Writing a Simple Word Processor

The elements of design for an interactive text editor with word-wrapping

Chuck Groom
December, 2002

© Copyright Blue Mug, Inc. 2002
www.bluemug.com
Contents

1 Introduction 2
  1.1 Examples ................................................. 2
    1.1.1 PicoGUI ......................................... 2
    1.1.2 Typesoft’s Vera .................................. 2
  1.2 About Blue Mug, Inc. .................................. 2

2 Goals and Features 2

3 Wrapping 3
  3.1 Newline and Carriage Return Characters .................. 5
  3.2 Tab .................................................... 5
  3.3 Efficient Wrapping ..................................... 6
  3.4 Example: PicoGUI TextEdit Widget ....................... 6

4 Data Structures 6
  4.1 Plain Text Storage ...................................... 7
    4.1.1 Primitives ........................................ 7
    4.1.2 High-level Text Storage ........................... 8
  4.2 Text Description Meta-data .............................. 9
  4.3 Example: PicoGUI TextEdit Widget ....................... 9
    4.3.1 Internal Fragmentation ............................ 10

5 Cursor Issues 10

6 Drawing 12
  6.1 Text .................................................. 12
    6.1.1 Optimization Hints ................................ 13
    6.1.2 Clear to End of Line ............................... 13
    6.1.3 Example: PicoGUI TextEdit Widget Hints .......... 13
    6.1.4 Example: Vera Wraps While Drawing ................ 13
  6.2 Virtual Drawing Pane .................................. 13
  6.3 Backing Store .......................................... 14
  6.4 Cursor ................................................ 14
  6.5 Summary of View Port and Backing Store ................ 15
  6.6 Scrolling .............................................. 15

7 Selection 15
  7.1 Scrolling .............................................. 16
    7.1.1 Click Below Text ................................ 16
  7.2 Formatting ............................................. 16
  7.3 Drawing Selections .................................... 17

8 Cut, Copy, and Paste 17

9 Focus 19
  9.1 Text Widgets .......................................... 19

10 Bibliography 19
1 Introduction

Without being too flip, we can say that it’s easy to write a text editor except for all the hard bits. An editor design which appears straightforward on the drawing board becomes complicated in practice as a myriad of subtle exceptions and corner cases crop up.

This document shares our experience writing text widgets and simple word processors. We will address text data storage, formatting, wrapping, and rendering for simple, general-purpose text editors with word processor-style line wrapping. This discussion applies to both proportional-width and character cell fonts.

We will not discuss undo/redo, a worthy topic in its own right but far too involved for this document. We only briefly outline the issues involved in cut, copy, paste and the clipboard.

We will evolve this document to meet reader expectations. If there is a topic you would like to see discussed in more detail, or if you have a question which was not answered, please send an email to text-feedback@bluemug.com.

1.1 Examples

We will illustrate specific designs choices with our work on the PicoGUI TextEdit widget and Vera textpad device.

1.1.1 PicoGUI

PicoGUI (http://pgui.sourceforge.net/) is small, fast, and full-featured client-server windowing and GUI environment which runs on a wide range of platforms. It is ideal for embedded graphical devices. Blue Mug contributed the TextEdit widget, a fast multi-line text object suited to uses ranging from displaying a few lines of text to word processing.

1.1.2 Typesoft’s Vera

Vera is a low-cost full-sized keyboard with text display. Blue Mug was contracted to write a full text editor, memory manager, display driver, USB driver, and keyboard driver that fit into an 8K ROM, using an 8-bit microcontroller. See http://www.typesoft.com/ for more details.

1.2 About Blue Mug, Inc.

Blue Mug, Inc. is a software engineering company that specializes in designing and implementing system software for mobile consumer devices. For more information about Blue Mug, our background and our services, please visit http://www.bluemug.com/ or email info@bluemug.com.

2 Goals and Features

A text editor enables a user to complete a writing task. This may range from jotting notes to writing long documents with complex formatting to writing a computer program. If you have knowledge about what specific tasks your users are trying to accomplish, you can incorporate this knowledge into your editor. For example, a text editor specifically designed for programming would expect many short lines whereas a text editor for prose would anticipate documents comprised of paragraphs, each an average of maybe 50 words. Anticipated usage patterns should influence how data is stored and presented.

