MetaFog: Converting METAFONT Shapes to Contours

Richard J. Kinch
1500 S Ocean Blvd, Suite 407
Pompano Beach FL 33062 USA
(407) 379-4480
Internet: kinch@holonet.net

Abstract

The Computer Modern Typefaces have their original specification in terms of the METAFONT language. The individual glyph programs rely on the sophisticated algebra and marking methods of METAFONT. Many of the METAFONT primitives, such as stroked pens and overlapping ink, are not directly expressible in outline typeface formats such as Type 1 and TrueType, which support only topographic contours expressed as non-overlapping Bézier curves.

We explain the computational geometry involved in the conversion from METAFONT shapes to outlines, why this is a difficult problem, and why previous efforts have fallen short. We describe MetaFog, a set of programs written to post-process METAPOST output to complete such conversions, and the algorithms implemented to solve the mathematical problems. The two most significant problems are (1) finding the envelope of an elliptical pen stroked along a Bézier curve (an algebraic problem), and (2) reducing overlapping paths to an equivalent, non-overlapping contour (a topological problem). We propose a scheme to embed Type 1 and TrueType hint technology into METAFONT sources to reduce the duplication of effort to produce well-hinted fonts. We compare the accuracy of MetaFog's analytic conversions to approximations based on auto-tracing of METAFONT's bit-mapped output, and show examples of errors in the Computer Modern Typefaces which are hidden in METAFONT proofs but visible in MetaFog proofs.

TEX and its Fonts

Modern implementations of TEX like TRUETEX® have eliminated bit-mapped meta-fonts in favor of outline formats such as TrueType or Type 1. TEX did an admirable job of producing its own font bit-maps in the days before operating systems supported fonts. But today the most popular operating systems and print engines require outline fonts. These scalable formats facilitate previewing and printing TEX documents in a powerful, portable, and flexible fashion which bit-mapped fonts cannot achieve.

While pure TEX is independent of any particular fonts, TEX is nevertheless just as dependent today on Computer Modern and other METAFONT-based fonts as ever. Thus arises the need for conversion of METAFONT programs into equivalent outline forms.

While METAFONT programs can describe a glyph in terms of complex, overlapping paths, the outline formats require that we specify glyphs as a set of *contours* (non-overlapping outlines). Herein lies the most difficult aspect of conversion: METAFONT's primitive shapes are built from third-degree parametric curves modulated by third-degree paths, and such shapes can overlap, add and subtract in arbitrarily devious ways.

Conversions: analytic versus approximate. MetaFog is a system for exact, analytic conversion of METAFONT shapes to contours. That is, MetaFog always store shapes in terms of their pure, parametric curves. By "analytic" we mean that the methods we use analyze and solve the underlying equations for the parametric curves. We use no intermediate approximations such as converting curves to polygons, so that every result curve is a direct derivation of an input curve and every input point is unchanged in the output.

By "exact" we mean that the result curves follow the METAFONT shapes to within one pixel in the 1024 or 2048 pixels/em grid used in typical outline font formats. In some cases METAFONT design envelopes cannot be represented exactly by Bézier curves, and we use this metric to determine the degree of curve-fitting needed. We use a METAFONT `mode_def` for a "perfect" output device needing no corrections for fill-in or overshoot.

Automating an analytic conversion of METAFONT shapes requires a major effort in both mathematics and software. It requires solutions to problems which Knuth managed to avoid in METAFONT by using numerical tricks and simplifications. Earlier projects have attempted the task, but either fall short of or approximate the full solution (Berry 90; Carr 87; Henderson 89).

Outline conversions of meta-fonts have also been done before using approximation techniques, thus avoiding the difficulty of an exact, analytic conversion. For example, autotracing attempts to fit an outline to a high-resolution bit-map. With enough skilled labor, autotracing yields an aesthetically pleasing result, although the shapes will tend to have certain artifactual deviations from the precise METAFONT originals. The BlueSky-Y&Y conversions of Computer Modern and other meta-fonts show that careful autotracing and hand-tuning can produce a result equal to that of a conventionally-designed commercial font.

More recently Malyshev (1994) has published the BaKoMa fonts, which contain very precise outline conversions of Computer Modern. Malyshev's publication is limited to the results (that is, the outline fonts themselves); he has not revealed the details of his technique, although he claims that it is analytic and not an autotraced or otherwise a digitized approximation. We will show below examples of font details which an analytic conversion would preserve, but which are missing from the BaKoMa fonts. Malyshev's claim of analytic perfection could nevertheless be true, if such errors were introduced, for example, by bugs in his conversion software. On the other hand, if a hidden approximation is involved somewhere in the BaKoMa conversion process, the result would not meet our strict definition of being both "exact" and "analytic". This is not to say that the BaKoMa fonts are poor conversions; it is evident that the shapes are excellent in every way important to font designers and that they are generally faithful to the METAFONT originals.

The Nature of the Conversion

Let us consider the nature of the conversions involved. METAFONT can actually do more sophisticated things than we are about to describe, but we will restrict our consideration to those METAFONT features that are actually used in typefaces like Computer Modern.

Bézier curves. We will consider Bézier (Glassner 90) contours to be our target format. A Bézier *curve* (Figure 1) is a parametric curve governed by the equation:

$$z(t) = (1-t)^3 z_1 + 3(1-t)^2 t z_2 + 3(1-t)t^2 z_3 + t^3 z_4$$

Parameter t is called the *time* along the curve and ranges over the interval $[0, 1]$; the point at time t is $z(t)$. A Bézier *path* is a set of Bézier curves which connect in a chain at their endpoints to form a more complex curve. A closed path which does not overlap describes a complete circuit and encloses an area. A set of such paths make a Bézier *contour*, which can describe the outlines of a glyph. The paths in the contour of a well-formed glyph do not intersect each other, and as well they do not intersect themselves. This is the representation used in the Type 1 font format (Adobe 90). Conversion of Type 1 glyphs to TrueType glyphs (which use lower-order parametric curves) is a straightforward conversion. In METAFONT (as documented in the literate source code), Knuth calls the Bézier paths *cubic splines* (an equivalent mathematical term), and uses a data structure consisting of knot locations and control points to specify paths. This is the terminology we use in MetaFog. In Figure 1, points z_1 and z_4 are knots, and z_2 and z_3 are control points.

