Some New Ingredients for the Cookbook Approach to Anti-Aliased Text

Avi Naiman

Computer Systems Research Group
Department of Computer Science
University of Toronto
Toronto, Ontario, M5S 1A4

ABSTRACT

Recently, much attention has been devoted to the creation and display of high-quality text on raster scanned display devices with gray scale capability. While the filtering and sampling methods needed to create the necessary character fonts have been outlined in great detail, the problems of font storage and proper character positioning have not been adequately dealt with. This paper presents a modified run-length encoding algorithm which takes advantage of the high coherence of character forms to achieve increased compaction rates. In addition, an inexpensive method for positioning the anti-aliased characters at subpixel resolution is described. Finally, *kerning*, a spacing technique which has been incorporated into the system, is discussed, along with some of the problems associated with trying to automate this process.

RESUME

Recemment, beaucoup d'attention à été porté sur la création et l'affichage de texte de haute qualité sur système d'affichage à quadrillage avec échelle d'intensitè. Les méthodes pour filtrer et échantilloner les jeux de charactères sont bien connues mais les problèmes de mise en mémoire des charactères et du choix adéquat de leur placement demandent encore de l'étude. Nous presentons ici une modification de l'algorithme d'encodage par longueur de série qui utilise la grande cohérence des formes des charactères pour augmenter le taux de compaction. Nous décrivons aussi une méthode peu coûteuse de placement de charactères anti-aliassés à une fraction de pixel près. Nous discutons le *crénage*, une technique pour espacer les charactères qui a été incorporée dans le système et des problèmes associés avec l'automatisation de ce procédé.

Key Words: anti-aliasing, binary-picture representation, character representation, computer graphics, data compaction, fonts, high-quality text, kerning, proportional spacing, raster scan typography, subpixel resolution.

Introduction

The motivation for the work presented in this paper comes from research into the problems of developing an interactive display system that allows users to design high-quality slides of text for talks and presentations. A diverse set of primitives is supplied, providing great flexibility in design styles. The user can specify the text to be displayed on the raster scanned display device, where to display it, and what ink color to use. Additional attributes include shadowing, underlining, blending, justification, and spacing constraints. After judging the aesthetic quality of the displayed text, the user can repeatedly modify any of the parameters (e.g., the spacing constraints) and redisplay the text, until the image is satisfactory.

To reduce the amount of computation necessary during the interactive session, character representations are precomputed as multiple-bit-per-element matrices, where a value in the matrix indicates the weighted area of the corresponding pixel which is covered by the character. While recent research has led to the development of several algorithms for computing filtered, multiple-bit-per-pixel low-resolution character fonts from single-bit-per-pixel high-resolution master fonts [CROW78, KAJI81, SCHM80, and WARN80], it is expensive, both in storage and compute time, to create many fonts in various styles and sizes.

One should further note that, although the efficiency and simplicity of matrices is very appealing, scalings and rotations are not easily performed on images in pixel form (but see [CROW78, CATM80, and WEIM80]). Even worse, as the size of a font changes, the width of the characters changes disproportionately with the height, serifs must be thickened or thinned, and ascenders and descenders must be shortened or lengthened relative to the x height of the font [BIGE83]. More information about the character's shape, available in a higher-level representation, is needed to scale properly; this information is lost when characters are represented as pixel arrays.

Alternatively, information about the outline of the characters can be retained — by approximating with anything from simple lines to cubic splines [FREE61, KIND76, KNUT79, PAVL83, and PLAS83] — in order to more easily perform 2-D and 3-D transformations. However, the complexity of encoding and scan converting (to a raster representation) characters which are represented as outlines, has caused implementations to utilize the simpler matrix representations of the fonts. In this light, we have concentrated on some of the difficulties encountered in these systems:

- to reduce storage costs, we have devised a variation of the run-length encoding algorithm which takes advantage of the spatial coherence of characters to minimize storage of the master fonts;
- we have devised a rendering algorithm which automatically calculates new bit-map representations of characters from existing ones to achieve sub-pixel positioning resolution;
- as traditional automated spacing techniques are not sufficient, since they cannot handle the unbalanced spacing introduced by characters of varying densities, we have implemented a method of *kerning* characters to achieve a more perceptually uniform spacing. However, since the aesthetics of spacing are not yet fully understood [KIND76], we have left ultimate control of the spacing constraints with the user, by allowing him to override the automated process.

