Example-Based Hinting of TrueType Fonts

Douglas E. Zongker[1,2] *Geraldine Wade*[1] *David H. Salesin*[1,2]

[1]Microsoft Corporation [2]University of Washington

Abstract

Hinting in TrueType is a time-consuming manual process in which a typographer creates a sequence of instructions for better fitting the characters of a font to a grid of pixels. In this paper, we propose a new method for automatically hinting TrueType fonts by transferring hints of one font to another. Given a hinted source font and a target font without hints, our method matches the outlines of corresponding glyphs in each font, and then translates all of the individual hints for each glyph from the source to the target font. It also translates the control value table (CVT) entries, which are used to unify feature sizes across a font. The resulting hinted font already provides a great improvement over the unhinted version. More importantly, the translated hints, which preserve the sound, hand-designed hinting structure of the original font, provide a very good starting point for a professional typographer to complete and fine-tune, saving time and increasing productivity. We demonstrate our approach with examples of automatically hinted fonts at typical display sizes and screen resolutions. We also provide estimates of the time saved by a professional typographer in hinting new fonts using this semi-automatic approach.

CR Categories: I.7.4 [Document and Text Processing]: Electronic Publishing

Keywords: automatic hinting, digital typography, gridfitting, shape matching

1 Introduction

The demand for high-quality hinted fonts is outstripping the ability of digital typography houses to produce them. Hinting is a painstaking manual process that can only be done well by a handful of highly skilled professionals. It requires a blend of typographical artistry with technological ability. In order to provide a full appreciation of the hinting problem, we begin here with a review of how digital fonts are scan-converted onto a raster display.

In digital typography, each character in a font is described by a set of *outlines*, usually represented by splines. When the character is rendered onto a grid of pixels, the outlines are scaled to the desired size, and then each pixel whose center lies inside of an outline is set to black. When fonts are displayed at sufficiently high resolutions this approach works beautifully. But for sizes below about 150 ppem,[1] severe aliasing problems can result when this naive outline filling process is applied, especially for delicate features such as serifs. Figure 1 shows an example. The left image is generated by the naive algorithm. This pixel pattern does not look much like a lowercase 'a'. A simple *dropout control* mechanism added to the fill algorithm turns on additional pixels to preserve the character's topology, resulting in the center image. The right image, though, shows the work of an experienced hinter. The pixel pattern has been subtly altered to both improve readability and better preserve the character of the original outline.

[1]Hinters express font sizes in *pixels per em*, or *ppem*. This measure counts the number of device pixels in the em of the font. In traditional typography, the *em* of a font was the height of the metal block of type. With digital typography, there is no actual metal block to measure, but the digital outlines are still expressed in coordinates relative to this hypothetical size. The *point size* of text refers to the size of its em expressed in points (a point is 1/72 of an inch). Thus, "12-point text" corresponds to 12 ppem on a 72 dpi screen, or 100 ppem on a 600 dpi printer.

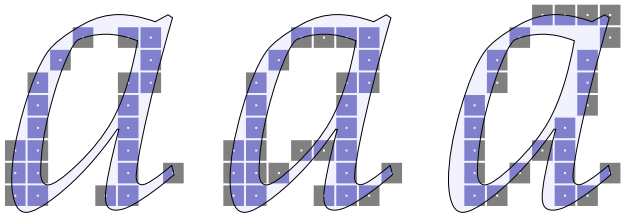

Figure 1 Outline for the Palatino Italic 'a', along with the pixel pattern generated by rasterizing the outlines for display of 18-point text on a 72 dpi device. The left image shows the results of the naive fill algorithm. The middle image shows the result of enabling the rasterizer's dropout control mechanism. The right image shows the results of hinting the character by hand.

The hinting process is not just about optimizing individual characters. The hinter must balance the needs of a single glyph with the desire for *consistency* across all the characters of a font. It is important, for example, to ensure that all the vertical stems of a font are the same number of pixels wide at a given size. If the scaling and rounding process produced one-pixel-wide stems on some characters and two-pixel-wide stems on others, then a passage of text would look blotchy and be difficult to read. The goal of the hinter is to produce a smooth transition from very high sizes, where merely filling the outlines suffices and hinting is unnecessary, down to lower sizes, where legibility must be preserved even when that means a departure from the outlines drawn by the original font designer.