The goal of MetaFog conversion is to produce Bézier outlines which accurately represent the METAFONT designs. This will be close to the minimal set of knots needed to fit the design, because both METAFONT and Computer Modern are economical in their use of reference points, and the reference points in a METAFONT program generally expand into the minimal set of knots to implement a fitted curve. Because METAFONT divides curves into octants, METAFONT's curves tend to have control points every 45 degrees or so, versus Type 1 fonts which often subtend curves of 90 or 180 degrees per control point. So in this sense METAFONT designs have *more* control points than good Type 1 designs. On the other hand, the Type 1 format mandates rules for tangents and extrema points that tend to add redundant control points to designs, so in this sense METAFONT designs have *fewer* control points than good Type 1 designs. MetaFog preserves the pure METAFONT design, such as the addition of 45 degree control points and the absence of redundant extrema points versus a likely implementation in Type 1. The final conversion code will optionally add redundant points to meet the Type 1 mandates.

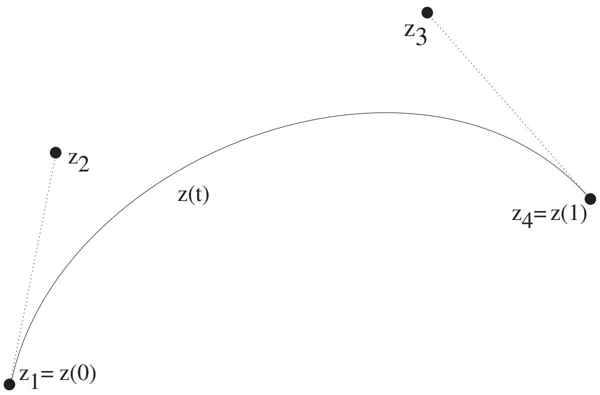

Figure 1: Bézier curve, starting at z_1 and ending at z_4. The outgoing control point is z_2, incoming is z_3.

Before METAFONT digitizes a glyph into a bit map, it represents the glyph as a collection of shapes. Each shape can be an outline determined by a set of Bézier curves or the envelope of an ellipse stroked along a path. Each shape also can add or subtract ink. This is the internal representation which we wish to reduce to an equivalent set of Bézier outlines, which are the shapes which a Type 1 font uses directly or which can be easily converted into the shapes for TrueType font.

METAFONT shapes also have color; in practice this means that we can think of each shape as either additive black ink or "white" ink that subtracts black ink already drawn. We can see that the order of drawing shapes in a glyph must therefore be preserved.

Within MetaFog we use the winding-number convention (like PostScript's) for controlling color (black versus white), while METAFONT stores an explicit color for each shape. METAFONT shapes usually, but not always, follow a consistent winding direction for the associated color. MetaFog is careful to check shapes on input so that the winding number and color are consistent. When MetaFog discovers an inconsistency, it reverses the input path.

The different models treat edges differently when rendering bit maps. We have yet to take this into account in our conversions.

Bézier tools. MetaFog uses quite a few algebraic tools to manipulate curves. Some are reimplemented or generalized algorithms from METAFONT, and some are entirely new concepts:

- Find the coordinates of a given time value on a curve.
- Find closest time on a curve to a given point.
- Audit a curve, path, or contour data structure for consistency.
- Test whether a path is degenerate (zero winding number).
- Test whether a path is redundant (contained within) with respect to another.
- Test whether a path (possibly pivoted) duplicates another.
- Test whether two paths overlap (that is, have a common segment).
- Find all intersections between two curves, associating mutual intersecting locations.
- Find all intersections in a contour, associating them with each appropriate curve in terms of time.
- Sort intersection times associated with curves in a contour.
- N-sect a curve into N curves given a set of times.
- Given an ellipse, generate a four-curve approximation.
- Given an ellipse, find the point on the ellipse at a given angle from the major axis.
- Given an ellipse and an angle of rotation, find the maximum point (horizontal tangent) on the ellipse.
- Test if two line segments cross.
- Given the parallel curves of a stroked path, stretch or shrink the endpoints to fit a given ellipse with arbitrary rotation.
- Given a curve and a rotated ellipse, return the 6 to 8 curves fitting the envelope.
- Given the positions and tangents of a curve endpoints, and a midpoint position, locate the endpoints which fit the constraints.
- Test a curve for "simpleness" (turning angle $\leq \frac{\pi}{2}$ and no inflections).
- Given an open path and an ellipse, return the envelope, reducing overlapping segments.
- Test whether a given point is on a curve (with given tolerance).
- Test whether a given point is inside a closed path.
- Test whether a given path is interior to another path.
- Find all circuits in a contour.

The above algorithms, plus syntactical and data-structure chores, make up about 12,000 lines of C program code.

Loading Shape Information from METAPOST. METAPOST (Hobby 89) relieves us of the difficult task of running METAFONT and extracting the Bézier curve information relevant to a character. We chose to have MetaFog interpret the PostScript output from METAPOST and to construct the MetaFog contour data structures during this interpretation, rather than trying to modify METAPOST to make output in a more convenient form. This allows us to stay current with METAPOST improvements.