Basic Recipe

Based on the work reported in [WARN80], character masters of a Helvetica font were created by digitizing 12-inch high Letraset characters at a resolution of 256 by 256 and thresholding to obtain a single-bit-per-pixel representation. Care was taken that all characters were aligned on a common baseline during digitization. However, since the text is proportionally spaced during the rendering phase, positioning of the characters in the horizontal axis was of little importance. An interactive paint program was then used to remove noise along the borders of the characters and more exactly position the characters on their baseline (see [SCHM80 and NEGR80] for more on the problems of obtaining high-quality masters). The compaction technique to be discussed below was then invoked, and the master font stored.

To display text, a two-phase process is invoked. In the first phase, those fonts which will be used are computed by specifying a master font, the size of the font to be computed, the type of weighting function (filter) and its parameters to apply (see

[WARN80]), and the phase of the sampling grid relative to the master characters to use. Once all of the necessary fonts are created, the second phase is invoked for interactive specification of the desired text and text attributes.

Font Compression

The total storage requirements for the 94 printable ASCII characters that we digitized would be 752 K bytes $((94 \times 256 \times 256 \times 1)/8)$ without any compaction. A first look at compaction techniques motivated us to applying a run-length encoding algorithm, whereby two-byte fields would indicate lengths of runs of pixels of either the foreground (i.e. the pixels that define the character) or background. However, since most of the runs were under 256, we restricted runs to being contained on rows of the character matrix, so that run-lengths could be encoded in 1-byte fields. The new compaction algorithm, then, ran a run-length encoding algorithm on each row of the matrix, so that 256 *row structures* were created and stored sequentially.

Run-Length Encoding

Traditional run-length encoders store values which alternately specify the lengths of foreground runs and background runs, with (a one bit) overhead of specifying what type of run the first one is. During decoding, then, a row is constructed by using the successive values to alternately lay down foreground and background pixels, stopping when the last pixel on the row has been specified. If there are n runs on a row, then it takes n bytes plus one bit to encode the row.

Alternatively, one can record the starting and ending locations of just one type of run, preceded by a count of how many runs of that type there are on the row. We will assume that, since there are 256 positions on a row, it will take one 8-bit byte to store the starting location of a run, and another to store the ending location (although fewer bits may be sufficient if assumptions can be made about the characteristics of the runs). Therefore, if there are m foreground runs on a row, we can encode the row in $2m+1$ bytes. Note that if there are only m-1 background runs, we could have encoded the row in $2m$-1 bytes. However, typically, the first foreground run does not start at the first position on a row and the last foreground run does not end at the last position on a row; so there are usually fewer foreground runs than background runs.

Alhough character forms are very coherent, the presence of noise in the image should not cause the compaction technique to require more storage than the original image would have needed if stored on a pixel-by-pixel basis. Furthermore, if *special* characters (arbitrary images) are to be allowed and dealt with, no assumption of coherence can be made. Since only 32 bytes are required to store the values of 256 bits (i.e. the raw data of a row), if there are more than 16 runs of foreground pixels on a row, it will take more bytes to *encode* the data than to store it in raw form. Therefore, only 4 bits are needed to specify the number of runs on a row, including an escape value to indicate that a row is being stored unencoded.

Replication Fields

In looking at certain characters, we noted that there was often a tendency of extreme row-to-row coherence. For example, the Helvetica character 'E' has a very regular shape and could be geometrically encoded as four rectangles of foreground pixels. Its simple, coherent structure causes many successive rows to be encoded identically. Therefore, a replication count field can be included prior to a row structure to indicate how many exactly identical, successive rows will be encoded by the following row structure. Since there can be, at most, 256 identical, successive rows, it takes one byte to store the replication count. In this manner, the 'E' described above would only take 17½ bytes to encode, regardless of its height, compared with $2\frac{1}{2}n$ for an n-pixel high 'E' compacted without a replication count field.

