ELECTRONIC PUBLISHING, VOL. 6(3), 117–132 (SEPTEMBER 1993)

Infinifont: a parametric font generation system

CLYDE D. MCQUEEN III AND RAYMOND G. BEAUSOLEIL

ElseWare Corporation
101 Stewart Street, Suite 700
Seattle, WA 98101–1048, USA

email: clyde@elseware.com, ray@elseware.com

SUMMARY

We have developed a high-performance parametric font generation system for the creation and commercial supply of digital fonts, and in particular, for generating a wide variety of digital typefaces using a single compact representation of typographic knowledge and characteristics. Typographic feature detail can be added or removed, depending on the application. The system does not rely on master outlines for interpolation between or extrapolation from static typefaces. Our current implementation of this technology generates 50 Latin text characters per second on a 25-MHz 80386 platform without the use of a mathematics coprocessor.

KEY WORDS Infinifont Parametric Font generation system

1 INTRODUCTION

Digital fonts have become ubiquitous in the age of the personal computer. While these fonts have significantly increased the range of typographic expression available to digital documents, they have created a new class of problems. In particular, a document created on one computer system, using digital fonts available on that system, may not display with high fidelity on a second system (even if the computer models and/or operating systems are identical) if the same font files are not available on both systems. While it is possible for the second operating system to map a font request by the viewing application into a request for another font, this replacement font will not in general be a replica of the font used to create the document on the original computer system.

In addition, font files require so much storage space that large font libraries are not practical for the vast majority of individuals and organizations. These large font file sizes compound the document portability problem, since it is impractical to embed the font files within a document file prior to rapid electronic distribution within a large, networked organization. The solution adopted by most workgroups is to restrict the number of available fonts to 35–40 fonts or less.

In Figure 1, we show the total system storage size required for two digital type libraries, comprised of either TrueType or PostScript font files, as a function of the number of fonts in the system. Note that a 200-font TrueType library requires approximately 12 MB of disk storage space, while the corresponding PostScript Type 1 library requires 6.5 MB. On systems with severely restricted storage (such as handheld personal computers), font library size would have to be significantly constrained.

CCC 0894–3982/93/030117–16

Received 15 August 1993
Revised 1 December 1993

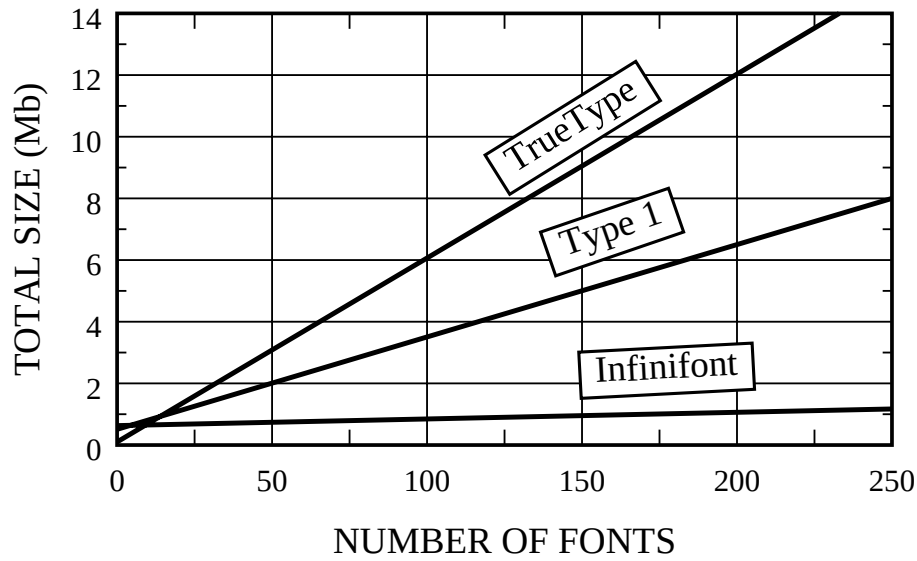

Figure 1. Comparison of the total system storage size required for TrueType, Type 1, and Infinifont fonts as a function of the number of fonts

Here we describe *Infinifont*, a general method and architecture for parametrically generating characters and fonts in digital format for use in electronic computer systems. The Infinifont system uses a very compact font representation scheme and a high-performance font generation engine to produce high-quality typefaces. The architecture consists of three primary components:

1. The *Terafont*, a body of typographic knowledge represented by machine instructions in a binary computer file format;
2. *PANOSE files*, one or more compact binary parameter files that specify the details of the font(s) to be created; and
3. The *Font Engine*, a computer software program capable of rapidly executing the typographic instructions using one of the PANOSE files as an input.

