May 15, 1986

Data Structures in the

Andrew Text Editor

Wilfred J. Hansen
Information Technology Center
Carnegie-Mellon University
4910 Forbes Avenue
Pittsburgh, PA 15213

With bit-mapped graphics systems, like IBM's RT PC, text can be much more than a stream of ASCII characters. There can be many variations as shown in Figure 1: differences in font, face code, size, justification, indentation, sub- or superscripts, and so on. In this article I want to describe how the Andrew system stores text and displays it on the screen. As a focus, I will look at the question of how a character gets to the screen after being typed. In older hardware this echoing was done by the display device itself, but in modern workstations considerable software is involved. My emphasis will be on the data structures.

Figure 1 about here

Before getting on, however, I need to say just a few words about workstations and Andrew. For this article let me define a workstation as a deskside computer with:

a hard disk--twenty or more megabytes,

at least two megabytes of real memory,

a virtual memory management system,

a speed of at least a million instructions per second,

a network connection,

a mouse,

and a mega-pixel bit-mapped display.

A bit-mapped display is implemented with one bit of memory for each pixel on the screen. To draw an image, bits in the memory are turned to one or zero to cause the screen to be black or white. To draw a character, a rectangle is copied from a font file which has a rectangular array of bits for each character. To have multiple font families, face codes, and sizes means having multiple font files, one for each combination. The Andrew system, for example, has had over a thousand different font files, though it has now settled on eighty-four.

Andrew is the name of the software produced by the Information Technology Center, a joint project of IBM and Carnegie-Mellon University. One aspect of the project is a distributed file system designed so that if you have an account you can sit down at any of five thousand work stations and get your files. You can also give anyone permission to read your files, without the inconvenience of making and transporting a floppy disk. This same communication network is the basis of Andrew's comprehensive electronic mail system.

The Andrew user interface software includes a window manager, a subroutine package for dealing with text, an editor and mail system that utilize the text package, and numerous other facilities. I find the large screen to be a significant advantage in my work. I often look at two or more source files at once to write a third or answer a mail message. In the screen image shown in Figure 2, I am writing this article and checking a source file for the definition of the data structure for documents.

Figure 2 about here

This article attempts to describe how the Andrew system stores and displays text. However, I have taken liberties with the names of routines and simplified many of the details so they are not the same as the actual implementation. Moreover, an improved Andrew formatting system is being built and it differs in many ways from what is described here. Much of the software described below was originally written by James Gosling who based it on his version of the Emacs editor.

Documents: How to Store Text

Documents are the heart of the Andrew text management system. A document is simply a stream of text. It may be as small as a single line; the bottom line of each of the windows in Figure 2 displays a document used to request parameters from the user and show messages. Other documents in that figure are the files being edited and the shell command area in the typescript. One application, the CMU-tutor lesson writng system, even uses a (non-displayed) document to store the results of compiling its lesson.

Figure 3 about here

At this point I could tell you how documents are implemented, but I prefer to let you puzzle over that while I describe how applications manipulate documents. Conceptually, the program refers to the document as a stream of characters, with the first numbered as

zero. See Figure 3. To have a document, a program first declares a variable to refer to it:
struct document ∗ doc;
This declares doc to be a pointer to a control block for a document; one element, length, is the number of characters in the document. The principle operations on documents are these four routines:

NewDocument(initiallength) - returns a pointer to a control block for a newly created document with capacity for **initiallength** characters. The document can get bigger than **initiallength**, so the exact value is not particularly important.

CharAt(doc, position) - returns the character presently at location **position** in document **doc**. Will return nonsense if **position** is negative or as large or larger than the number of characters in the document. If doc has the contents shown in Figure 3, CharAt(doc, 1) returns the value 'h'. (For performance, CharAt is implemented as a macro in C.)

InsertString(doc, position, string, length) - inserts **length** characters from **string** into document **doc**. The insertion is such that the first inserted character will wind up in location **position**. The call InsertString(doc, 9, "talking ", 8) will convert Figure 3 to discuss a `"talking raven"`.

