurs when incremental changes are made to the text. Consider, for example, the effect of adding a single character to a PagingView. When the view repairs its character composition, it can narrow its attention to the paragraph that contains the new item because line breaks in one paragraph cannot affect those in another. Furthermore, the new set of line breaks will probably be similar to the old set: most often, the change will be confined to a single line; the composition need build only a single new box for the modified line.

At the next composition level (the lines composition), the new glyph will replace the previous line box. But the new box will usually have the same size as the old box, since for running text from a single font, both the line length and the line height are constant. Compositions check for this common occurrence: if an incoming glyph has the same geometry requirement as the glyph it replaces, the composition will not propagate the change because such a change can have no effect on formatting.

The net effect is that most incremental changes to a document do not propagate beyond the character composition. Only when a line changes size (perhaps the user specified a larger font) or a new line is added will the effect be more far-reaching. The same

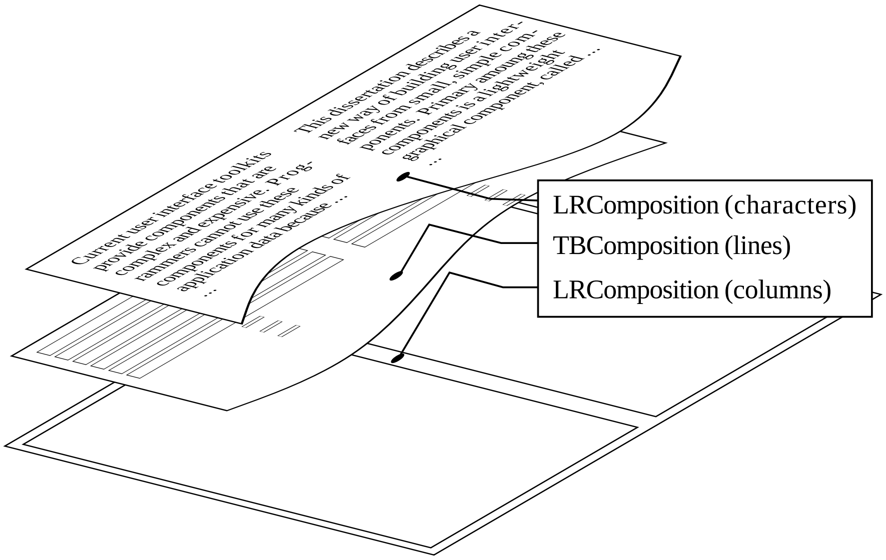


Figure 6: PagingView structure

confinement also operates at the level of the lines composition; many line-level changes are confined to a single column and thus do not affect the columns composition. These optimization allow the editor to keep up with rapid typing while maintaining full formatting for the entire document.

The second important way in which PagingView differs from MinipageView is in its support for shaped compositions. Compositions record a separate formatting size for each composite they create; views that use compositions can thus control the target sizes of the composites by individually setting the lengths of the pieces.

Document editors can use composition shaping to leave cutouts in the formatted text for secondary material such as floating figures. For example, Figure 7 shows how an editor could leave a semicircular-shaped hole in the right margin by individually shaping each line in the composition.

PagingView uses shaping at the lines composition level: it shapes the composition to leave room for floating figures at the top or bottom of columns. Before it repairs its lines composition in response to a text change (or when a float is moved), PagingView sets the top and bottom margins of each column in the composition, which then recalculates its structure to accommodate any changes in the sizes of its pieces.

6 Performance

The full-scale use of objects for pages, columns, lines, and characters greatly simplifies the Doc implementation. However, we were concerned that the associated runtime costs might make this approach impractical. To evaluate the costs of this approach, we measured three aspects of Doc's performance: code size, memory usage, and execution time. We examined the runtime costs for a range of document

Current user interface toolkits provide
components that are complex and
expensive. User interface program-
mers cannot use these compon-
ents for many kinds of applic-
ations because the resulting
implementation would be
awkward and inefficient.
This dissertation describes
a new way of building user
interfaces from small, simple
components. Foremost among
these components is a lightweight
graphical component, called a glyph,
that programmers can use in large
numbers to define the appearance of
user interfaces.