METAPOST outputs outline curves in PostScript by first defining the path with `newpath`, `moveto`, `curveto`, `lineto` and `closepath` commands, followed by a zero-pen-width `stroke` and a `fill`. For "white" ink METAPOST uses `setgray` before stroking or filling. For elliptical pens and slanted coordinate output transformations METAPOST uses `dtransform`'s to apply affine transformations. MetaFog contains an input interpreter that converts METAPOST output to internal data structures.

Rendering ellipses stroking paths. One of the problems which Knuth sidestepped in METAFONT was computing the envelope of an ellipse stroking along a Bézier curve. Knuth here chose to use Hobby's method to compute the envelope in terms of the raster instead of scalable curves; the computational geometry then reduces to a matter of manipulating line-segments and polygons instead of polynomial curves (*The METAFONTbook*, §524).

We instead want to compute a Bézier curve outline for stroked-ellipse envelope. Algebra tells us that stroking a 3rd degree polynomial curve (the ellipse approximated by Bézier curves) along a 3rd degree polynomial curve (the Bézier curve of the stroked path) results in a 6th degree envelope curve. We will have to approximate these 6th degree exact envelope curves with 3rd degree (Bézier) curves.

Figure 2 shows how an ellipse contour may be approximated by a contour made from Bézier curves. This is similar to the four-curve approximation to a circle cited by Knuth. The Bézier control points for a unit circle are located symmetrically $\frac{4}{3}(\sqrt{2}-1) \approx 0.552$ units away from the end points. (This quantity does not appear explicitly in METAFONT, but we can solve for it by substituting the known angles and locations at the ends and midpoints of the curves.) The affine properties of Bézier curves permit us to linearly distort the Bézier control points of the unit circle in proportion to the eccentricity of a unit ellipse to fit a Bézier contour to that ellipse. We can also apply linear transformations of rotation, scaling, and translation to tilt, size, and place a unit ellipse as desired.

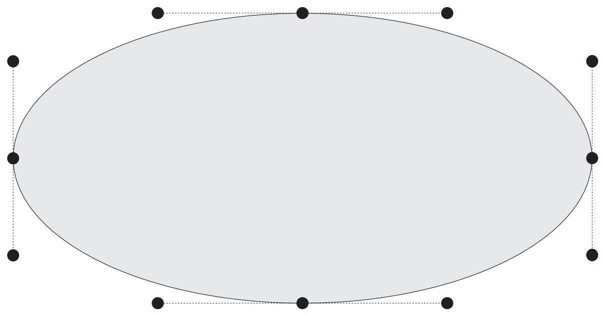

Figure 2: Contour of four Bézier curves which approximate an ellipse.

To proceed to the envelope problem, let us assume that the situation looks like Figure 3, where (without loss of generality) we have rotated and translated the coordinates such that the start of the stroking path is at the origin and has zero slope there.

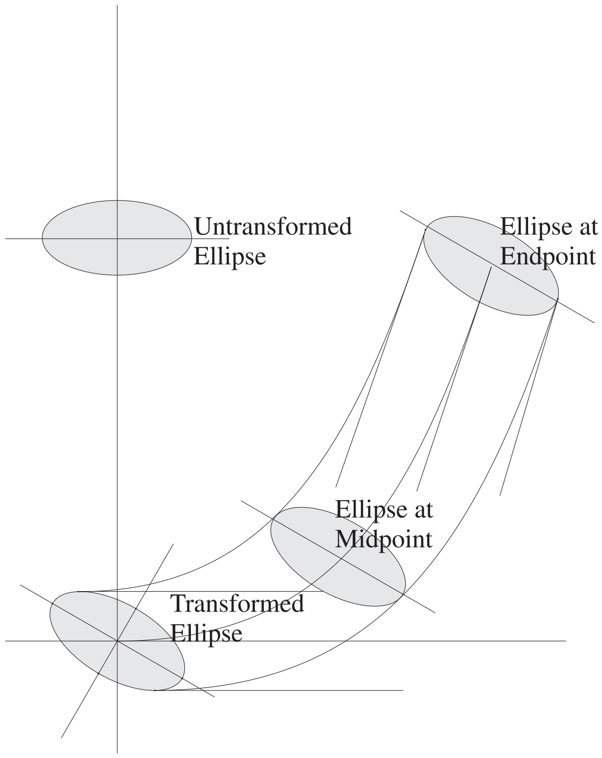

Figure 3: Stroking an Elliptical Pen on a Bézier Path.

We will fit an envelope consisting of two ends and two sides. The sides are "parallel" to the stroking path, and the ends are subsets of the ellipse at the start and finish of the stroke. We use a set of boundary conditions for the approximation which will be natural and visually appealing: The slopes of the side curves start at zero and end with the same angle at which the stroke curve ends. We fix the midpoints and angles of the side curves based on the location of the ellipse at the midpoint of the stroke, using the tangent points of the ellipse matching the angle of the stroke at its midpoint. This approximation is quite good when curves are not too "sharp"; that is, they do not turn through more than 90 degrees, and are not too "tight"; that is, they do not have a high 2nd spatial derivative. We can always bisect sharp and/or tight curves to improve the accuracy of the approximation as needed; in practice the curves are almost always so gentle as to be well-fitted without bisection.

To compute the envelope curves, we must find their endpoints and their control point locations. We first translate and rotate the coordinates of the problem to the normalized coordinate system to fit the model. Using the boundary conditions—namely the endpoint locations, endpoint tangent angles, and the midpoint locations—a bit of polynomial algebra and a solution of simultaneous equations yields a closed-form solution to where to put the endpoints and control points of the envelope curves. Given these two curves, we can compute the subset of the ellipse curves as a maximization problem in another transformed coordinate system. Inverting the rotation and translation of coordinates yields the desired solution.

Figure 4 shows some examples of envelopes computed with this method. Careful attention to generality and numerical domains yields a robust algorithm, which is crucial to the wild data characteristic of graphical shapes.