DeleteChars(doc, position, length) - Deletes **length** characters from **doc**, beginning with the character at location **position**. The call DeleteChars(doc, 22, 8) would result in `"Why is a raven like a  desk?"`

With just these routines we can implement all of the operations on documents that are usually available in text editors. For example, when the user types a letter to be inserted in the document, the routine called is InsertCharacterCommand, as shown in Algorithm 1. Here and below the Algorithms are written in a subset of C similar to Pascal. Note that a declaration containing an asterisk before the identifier declares that variable to be a pointer to the indicated type. As a special case, if variable str is to refer a string of characters it is declared as "*char* ∗ str". Given a pointer one may either extract an element--as in doc->length to get the length of a document--or subscript it to get an element of an array of the given type--as in str[i] to get the $(i+1)^{st}$ character of str. (The first character is str[0].)

Algorithm 1 about here

For a more extensive example, consider the global replace operation. The user is prompted to provide an old string and a new one, then the editor replaces every instance of the old string in the text with the new one.

We need first the routine in Algorithm 2 to find an instance of a string in a document and return its location. Note that the outer while loop terminates when the length remaining in the document is shorter than the string str. The inner while loop terminates either when it finds the string, when i>=len, or when the i^{th} character of the string does not equal the $(pos+i)^{th}$ character of the document.

Given the routine find we can write Algorithm 3 to do the global replace. In practice it would be called by the command processor after it has requested the old and new strings from the user.

Algorithms 2 and 3 about here

How then is a document implemented? By brute force. The struct document control block points to a single array of characters large enough to store the text of the document. You can perhaps imagine two problems with this scheme. The first is that each insertion might entail moving the entire rest of the document for each character inserted. This is avoided by leaving a *gap* in the middle of the text array at the location of the most recent insertion or deletion. If I insert a character in the first paragraph and then move to the last paragraph and make an insertion there, the gap is moved by moving all intervening characters, filling the old gap and leaving a new one. After too many insertions the second problem is reached: the document text exceeds the array size. But some documents will never grow large. How then can the size of the text array be adjusted to accomodate both small stable documents and large growing documents? Again Andrew uses a brute force solution: When an insertion would make the text too large, a new array fifty percent larger is allocated and the existing text is copied to it.

The solutions to both problems potentially require copying large portions of a document, which might be thought to be disconcerting to the user. I can report, however, that after two years of experience with the system I have never noticed a delay for copying the text. After all, a typical document is less than a 100,000 characters and a typical copy loop has about six instructions. At one million instructions per second, the entire copy takes no more than 0.15 seconds, when moving four characters per cycle. Most documents are shorter, so the time to copy the text is insignificant; it takes longer than that to paint a full screen of text.

Instead of the data structure in Figure 4, some editors use an alternative with a linked list of control blocks, one for each line. It is undeniable that this structure can be much faster for the operations of insertion and deletion of characters; there is never a copy anywhere near as long as 0.15 seconds. But other delays are encountered, especially in a paging environement. Not only does the data structure take considerably more space--sometimes twice as much--but the control blocks and text lines can become scattered over numerous virtual memory pages. When that happens a single screen repaint may require touching twice as many pages as there are lines on the screen. If they cannot all fit in memory, lengthy paging delays occur. With the Andrew data structure paging is minimized.

Figure 4 about here

The complete Andrew document data structure is shown in Figure 4. With this structure InsertCharacter and DeleteChars are written in terms of two subroutines. GapTo(doc, position) moves the gap so it occurs just before the character at the given position. Then a deletion can be made simply by decreasing the size of the document and increasing the value which says where the text after the gap begins. (The initial part of the text after the gap is thus deleted.) For insertions the routine RoomFor(doc, size) is also used. It ensures that the gap is big enough for an insertion of size characters.