All text editors share a few essential design goals:
• No keyboard action should be slow enough to bother the user; insert, in particular, must appear to be be instant.

• Speed and memory usage should scale with document size and complexity as, at most, $O(n \log n)$.

As mentioned in the introduction, this document will focus on writing simple, general-purpose text editors with line-wrapping.

3 Wrapping

Text wrapping operates on strings which end with newlines; each such string is a paragraph. If a string is wider than the window, we will display it across several screen lines such that each screen line is broken at a natural position for a reader – usually the space between two words or a hyphen.

For example, suppose we were using a 40 column monospace display. We would wrap the first two paragraphs of this section as:

```
Text wrapping operates on strings which end with newlines; each such string is a paragraph. If a string is wider than the window, we will display it across several screen lines such that each screen line is broken at a natural position for a reader -- usually the space between two words or a hyphen.

For example, suppose we were using a 40 column monospace display. We would wrap the first two paragraphs of this section as:
```

Wrapped text properties include:

• The first character of a paragraph starts at a known $x$ position, the paragraph indent.

• Trailing whitespace characters are allowed to dangle off the right edge of the display. For example, in the above paragraph there is a space after “between.”

If there is a long word (a sequence of characters without natural breaks, like whitespace or a hyphen) wider than the screen, we have two wrapping options:

1. Start the long word right after the last displayed word.

2. Start the long word on a new display line.

```
Option one for dealing with a_very_long_word_which_is_hard_to_wrap.

Option two for dealing with a_very_long_word_which_is_hard_to_wrap.
```

The former allows tighter text packing, the latter improves legibility. Most text editors choose the latter option of starting very long words on their own line. Either is fine.

This suggests a simple wrapping algorithm. To wrap a paragraph, start at the head of the paragraph and set $x=$paragraph indent. Then enter a loop:
Get the next word width, $w$, not including trailing space

If $w + x$ is less than the screen width

– The word fits on the current screen line

Otherwise,

– Advance to the next screen line, set $x=0$

Increment $x$ by $w + \text{width of whitespace}$ after the word.

Of course, you will need to add an exception for words wider than the screen. But you get the idea.

This algorithm is not very good for interactive wrapping. You don’t want to scan the rest of the paragraph after each keystroke. We will discuss better strategies in a little while.

Before getting into that, it’s useful to delineate when to rewrap. We should rewrap a line that grows too long to fit on the screen or shrinks to be short enough to accept part of the next line. This happens when:

• **Insert breaking character (space, tab, hyphen)**
  – A breaking character could push a word off the right edge of the line.
  – A breaking character inserted in the middle of the first word in a screen line could make the first word segment short enough to fit at the end of the previous line (Figure 1).

• **Insert non-breaking character**
  – A non-breaking character expands the current word, possibly pushing the last word in the line onto the next line.

• **Delete non-breaking character**
  – A delete in the middle of the line could reduce the size of the screen line such that the first word of the next line could be moved to the end of the current line.
  – A delete in the first word of a line could make that word small enough to fit at the end of the previous line (Figure 2).

• **Delete breaking character**
Figure 2: User deletes in middle of first word of line, making it short enough to fit at the end of the previous line.

- Deleting a breaking character in the middle of a line could reduce the size of the screen line such that the first word of the next line could be moved to the end of the current line.
- Deleting the break between two words could form a word wider than the screen width.
- Deleting at the start of a line could join the last word of the previous line with the first word of the current line, moving the last word of the previous line to the current line and increasing the width of the current line. This in turn could push several words at the end of the current line onto the next line.

- **Format change**
  - Format changes increasing character size may push words onto the next line. Format changes decreasing character size may require wrapping from the previous line to fit the smaller words at the end of the line.

3.1 Newline and Carriage Return Characters

In the above discussion, we used the term “newline” as a generic term for a character sequence which is used to mark the end of a line of text. Each paragraph is a single line of text which the editor wraps so it will be displayed on screen. Non-wrapping editors would allow such lines to trail off the right edge of the screen.

Unix uses the line feed character (\n, hex 0x0A) as its text line terminator. Microsoft Windows uses the carriage return character (\r, hex 0x0D) followed by a line feed to mark the end of a line. Versions of MacOS prior to MacOS X, which is based on Unix, use the carriage return character.