We must be careful about how much overhead is being introduced. For example, the character 'O' typically has no two successive rows which are exactly identical. If we record a replication count of 1 for each of the rows of the character, we have increased our overhead quite substantially (possibly by as much as 256 bytes). The solution is to incorporate a way to turn off inclusion of a replication count field. Since a replication count of zero is less than meaningless, we can use that value to indicate that the replication field will not exist for the number of rows specified by the following byte.

For the letter 'O', then, we would have a two-byte overhead in the encoding of the character: the first byte is a replication count of zero to turn off the replication mechanism, and the second byte is a row count indicating the height of the character. Note that turning off the replication mechanism for a single non-replicated row is more expensive than specifying a replication count of 1 for that row. In fact, there is no savings in turning off the replication mechanism unless there are at least three successive rows which are not replicated.

Row Order vs. Column Order

Some other observations that we made concerned the geometry of characters. For example, the character 'H', while quite amenable to the replication count approach, can be compacted with even greater efficiency if we change our point of view and, instead of encoding on a row-by-row basis, encode on a column-by-column basis. Though, in this case, the savings is minimal (due to the replication count scheme having achieved such a high compaction of the 'H'), it is clear from this example that one should look at compacting both ways, and then determine which is cheaper in storage and include an additional flag as to whether the encoding is row-by-row or column-by-column.

Even greater compaction rates can be achieved if we consider encoding the *rate* at which runs change from row to row, and exploit the symmetry of most characters around the $x=c$, $y=c$, $x=y+c$, or $x=c-y$ axes (where c is some constant). However, although decoding is often trivial, if the original character form has no higher-level representation by which such slopes and symmetries can be found, it can be quite difficult to glean such information merely from the bit-matrix representation (but see [PLAS83]). The intent here is not to find an extremely efficient way of representing a character form, but rather to improve on standard encoding schemes by exploiting the coherence of characters.

Simple run-length encoding reduced the storage requirements to about 57 K bytes. Simple run-length encoding using the minimum of the row-order and column-order compaction passes reduced the storage requirements to about 39 K bytes. Run-length encoding employing replication fields reduced the storage requirements to about 28 K bytes — over a 96% compaction rate compared with the unencoded 752 K of data.

Proportional Spacing

The aesthetic quality of the displayed text depends not only on the colored contours of the characters, but also on the positioning of the individual characters to achieve perceived uniform spacing. Keeping in mind the dictum that, as objects become too small to resolve spatially, size and intensity become interchangeable [CORN70], we can consider the spatial position of the edge of a character as integration of luminance over area. This implies that changing the intensity values along the sides of a character's matrix representation can effectively move the boundaries of the character a fraction of a pixel [SCHM80]. This can result in an improvement of the apparent spacing.

If character matrices can only be positioned on pixel boundaries of the display device, then the spacing can be off by as much as one half of a pixel. For example, if the right-hand column of the previously displayed character only covers one quarter of the pixel column in which it is displayed (i.e. it has intensity values of 25%), then that pixel column will be interpreted as one quarter of a pixel of character, and three-quarters of a pixel of space. If 2 pixels of spacing are required and the left-hand column of the current character being displayed also covers only one quarter of the pixel column in which it will be displayed, then a choice must be made between skipping a whole pixel of space before laying down the current character — causing a perceived spacing of 2 and a half pixels — or not skipping a pixel — causing a perceived spacing of 1 and a half pixels. Either way, the spacing is one half of a pixel off from the 2 pixels that were required. Worse yet, one pair of characters may have one half of a pixel too little, while the next pair has one half of a pixel too much (see figure 1 for an example where one pixel of spacing was required).