The Font Engine is a software virtual machine that executes a sequence of special instructions for manipulating font constructs, including points, Bézier curves, and hints. The PANOSE file input to the font machine must at least include a minimal set of measurement data, such as a PANOSE number, which can be used to compute global variables representing typographic characteristics common to all of the glyphs in the font. Additional global variables, and local variables needed to capture nuances of individual characters, can then be computed or assigned default values by the Font Engine. Metric data embedded in the PANOSE file can be used to coerce the individual characters into predetermined bounding boxes. As needed, any global or local variable can be overridden at any time during the execution of the instruction stream by additional data values stored in the PANOSE file. Therefore, the fidelity with which a traditional typeface can be replicated depends on the number of overrides stored in the input PANOSE file. The instruction sequence is sufficiently rich that a wide variety of fonts within a large design space can be instantiated in

the field using a relatively small parameter set, without extrapolating from a single master outline or interpolating between two or more outlines. Since the generated character shapes are defined in terms of conventional mathematical constructs, such as points and Bézier curves, they can be easily translated into other digital font formats if necessary.

An essential feature of Infinifont is the ability to closely replicate existing or traditional typefaces. There are two problems which must be solved before high-fidelity replication can be achieved: matching the character advance and other metrics, and matching the glyph shape. A failure to correctly match metrics within a target typeface is easily noticed; if even a single character in a font has been synthesized with a width that is only 1% too large, the document may not paginate identically in two different operating environments. However, metrics matching is reasonably well understood and is solved by many font vendors simply by copying the relevant metrics from the target font to the newly generated font. Metric data generally comprises less than 5% of the size of a typeface; hence it is reasonable to copy the metrics into the PANOSE file (and then into the document itself). It is crucial to reconcile the metrics with the glyph shapes during generation.

The glyph outline conformance problem is solved by using first a PANOSE number, then global override data, and then local override data, to update at synthesis time those parameters which will significantly affect the final glyph shape. The override system is designed so that the larger the number of override inputs, the closer the contour conformance, and the most general override inputs are presented and applied first. This provides for a reasonable 95%–5% solution, where a very close replica may be represented in about 5% of the overall space required to store the typeface using a conventional font format. Closely replicating well known typefaces while maintaining the size advantage of parametric type is a key advantage of the Infinifont system. The architecture can therefore always be scaled from low to high sophistication by increasing the number of detail strings used to describe a target typeface. On some platforms where both volatile and nonvolatile storage space is severely limited (such as a handheld personal computer), size (and degree of conformance) can be reduced; on a workstation or a printer, the degree of conformance (and therefore the required size) can be much higher.

Our current implementation of the Infinifont system is capable of generating over 50 Latin text and display characters per second on a 25-MHz 80386 platform without the use of a mathematics coprocessor, or over 150 characters per second on a 50-MHz 80486 platform. As shown in Figure 1, a system containing 200 fonts (each of which include more than 300 characters), consisting of the Font Engine, the Latin Text and Display Terafont, and 200 PANOSE files, requires approximately 1 MB of hard disk storage.

2 REQUIREMENTS FOR A FONT SUPPLY TECHNOLOGY

Generally, digital electronic representations of fonts are physically stored on components of the hardware that constitute the computer system, such as on a hard disk or read-only memory (ROM). Fonts are then ‘requested’ by a document that has been opened by computer application software for display by the computer system. If a requested font is available, then the computer operating system accesses the digitally formatted font, converts the data into native graphical display objects, and then displays the necessary characters. If the font is not available, then the operating system can substitute another available typeface or attempt to synthesize the requested font.

The digital font generation and supply process can be described as a sequence of operations applied to some initial artistic representation of a font. The first set of operations, performed in the 'shop' using computer software, converts a font from an initial visual representation to a digital format suitable for distribution to the 'field.' In the field, a second sequence of operations is performed to display the font on a computer system, either on a computer monitor, on the output of an electronic printer, or using any device capable of producing a bitmap.

Consider a font F defined as a set of characters c_n, which have outlines that in turn consist of paths p_{nk}, or

$$F = \{c_n\}, \tag{1}$$

and,

$$c_n = \{p_{nk}\}. \tag{2}$$

These paths can be represented graphically as a collection of on-curve and/or off-curve points, with rules which specify how to connect the on-curve points given, for example, the positions of the off-curve points. Generally, there are additional components of a digital font designed to enhance its visual appearance when displayed electronically by an operating system (such as on a computer monitor, or on an electronically printed page). First, virtually all *scaleable* fonts (i.e. font outlines which can be scaled to many different sizes) also carry special programs, called 'hints,' that instruct the operating system to adjust the shapes of characters in that font prior to rasterization (conversion to a bitmap) for display, particularly at small point sizes where the resolution of both the display device and the human eye significantly affect the apparent shape of the characters. Second, digital font files also contain 'metric data,' which instruct the display device to place characters on a page in certain relative positions, specifying a certain amount of white space in both ordinary (character advance widths, line spacing) and extraordinary (kerning widths) situations.

Consider a measurement system **M** capable of operating on a target font F, which may or may not be initially represented digitally. In the 'shop,' we apply a (possibly nonlinear) measurement operator M to this font, yielding a set of measurements E (the measurement data, or simply 'measures'):

$$E \equiv \{e_m\} = M(F), \tag{3}$$

where e_m is measure m of the set. Note that if the measurements are made on the entire font, rather than on individual characters only, the measures can include intercharacter metric data. In some cases, the measures E are generated without explicit reference to an existing target font; instead, the measures are designed ab initio for the purpose of instantiating an original font.