METAFONT usually uses circles (of course, a circle is a special case of an ellipse) to stroke pens. The exceptions where METAFONT uses elliptical pens are the calligraphic capitals and a few math symbols. Knuth also used circular pens quite liberally in Computer Modern. For example, circular pens draw the rectangular stems, since the technique makes parameterization of stem widths and rounding of corners somewhat easier, and the serif programs take care of squaring off the round corners for Roman faces.

The logical shape primitives OR and NOT Once MetaFog expands any stroked paths to envelopes, it can proceed to intersect overlapping paths. MetaFog must compute all possible multiple intersections of each pair of curves in a path, instead of assuming only one possible intersection as in METAFONT. (two Bézier curves can have 8 intersections–try to find an example using your favorite drawing program!). MetaFog computes all intersections using an exhaustive extension to the recursive, numerical solution Knuth used in METAFONT; a closed-form solution employing zeroes to cubic polynomials is also possible but not implemented. Computing all such intersections and reconstituting the shapes with new knots at all intersections in the general case is a difficult problem which consumes most of MetaFog's running time.

Figure 4: Envelopes computed for various Bézier strokes of an elliptical pen.

Weeding. The MetaFog weeder is a visual tool which allows a human operator to examine and hand-correct the output from automatic conversions. Manual input to the conversion process is vital, because METAFONT output often has degenerate shapes and intersections that defy an automatic solution. In such cases, MetaFog cannot determine which shapes are overlapping, and so outputs a partial solution to the topological problem; the weeder allows the human designer to choose the proper Bézier shapes from intermediate METAFONT elements.

Figure 5 shows the weeding display for character 'm' of `cmti10`. The display shows each of the Bézier curves of the input shapes, intersected and broken into separate pieces. The user has invoked MetaFog in "minimal" mode (which is guaranteed to succeed), which means that all curves used in computing envelopes of strokes are retained; an "intermediate" mode (which does not always succeed) reduces each envelope to the exterior curves. Note how MetaFog has stroked a circular pen along Bézier paths and produced curves for the envelope. MetaFog has also inserted new knots where curves intersect; this computation can be quite complex since a given curve can have arbitrarily many intersection points, resulting in a repeated bisection of the curve. The human operator uses the mouse to observe and toggle Bézier segments which make up the correct envelope of the character; each segment changes color as it toggles on (blue) or off (red). Toggling proceeds quickly because the mouse click need only be near (not necessarily on) the desired curve, clicks are buffered when the operator outpaces the CPU, and a second click will toggle off any inadvertently erroneous selections.

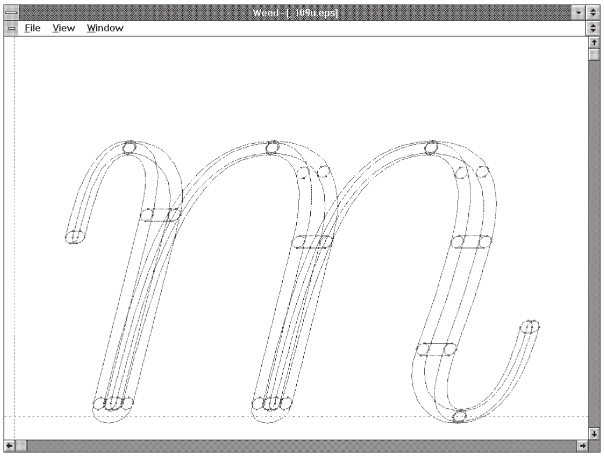

Figure 5: Weeder display for `cmti10` 'm'

The weeder's user interface is optimized for speed. The PC's numeric keypad provides convenience functions, so that the operator keeps one hand on the mouse and the other hand on the function keys. The operator can quickly switch between displaying all curves versus displaying selected curves only. Previewing selected curves only gives an accurate check that no segments are missing and that no extra segments are selected. Zoom-in and zoom-out allow the operator to pick through "busy" areas where many curves lie very close together. After toggling all the exterior edges of the glyph, the operator visually checks the glyph for proper construction, and finishes by exporting the character. During export, the weeder takes care of optimizing the output curves by removing redundant control points. Keypad functions allow the user to flip quickly through all the glyphs in a font. This allows careful previewing and weeding of any glyphs that need touch-up from MetaFog. A checkplot program produces a printout of all the glyphs in a font for checking and documentation.

In the worst case, MetaFog can always produce a fully-intersected set of shapes which elliptical pen envelopes (if any) already expanded. The operator of the weeder then has a more detailed pointing job, but the result will be just as perfect as an automatic solution.

Figure 6 shows the MetaFog weeder view of `cmsy10`'s calligraphic $\mathcal{A}$. This illustrates how a tilted elliptical pen strokes a Bézier path in slanted coordinates.

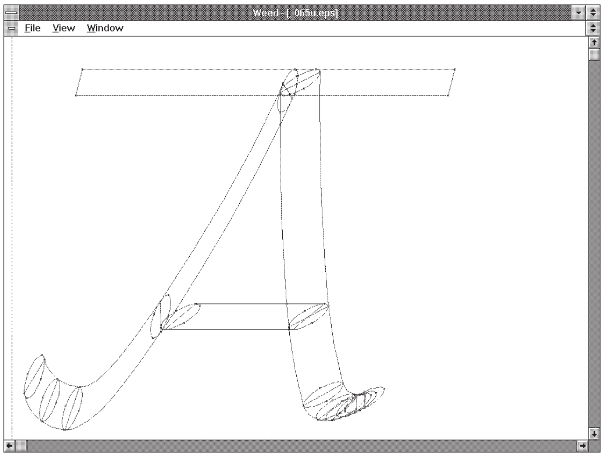

Figure 6: Weeder display for `cmsy10` calligraphic $\mathcal{A}$, illustrating envelopes of elliptical pen strokes

A few cases of the calligraphic capitals contain tightly turning curves which require hand-corrections using the weeder.