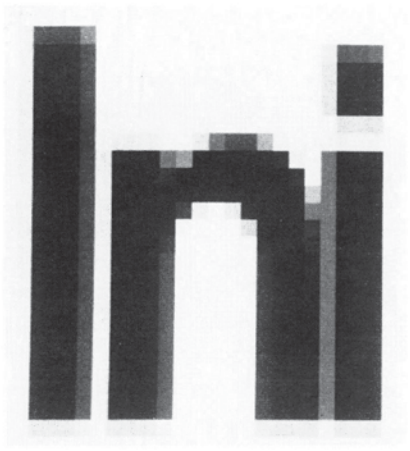


Figure 1. Unbalanced spacing due to roundoff.

Multiple Representations

One method for improving the perceived spacing is to produce multiple versions of each character font, differing from each other only by the phase of the sampling grid relative to the filtered master character [WARN80]. To increase the apparent accuracy of the spacing, the version of the characters which adheres most closely to the spacing constraints is used. In the example above, a search of the various representations of the current character being displayed would be made, and the one whose left column most closely covers three-quarters of the pixel column would be selected. Using this technique, to guarantee an accurate subpixel positioning within $1/n$ of a pixel, $n/2$ different representations of the character font would have to be computed, using $n/2$ equi-spaced phases of the sampling grid relative to the filtered masters.

A serious drawback of this technique is that many copies of each font must be computed and stored. Though the time it takes to retrieve the multiple versions and decide which is most useful may not be great, the time it takes to compute the many versions, and the storage requirements to keep them on-line, can be prohibitive.

Average Representation

One possibility for improving the spacing constancy is to compute only a single representation of the characters, but to guarantee that each character's left-hand column covers approximately half of the pixel column in which it will be displayed. In this manner, though, in the worst case, the spacing will still be off by a half of a pixel (e.g. when the previous character's right-hand column completely covers its pixel column), the *average* case will have improved to an error of only one quarter of a pixel. However, this is, in general, not sufficient (Figure 2).

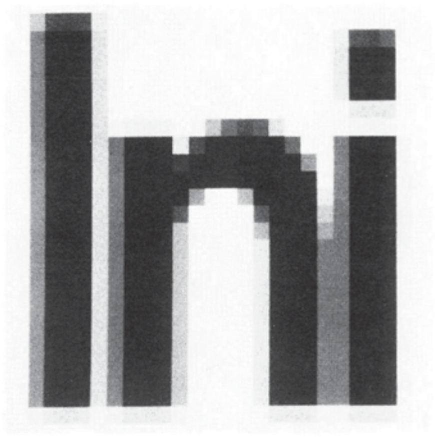


Figure 2. Improved spacing through average representation.

Subpixel Adjustments

This line of thinking has helped us to devise a method whereby a single representation of the characters can be computed, and adjusted right or left at subpixel resolution to reduce the spacing error. The adjustments correspond to taking a percentage of each column of the character's gray scale matrix representation, subtracting it from that column's values, and adding it to either the column to its left or right if the character is to be moved to the left or right, respectively. The percentage that needs to be used is calculated from the amount of space that is needed to complement the amount already displayed. In our first example from above, we would like the current character's left-hand column to contribute exactly one quarter of a pixel of space. If the character has been computed to cover one half of the pixel column it lies on, then it needs to be moved left one quarter of a pixel (i.e., that is the percentage used in computing an adjusted version of the character; see Figure 3).

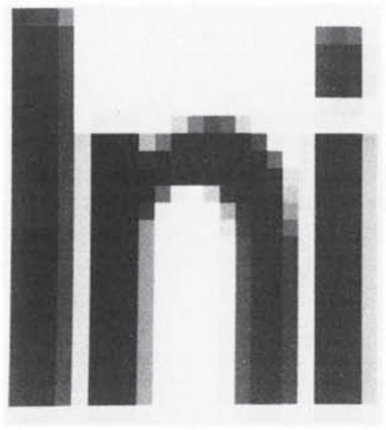


Figure 3. Balanced spacing through subpixel adjustments.

Care has to be taken to employ some heuristics along the sides of the character being adjusted. Since, at character-display time there is no longer enough information to determine how a particular intensity value was arrived at, one must, in general, assume that the intensity value of a particular matrix entry is evenly distributed over the pixel in which it is displayed. However, the same need not be true along the sides of the character. In the left column, we can assume that the intensity values imply coverages evenly distributed over the *right half* of the pixels they will lie on.