Although the ever-improving resolution of hardcopy devices is beginning to approach the point at which hinting is not necessary, the technology is not there yet: 10- or 12-point text on a 300 or even 600 dpi printer still needs hinting for best results. More importantly, the increasing emphasis on reading text on-screen—from visions of the "paperless office" to the emergence and proliferation of handheld computers and eBooks—means that more and more text is being viewed on devices in the 72–100 dpi range. Though resolutions of these displays are improving as well, for the foreseeable future hinting will be an absolute necessity in order to provide clear, legible text.

Although attempts have been made to design automated hinting systems in the past [2, 5], even the best of these produce hints that are good, but still not up to the standards of professional typographers. This previous work assumed that in order to be useful, an autohinter had to be a monolithic, self-contained package: outlines in, quality hints out. That is an admirable goal, and it may be achieved someday. However, given the detailed, aesthetically-based nature

of the work, we think that it is currently more useful to view the autohinter as one piece of a system that includes a human hinter.

The subject of this paper, then, is not a tool for automatically *generating* hints so much as a tool for automatically *translating* hints from one font to another. An important advantage of this approach is that it preserves the basic strategy and structure of the original hints, which were hand crafted by a professional typographer for each individual glyph of the font. Generally, these translated hints provide an excellent starting point for a human to fine-tune and adjust. We demonstrate our approach with examples of automatically hinted fonts at typical display sizes and screen resolutions. We also provide estimates of the time saved by a professional typographer in hinting new fonts using this semi-automatic approach.

2 Background

There are two major font standards in widespread use today: Type 1 and TrueType. Type 1 fonts [1], often called "PostScript fonts," were developed by Adobe and are popular in the world of publishing. Printing applications were the target when this system was developed, though utilities are now available to enable on-screen display of Type 1 fonts. The TrueType format [3], originally developed by Apple, was intended to unify type on the screen and on paper, and is used in both the Macintosh and Windows operating systems. TrueType has something of a reputation for being of low quality, but this is mostly due to the fact that TrueType was always an open standard while Type 1 was not, and so the public domain is flooded with a large number of poorly designed, unhinted TrueType fonts. The TrueType standard does contain extensive facilities for high-quality hinting, though, and more and more quality fonts are now available in TrueType.

Though both formats represent characters as spline-based outlines, the hinting styles are radically different. Hinting for Type 1 fonts works by marking sections of the outline as corresponding to particular typographic features of the character—stems, bowls, counters, and so on. It is the job of the rasterizer to take advantage of these *hints* about the character shape to produce the best possible pattern of pixels. This scheme has the advantage that enhancements to the rasterizer can produce improvements to all fonts on the system, but means that a designer of digital type cannot specify exactly what an outline will look like when rendered at a given size.

The TrueType font technology takes a different approach. Instead of leaving control over the glyph's final appearance to the rasterizer, a TrueType font contains explicit instructions about how particular control points should be shifted to fit the pixel grid. These instructions take the form of a *program* in a special, TrueType-specific bytecode language. Since both the behavior of each instruction and the rasterizing algorithm are defined in the TrueType standard, the designer of a TrueType font can predict exactly which pixels will be turned on for a character at a given size, no matter what the output device is.

In TrueType, each contour of an outline is specified with a sequence of point positions. (See the outline curves of Figure 3 for some examples.) Each point is flagged as either *on-curve* or *off-curve*. TrueType defines the outline as follows:

- Two successive on-curve points are connected with a straight line segment.
- When an off-curve point falls between two on-curve points, the three are treated as the control points for a quadratic Bézier segment.
- When two adjacent off-curve points appear, the midpoint of the segment connecting them is treated as an implicit on-curve point between them, allowing reduction to the case above.