Why are there both line feed and carriage return characters? In the days of yore, there was a machine called the Teletype Model 33. It printed a stream of characters to a roll of paper at a rate of 10 characters per second. Moving the print head to the left side could take up to 2/10s of a second, the time it took to receive two characters. If a character arrived while the print head was in motion, it wouldn’t be printed. The solution was to separate the commands for moving the paper up one line (line feed) and moving the print head to the left edge (carriage return). Starting a new line required two characters, thus taking up the requisite 2/10s of a second.

3.2 Tab

A tab character may be interpreted as either a fixed width of whitespace, usually a multiple of the font’s average character width, or a tab stop.
Tab stops are invisible horizontal positions, generally regularly spaced, such that a tab takes up the entire width up to the next tab stop. This means that a tab character’s width depends on its horizontal position. If a text editor supports tab stops, any display line containing a tab will need to be resized and possibly re-wrapped whenever the text before the tab changes.

3.3 Efficient Wrapping

When the user inserts or deletes text, we only want to rewrap the affected region.

The majority of inserts and deletes will affect only the current line. When a line’s wrapping changes, other wrapping changes will usually cascade through to the bottom of the paragraph. A change may ripple part-way down a paragraph and happen to restore a previous wrapping pattern, but this is rare.

This means that to wrap efficiently, you only need to store information about line widths from the first line to the current line. You will also want to know the width of the last word in the previous line and the first word in the next line.

- When the cursor moves backwards through a paragraph, its wrapping context is known.
- When the cursor moves forwards through a paragraph, the wrapping information can be determined by wrapping forward from the last line known to be accurate.
- If wrapping changes, recalculate wrapping from the change through to the end of the paragraph.

You may ask, why store wrapping information from the start of the paragraph? Why not just store wrapping information for the previous, current, and next line? If the cursor moves up, just wrap backwards from the previously known line. It turns out that this is impossible; you can only wrap forward from a known starting line. Wrapping backwards with a greedy algorithm will not yield the same result as wrapping forward, and a non-greedy algorithm would require information that can only be determined by working forward from a known starting point (Figure 3).

3.4 Example: PicoGUI TextEdit Widget

The PicoGUI TextEdit widget stores paragraph objects which contain atoms. Atoms describe bits of text. Atoms do not span screen lines. Each atom has a bitfield of flags. If the LEFT flag is set, the atom’s left edge is on the left side of a screen line; the converse applies for the RIGHT flag. These flags give us enough information to wrap and render quickly.

When text is inserted or deleted, we check to see if the wrapping changed; if so, atoms and flags are recalculated from the change through the rest of the paragraph.

4 Data Structures

Text editors store both plain text and meta-data describing the text (usually formatting and wrapping hints). Depending on the editor design, there may be little overlap between the data structures for plain text storage and meta-data, so it may be easier to use two parallel data structures instead of a unified structure. For example, plain text may be stored in a linked list and formatting information may be stored in a tree. For the purposes of this document, we will discuss the two as separate structures.
Figure 3: Wrapping text working backwards from a known screen line may not yield the same result as wrapping forward from the start of the paragraph. If the text “You may ask, why store wrapping information from the start of the paragraph? Why not just store wrapping information for the previous, current, and next line?,” is known to start the 3rd screen line with “information for...”, wrapping backwards gives a first line ending with “from” whereas forward wrapping gives a first line ending with “the.”

4.1 Plain Text Storage

Plain text should be stored such that:

- Insert and delete are fast
- There are minimal memory (re)allocations
- Memory is not wasted

To meet these goals, it is necessary to choose useful primitives to manage chunks of plain text and organize these primitives into an appropriate higher-level structure.

4.1.1 Primitives

Text could be stored in an array that is precisely the size of the text data. No memory is wasted. This is not feasible, however, because each insert and delete would require allocating a new array, copying the contents of old array, and freeing the old array.

One way to avoid reallocating-and-copying on every insert and delete is to maintain a gap of unused characters at the end of each string array. On delete, shift characters after the cursor left; on insert, shift characters right. When the gap is filled up or becomes too long, reallocate the array.

This technique will, of course, internally fragment memory. But the real problem is that each insert and delete requires shifting memory. This could be slow if the array is large.

If text were stored in many small sized arrays, e.g. one per word, insert and delete would be fast but the memory overhead of having gaps at the end of so many sized arrays may be unacceptable.