It is a bit of C arcanery, but here is the full declaration of CharAt:

```
/* CharAt(d,n) accesses character n of document d.
   d and n must be side-effect free. */
#define CharAt(d,n) ((n)<(d)–>part1len \
      ?  ((unsigned char  * ) (d)–>body)[n] \
      :  ((unsigned char  * ) (d)–>part2body)[n])
```

The test of n<d–>part1len determines if the desired character is before the gap. If so, the second line accesses it by subscripting directly into d–>body, the text area; but if not, the third line subscripts into an aritifical array d–>part2body, which begins d–>part1len characters before the first character after the gap. (You may want to think about this. Or then again, maybe you don't.)

Markers: How to Update the Display

Let us return to our focal question: How does a character appear on the screen when it is typed. As you check the code above, you will find nothing that updates the screen. This is done subliminally with the magic of markers. A marker is a data structure that refers to a portion of the text of a document; the portion starting at some character and extending for some length. Consider Figure 5 which shows three markers attached to our document of 35 characters.

Marker magic occurs because they are updated two ways by InsertString and DeleteChars. These routines adjust marker limits so they always refer to the same part of the text. If the string "`talking `" is inserted in Figure 5 at position 9, just before the `'r'`, the position value of marker C is incremented by eight, the length value for marker B is increased by eight, and marker A is unchanged. While adjusting limits a changed flag is set in a marker if the text it refers to is modified. For the insertion of "`talking `", the flag is set only for marker B. (The text referred to by C has moved, but not changed.) Once the flag has been set, it remains set until some routine outside the document package turns it off. Usually this is a routine associated with the routine that created the marker in the first place.

Figure 5 about here

From the discussion so far we can deduce the fields of a marker control block. Each *struct document* has a pointer to the list of markers associated with the document, and each marker has a pointer, doc, to its document. The extent of the text referred to is given by position and length. If the referenced text is changed the changed flag is set. And finally we have next and prev fields to connect the markers together in a doubly-linked list.

The routines to update markers are straightforward except for one decision that must be made: If an insertion is made at either end of the text referred to by a marker, is the length field of the marker made bigger? The Andrew answer is that the marker is made longer if the position of the insertion is a character that is referred to by the marker. Thus the length of a marker m will be increased only if m–>position <= InsertionPosition < m–>position+m–>length. Note that this rule will never extend the length of a marker that has length zero.

How then do markers aid in updating the display? The trick is that the display management portion of the editor keeps a marker for each line displayed on the screen. The line is redisplayed on the screen only if the text it refers to has changed. To make this possible, the editor is carefully partitioned between the routines which respond to user inputs and those which update the display.

At the highest level is a main loop which determines whether there is user input and processes it. The loop defers calling the screen update routine until no input is pending. This main loop utilizes a data structure for each portion of the screen. For a document the data structure representing its screen image is the view, a structure that keeps all the information needed to format the document for display. Among the fields of a view are these:

ViewTop - a marker whose position indicates the first character to be displayed on the top line of the image for this document.

Line - an array of LineImage data structures, one for for each line to be displayed. The LineImage for each line includes m, a marker for the text displayed on the line; y, the screen *y* coordinate of the top of the line; and height, the height of the line.

NumberOfScreenLines - how many lines are displayed for this view.

left, top, width, height - these describe the subrectangle of the window devoted to this text view. Left and top are the *x* and *y* coordinates of the upper left corner; width and height give the size of the window.

The coordinate system for the Andrew window manager begins with (0, 0) in the upper left corner and extends with increasing *x* to the right and increasing *y* downward. Distances are measured in *points*, the printer’s term for a unit about 1/72 of an inch. However, this is a pun because the implementation is really in terms of pixels on the screen; this works because many workstations have about 72 pixels to the inch. On work stations, like the RT, with more pixels to the inch the image will be smaller than would be found by measuring in true printer's points.

The marker for the i^{th} line is view–>Line[i]–>m. Curiously, it must refer to text beyond the end of the i^{th} line because insertion of a space in the first word of the $(i+1)^{st}$ line may require the i^{th} line to be redrawn with a short new word at its end. Thus the marker must include all the text on the line and also the first word of the next line.