For example, instead of interpreting an intensity value of 25 percent in the left column to mean that the character covers one quarter of the pixel it lies on (evenly distributed over the whole pixel), interpret it to mean that the character covers one half of the right side of the pixel (evenly distributed over the right half of the pixel). We can, in fact, guarantee this when precomputing the character's representation, by choosing the particular phase which centers the filter function matrix over the left-most column of the character. In this manner, our heuristic becomes a fact along the left side of the character.

When adjusting left one quarter of a pixel, we know that the left column's values all remain in the left column after the adjustment, since the adjustment is less than one half pixel. So, whereas non-side columns will *lose* some of their values (that part which is computed to be on the left side of the pixel, and hence moved to the pixel to its left), we do not move any percentage of the values in the left-hand column out of that column. We simply *add* to them the appropriate percentage of the values from the column on its right.

In an analogous fashion, we assume that the values in the right-hand column are evenly distributed on the left side of the column. Even though this can't be guaranteed (and, in fact, is wrong when considering a horizontal serif), this is a reasonable heuristic as the right side of the character will, in general, be closely connected with the rest of the character. Therefore, we simply *subtract* from its value the amount of the adjustment, and add that to the column to its left. If anything remains in the right-hand column, then it is displayed; otherwise, the right-hand column is removed. Similar action is taken when adjusting the character to the right.

For example, if the character is moved one quarter of a pixel to the left, and the value in the right-hand column is 15%, then we add the 15% to whatever remains in the column to its left. If the value in the right-hand column is 35%, then we add 25% to the column to its left, and display the value of 10% in the right-hand column.

Since the assumption that the intensity values are uniformly distributed over the whole pixel is not accurate, this scheme does not remain strictly faithful to the character's original representation. However, we have found that, in practice, the characters do not noticeably degrade, while the spacing improves considerably — especially when the amount of space required between characters is relatively small. This is mainly due to the fact that the adjustments are, on average, on the order of one quarter of a pixel in either direction (and always less than one half of a pixel), while the character fonts being displayed are typically of display size (greater than 20 pixels high). At smaller font sizes (less than 10 pixels), though the spacing improves, the characters sometimes become severely distorted and, at times, unrecognizable, since the percentage of adjustment is large relative to the absolute width of the characters. This obviously affects thin, non-dense characters more severely than wide, dense ones.

Kerning

Of even more consequence in producing perceived uniform spacing are the gaps and holes introduced into the text when characters of widely varying densities lie side by side. An example of this problem occurs when an 'A' is followed by a 'Y', as in 'AYE'. Simplistic, automated spacing algorithms position the left edge of the 'Y' one unit of inter-character spacing to the right of the right edge of the 'A'. However, since the 'A' is sparse in the upper right-hand corner, and the 'Y' is sparse in the lower, left-hand corner, we perceive too much spacing between the characters (Figure 4a). The solution to this problem is to back the 'Y' into the 'A' until the spacing *appears* proportional; e.g., 'AYE'. This process, referred to as *kerning*, was once carried out by hand, when characters were mounted on rectangular pieces of metal of fixed width. The human typesetter had the opportunity to judge the perceived spacing between the characters, and physically cut into the metal blocks to make them fit better (Figure 4b). With the advent of automated computerized typesetting, we often have no control over these aesthetic decisions.

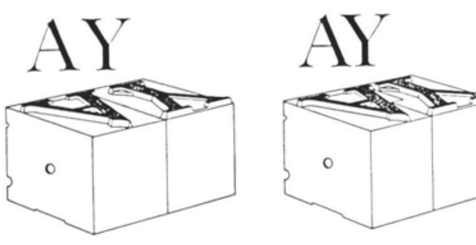


Figure 4a. [BIGE83] Figure 4b.

Width Tables

One of the first methods employed to take care of the worst of these problems was that of spacing tables [SCHM80 and BIGE83]. In this approach, a character's representation would have two values associated with it. The first would be a character width, indicating the width, in pixels, of the character's matrix. The second would be a spacing width, which would indicate how many pixels to move over from the right side of the character before laying down the next character. The spacing widths could be negative, allowing the next character to back into the current one.