The glyph renderer starts by scaling the outlines to a particular size, then executing the attached program to shift control points around in a size-specific way before filling the altered outline. By itself, this approach cannot produce the necessary consistency among different characters of a font, or even between different parts of the same character, since each action is necessarily local. Global synchronization of outline alterations is achieved through use of the *control value table*, or CVT. This is a shared table of distances, which can be referenced by instructions in each glyph's program. When the rendering is initialized for a given size, the values in the CVT are scaled and rounded to the current grid size. Point movements can then be constrained by CVT entries. For instance, a person writing hints for TrueType may decide to use CVT 81 to represent, say, the width of vertical black stems in lowercase letters. He or she will then write instruction sequences for all appropriate lowercase letters, all referring to CVT entry 81, so that all the vertical black stems at a given size will have the same width.

The TrueType language is an assembly-style stack-based language. The intent of the designers of TrueType was not to make typographers learn and write in the TrueType language itself, but rather to facilitate the development of high-level languages and tools that generate TrueType code. The Visual TrueType (VTT) package from Microsoft [7] is such a tool. VTT provides a high-level language, called *VTT Talk*, for expressing relationships between points. VTT Talk provides statements for expressing the following classes of hints:

- *Link* constraints: the vertical or horizontal distance between a pair of knots is constrained by an entry in the CVT.
- *Dist* constraints: the "natural" vertical or horizontal distance between a pair of knots is maintained, so that if one point is moved the other moves in parallel.
- *Interpolate* constraints: a knot's fractional distance between two parent knots is maintained.
- *Anchors*: specific knots can be rounded to the nearest gridline, or to a gridline specified by a CVT entry.

These types of hints are demonstrated visually for two characters from the Georgia Roman font in Figure 6. The VTT Talk hints are compiled into a TrueType program stored in the font file. One advantage of working with VTT Talk is that each statement simply asserts a relationship between two points, and there is little dependence on the order of the statements. If one statement is omitted, the meaning of the others is unchanged. In contrast, TrueType assembler is a sequential language that maintains a fairly complex state. Most instructions in TrueType have side effects that modify this state. If we tried to translate the assembler code directly, and were for some reason unable to translate a particular instruction—for instance, due to a sufficiently large difference in the matched glyphs' outlines—the effects of subsequent instructions could change entirely.

Our approach is primarily motivated by the work of Hersch and Betrisey [4, 6]. In their method, hints are generated for each glyph by matching its outline to a human-constructed generic model of that character's shape (for example, a generic uppercase roman 'B'). The model consists of two representations of the generic character shape. The *skeleton* model builds the character out of solid parts, labeled as stems, bowls, serifs, and so on. The *contour* model is an outline representation of the character, constructed to have as few control points as possible while still spanning the space of possible character shapes. The correspondences between the two models are known, being specified by hand when the model is built. In their method, the outlines of the glyph to be hinted are matched to the corresponding contour model by a fairly complex process that takes into account both global and local features. Points are classified by their position relative to the baseline, cap-height and x-height lines, and left and right sidebearings. Local features distinguishing points are based on the curvature, direction, and orientation of the adjacent curve segments. Once the correspondence between the unknown

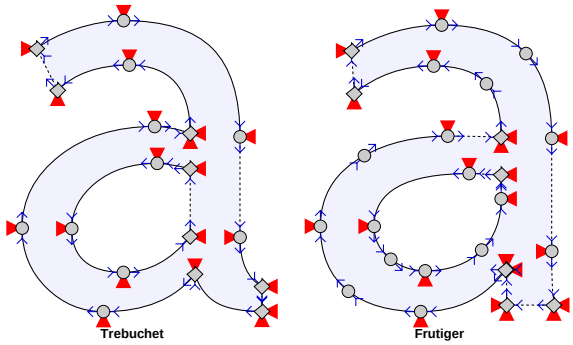

Figure 2 Features used for matching on-curve points. Diamonds indicate corner points; circles indicate smooth points. Incoming and outgoing directions are quantized to the eight compass directions, indicated with blue arrows. Local extrema are shown with red triangles. Each point is also marked to indicate whether the segments on each side are curved (solid lines) or straight (dashed lines).

outline and the model outline is established, the known correspondence between the model outline and the model skeleton can be used to label parts of the unknown outline as belonging to significant features such as stems and serifs. From this labeling a set of Type 1-style hints for the new outline can be derived.