Figure 7: Shaped composition

sizes. For the execution time, we measured both the "batch" cost of redrawing a full document page and the interactive cost of processing a character insertion. All the measurements were gathered on a DECstation 3100 running Ultrix 3.1.

6.1 Code Size

Doc consists of about 15,000 lines of C++ source code spread over 29 classes. About half the classes (15) represent views, one-third (10) represent document items, and the remainder are general support classes. Over one third of the item code is devoted to reading and writing the data to a file. Other significant portions of the item code are for building and maintaining the document's structure and implementing the cross-referencing and automatic numbering.

Many of the view classes are trivial; the implementation of these classes is straightforward. Only DocumentViewer, TextView (together with its subclasses MinipageView and PagingView), PSFigView, and TabularView contain significant amounts of code. For these classes, a total of about 1200 lines were devoted to building and maintaining the views' structure, about 900 lines to handling and interpreting user input, and about 800 lines to reading figures from files and building graphical views.

6.2 Memory usage

We instrumented Doc to collect object instance counts and heap size information during execution. Table 3 lists the documents that we used as input. The documents are all technical papers, but included a single-page abstract, several medium-size papers or dissertation chapters, and a complete dissertation.

Document	Description	Words	Pages
intro	paper abstract	750	1
uist	conference paper	5000	10
ch1	thesis chapter	1000	5
ch6	thesis chapter	7100	30
thesis	complete thesis	27000	130

Table 3: Sample documents

6.2.1 Instance counts

Table 4 shows the count of each type of item created for a subset of the documents. As expected, most items were characters. To determine the effectiveness of the sharing of character glyphs, we recorded the item and style codes of each character item. From this data, we computed the number of glyphs created and the average number of uses of each character glyph, shown in Table 5.

The occasional use of different formatting styles and the natural range in character frequency for English text resulted in a wide range in the degree of sharing for individual glyphs. The most commonly-used glyph for all documents (accounting for almost 15% of all character items) was a discretionary representing an inter-word space. In the thesis document, for example, this glyph was used 24,440 times. In contrast, 57 glyphs were used only once. In total, the 20 most widely-used glyphs accounted for about 80% of the character items in each of the documents.

These results show that sharing greatly reduces the number of glyph instances created to display a large document. Even for very large documents, the total number of character glyphs should remain less than 1000. This characteristic is a consequence of the small number of styles used in most documents.

In addition to the glyphs that represent the data items in the document, views create composite glyphs that represent formatted units. Most of these composites are boxes that represent lines of text. To determine the number of instances of these glyphs, we instrumented the TeXCompositor object (which implements the TeX line-breaking algorithm) to report the size of paragraphs and the number of formatting breaks for each paragraph. Table 6 summarizes the results.

The "compositions" column lists the total number of times the compositor was called. Many of these calls corresponded to non-paragraph constructs: section headings, table cells, figure captions, and fixed-format text such as verbatim code listings. In studying the compositor's behavior, these constructs are uninteresting because they do not exercise the compositor algorithm (they generate only one line per

Document	Character	Text	Tabular	Cell	Float	PSFig	Counter	Label	Ref
intro	5148	3	1	15	6	4	2	0	2
uist	33218	27	7	148	20	4	20	0	20
thesis	185833	94	35	866	74	14	156	92	289

Table 4: Document item counts

Document	Characters	Glyphs	Average uses
intro	5148	220	23
uist	33218	454	73
thesis	185833	486	382

Table 5: Glyph sharing for character items

Document	Compositions	Paragraphs	Par lines	Lines/par	Chars/line
intro	73	15	170	11.3	36
uist	459	77	660	8.6	51
thesis	1900	365	2152	5.9	78

Table 6: Composition information

composition). We excluded these calls from further study by filtering out all single-line compositions.

The “paragraphs” column in Table 6 lists the number of non-degenerate compositions for each document. For these paragraph compositions, we also computed the number of line breaks chosen and calculated the average number of lines per paragraph and characters per line. The short lines of the intro and uist documents reflect the page layout for these documents: intro was formatted with three columns per page and uist with two.