Another way to avoid the overhead of shifting characters on insert and delete is to store a contiguous gap of unused characters in the middle of the array. The first part of the string is
in the array before the gap, the second part of the string extends from the end of the gap to
the end of the array. This is called a buffer gapped array.

It is trivial to insert and delete when the gap is positioned at the cursor position. To insert
a character, copy the character into the start of the gap and decrease the gap size counter. To
delete a character, increase the gap size counter and move the gap start by a character.

For example, consider a buffer-gapped array storing the string “This is a buffer-gap
string”, with the cursor and gap positioned after the ‘t’.

The user presses delete. The gap grows to the left by one character. Although the buffer still
contains ‘t’, it is in the gap and will be ignored.

The user types ‘p’. ‘p’ is copied to the head of the gap, which shrinks to the right by one
character.

Moving the gap requires shifting parts of the buffer. As the buffer size increases, these
shifts can take longer.

Buffer gapped arrays are a good solution for text editing because users usually focus their
activities in a few areas rather than dispersing tiny edits throughout a large document. The
cost of shifting text around on the first edit is repaid by the speed of subsequent edits to that
region.

To preserve programmer sanity, it is a good idea to write accessor functions which translate
between the buffer gap’s physical layout of a string and the logical string position. In the above
example, array element 27 is in the gap whereas the 27th character of the string is ‘r’. Other
parts of the editor should not have to know that strings are stored in a buffer gapped array.

4.1.2 High-level Text Storage

While we have voiced our preference for storing plain text primitives in a buffer gapped array,
we have yet to address how to incorporate text primitives into a higher-level design for text
storage.

It is tempting to use a single buffer gapped string for the entire document. This would
not scale for large files, however. Growing or shrinking the array, or moving the gap a long
distance, would result in an unacceptable delay.

It is equally tempting to store a document as a tree organized by language syntax, i.e. a
document contains paragraphs containing sentences containing words. A string object would
be used for each leaf, e.g. each word. The advantages are that edits are fast and it is simple
to move text. On the downside, storing many buffer-gapped leaves causes excessive internal
fragmentation and wastes memory. A tree infrastructure is not amenable to certain text
operations, such as deleting a range of text.

Text could be stored in a linked list of fixed-size buffer gapped arrays (blocks). When the
system is idle, text is shifted between blocks such that no block is too full or too empty. This
design minimizes the number of memory allocations and allows developers to target an optimal level of internal fragmentation.

While text lookup in linked list of fixed-size blocks does not scale as well as tree-structured lookups, it is still suitable for most realistic scenarios. For example, if blocks are 4KB, the optimized gap size is 10%, and the user opens a 1Mb file, the worst-case access will have to scan through a linked list of about 285 blocks, which is not prohibitively slow on most systems.

The downside of a linked-list of fixed-size buffer gapped arrays is that it requires significant infrastructure to provide abstractions between logical strings and the block layout. For example, the action “insert a string 354 chars long at an offset 10531 chars into document” could require allocating new blocks, inserting the blocks into a linked list, shifting text from other blocks into the new blocks, and copying data into the new blocks.

4.2 Text Description Meta-data

In addition to storing the plain text, an editor should store a description of this text to define formatting and enable efficient wrapping. We will refer to this text description information as the meta-data.

The meta-data structure should connect to the plain text data structure in a way that does not require undue maintenance. For example, inserting a character should not force us to update every formatting ‘paragraph’ object’s text offset.

It usually suffices for each meta-data object to track its character length and cache the actual offset into the physical plaintext structure. For example, when the user inserts into a word, a meta-data structure may update the length of the word, parent sentence, parent paragraph, and document. To calculate the physical text offset for a meta-data object, work from a known offset and add or subtract the string lengths of the in-between objects.

What should the meta-data primitive be? Characters are the only object assured to be atomic with respect to wrapping and formatting. We may have to wrap a word at an arbitrary position if it is wider than the screen. Format changes may apply to half a word. It would be impractical from a memory standpoint to use a tree which included a node for each character, however.

Our suggestion is to use a document-paragraph division for both formatting and wrapping. Within a paragraph, wrapping is based on a list of visible lines and formatting may be based on nested ranges of characters. For example, characters 23-99 in a paragraph are bold and characters 12-16 within the bold region are also italic. Depending on the editor, a meta-data structure could combine the wrapping and formatting trees or store them separately for each paragraph.