3 Method

Hersch and Betrisey's work requires a manually constructed model in order to link points on the outline with the "semantic" features needed for hinting. Hinting in TrueType does not require an explicit labeling of these features; this information is implicitly used by the human typographer when deciding on a hinting strategy for the character, but the end result expressed in the font is just a set of relationships, or constraints, between control points. These constraints obviate the need for the skeleton model—once we find the correspondence between a contour model and the outlines of the target glyph, we can immediately produce hints for the target outline without transitively applying a second correspondence.

We've therefore reduced our needs to having a contour model with control-point-level hints attached to it. A shortcut now becomes obvious: use an already hinted TrueType font as the model! This has a number of advantages over using a specialized model built expressly for the auto-hinter. First, we already have a wide variety of fonts from which to choose as templates. Moreover, choosing a template close to the target font will increase the likelihood of a good match and consequently the quality of the resulting hints. This raises the possibility of having the template font be selected from the library automatically, or even choosing different template fonts for different characters of the target. Another advantage of using real hinted fonts as templates is that typographers, rather than computer scientists, can build templates using tools they already know; furthermore, each typographer can build templates to suit his or her own hinting style.

3.1 Matching the outlines

Suppose now that we have two glyphs representing a single character. One, the *source outline*, will be the hinted character that we are taking hints from. The goal is to translate those hints to refer to control points on the *target outline.* In the illustration here we'll show the process of translating hints from the lowercase 'a' of Trebuchet to the 'a' of Frutiger.

Our algorithm attempts to match up explicit on-curve knots using features such as contour direction and the presence of extrema. The on-curve knots typically have far more significance to the shape and extents of the contour. Once a match is computed between the

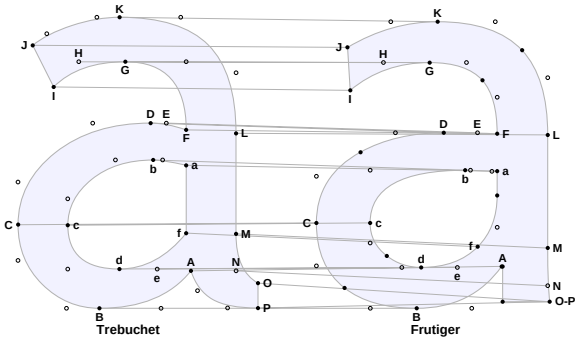

Figure 3 The final match for the two 'a' glyphs. On-curve knots are solid dots; off-curve knots are open circles.

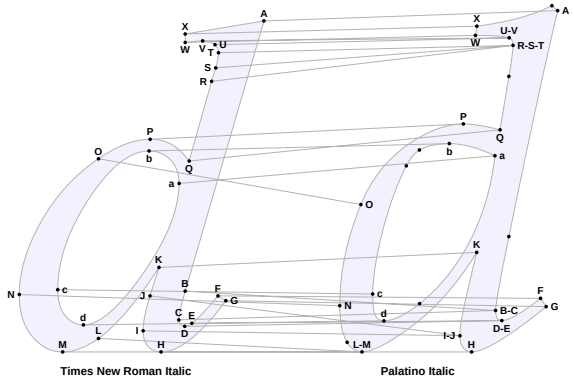

Figure 4 Results of matching for a more complex pair of glyphs. No matches involve off-curve knots in this example, so these knots are not shown.

on-curve knots, we attempt to pair up the remaining knots by simply counting the number of off-curve knots between each pair of matched on-curve knots. If the numbers are equal, we pair the off-curve knots based solely on their order. Only a very small fraction of hints involve these off-curve knots, but we want to preserve as many of the source hints as possible.