The measures are subsequently converted to their digital format D using a conversion operator C, or $D = C(E)$; this conversion operation may include hints which are necessary for a faithful rendering of the original typeface, and/or may include other data needed to initialize the font generation subsystem.

In the field, the digital font generation operator G acts on the digital format D to create a digital representation F' of the target font F:

$$F' = G(D) = G[C(E)]. \tag{4}$$

In principle, there may be another conversion operator required to translate the digital representation into a format understood by the operating system.

Clearly, if we desire a high-fidelity digital rendition of the original typeface, or $F' \approx F$, then we require that the consecutive application of the conversion and generation operators C and G essentially invert the transformation performed by the measurement operator M on the original font F, or

$$G[C()] \approx M^{-1}(), \tag{5}$$

where the inverse measurement operator M^{-1} is defined by $M^{-1}[M(F)] = F$. Of course, if the measurement operator M describes a 'many-to-few' transformation, the inverse operator M^{-1} may not exist; the corresponding font replication fidelity would be limited by the number of measurements in the set E.

There are additional requirements which should be satisfied by an ideal font supply system, including:

1. The operator G should be able to generate characters from the digital font data D much faster than the fastest human typist, and ideally as fast as the display rasterizer;
2. The operator G and the digital font data D should occupy only a small fraction of the electronic storage space (either static or dynamic) required by the operating system; and
3. The operator G and the font data D should be easily portable from one operating system to another.

Referring to Figure 1, these requirements can be translated into hard specifications for an implementation of a font delivery technology as microcomputer software:

1. The run-time component of the technology should be able to generate Latin characters at a rate of 40–80 characters per second;
2. The storage space required by the run-time component (i.e., the y-intercept in Figure 1) should be 500 KB or less, and the space occupied by each PANOSE file should be 2 KB or less when high-fidelity replication is desired; and
3. The run-time component should be written in the C computer language.

As discussed below, the Infinifont system allows the slope of the line in Figure 1 to be scaled, according to the required replication fidelity: when the desired resolution is low, then the size of the PANOSE files can be reduced from that required for high-fidelity replication.

3 SURVEY OF THE FIELD

If we restrict our attention to commercially available digital font distribution technologies, then the current field includes bitmapped fonts, scaleable fonts, and distortable fonts.

3.1 Bitmapped fonts

Fonts represented as bitmaps have been tuned for a particular system resolution. That is, each character in the font consists of a certain number of black pixels that have been laid into a square grid; different digital bitmap files must be created for different display

sizes of a given typeface. As prior art, the most significant manifestation of a conversion operator C_b for bitmapped fonts is the METAFONT program, designed by D. E. Knuth [1]. Superficially, METAFONT is similar to Infinifont, since it consists of procedural software which constructs digital representations of fonts based on numerical input data. However, the instruction stream generated by compiling the METAFONT source file must be executed once per desired point size, since the output digital font format D_b is bitmapped; the desired fonts cannot be generated in the field using a subset of all possible input parameters, since the executor exists in the shop only; the binary files distributed to the field (containing the bitmapped fonts) are in fact much larger than the original data files used to create them; and the range of fonts which can be constructed by arbitrary input values is necessarily limited.

3.2 Scaleable outline fonts

Fonts represented by scaleable outlines are generally stored as a collection of on-curve and/or off-curve points which describe fundamental graphical constructs such as lines and circular arcs, or second-order and third-order Bézier curves. They can be displayed accurately at nearly any device resolution; at low resolutions, hint programs are usually executed prior to rasterization. Scaleable font files are considerably smaller than the collection of bitmapped font files that would be required to represent the same range of point sizes for a particular typeface; that is, $D_s \ll D_b$. Nevertheless, since all points, hint programs, and metric data are stored in the data file, the file size typically required to provide high fidelity replication of the original font is approximately 35–70 KB; hence 200 such fonts would require 7–14 MB of storage for data alone. Commercially available examples of digitally scaleable outline font formats are PostScript Type 1 and TrueType 1 (which use Bézier curves to connect points), and Intellifont (which uses lines and arcs). Generally, outline fonts based on either of these formats are generated in the shop and then distributed to the field. The operating systems of the computers on which these fonts are used can generate the required characters by converting the digitally formatted font files into graphical display objects directly, or utility software can be provided by the font vendor which enables the operating system to perform the conversion. Utility software also exists which can convert from one format to the other, and/or distort the original outlines.