6.2.2 Heap size

We recorded the heap size of the Doc process at key stages of execution: after startup, after reading a document, and after creating one, two, and three views. The heap usage can be divided into two parts: fixed allocation associated with the entire editor, and variable allocation that depends on the number of characters in the document and the number of characters visible in a top-level view. We report here on these measurements for the ch6 document.

We found that about 650K of the memory was fixed overhead for toolkit objects, including menus, windows, and dialogs. For each character in the document, Doc allocated 10 bytes for the character item and 6 bytes for the view. For each visible character, it used an additional 30 bytes to cache position information. Finally, each formatted line in the view used 100 bytes, and each visible formatted line used an additional 300 bytes.

6.3 Execution Speed

We measured three aspects of the execution speed of Doc: *compositing*, *formatting*, and *drawing*. Table 7 shows the results of these measurements on the ch6 document.

All measurements were done on an otherwise unloaded DECStation 3100. Both the document editor and InterViews library were compiled without optimization. The measurements are user CPU time only; they do not include system call time or the time spend by the window system on the editor’s behalf.

6.3.1 Compositing

Compositing is the process of calculating formatting breaks. The compositing time is dominated by the cost of calculating line breaks, which are far more

Phase	Time	Pages/sec	Characters/sec
Compositing	2.01	15.0	20,000
Formatting	6.96	4.3	5,900
Drawing	2.62	11.4	17,500

Table 7: Execution timing for ch6

numerous than column and page breaks. For the ch6 document, the compositing algorithm found 1136 line breaks.

We measured compositing speed by forcing a full re-composition of the document, repeating the process 50 times, and taking the mean. Table 7 shows that the average composition time was 2.01 seconds for the 41,000 characters in the body of the document. This time corresponds to a composition rate of about 20,000 characters per second, or about 15 pages per second.

The largest single component of the optimal fit algorithm's running time comes from the possible_break method, which encapsulates the compositor's inner loop. This method, which accounts for about 31% of the total running time, is called once for each place where a formatting break can legally be inserted. For paragraph text, the method will usually be called at each word space; for other material it will be called more often. In the ch6 document, it was called 13,400 times, an average of about once for every three characters.

6.3.2 Formatting

The formatting phase consists of building the glyph hierarchy that represents the formatted document, and calculating the geometry allocations of the glyphs. We measured formatting speed by instrumenting the code to force a full rebuild and reallocation of the current page. Then we paged through the document, measuring the formatting time for each page. We totaled the page format times to find the format time for the entire document, then repeated this entire process 10 times. The mean formatting time was 6.96 seconds, which corresponds to an average format rate of 5900 characters per second or about 4.3 pages per second.

6.3.3 Drawing

We measured drawing speed by adding code to force a full redraw of the current page. As for formatting, we paged through the document and timed the redraw speed of each page individually and then summed the times for the redraw time of the entire document. Table 7 shows the draw time was 2.62 seconds for the document, corresponding to an average drawing rate of 17,500 character per second or about 11.4 pages per second.

In addition to the text, the document included several figures that contained complex graphics. To measure the effect of these figures on drawing speed, we collected the same measurements with all figures replaced by simple rectangles of the same size. The average draw time for the document fell to 2.27 seconds, showing that about 15% of the total draw time was due to figures.

6.4 Interactive Performance

From a user's perspective, the most important aspect of an editor's performance is its responsiveness during common interactive editing operations. These operations involve a combination of compositing, formatting, and redrawing. For example, typing or deleting a character causes Doc to re-compose the current paragraph, reformat any changed lines, and redraw the affected parts of the screen.

To measure interactive performance, we simulated the process of re-typing the thesis document. We started with an empty document and recomposed, reformatted, and redrew the document as each character was added. We recorded the user CPU time after each 1000 characters.

From these timings, we computed the average typing speed that Doc can maintain in characters per second. Figure 8 plots the average typing speed as the size of document grew during the simulation. The typing speed fell from around 25 characters per second for small documents to less then 10 characters per second for large documents. For moderate-sized documents (around 20 pages), the typing speed is about 15 characters per second or about 180 words per minute.