For example, the 'T' might have a negative spacing width so that the next character would not seem so far removed from it. This method is seriously deficient in that it does not take into consideration the next character to be displayed, and how the shape of the left side of the next character interacts with the shape of the right side of the current character. If, for instance, the character following the 'T' is an 'I', then no kerning should take place, lest the two characters touch each other; e.g., 'TI'.

Table Kerning

The next development involved the creation of kerning tables for character fonts. These tables specify, for character pairs, how much the second character should be kerned into the first. The main drawback of this technique is the storage required to store all of the kerning values. An exhaustive table would have n^2 entries where n is the number of characters in the font. If a font includes all of the upper and lower case characters, the numerals, punctuation marks, and some special characters, n could easily reach 100. The exhaustive table would then have 10,000 entries per font! (Modern computerized typesetting houses typically need at least 100 font faces on-line.)

Fortunately, the tables need not be exhaustive. Many characters, completely vertical on both sides, cannot be kerned at all, and, therefore, need never appear in the kerning tables (e.g. the 'H'). Other characters have this property on only one side (e.g. the 'P'). Furthermore, many characters, though likely candidates for kerning, cannot be kerned when juxtaposed with certain other characters on one side or the other (e.g. the 'T' with the 'I'). Quite often, the tables can be reduced further by grouping together, under the same entries, several characters which behave similarly on one or both sides. For example, the 'P' and 'F', and the 'W' and 'V'. However, this is very dependent on the font style.

Another drawback of this technique is that it is based on an overly simplistic assumption: that kerning is dependent only on the pair of juxtaposed characters. Figure 5 demonstrates how the 'a' has to be kerned to the 'T' by different amounts depending on the characters which follow the 'a'. In general, the amount which two characters need to be kerned seems to depend on, at least, the rest of the characters in the word, though the spacing between words and lines, and within other words on the line and page, may also influence the amount of space needed between two characters [KIND76].

Recently, many systems have introduced the capability of specifying kerning triplets; that is, values by which the current character should be kerned to the previous character, based on the previous, current, and following characters. Although the exhaustive kerning tables based on character triplets is n^3, the actual 3-dimensional matrix is very sparse. However, determining which entries are non-zero, and determining their values, is no trivial task.

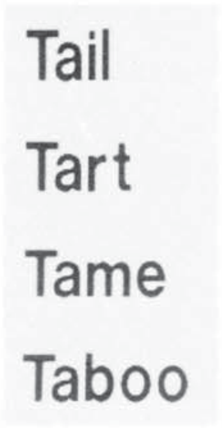


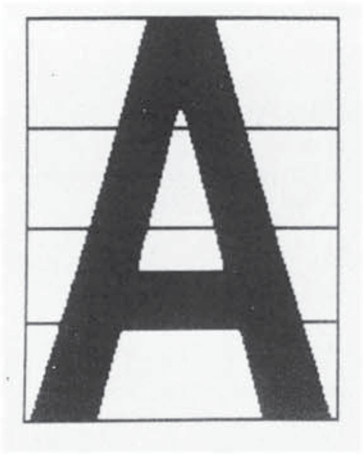
Figure 5. The characters following 'Ta' influence how much the 'a' has to be kerned to the 'T'.

Sector Kerning

A more sophisticated approach called *sector kerning* has appeared in recent years. This method stores a crude representation of the character's shape on the left and right side and uses that information to calculate, on the fly, how much to kern each pair of characters.

Some number of horizontal sectors (not necessarily of equal height) are specified for a particular font face (Figure 6a). Then, for each character in the font, one specifies the distance within each sector that another character is allowed to penetrate from both the left and right sides of the character (Figure 6b).

Before a character is displayed, it is determined which sector allows for a maximum penetration of the current character with the previous character. This is done by taking the minimum of the pairwise sums of the penetration values from the left side of the current character and the right side of the previous character. The resulting value is used as the amount by which the the pair of characters is kerned.