3.3 Distortable outline fonts

Fonts represented by distortable outlines are generally stored as one or more scaleable outline fonts with rules that specify either how to extrapolate from a particular outline font, or how to interpolate between two or more outline fonts. The extrapolations and/or interpolations occur along a limited number of well-defined axes or line segments. For example, suppose that two scaleable typefaces which differ only in ‘contrast’ are defined as the end points on a line segment. (The typographic term ‘contrast,’ as defined by the PANOSE typeface classification system, denotes the difference between the widths of predominant thick and thin stems of the characters in the font.) The lower contrast font is assigned the value 0.0 on the ‘contrast’ axis, while the higher contrast font is assigned the value 1.0. The rules applied to the distortable font data specify the interpolation of the on-curve and off-curve point positions as the ‘contrast’ parameter is varied. These

rules, the two outline fonts, any hint programs, and the subsystem that applies the rules to the available outlines, constitute the generation operator *G*. The metric data and the 'contrast' value constitute the digital parameter file *D*. In principle, then, any value of the contrast that can be found on the defined line segment can be used to create a new font. However, since the outlines that define the endpoints of the 'contrast' line segment must have the same number of points, distortion cannot occur between two fonts that contain fundamentally different topologies. Furthermore, other typographic characteristics, such as 'weight,' cannot be altered by adjusting the 'contrast' parameter; hence, an additional axis, and two additional outline fonts must be added to define new interpolation endpoints. In general, if n typographic characteristics are to be represented by a distortable font, then 2^n outline fonts are required to define the endpoints of the necessary distortion axes, and n input parameters must be specified to instantiate.

Clearly, then, a single distortable font file cannot be provided that will generate a wide variety of fonts (with a corresponding variety of character topologies and typographic nuances) without becoming exponentially large and requiring an unacceptably long generation time. This problem is substantially compounded if additional scaleable typefaces are assigned to points internal to the n-dimensional hypercube defined by the distortion axes; these intermediate typefaces allow more complex interpolation rules to be defined. In practice, then, distortable typefaces implemented for a computer system with finite storage and speed can instantiate a large number of fonts but only within a small design space.

4 INFINIFONT

The Infinifont parametric font generation system provides distinct advantages over these technologies. In particular, it satisfies all three of the requirements of an ideal font supply system defined above.

The unique ability of Infinifont to override the execution state of the font generation system at any time allows high-fidelity font replicas (as well as original typefaces) to be instantiated without explicitly providing the values of all possible variables in the input parameter files. Instead, the majority of variable values are computed from global variables or constants that are valid for all characters in the font, or are assigned unique default values in the function describing the character itself. As a result, only a few dozen of over 10,000 possible input parameter values must be stored in the binary parameter files; these files will typically occupy less than 2 KB of hard disk storage space.

An additional advantage of Infinifont technology is the ability to easily extend the character generation instructions (the 'Terafont') to include new glyph topologies. In the shop, a new function containing source code describing the new topology can be added to the list of active Terafont functions and then compiled into the Terafont binary. In the field, compiled glyph binary code describing an outline unique to a particular typeface can be read from a parameter file for runtime addition to the Terafont binary.

The Infinifont measurement operator *M* represents the PANOSE typeface measurement system and the user interface of the Infinifont font development system. The font development system is represented by the operator *C*: it reads the measurements and writes the digital font data *D* into the PANOSE files. The run-time Font Engine and the binary form of the Terafont are implementations of the generation operator *G* in this system: the Font Engine executes the Terafont after reading a PANOSE file to obtain input data.

4.1 The PANOSE typeface classification system

The PANOSE classification system [2] (the Infinifont measurement operator *M*) assigns a multi-digit number to a typeface that describes its predominant visual characteristics; the system provides a procedure for measuring a typeface to generate a PANOSE number. In general, a PANOSE number is a data structure containing PANOSE 'digits' (each 'digit' is actually represented on computer systems as one or two bytes, and may in fact have several decimal digits), each of which represents a visual characteristic like *weight* (heaviness of the main vertical stems), *contrast* (ratio of the thick-to-thin stems), and *serif style* (e.g., sans serif, cove serif, or square serif). There is a separate PANOSE classification system for each Script (e.g., Latin, Cyrillic, or Kanji) and each class of type, or Genre (e.g., Text and Display faces, Decorative faces, and Woodcut faces). Unlike other type classification systems, PANOSE classification systems work whenever possible on physical measurements of the type, rather than any subjective historical or artistic analysis. (In practice, physical measurements are only sufficient for the 'base' genre within each script, such as Latin Text and Display. Other genres require somewhat subjective, although not artistic or historic, judgments.)

PANOSE numbers are used by microcomputer application vendors to substitute an available font when a requested font is unavailable. The 'distance' between the PANOSE number of the requested font and that of each available font is computed; these numbers are then compared, and the font with the closest PANOSE number to that of the request can be substituted. The PANOSE system for any Script and Genre is designed to make this possible. Substitution may occur within the same classification scheme, or across classification schemes in the case of 'cross-literal' mapping.

PANOSE numbers offer a convenient set of measures for generic font instantiation. Provided that the PANOSE measurement system for a particular Script and Genre is invertable (that is, the inverse of the PANOSE measurement operator *M* exists), a given PANOSE number provides specific measurement data that provides a numerical description of the visual appearance of a typeface. Therefore, global variables are defined which numerically represent those typographic features which are measured to subsequently calculate PANOSE numbers. This approach offers a special advantage for any computer environment that uses PANOSE numbers for substitution: if a close PANOSE number match cannot be found for a specific font request, then a replacement font may be synthesized using the requested PANOSE number as input.