4.3 Example: PicoGUI TextEdit Widget

The PicoGUI TextEdit widget offers a specific example of a text editor drawing from the above ideas for plain text storage and description.

TextEdit text data is stored in a linked list of buffer gapped arrays. Each array is has a fixed size of 4KB. Some syntactic text characteristics map to the contents of each buffer gapped array:

- The widget has a linked list of blocks.
- A block has a buffer-gapped array.
- A block may contain multiple paragraphs.
• Paragraphs may contain multiple atoms.

The object properties are:

• Paragraphs cannot span blocks.
• Atoms cannot span screen lines.
• Every character within an atom shares the same format.

The PicoGUI TextEdit widget does not support a formatting hierarchy; it is possible to change the attributes of a region, but attributes cannot be cascaded.

The fact that paragraphs cannot span blocks enormously simplifies text manipulation abstractions such as reading or deleting a range of text because few text operations span paragraphs, and those that do are easy to special-case.

The linked list of blocks, each with a list of paragraphs, is a 2-layer description of the text in a way that isn’t far divorced from the physical contents.

If a paragraph is very long, the block will grow to contain the paragraph. The PicoGUI text widget does not scale for very long paragraphs. We feel this is acceptable because the widget is intended for generic prose and few paragraphs will ever be this long. Assuming words are on average 5 characters long, a 4Kb block could hold a paragraph that is about 800 words. Such a paragraph would be 2-3 pages long when printed out, which is rare outside of Faulkner and Russian novels.

The widget periodically scans for too-full or too-empty blocks and creates blocks, deletes blocks, and moves content accordingly.

4.3.1 Internal Fragmentation

The PicoGUI TextEdit solution of using a list of fixed-size buffer gapped arrays which contain entire paragraphs does not waste memory under normal usage.

Intuitively, it makes sense that choosing a block several times larger than the anticipated normal paragraph size will not waste more memory than one would want for a gap. A small block may only hold one or two paragraphs with half a paragraph’s space left over, which is wasteful, but half a paragraph’s space is appropriate for a block containing ten or twenty paragraphs. However, we were uncertain whether this reasoning applied to more realistic cases of packing variable-length paragraphs into blocks.

We used a simple analysis model to determine if our solution would result in excess internal fragmentation under normal use, and if not, what fixed-size buffer to use. We simulated packing 1,000 paragraphs whose lengths were a normal distribution (median 400 chars) into fixed-sized blocks of different size (Figure 4). We concluded that a 4KB block offers a good trade-off between a reasonably small block size (so shifting text won’t be slow) and efficient memory use (85-90% efficiency, or 10-15% internal fragmentation). This level of internal fragmentation is a good tradeoff between not wasting memory and allowing the user to enter a reasonable amount of text before needing to allocate memory.

5 Cursor Issues

Cursors tend to be inverted regions either the size of a character or the size of a narrow (1-2 pixel) bar. Editors with narrow bar cursors tend to blink the cursor every 200-400ms when idle, but keep the cursor solid while the user is typing.
Figure 4: Block size vs. percent of block memory used by text data if paragraphs are packed into blocks such that paragraphs do not span blocks (mean paragraph length is 400 characters, standard deviation 200 characters; blocks maintain at least 256 character gap). The height above the red line represents internal fragmentation.

Cursors are displayed to the right of the last entered character, at the position where the next character will go. When the cursor is at the end of a wrapped line ending in a space, most text editors display the cursor at the start of the next line (Figure 5).

How should an editor display a cursor that’s at the end of a word which comes exactly to the edge of the line width? The word fits, but the cursor won’t fit after the word. Options include:

1. Make sure the editor display area is wider than the line width by at least a cursor-width of space.
2. Assuming a narrow cursor, move the cursor to the rightmost edge of the screen. This obscures the right side of the last character, but if the cursor blinks the user could still read the text.
3. Draw the cursor at the start of the next line.
4. Leave the cursor at the end of the line.

Most editors choose option 1. I don’t know of any editor that uses option 2.