Hinting. Rendering fonts on low-resolution devices like video displays and laser printers requires heuristic help to yield a pleasing result. Without such help, the bit maps will have unnatural bumps, stems will be of uneven width, and drop-outs will occur.

METAFONT handles these matters in the chapter "Discreteness and Discretion" of the *The METAFONTbook* and via `mode_def` items like `fillin` and `o_correction`. The Type 1 font language allows the designer to add "hinting" for the same purpose. TrueType calls the same notion "instructions." Since most of the industry effort in this regard has been expended on Type 1 font hinting, "hinting" has become the generic term for this aspect of font design.

METAFONT has significant modeling differences from the outline font hinting methods, so there is no translation possible to automatically make a hinted outline font from a METAFONT design.

Hinting can be applied after the translation, either automatically by auto-hinting software (yielding a poor to modest result) or by a skilled programmer (yielding the best results, given enough expert effort). The big problem with adding hints in this "post facto" manner, is that the hints become detached from the original METAFONT programs, and any change to the meta-fonts will require repeating the manual hinting effort. The biggest loss here is that the meta-ness of METAFONT programs does not carry over to post-facto hand-hinting; one METAFONT character program makes variants like bold, italic, sans serif, etc., but each variant must be independently hand-hinted. Another important example is that most of the METAFONT programs for the DC fonts repeat programs from Computer Modern, but such redundancies would not be usable after translation to outline formats. Outline-format fonts do suffer from an inability to exploit these redundancies, and serious font designers typically have in-house tools to overcome this problem.

Since METAFONT hinting does not carry over to Type 1 or TrueType hinting, the ideal solution would be to enhance the Computer Modern METAFONT programs to contain new hinting information suitable for translation to other forms. Type 1 and TrueType hinting employ a limited number of techniques, which depend on the exact coordinates and design of each particular character. A programmer could add each hint and the associated coordinates to each character's METAFONT program in the form of pseudo-comments. A hint-translator program would convert the METAFONT pseudo-comments into Type 1 hint programs or TrueType instructions. Making the shape translation independent of the hint translation would allow adjusting shapes or hints independently, without having to re-run both aspects of the translation. The pseudo-comment language would be designed to represent the various hinting technologies and to exploit any commonality between them. The METAFONT language is well-suited to such an extension.

For example, Type 1 "flex" hinting needs to know the size and position of what is called the "dish" concavity in the Computer Modern serifs. Addition of this information to the Type 1 fonts improves the rendering of serifs. While this information is present in the METAFONT programs, it is lost in the process of translation to output shapes. The proposed method of pseudo-comments and hint-translator would preserve and translate this information. Hints are typically applied to stems, bowls, bulbs, and other character features, and METAFONT is quite aware of the pertinent coordinates of these items.

This is also a database problem. One of the difficult tasks of translating TEX fonts is the surfeit of them. Just between Computer Modern and the DC fonts, spread across various optical sizes, there are several hundred fonts each having 128 or 256 glyphs. Given that the typical glyph outline contains dozens of endpoints, each having 3 pairs of coordinates, one can see that the translation enterprise involves millions of coordinates. Organizing this information into glyph data, character names, fonts, character metrics, encodings, accent composition rules, version controls, kerning pairs, ligature rules, font families, output formats, hinting data, and so on is a substantial database problem. Since we want to exploit redundancies like common subsets between `OT1` and `T1` encodings, we especially need a capable database approach to managing this information.

MetaFog uses more of a rapid-prototype approach. Shell scripts manage the various steps in translating a given font: running METAPOST to get intermediate conversions; running MetaFog itself to convert all or part of a given font to outlines, assembling various files for a C program `makefont` which assembles individual character data into complete Type 1 fonts, including insertion of extrema points, initial production of an AFM file, and a TEX virtual font file. Tables keep track of redundancies between characters and fonts so that a given METAFONT glyph need only be translated once. Tables such as encoding vectors are typically kept in ASCII form and look-ups are performed by shell scripts. Glyph information is kept in PostScript or pseudo-PostScript form and rapidly manipulated by C programs built from common function libraries.

To finish the fonts, we use several outside utilities. The programs of Hetherington's `t1utils` collection take care of the details of conversions to and from the encrypted Type 1 font format, so that MetaFog need be concerned only with ASCII Type 1 output. We also test the fonts with all the commercial font editors currently available: *Fontographer*, *Fontmonger*, *Type Designer*, and *FontLab*; we use

Fontmonger to convert the Type 1 fonts to TrueType form.

If we were to repeat the implementation, one might consider using a relational database to store the information, with query scripts and C programs doing the detailed work.

Optical Overkill. Fonts as they are used in operating systems today do not favor the optical scaling which TEX is adept at exploiting. For example, TEX uses eight optical sizes of the Computer Modern Roman font (5–10, 12, and 17 points). This is too many optical sizes—do we really need every step from 5 to 10 points? No doubt this was encouraged by the METAFONT facility at optically scaling with a simple parameter change. But with the various embellishments of bold, italic, and so on, a minimally complete Computer Modern font set yields over 100 discete fonts.

Users today are not accustomed to seeing so many fonts associated with an application. TEX has a distinctly archaic atmosphere in this regard. Operating systems that manage fonts are taxed to handle the plethora of tiny variations in TEX fonts.

Lately this overkill of optical sizes has worsened with the NFSS, which does a good job of hiding optical sizes from the user, but encourages the style designer to multiply them.

The pain is excruciating with regard to outline translation, where essentially identical problems with slight variations are repeated many times. We would urge restraint on TEX experts when it comes to selecting optical sizes.