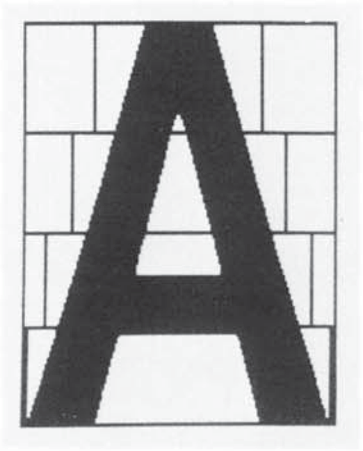


Figure 6b. Penetration values.

Since the position and height of the sectors can be established by the font designers, more weight can be given to those areas which are more critical in spacing techniques. It has been suggested that, since the eye follows just along the x height when reading lower case text, the ascenders contribute very little, if any, to the spacing, and the descenders even less (Figure 7 — [KIND76]). Therefore, one may be able to place all of the kerning sectors within the x height of the characters, and ignore the ascenders and descenders as far as spacing is concerned.

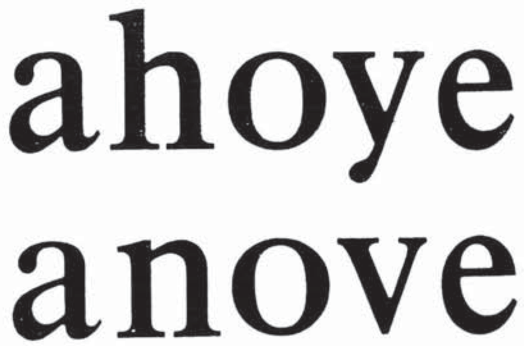


Figure 7. Ascenders and descenders do not affect the spacing [KIND76].

The situation becomes even more complex when reading upper case text (where the eye tracks along the top of the capitals) or mixed upper case and lower case. For example, the 'T' in 'Toe' needs a closer fit to the 'o' than it needs to the 'h' in 'The' (Figure 8). This may indicate that, in the first case, the eye is interpreting the 'T' as a lower case letter with an ascender, and therefore the spacing needs to be uniform at the x height, whereas, in the second case, the eye is interpreting the 'h' as a capital, and therefore the spacing needs to be uniform at the cap height [KIND76]. This suggests that alternate sets of penetration values may be useful depending on the nature of the text being read.

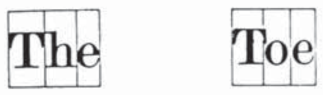


Figure 8. Case sensitivity is necessary to properly space text [KIND76].

The most immediate advantage of this technique is that the storage requirements are relatively low while the information content is high. If *n* sectors are used, and it takes one byte to store a penetration value, then 2*n* bytes are needed to store the kerning information for each character. For a 100-character font, to store 5 sectors per character requires only 1,000 bytes. Not only is there a corresponding reduction in the amount of human interaction needed to specify the kerning information, the process of specifying the penetration values lends itself to automation. Another advantage is that, although this technique does not provide an immediate solution to the problems of the influence of the surrounding text on the spacing of character pairs, it allows greater flexibility in experimenting with different spacing algorithms to achieve a more uniform perceived spacing of the text.

Summary

We have concentrated on three problem areas inherent in producing high-quality text on raster scanned display devices from pixel-oriented representations of the character forms:

1) Taking advantage of the high spatial coherence of characters, we managed to increase the compaction rate of the standard run-length encoding algorithm by encoding each row separately as a structure of runs of the foreground pixels, and including a replication count field to avoid repeating identical, adjacent rows. Care was taken to ensure that the overhead incurred by including the replication count field never increased the size of the compaction in cases where there were no identical, adjacent rows.

2) In order to improve the spacing of characters at subpixel resolution, we computed an *average* representation of the characters and adjusted them by shifting small percentages of the intensity values in each column to the left or right. By applying heuristics along the sides of the characters, we were able to avoid computing multiple representations of the characters and still achieve a highly satisfactory improvement in the spacing. However, the technique broke down when very small fonts were used, since the amount of adjustment was large relative to the width of the characters.