If space is too limited to devote a column to a cursor gutter, we are limited to options 3 and 4. Both lead to strange behaviors, however, because different cursor positions in the text might map to the same cursor position on screen. In option 3, the cursor may be at the end of the line if the cursor operates on the last character or the one before it. In option 4, the cursor may be at the start of the line if it operates on the last character of the previous line or the one before it, if the last character is a space.
6 Drawing

Text should be drawn to the screen as quickly as possible without any flicker. Drawing happens when:

- The text editor is created, or
- The editor region has been scrolled, or
- The text has changed

The typical sequence for dealing with text changes is:

1. Modify internal structures (plain text storage, the text description)
2. Check wrapping
3. Determine the cursor position and scroll as necessary so the cursor will be visible
4. Draw changes

Most text editors manage their own drawing. They directly draw into a rectangular area using system drawing primitives. Most editors eschew the overhead of using higher-level abstractions like the “label” widgets provided by most GUIs.

6.1 Text

It is in the drawing stage where you reap the benefits of having a good data storage models to tell you what the text is, what it looks like, and how it is wrapped. If you can easily generate a list of text chunks such that each chunk has the same formatting and is part of a single screen line, then it is trivial to draw from left to right, top to bottom.

To draw a chunk of text, first draw the background color over the text region to clear old text, then draw the new text. An exception to this is the case of where a line’s leading (height) is less than that character height. Undrawing just the leading height would cause leave the old bits of characters poking outside the leading region undrawn. Undrawing the full character height will erase part of the previous or next line.

The solution is to first do all the undrawing, then to do all the text drawing. Lines with leading less than the character height should be undrawn using the full character height, and the redraw step should include any lines immediately above or below such a region.
6.1.1 Optimization Hints

We strongly recommend adding drawing optimization hints wherever possible. If we can know that a particular line but no other lines need to be redrawn, we should only draw that line. If a character was inserted which did not affect the wrapping of the line, we may simply move the rest of the line right one character’s width (by blitting to a dedicated pixmap, blitting back to source pixmap) and draw just the one character.

6.1.2 Clear to End of Line

One tricky aspect of drawing text to screen is knowing which text to undraw. Old text may extend beyond a visible line when:

- A line becomes shorter
- The document is shorter than the editor window size and a user action makes the text move up several lines. Previously drawn text extends beyond the bottom of the document area.

We suggest setting a flag on lines which have been newly wrapped and not drawn. When we encounter such lines in the drawing stage, we draw the background past the right side of the line to the edge of the screen, effectively undrawing old text.

Similarly, when there has been a delete affecting the total document height, set a flag so that in the draw stage the background will be drawn from the end of the text region to the bottom of the visible region.

6.1.3 Example: PicoGUI TextEdit Widget Hints

The PicoGUI TextEdit widget sets drawing hints on its atoms (which describe text) when it creates or alters them. These hints tell the drawing step to ignore the atom, to draw the entire atom, to draw the last character of the atom, and/or to undraw the region to the right of the atom.

6.1.4 Example: Vera Wraps While Drawing

The Vera text editor needed to have an extremely small footprint, so we checked for wrapping in the drawing stage. As long as the rendering/wrapping starts at a position known to be accurate with respect to wrapping, the two steps can be merged together to save code. This comes at the expense of code modularity and legibility.

6.2 Virtual Drawing Pane

Documents could be many screens in height when rendered. An editor presents a small view into a much larger virtual drawing the size of the entire document.

It would be impractical to maintain a physical canvas in memory the size of the entire drawing area. The backing bitmap could easily become too large for memory, and any sizing changes in the middle of a document would require a redraw of the rest of the document, even though this region would be off-screen.

Therefore, text editors draw to a virtual canvas the size of the entire document. Actual drawing is only done in the region of the virtual document within the view port. The view port translates virtual coordinates to real coordinates. Rendering starts at the virtual top of
the view port and finishes at the bottom of the view port. To scroll, change the virtual top of the view port and redraw.

Most editors align the first visible line to the top of the screen such that it is completely visible. If the editor viewable area is sized such that the last visible line would be partially obscured, most editors will not display that line.

6.3 Backing Store

Text rendering usually involves drawing a rectangle with the background fill followed by drawing a text string. If drawing operations were updated directly to the screen in this order, there would be a slight flicker. It is especially noticeable for a text editor because there are often a lot of draw operations which can slow the system down enough for the flicker to be visible, and the user’s eye is trained on the region which is being redrawn.