Comparisons of Various Approaches

Let us compare a typical Computer Modern glyph as translated to outline form by various methods. Figures 7 through 12 show the output for 'R' of `cmr10` from various conversions.

Note that Figure 7 and Figure 8 show extra, relocated, missing, or artifact control points which have lost the symmetry of the METAFONT control points. The autotracing method used is evident in these examples.

Figure 9 has retained most of the METAFONT control points but also inserted artifacts. Figures 10 and 11 show the set of true METAFONT pieces from an intermediate step, where MetaFog has expanded the circular pen strokes into their Bézier envelopes. Figure 12 shows how MetaFog retains the METAFONT control points exactly, including all the octants and all the symmetry; there are no extra or artifactual control points. In comparing Figures 9 and 12, note that the tip of the leg in the BaKoMa conversion develops an asymmetry, that the flat top of the tip has narrowed, and that the 45 degree control points are missing from the bowl and serif curves (which will underspecify these curves). The MetaFog conversion retains the proper symmetry, flatness, and precision, which are all aspects of this character readily observed in the METAFONT proof in *Computer Modern Typefaces*.

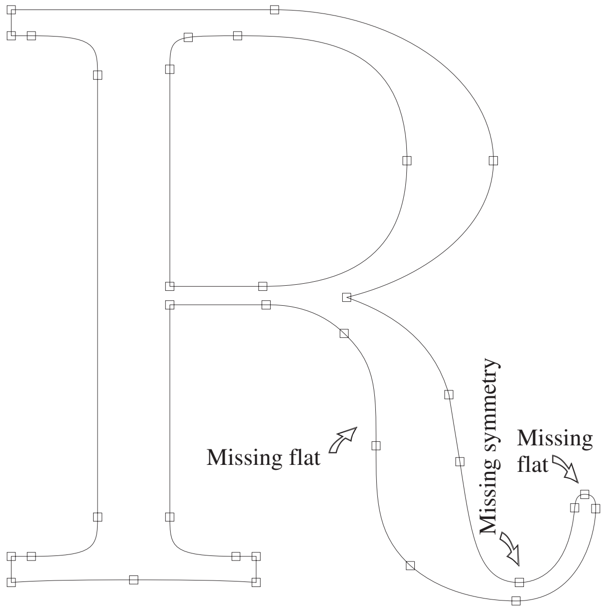

Figure 7: Blue Sky Research autotraced conversion.

X-Rays reveal bugs in Computer Modern. MetaFog allows more detailed visualization of character designs than METAFONT proofs. While METAFONT proofs show reference points and marked areas, they cannot show most of the relevant geometric information. Indeed, few of the knots, not all the outlines, and none of the stroked pen envelopes are accessible in METAFONT. Since MetaFog converts and manipulates all these items, it can also plot them in a convenient form. This yields a new and sometimes surprising "x-ray" view of a character—a view unavailable in METAFONT. MetaFog's output files use a PostScript format so that proof pictures plot on any PostScript output device. The weeder is also a convenient visual tool for such views.

Surprising aspects to some of the Computer Modern designs show up in the "x-ray's". The parametric nature of meta-designed features becomes visually apparent, and bugs in the design are clear where they were not before. Figure 13 shows how the serif subroutine has introduced an unexpected

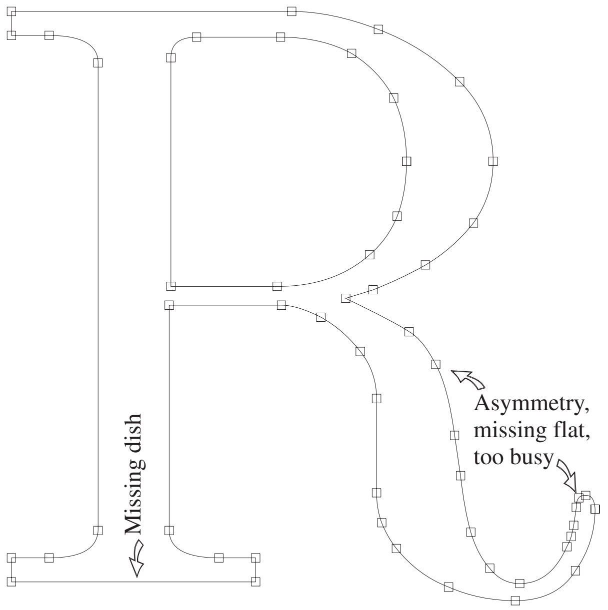

Figure 8: PC TeX autotraced conversion.

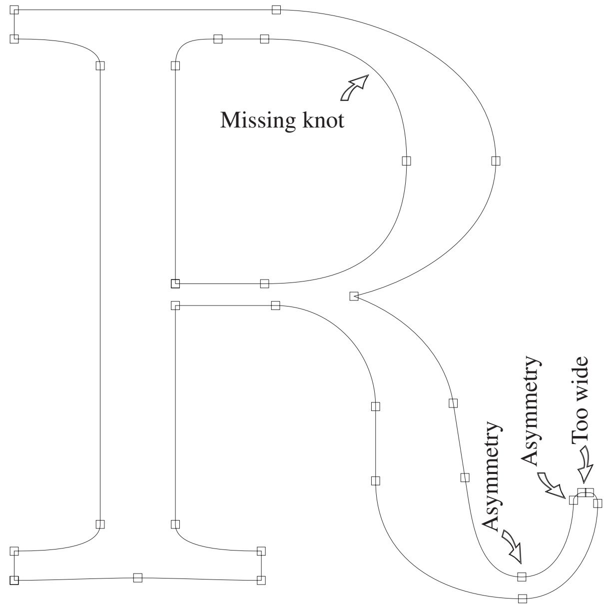

Figure 9: BaKoMa conversion.

inflection in `cmbx5` letter 'x'. Since the loss removes just a few pixels likely to be filled in physically by most marking engines and optically by the human eye, the error is not obvious in normal usage or on METAFONT proofs. It becomes clear, however, during the MetaFog weeding.