Currently, the PANOSE system is available at two levels of sophistication. PANOSE 1 classifies typefaces using a 10-byte number, or data structure; it is ideal for closed systems where storage requirements are strict and the available fonts are well known. PANOSE 2 classifies typefaces using a 72-byte data structure describing 36 typographic features; it provides significantly richer measurement data than the PANOSE 1 system. It is designed for extensible systems where fonts are regularly added and removed, using mixed languages and mixed font formats (including distortable fonts).

4.2 Pecos development environment

Figure 2 shows the general layout of the graphical user interface of the Pecos font development environment, as well as a representative glyph function for the cursive topology of

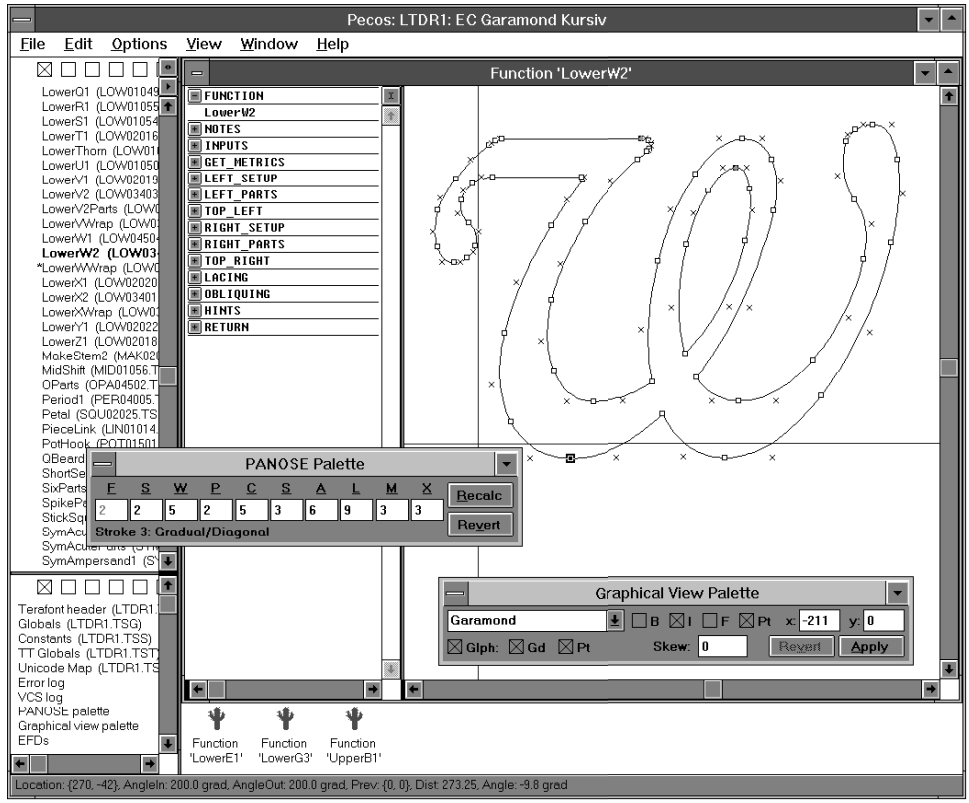

Figure 2. The Pecos development environment

a Garamond lowercase *w*. Note the list of available Terafont component names, which can be used to navigate among Terafont global and function files. The glyph function for the cursive lowercase *w* has three views, two of which are shown in Figure 2. The text view allows the function text to be divided into sections for convenient editing. The graphical view displays the character outline path resulting from the execution of the compiled function binary by the Font Engine. When visible, the third view (the 'value bar') appears to the left of the text view, and displays in text format the values of all variables local to the glyph function `LowerW2` at the end of its execution.

Pecos uses the same version of the Font Engine for glyph screen display as that shipped with the run-time Infinifont system (although through a necessarily different application programmer's interface). As the Terafont and PANOSE files are developed, individual glyphs can be compiled and displayed in the graphical view, and TrueType files for all active, compiled PANOSE files can be written to disk for testing. Pecos also includes a TrueType character debugger that can be used to read and execute hint programs bound to a particular glyph, and then rasterize the glyph for display on a grid. When the Terafont

and the desired collection of PANOSE files have been tested, they can be written in binary form to disk for distribution with the run-time Font Engine.

4.3 The Terafont

The Terafont is fundamentally an abstract body of typographic knowledge, represented as a collection of software *functions* that instruct the Font Engine to build the character outlines in a font. Generally, the Terafont programs include instructions that compute the values of variables that will be used globally throughout the font (the Terafont globals), as well as instructions needed to build glyphs, parts of glyphs, composite glyphs, and perform general-purpose computations (the Terafont functions). To synthesize a complete font, the Font Engine must first execute the global program, and then the program for each character (A, B, C, etc.) in turn. To synthesize a single character, the Engine executes the global program and then the program for a single character. The Font Engine employs a caching scheme to store the results of the global program execution so that subsequent characters may be synthesized without re-executing the global code.

The Terafont supports the function construct much like stack-based languages such as C, Pascal, and FORTRAN. Like these and other languages, the use of functions allows the total Terafont to be significantly smaller and easier to program and maintain. The Terafont Source Language, or TSL, is essentially like the C programming language, with support for special data types and opcodes that allow for efficient construction of outline curves and hint constructs. It is important to understand that the Terafont is not a 'master outline', since it contains no points or curves. Rather, it is a sequence of instructions that are executed by the Font Engine using a PANOSE file to provide input data; the points and curves which represent the character outlines are *created*, not transformed. The complexity of the Terafont is governed by the range of distinct topologies (such as a and *a*) encompassed by the desired glyph space.