To avoid flicker, most editors draw to an off-screen pixmap which is blitted to the screen. Such editors are said to use backing store. This bitmap could be the size of a single line or the entire editor window. One reason to use a buffer the size of the editor window is that it is trivial to expose events generated when a window occluding the editor window moves. On the other hand, a buffer the size of a single line uses less memory.

It is possible to use a backing bitmap a few (2-10) lines taller than the editor window to permit fast scrolling. When the user scrolls up or down by a few lines, the editor can instantly blit the corresponding portion of the bitmap that was off-screen, then redraw to the bitmap.

6.4 Cursor

Some cursors are drawn as an XORed rectangle. To undraw the cursor, XOR the cursor rectangle again. Using an XORed cursor allows editors without backing store to avoid re-rendering the text when the cursor moves or blinks.

If the editor uses backing store or temporarily copies the cursor region to an off-screen bitmap, the cursor can be drawn in any manner. After blitting the backing pixmap, draw the cursor directly to the screen. There may be flicker, but redraws coincide with cursor movement, so this is fine. To undraw the cursor, blit over the cursor region.
6.5 Summary of View Port and Backing Store

The text editor scrolls by setting the virtual top of the pane and redrawing itself. Scrolling happens when:

- The user manipulates the scrollbar
- Due to a user keyboard action, the cursor would be displayed off-screen. Scroll so the cursor is visible.

If a user is paging through a document quickly, e.g. pressing “page up” or dragging the scroll thumb, it may be a waste of time to render a screen fully only to re-render it at another position. It would be better to render only part of the screen, then check to see if the screen needs to be re-rendered at a different position.

7 Selection

Users select a region of text by clicking and dragging over a text region or by holding down the shift key while moving the cursor. Some editors extend selection techniques to include double-clicking on a word to select the entire word, triple-clicking to select the entire sentence, and snapping selection to entire words.

Selected text may be:

- Replaced if the user inserts or pastes text
- Deleted if the user deletes or cuts
- Copied
- Moved if the user drags the selected region to a new position
• Modified if the user changes the text format

Cut, copy, and paste will be addressed later.
Most text editors internally treat a selection as the region between two marks in the metadata. The cursor is one of these marks.

7.1 Scrolling

Selections may extend beyond the top or bottom of the visible region. The editor should scroll so the growing edge of the selection, marked by the cursor, will be visible.

When the user selects text using the keyboard shift and cursor movement keys, scrolling is already managed by the rule that the editor scrolls such that the cursor is always visible after a keyboard event.

The user may select text by clicking and dragging the mouse from within the editor region to above or below the editor region. The text editor should scroll up or down at a constant rate, moving the selection marker as it goes along.

It can be tricky to choose the correct scroll rate and number of lines per scroll step. If you scroll too fast, the user may have to use lightning-fast reflexes gained from years of video gaming experience to stop at the right place. If you scroll too slow, users will rapidly become very annoyed. One prevalent technique is to set the scroll velocity as the distance between the top/bottom of the editor region and the mouse. This puts scroll speed under user control. If you use this technique, make sure that the difference between lightning-fast and molasses-slow is more than just a few pixels so those with poor mouse skills may effectively use your text editor.

The text editor should intercept all mouse events which started as a drag within the editor window. For example, buttons outside the text editor should not receive mouse-enter events if the user is in the middle of making a selection.

7.1.1 Click Below Text

If the editor text ends part way down the editor window, the editor should interpret clicks below the text region as setting the cursor to the end of the text region. Therefore, a mouse drags starting below the editor text should select from the end of the text.

7.2 Formatting

When the user selects a region of text which contains different kinds of formatting, how does the editor report the formatting of the selected region? If a selection includes an italic word in otherwise plain text, is the entire text region plain, italic, or neither?

What happens when the user changes the formatting of region containing text of different formats? Will all the text change to the same format, or is the format change added to the existing formats?

If the user deletes a selection spanning format types, are there cases where the format of the region before or after the selection should change?

These are difficult issues to resolve, and the answers depend on the specific editor design and formatting markup model. This document is too limited in scope to give this topic the attention it deserves. Suffice to say that an editor supporting multiple format types will benefit from a well fleshed-out design which addresses these and other formatting cases.
7.3 Drawing Selections

Drawing selected text is like drawing normal text, except that selected text should be drawn with a different background and, in some cases, a different foreground color to distinguish it from non-selected text.