Given these considerations we can now describe the update routine; it reconciles the screen image with the new contents of the document in three phases. The first determines which text lines have to be redrawn based on the changed flags in the Line array. In this process it calls a subroutine, DetermineSpacing which marches across the line interpreting the formatting information so it can find the height and width of each character and the widths for spaces to perform justification. All this information is preserved in the LineImage structure for the line and the value returned by DetermineSpacing is the position in the document of the first character for the next line. Note that an unchanged line may have to be redrawn if its *y* coordinate changes or if the previous line ends at a different position in the text. The second phase of the update routine erases the old text from each portion of the display that is to be redrawn. The third phase then plots each line that has been identified as needing to be redrawn. The heart of this phase is a call on SendTextToDisplay which uses the information recorded in the LineImage by DetermineSpacing and actually sends the characters to the screen,

The three phases of the view update process are sketched in Algorithm 4.

Algorithm 4 about here

To review, here is how a typed character gets to the screen: the interaction routine updates the document with InsertString, which sets the changed flag in a marker in one of the elements of the Line array in the view. The update routine finds that the marker for the line is changed, erases the existing line, and replots the line--now with the new character. It is part of the miracle of modern workstations that this all happens fast enough to keep up with any typist.

(Incidentally, if Lewis Carroll had been asked the Mad Hatter's riddle, I suspect he might have answered that the desk wrote "Nevermore".)

Wilfred J. Hansen is a System Designer at the Information Technology Center, Carnegie-Mellon University. His major work there has been with the Edittext text editor described here and its successor. His PhD thesis project, with Stanford University, was the first hierarchical syntax-driven editor and he has co-authored two texts with E. M. Reingold: Data Structures, *1981, and* Data Structures in Pascal, *1986, Little Brown and Co., Boston.*

In a given font family, like the seriffed font in this sentence, there are relatives in several sizes, both smaller and larger. For each of the sizes the family will have variants with **bold** and *italic* faces, and sometimes even ***bold-italic***. For displaying computer listings there will be a font family with `fixed width characters.` For a modern look there are other font families having no serifs.

This paragraph illustrates indentation and right justification. With a superscript$^{\text{here}}$ and a subscript$_{\text{here}}$. Continuation lines are justified to the right side, so the last line may have a large indentation on the left.

This is centered, bold, and italic, all at once.

Figure 1: *Varieties of text formatting. Note the variations in font, face type, size, justification, indentation, and subscripts. This caption has normal margins, the standard italic font, and is justified left and right.*

Figure 2: *An Andrew Screen Image. In this example I am writing this article and checking a source file. Top to bottom in the left column are a system status window, a shell window for giving commands, and an editor for the styles in this article; the latter is open to the style used for Algorithms. In the right column are the editor working on this document and another editor where I am checking source code.*

```
Why is a raven like a writing-desk?
```

character 34 (?)

character 22 (w)

character 9 (r)

character 0 (W)

Figure 3: *An Andrew document. The length of this document is 35 characters.*

body
40 size
27 length
22 part1len
13 gaplen
part2body

gaplen part1len

`Why is a raven like a       writing-desk?`

part1len gaplen

size

Figure 4: *The document data structure. The* body *field of the* struct document *points to a single array of characters containing the text, which is generally in two pieces with a gap between. This example shows the situation after deleting* "`writing-`" *from Figure 3. The address* part2body *is such that if* i>part1len *the* i*th* *character is at* part2body[i].

```
Why is a raven like a writing-desk?
```

C - position:15 length:14

B - position:7 length:12

A - position:0 length:6

Figure 5: *Three markers on a document. Marker* A *refers to the text "Why is",* B *refers to "a raven like", and* C *refers to "like a writing". Note that the text refered to by markers can overlap.*