4.3.1 Terafont globals

The global program consists of instructions to calculate several hundred global variables. These variables define properties and measurements, and are used by the individual character programs. In addition, the global program includes instructions for allocating and defining special constructs used for generating hints.

Terafont globals fall into one of several broad classes, including:

1. *Character map.* A list of characters to be included in the synthesized font, with a corresponding list of Terafont functions needed to generate those characters. The characters are assigned a Unicode Worldwide Character Encoding standard identification number (defined by the Unicode Consortium).
2. *Constants.* A list of names representing allowed Terafont source data types which are assigned constant numerical values. These constants can be accessed by other Terafont globals and any Terafont function.
3. *PANOSE globals.* A set of global variables whose values represent the measurements made on a target font to determine the PANOSE number. These variables are computed using a PANOSE number as input.

4. *Global variables.* Other global variables whose values represent numeric typographic characteristics of a font that are not measured to determine a PANOSE number, but that are measured to provide high-fidelity replication of a target font. In principle, all variables in the Terafont could be considered to be global; however, since some of these variables would be used by only one glyph under certain circumstances, this choice would be highly inefficient, particularly when the Font Engine is operating is generating a single character. Hence, the choice of whether or not to define a particular variable as global is made on the basis of efficiency.
5. *Format-specific globals.* Additional global variables can be named and defined as needed to support a specific font format, such as TrueType.

With the exception of overrides, the Font Engine executes instructions in the same order that they are written into the Terafont source files.

4.3.2 Terafont functions

Each character program uses the global variables and synthesizes a single character. The results include both the character outline as well as the hinting constructs required by the operating system to generate a high-quality raster image. For the TrueType font format supported by Microsoft and Apple, the hinting constructs themselves are small programs that are executed by the rasterizer provided by the operating system. In a very real sense, the Infinifont character programs are automatic code generators.

Terafont functions fall into one of several broad classes, including:

1. *Wrapper.* A top-level function that binds one of several different topologies to one character Unicode value. (See the discussion of the Unicode character mapping standard in the description of parameter files below.) A wrapper function will generally call a particular glyph function depending on the requested topology. If a particular character in a Terafont has only one basic topology, then the Unicode value may be bound directly to the corresponding glyph function. The Terafont function required to select between the lowercase a and *a* topologies falls into this class.
2. *Glyph.* A function that contains all Terafont source needed to generate a glyph for a given topology, including metrics coercion and computation, and hinting code.
3. *Composite.* A special type of glyph function that binds two or more glyph functions to a single Unicode value. The Terafont function required to build a character such as ä falls into this class.
4. *Part.* A function that contains the Terafont source code needed to generate part of a glyph, such as a stem, a serif, a hook, or a finial. It requires input parameters and does not contain metrics source code. In some cases, a part may contain hint code, but generally it creates point references which are returned to the calling function for hint generation.
5. *Calculator.* A function that provides a general service that could be called by any glyph or part.
6. *Hint.* A function that processes and generates hint fragments, which are technology-dependent.

In general, wrapper and composite functions call glyph functions, which in turn call part and hint functions; a calculator function may be called by any other function.

Significantly, the local variables defined in the Terafont glyph source code are either provided with default values, or computed in terms of other variables that have been assigned default values. These default values are set in terms of global variables, global constants, or local constants. Hence, every glyph function is capable of generating an outline even in the absence of input variables. At any point during the execution of the Terafont binary representing this glyph function, values of variables local to the glyph function or its dependents can be overridden using data provided in a parameter file. Heuristics are used to compute default values for all variables in the system that do not directly determine the PANOSE number for a given font; consequently, only a subset of the local variables need be overridden to provide high-fidelity replication of a target font. Therefore, a significant reduction in the size of the PANOSE file is obtained.

4.3.3 Terafont hints

Hints are small software programs which modify a glyph outline during rasterization to enhance its visual appearance at display time, particularly at low resolution and/or small point sizes where distinct visual features of a character can be difficult to represent. Hint formats generally depend on the computer platform being used, since properties of the rasterizer vary from one operating system to another. Infinifont does not require the use of a specific point-based hinting format; rather, the Infinifont hinting model provides an architecture that will support a variety of popular hinting formats. However, the first implementation of the Terafont has been optimized for use with a TrueType rasterizer; in fact, hint fragments are entered into the Terafont source files directly in TrueType assembly language.

In Infinifont, typographically interesting features such as stems and serifs are specified by the PANOSE file and identified at synthesis time as the Terafont binary is executed. They are neither explicitly installed in the shop for all fonts in the design space (as in the case of distortable fonts), nor are they identified by the hint instructions at rasterization time (as is required by some scaleable font design tools). Rather, as the glyph is built, hint instructions are bound to the glyph that are specific to the features that are incorporated into the glyph.