Most editors display the selection background color in a rectangle the height of the line, not each particular character. Even if a line contains text using different font sizes, the selection rectangle will be the same height for all characters.

Text editors may display selected text in several different ways.

- How will a selected region be shown?
  1. Invert the selected region. This will be high contrast, but may not be visually appealing.
  2. Draw the text over a highlight color. This may look nice, but the highlight color must be high contrast with respect to both the text color and background color. If the editor supports colored text formats, the editor should probably alter the appearance of colored text which will be similar to the highlight color.

- Will the selection decoration only be drawn over the selected text itself?
  1. Only display selection over the selected text chunks. The selection will not be drawn over indented regions. Note that empty lines do not have an associated text chunk, so a special case should be introduced to indicate empty line selection.
  2. Display the first line’s selection from the start of the selection to right edge of the screen, and display the last line’s selection from the left edge of the screen to the end of the selection. Display the selection across the full editor width from the second to the second-to-last lines.

- How should selection be displayed for a line with leading (height) shorter than the height of a character in that line? This means that lines overlap.
  1. Use the line leading as the selection height. Parts of the selected line text may extend outside of the displayed selection.
  2. Use the font height as the selection height. Two selection regions may overlap on screen, or a selection may partially overlap a line which is not selected.

Figure 6 gives two examples which illustrate the implications of the above decisions.

8 Cut, Copy, and Paste

Cut, copy, and paste are simple operations for plain text and complicated for formatted text. As we mentioned above, this document is too limited in scope for us to discuss formatting issues. However, we can raise several more interesting questions that arise when an editor allows users to cut, copy, and paste formatted text.

- How will the clipboard store formatted text?
  - How will the system clipboard make data available to other applications, in particular those expecting data in a particular format?
  - Will the clipboard persist after the editor is destroyed?
Figure 6: Two possible text selection display techniques. Selection type (A) highlights the chunks of selected text, selection type (B) inverts the entire selected region, extended to the margins. Note that (1) an empty line selection should be displayed even if it has no associated bit of text, (2) if a line’s leading is shorter than the height of characters in that line some editors may allow parts of selected characters to be drawn outside of the selection, and (3) it is important to make sure that colored text will show up when selected; simple inversion will work, as will changing the text color as needed.
• If there are formatting characters immediately before or after a selection, will that markup information be included when the selection is copied?

• If the user pastes into a formatted region, will the format be applied to the clipboard text? For example, if bold plain text is pasted into an italicized region, will the result be bold italic text?

9 Focus

For a dedicated text editor with a touchpad or other pointer, focus is straightforward. The text editor region receives all key events. The user manipulates other portions of the user interface via pointer operations or meta-key escapes.

9.1 Text Widgets

Focus poses many challenges to text editor widget designers. Once the widget has keyboard focus, how does the widget relinquish focus without using a pointer device? Requiring a pointer device is both bad design (users shouldn’t have to take their hands away from the keyboard) and also limits the range of devices that can use the widget.

Most GUIs intercept ‘tab’ to move focus forward through the widget tree, and ‘shift+tab’ to move focus backwards. Editors process ‘tab’ keys, however. ‘Ctrl+tab’ is a standard keybinding for moving focus forward from multi-line text editors, and ‘shift+tab’ can move focus backwards, but it is unacceptable to assume that users will know this.

One solution is for a text widget to move focus backwards when a user arrows the cursor up past the start of the document, and move focus forward when the user arrows past the bottom of the document. The application should be able to set whether this is the desired behavior. The widget should emit a signal when the user attempts to move the cursor past the start or end of a document so explicit focus handling routines can be called.

10 Bibliography

There are few documents which detail the how-tos of building a text editor from the ground up. Instead, there is a great deal of scattered lore on the Internet and, of course, a myriad of text editors to learn from. The simplest text editors can be found in GUI text widgets; I would advise interested readers to study the widgets in open source toolkits such as Gtk+, QT, and of course PicoGUI.

Craig A. Finseth’s *The Craft of Text Editing*, ©1999, published on-line at http://www.finseth.com/~fin/craft/, was enormously helpful in writing this paper and offers a broad discussion of many facets of text editor design. It is based on a 1991 book which was in turn based on a mid-80s doctoral thesis, so it is a bit dated.

The history of the Teletype Model 33 is explained by http://www.oualline.com/eol.html.