Figure 14 shows how joining of the "beak" to the arm on digit '7' becomes distorted at smaller sizes. This error is easily missed on proofs but is visible under magnification. If you magnify and carefully examine the actual-size proofs in the *Computer Modern Typefaces* (Volume E of *Computers & Typesetting*), this error is visible as a row or two of extra pixels at the top of the character.

A frank error in `dcr10`'s 'thorn' was easily discovered in this way, although it had escaped all the proof checks and actual usage for several years (Figure 15). The bottom serif's have an extra "step", which on bit-mapped proofs looks like a purposeful fillet. On the MetaFog conversion it appears clearly as an error. (This error has been corrected in the autographs pursuant to this discovery, and does not appear on more recent versions.)

Is There an Exact Translation?

Is an exact translation possible? We used the standard of 1 pixel on a 2048 pixel/em grid. No doubt the "noise" of digitization and hinting creates many more varying pixels than this standard of error. There is no outline that will render the same bit map in a Type 1 or TrueType engine as METAFONT would render for a METAFONT program. The font format itself requires that we must approximate sixth-degree pen strokes with third-degree pieces. METAFONT cannot draw proof picture of the underlying curves, it can only produce a high-resolution bit map. And finally, device-specific mode definitions in METAFONT result in significant changes to the "proof mode" device.

So there is no such thing, in a practical sense, as an exact translation, because there is no exact shape to what a METAFONT program describes! Perhaps we should instead speak in terms of an "ideal" translation.

Sample Output

A sample of MetaFog conversion, namely `cmr10` in Type 1 format, is available at:

`FTP://ftp.netcom.com/pub/Tr/TrueTeX`

This is an *unhinted* font suitable for viewing in a font editor, but not suited for textual use. MetaFog itself is a proprietary product, and is not in the public domain.

Colophon

We drafted this paper using the TrueTeX implementation of LaTeX 2_ε for Windows, which allowed WYSIWYG previewing and printing, including all graphic images. We used three kinds of figures, all processed through Corel Draw 5: ordinary drawings, MetaFog imports, and screen snapshots.

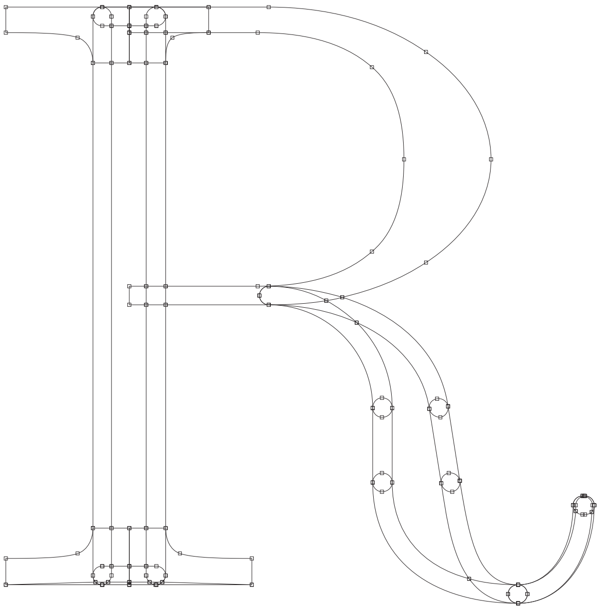

Figure 10: MetaFog intermediate step (overlaid), showing both explicit METAFONT shapes and Bézier envelopes of circular pen strokes.

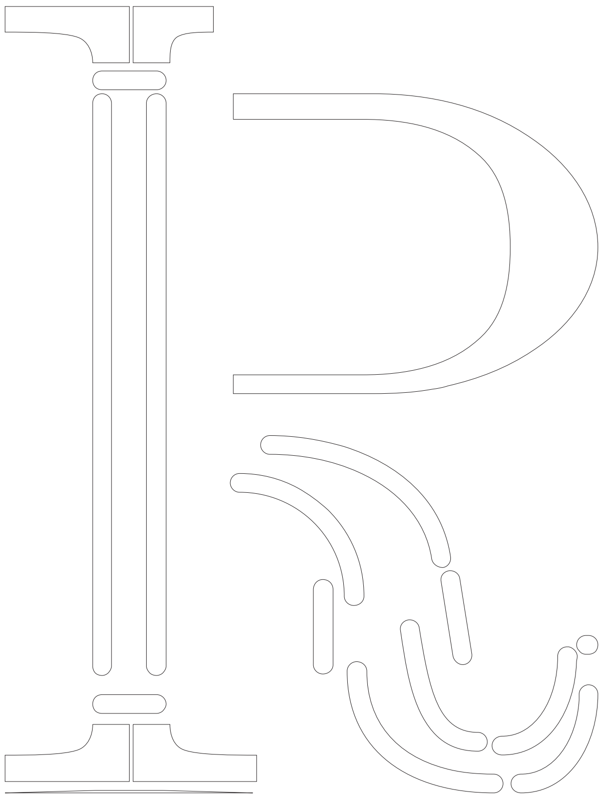

Figure 11: MetaFog intermediate step (exploded). The "dish" at the bottom is a white shape which subtracts from the serifs.