Algorithm 1: *Code called when a character is typed.* Doc *is the document being edited and* CaretLocation *is the place currently selected for insertion.*

```
InsertCharacterCommand (c)
   char  c;
{
   char  s[1];         /* string to pass to InsertString */
   s[0] = c;           /* copy the character into s */
   InsertString(doc, CaretLocation, s, 1);    /* insert it */
}
```

Algorithm 2: *Finding a string in a document*

```
/* find - search document doc forward from location pos for
         string str.  Return the location of str or – 1 if it’s not there */
   int
find(doc, pos, str)
   struct document  ∗ doc;    /* document to search */
   int pos;                   /* where to start looking */
   char  ∗ str;               /* what to look for */
{
   int len, i;
   len = strlen(str);          /* compute length of string */
   while (pos+len <= doc–>length) {
      /* check to see if str is in document starting at pos */
      i = 0;
      while (i<len && str[i]==CharAt(doc, pos+i))
         /* the first i+1 characters of str match the
            characters in the document at positions
            pos, pos+1, pos+2, ..., pos+i */
         i = i + 1;
      if (i==len)              /* the entire string matches */
         return (pos);
      pos = pos + 1;           /* no match at pos, go on to next */
   }
   return (– 1);               /* no match at all, report failure */
}
```

Algorithm 3: *Global replace operation*

```
/* subst - Replace every occurence of string old
                in doc with string new */
subst (doc, old, new)
   struct document  * doc;      /* where to do the global replace */
   char  * old;                 /* the string to be replaced */
   char  * new;                 /* the string to replace it with */
{
   int pos, oldlen, newlen;     /* declare local variables */
   oldlen = strlen(old);        /* compute length of strings */
   newlen = strlen(new);
   pos = find(doc, old, 0);     /* find first instance of old */
   while (pos >= 0) {
      /* there is an instance, replace it */
      DeleteChars(doc, pos, oldlen);
      InsertString(doc, pos, new, newlen);
      pos = find(doc, old, pos+newlen);   /* find next instance */
   }
}
```

Algorithm 4: *Updating a view. The work of understanding the style information is hidden within* **DetermineSpacing** *and* **SendTextToDisplay**.

```
/* Phase 1: Find lines that need to change
        in this view due to changes in doc.*/
{
   int NextPosition;       /* position of start of next line */
   int y;      /* y coordinate of top of next line to display */
   int i;      /* i sequences through the lines displayed */
   NextPosition = view–>ViewTop–>position;
   y = 0;
   i = 0;
   while (y < view–>height  &&  NextPosition < doc–>length {
      /* decide which lines need to be redisplayed and
         choose space width for justification */
      struct LineImage  * ThisLine;          /* address of the ith line */
         /* ("ThisLine" is used for lack of Pascal's with statement) */
      ThisLine = &(view–>Line[i]);
      if (NextPosition != ThisLine–>m–>position
                        ||  y != ThisLine–>y) {
         ThisLine–>m–>changed = True;
         ThisLine–>m–>position = NextPosition;
         ThisLine–>y = y;
      }
      if (ThisLine–>m–>changed)
         NextPosition = DetermineSpacing(ThisLine);
      y = y + ThisLine–>height;
      i = i+1;
   }
   view–>NumberOfScreenLines = i;
}
/* Phase 2: Erase text that is to be redrawn */
{
   int i, j;
   For each group of consecutive changed lines {
      set i to the first in the group and j to the last;
      erase the rectangle that has an upper left corner of
            (view–>left, view–>Line[i]–>y)
         and a lower right of (view–>right,
             view–>Line[j]–>y+view–>Line[j]–>height– 1);
   }
}
/* Phase 3: Send new text to the display */
{
```

```
    int i;     /* cycle through the lines */
    i = 0;
    while (i < view–>NumberOfScreenLines) {
        /* now redisplay the changed lines */
        struct LineImage  * ThisLine;          /* address of ith line */
        ThisLine = &(view–>Line[i]);
        if (ThisLine–>m–>changed) {
            SendTextToDisplay(ThisLine);
            ThisLine–>m–>changed = False;
        }
        i = i+1;
    }
}
```