Specific points in a glyph outline can be identified for later reference in a supported hint fragment construct. This method is implemented in TSL using the `pointref` (or point reference) data type; variables of this type can be used to specify a particular point on a path and then bound to the path. As the path is intersected, trimmed, transformed, and/or linked, the Font Engine updates the point reference so that it continues to indicate the original tagged point.

A considerable reduction in Terafont binary size is obtained by defining globally known hint fragments, or font programs, which perform common hint instructions that are likely to be required by more than one glyph, and can be called by other hint fragments. In TrueType, these font programs are stored directly in the `fpgm` table.

4.4 The Font Engine

The Font Engine is a software virtual machine that executes programs stored in the Terafont binary. The virtual machine resembles other stack-based systems, except that there are

special instructions for manipulating font constructs, including Bézier curves and hints.

During execution of the Terafont, the Font Engine maintains a glyph database that contains a runtime record of data bound to each glyph outline that has been constructed, including point references, hints, metrics, and alignment points to be used when building composite characters. In font-at-a-time mode, where an entire font is to be generated, after execution of the globals and all functions required to build the glyphs listed in the character map, the Font Engine binds all resulting outline, hint, and metric data together and then builds a font file under a supported format. In character-at-a-time mode, where only a single character is to be generated as quickly as possible, the Font Engine will execute only the Terafont globals and those glyph functions and called subfunctions necessary to construct that character. The executed global code can be cached for later use, so that subsequent characters can be generated even more rapidly.

Font Engine opcodes can be divided into the following broad categories:

1. *Path creators.* Opcodes which create paths and graphical elements that can be used to build paths, such as points and curves. The path elements comprise second-order or third-order Bézier curves, defined by an ordered set of on-curve and off-curve points.
2. *Path transformers.* Opcodes which convert existing paths into new paths using standard coordinate mapping transformations, including scaling, rotating, reflecting, and obliquing.
3. *Path intersectors.* Opcodes which find one or more specified intersections of two paths and then return one or more resultant paths. Intersection operations include trimming of unwanted path segments after the intersection is found, applying a rounded corner to the vertex of an intersection, updating references to points of interest for hinting, linking two or more paths into a single path, and closing a sequence of paths to form a single path representing a character outline.
4. *Mathematics.* Opcodes which perform basic arithmetic, including stack management, and compute special functions, such as transcendentals.
5. *State handlers.* Opcodes which can access and modify the current Font Engine system state. This set includes opcodes which: access and modify the stored metric values for the current glyph function (described below); update the hint state bound to a particular glyph so that a specific hint fragment is executed when the glyph is passed to the rasterizer; update data needed to build composite characters, such as alignment points; and override the current state of a global or local data space.

The ability of the Font Engine to override the values of variables stored in the global and local data spaces is a unique feature of Infinifont, allowing PANOSE file size and character generation rate to be functions of the desired target font replication fidelity. A detail string included in a PANOSE file instructs the Font Engine to override the value of a particular global or local variable at a particular point during the execution of the Terafont. If the Font Engine encounters an override opcode in the Terafont instruction stream, it accesses the override instructions indicated by an argument to the opcode and performs the requested operation. For example, an override string may ask that a particular value stored at a specified location in the local data space be incremented or decremented by a certain amount, scaled by a certain factor, or simply replaced with another value.

4.5 PANOSE files and outline conformance

The response of the Infinifont system to an instantiation request depends on the data provided by the PANOSE file, and can occur at one of the following levels:

1. *The generic Font Engine response to an input PANOSE number.* If a PANOSE 2 number is input, then global instructions are executed which extract measurement information from the PANOSE 2 digits and then assign that information to a set S_2 of global variables. The values of these variables are generally not overridden, and the Terafont is designed to ensure that a subsequent measurement of the generated font F' will yield the input PANOSE 2 number; in other words, self-consistency is enforced by design. If a PANOSE 1 number is specified, then global instructions assign extracted measurement information to a set S_1 of global variables. By design of the PANOSE mapping system, $S_1 \subset S_2$; consequently, those global variables contained in the set $S_2 - S_1$ must be assigned default values. These global variables may be overridden, but those overrides must not disable self-consistency. The Times uppercase E displayed in Figure 3a has been generated using a PANOSE 1 number only, while the E shown in Figure 3b has been generated using a PANOSE 2 number.
2. *The response of level 1 with the addition of metrics coercion.* While the generated font F' will not necessarily represent an extremely high-fidelity rendition of a target font F (if such a target font exists), it will have a simpler visual appearance and the same metrics.
3. *The response of level 2 with global detail strings specifying topologies that are not extracted from the PANOSE number.* These will be more common if a PANOSE 1 number has been selected, rather than a PANOSE 2 number.
4. *The response of level 3 with additional global detail strings that capture global typographic data not included in the PANOSE 2 measurements.* The E displayed in Figure 3c has been generated using both a PANOSE 2 number and global overrides.
5. *The response of level 4 with local detail strings that capture typographic nuances present in individual glyphs.* The E displayed in Figure 3d has been generated using a PANOSE 2 number, global overrides, and local overrides.