3) We explored various methods for *kerning* two characters which would otherwise leave gaps and holes in the spacing. We found that sector kerning provides excellent results in the general case, with little storage overhead and human interaction. However, more sophisticated techniques are needed to take into consideration all of the characters in a word when determining how much a pair of characters should be kerned.

References

BIGE83 Bigelow, C., and D. Day, "Digital Typography," *Scientific American*, v. 249, 2, pp. 106-119.

CATM79 Catmull, E., "A Tutorial on Compensation Tables," SIGGRAPH 1979 Proceedings, published as *Computer Graphics*, 13(2), August 1979, pp. 1-7.

CATM80 Catmull, E., and A. R. Smith, "3-D Transformations of Images in Scanline Order," SIGGRAPH 1980 Proceedings, published as *Computer Graphics*, 14(3), July 1980, pp. 279-285.

CORN70 Cornsweet, T. N., *Visual Perception*, Academic Press, New York, 1970.

CROW78 Crow, F. C., "The Use of Grayscale for Improved Raster Display of Vectors and Characters," SIGGRAPH 1978 Proceedings, published as *Computer Graphics*, 12(3), August, 1978, pp. 1-6.

FREE61 Freeman, H., "On the Encoding of Arbitrary Geometric Configurations," IRE Transactions on Electronic Computers, EC-10, 2, June 1961, pp. 260-268.

KAJI81 Kajiya, J. and M. Ullner, "Filtering High Quality Text for Display on Raster Scan Devices," SIGGRAPH 1981 Proceedings, published as *Computer Graphics*, 15(3), August, 1981, pp. 7-15.

KAWA80 Kawaguchi, E. and T. Endo, "On a Method of Binary-Picture Representation and its Application to Data Compression," IEEE Transactions on Pattern Analysis and Machine Intelligence 5, 1(January 1980), pp. 27-35.

KIND76 Kindersley, D., *Optical Letter Spacing for New Printing Systems*, Sandstone Press, New York, 1976.

KIND79 Kindersley, D., and N. Wiseman, "Computer Aided Letter Design," *Printing World*, October 31, 1979, pp. 12-17.

KNUT79 Knuth, D. E., *TEX and Metafont: New Directions in Typesetting*, American Mathematical Society and Digital Press, 1979.

NEGR80 Negroponte, N., "Soft Fonts," Proceedings, Society for Information Display, 1980.

PAVL83 Pavlidis, T., "Curve Fitting with Conic Splines," ACM Transactions on Graphics, 2(1), January 1983, pp. 1-31.

PLAS83 Plass, M., and M. Stone, "Curve-Fitting with Piecewise Parametric Cubics," SIGGRAPH 1983 Proceedings, published as *Computer Graphics*, 17(3), July 1983, pp. 229-239.

PRIN79 Pringle, A., P. Robinson, and N. Wiseman, "Aspects of Quality in the Design and Production of Text," SIGGRAPH 1979 Proceedings, published as *Computer Graphics*, 13(2), August 1979, pp. 63-70.

SCHM80 Schmandt, C., "Soft Typography," Information Processing 1980, Proceedings of IFIPS, pp. 1027-1032.

SCHM83 Schmandt, C., "Fuzzy Fonts," Proceedings of the National Computer Graphics Association, 1983.

SEYB69 Seybold, J. W., *The Market for Computerized Composition*, Printing Industries of America, Computer Section, Washington, D.C., 1969.

SEYB79 Seybold, J. W., *Fundamentals of Modern Photo-Composition*, Seybold Publications, Inc., Media, Pennsylvania, 1979.

WARN80 Warnock, J. E., "The Display of Characters Using Gray Level Sample Arrays," SIGGRAPH 1980 Proceedings, published as *Computer Graphics*, 14(3), July, 1980, pp. 302-307.

WEIM80 Weiman, C. F. R., "Continuous Anti-Aliased Rotation and Zoom of Raster Images," SIGGRAPH 1980 Proceedings, published as *Computer Graphics*, 14(3), July 1980, pp. 286-293.