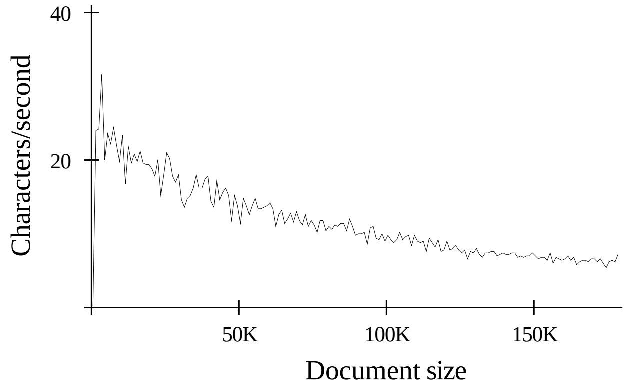


Figure 8: Simulated typing speed for thesis

The simulation results are conservative; Doc can keep pace with sustained typing speeds higher than those show by batching several keystrokes into a single repair operation. When the user types a key, Doc updates its internal data structures to record the event. Before repairing its views, Doc checks to see if another key event is pending. Only if no such event is available will the editor undertake the potentially expensive view repair operation. The net effect of this strategy is that, even for rapid bursts of typing, Doc updates its views without noticeable delay.

On the other hand, our simulation did not measure the variation in response time. Users are often more disturbed by variations in response time than in a slightly slower or faster average response time. Our experience is that the operations that are potentially slow, such as adding a character that causes a change in page breaks, are also the most visually distracting (the current page changes significantly). When fine-tuning the final draft of a document, a user wants to see the update regardless of how long it takes. For rough drafts, however, the user may wish to defer the repair operation until specifically requested. We may experiment with such a feature in future.

7 Conclusions

The heavy use of objects in the implementation of Doc has yielded a simple yet powerful document editor. The three key results of this work are (1) that using shareable objects for individual characters reduces the number of object instances to manageable proportions, (2) that general-purpose composition objects can be used to build multi-layered compositions for lines, columns, and pages, and (3) that the performance of this approach is practical on current systems.

The general-purpose nature of the composition class is a direct consequence of the representation of characters as glyphs–a composition simply manages a collection of glyphs that could be characters, lines, columns, or pages. Compositions can also be used for other kinds of layout, such as arranging buttons in dialog box or pretty-printing source code.

Although we have been careful to avoid performing unnecessary computation for text outside the currently visible page, we have spent little time otherwise tuning the Doc implementation. There are numerous opportunities for further optimization, including reducing memory usage though better encoding or more computation, and reducing redraw time by optimizing the inner loops. Given that the level of CPU performance of a DECstation 3100 is exceeded by most newer systems, our approach is clearly practical today and should become increasingly attractive in the future as a way to reduce the complexity of document processing software.

References

[1] P. Calder and M. Linton. Glyphs: Flyweight objects for user interfaces. *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, Snowbird, Utah, October 1990, pages 92-101.

[2] D. Knuth. *The TeX Book*. Addison-Wesley, Reading, Massachusetts,. 1984.

[3] D. Knuth and M. Plass. Breaking paragraphs into lines. *Technical Report STAN–CS–80–828*. Stanford University, November 1980.

[4] L. Lamport. *LaTeX User's Guide and Reference Manual*. Addison-Wesley, Reading, Massachusetts,. 1986.

[5] M. Linton. Implementing Resolution Independence on top of the X Window System. *Proceedings of the 6th X Conference*, Boston, Massachusetts, January 1992.

[6] Open Software Foundation. *OSF/Motif Programmer's Reference, Revision 1.0*. Open Software Foundation, 11 Cambridge Center, Cambridge, Massachusetts. 1989.

[7] A. Palay et al. The Andrew Toolkit: An overview. *Proceedings of the 1988 Winter USENIX Technical Conference*, Dallas, Texas, February 1988, pages 9-21.