We originated figures like the Bézier curve and ellipse examples using Corel Draw's drawing tools; MetaFog-output figures by importing MetaFog's PostScript-like output into Corel Draw; and screen snapshots by capturing with Corel Capture and pasting into Corel Draw via the Windows clipboard. We used the figures in Corel Draw to export Encapsulated PostScript (EPSF) files and inserted the files as TEX figures using the `epsfig` package for LATEX. The TRUETEX `dvips`-compatible special-handlers allowed both screen previews and printing of the EPSF figures, including printing on a non-PostScript laser printer. We used Corel Draw to print overhead transparencies of the figures. Draft copies and transparencies were imaged on an HP 4M Plus laser printer. The Proceedings editors use LATEX and `dvips`, so that no conversions were necessary between the author's submission and the final production.

References

Adobe Systems Inc. *Adobe Type 1 Font Format*. 1990.

Berry, Daniel, and Shimon Yanai. "Environment for Translating METAFONT to PostScript." *TUGboat* **11** (4), pp. 525–541, 1990.

Carr, Leslie. "Of METAFONT and PostScript." *TEXniques* **5**, pp. 141–152, August, 1987.

Glassner, Andrew (editor). *Graphics Gems*. Academic Press, Cambridge MA, 1990.

Henderson, Doug. "Outline fonts with METAFONT." *TUGboat* **10** (1), pp. 36–38, 1989.

Hobby, John D. "A METAFONT-like System with PostScript Output." *TUGboat* **10** (4), pp. 505–512, 1989. See also the software on CTAN.

Malyshev, Basil K. "Automatic conversion of METAFONT fonts to Type 1 PostScript." Proceedings of the 15th Annual TEX Users Group Meeting, 1994. See also the software on CTAN.

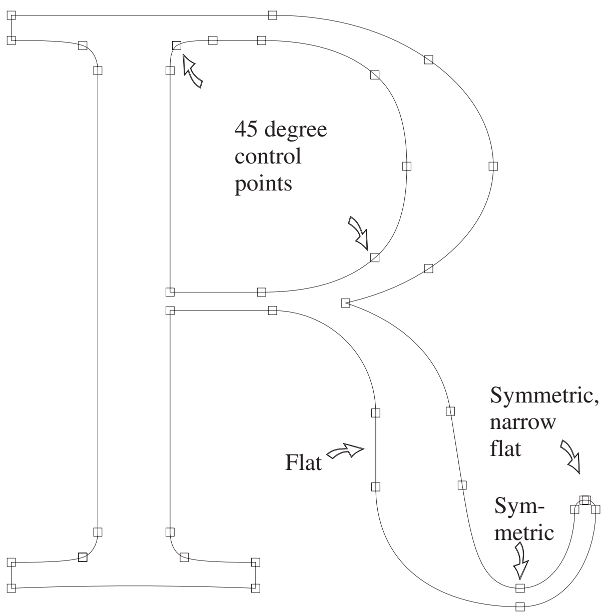

Figure 12: TrueTEX conversion via MetaFog.

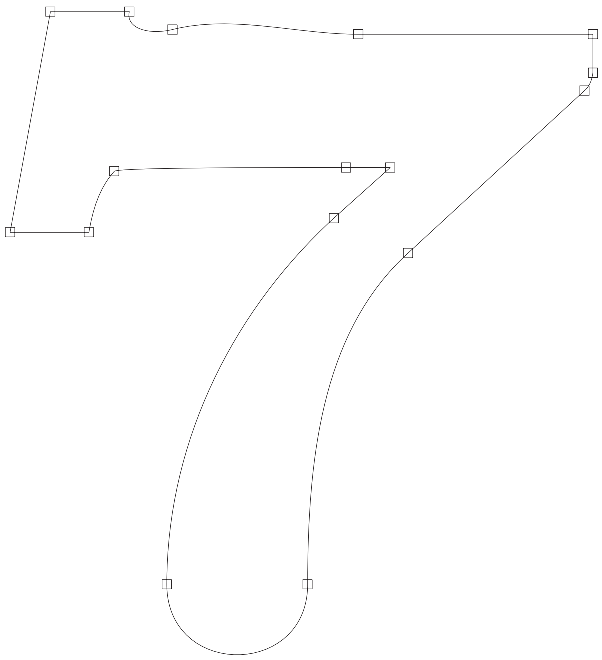

Figure 14: Error in CM digit 7 (`cmbx5`)

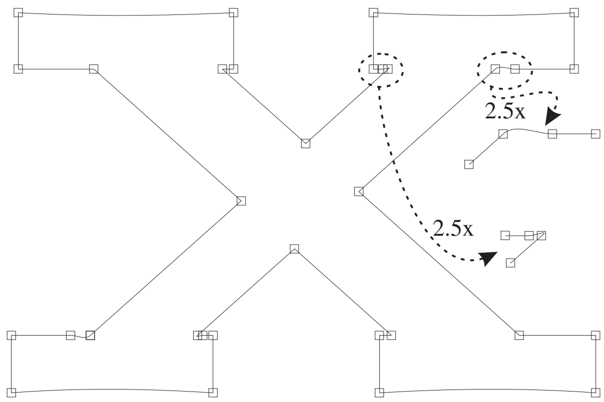

Figure 13: Error in CM serifs (`cmbx5`)

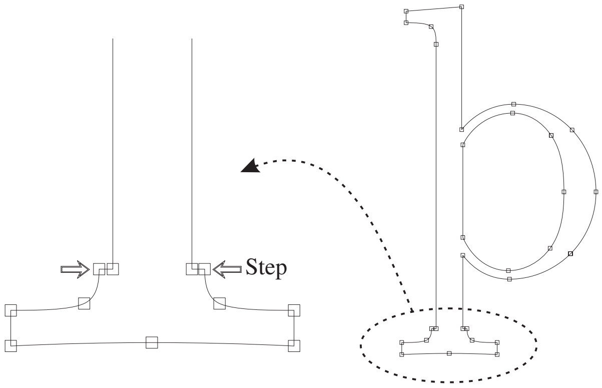

Figure 15: Error in DC thorn (`dcr10`)