Note that while the addition of global and local overrides can increase the level of glyph conformance, the response of the Infinifont system to PANOSE numbers provides an acceptable Times-like E with a minimum of disk storage.

5 CONCLUSION

The Infinifont font generation technology is a proven high-performance technology for creation and distribution of digital fonts. In the future, we will enhance the system in several ways:

1. We will continue to reduce the disk storage requirements and increase the performance of the system;
2. We will enrich the Latin Terafont to include more heuristics, in order to reduce the number of overrides required for high-fidelity replication;
3. We will implement expanded Latin character sets, such as small capital letters, more ligatures and other symbols, with relatively little effort and a small increase in the Terafont size; and
4. We will implement other character sets, including Cyrillic and Kanji.

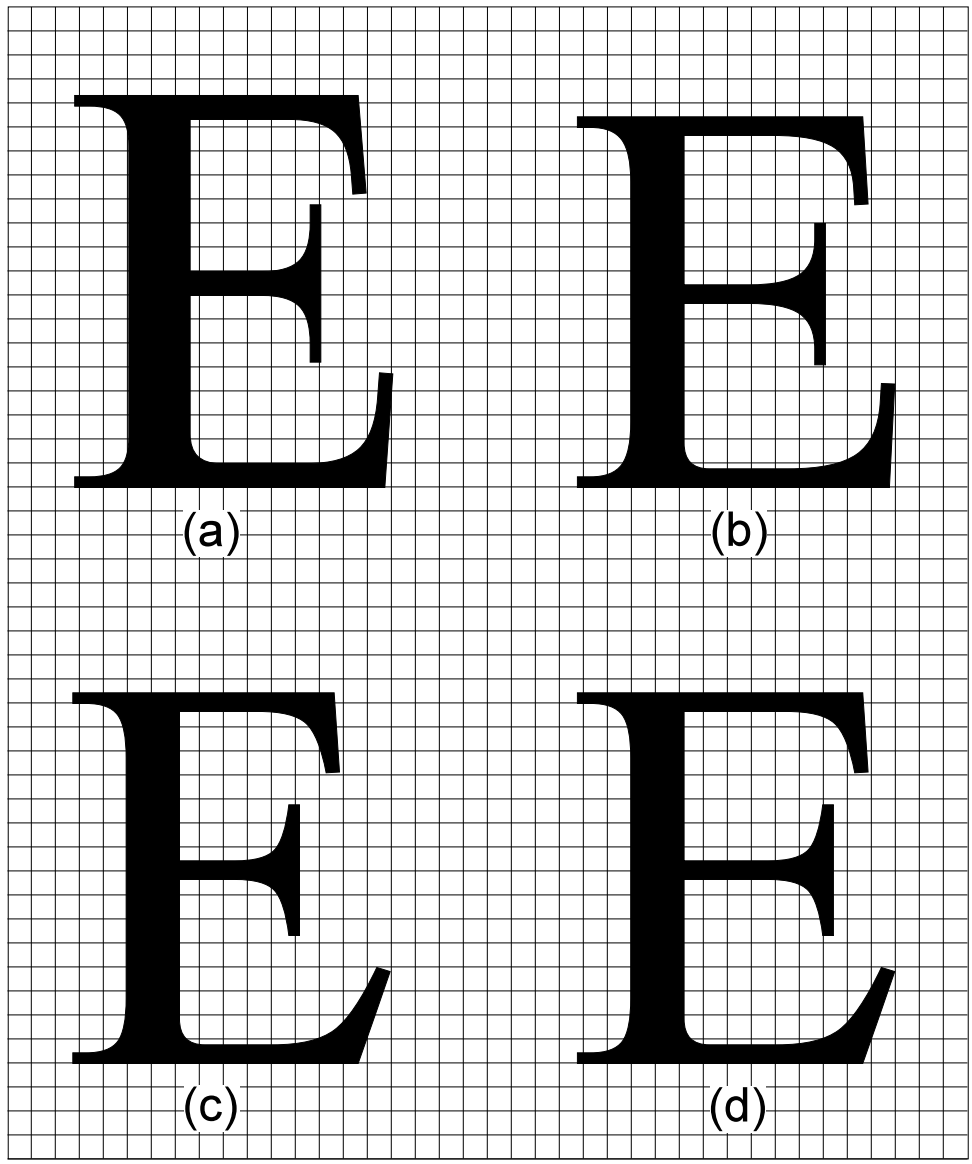

Figure 3. Times uppercase E generated by the font development system using a variety of inputs: a) a PANOSE 1 number only; b) a PANOSE 2 number only; c) a PANOSE 2 number and global overrides; and d) PANOSE 2 number, global overrides, and local overrides

In particular, we believe that a *trait-based* (or *feature-based*) system such as Infinifont will be much more successful at dramatically reducing the storage requirements of fonts containing huge character sets (such as those in Kanji alphabets) than those technologies which rely on distortable master outlines to instantiate each glyph.

REFERENCES

1. D. E. Knuth, METAFONT*: the program*, Addison Wesley, Reading, MA, 1986.
2. B. Bauermeister, *A manual of comparative typography: the PANOSE system*, Van Nostrand Reinhold, New York, 1988.