Many glyphs are defined by multiple contours, but there are no restrictions on what order the contours are listed in. Therefore, our first task is to determine which contour goes with which in the two glyphs. We do this be enumerating all the possibilities for a one-to-one pairing of the contours. (The hinter rejects input outline pairs with differing numbers of contours.) For each pairing we calculate a score as follows. Suppose that the target character is scaled and translated so that its bounding box is equal to that of the source character. For each individual contour within the characters, we sum together the absolute values of the differences between corresponding sides of the *contour* bounding boxes. This value, summed over all the contours gives the score for the match, with the lowest value being the best match. While this is a factorial-time algorithm, we have not found the running time to be a problem—for the Latin character sets we have been using it is rare to find a character with more than five contours.

The next step, the heart of the algorithm, is to match up the knots on each pair of contours. We begin by identifying a number of features at each knot, and assigning a point score for matching that feature:

1. Each knot has an incoming and an outgoing *direction*, based on the tangents of the curves touching that knot. The direction is quantized to one of eight possibilities, corresponding to the eight compass directions. A pair of knots is assigned from 200 to -200 points based on the similarity of each direction. For example, a knot with an incoming direction of "north," gets 200

points when matched with another “north” knot, 100 points for a “northeast” or “northwest” match, 0 points for “east” or “west”, -100 points for “southeast” or “southwest”, and -200 points for matching “south” knot. This score is calculated for both incoming and outgoing direction.

2. Each knot can be flagged as a local minimum or maximum in each of the x or y directions. A knot with one of these flags will contribute 150 points when matched with a knot with the same flag, or -150 points when matched to the opposite flag. A knot may not be an extremum at all in a given direction, in which case any match will not produce a score for this category.
3. Finally, each knot has a flag to indicate whether the incoming and outgoing lines are straight (within some tolerance) or curved. Matching these flags produces a score of 100 points, but not matching them produces no penalty.

Figure 2 shows our two ‘a’ characters, marked with the features used for matching.

To generate these matches, we pick an arbitrary starting knot on each contour to be matched, and pair these knots. We then go around the source contour, pairing each knot with the knot on the target curve whose fractional arc length relative to the starting knot is closest to that of the source knot. This generates a match with one pair for each source knot. We can sum the local-feature score of each pair to rate the quality of the overall match. We generate a match using each knot on the target outline as the starting point. The five matches with the highest local-feature scores go on to the next stage.

In this final stage, we attempt to improve the scores of these five best matches by small perturbations of the pairings. We remove knot pairs with a negative local-feature score, look for matches for unpaired source knots, and shift existing pairs to adjacent target knots, all subject to the constraint that the match respect the ordering of knots around the contour: if knot B follows knot A in the source contour, then the partner of knot B should not come before the partner of knot A on the target contour. Once we’ve performed this local improvement on each of the five top matches, we select the match with the highest final score as our final match.

The results of this matching algorithm are shown in Figure 3. These heuristics work well for a wide variety of character styles, including roman, bold, and italic characters. A matching for a more complex pair of glyphs is shown in Figure 4.

3.2 Hint translation

Having produced a match between the knots of the source character outline and those of the target outline, we’re now ready to translate the hints themselves. We parse the source font’s VTT Talk hints and copy them to the target font, replacing knot numbers as appropriate according to our match. If we do not have a match for a knot referenced in a particular statement, we simply copy the source statement unchanged, but comment it out, to mark it as a place that may need special attention by a person reviewing the font.

3.3 CVT translation

The CVT is a central feature of the TrueType hinting mechanism, and no TrueType autohinting scheme would be complete without addressing it. In VTT Talk, entries of the CVT are used via statements such as `YLink(14,0,87)`, which says, in effect, “move knot 0 up or down so that its vertical distance from knot 14 is equal to CVT entry 87.” Our matcher allows us to translate the references to specific knots to their analogues in the new font, and we can certainly keep using the same CVT entry numbers as in the original font. The question is, what *values* do we put in those entries? The old entries tell us little, since they represent distances measured in the source font, which may bear little or no relation to distances in the target font.

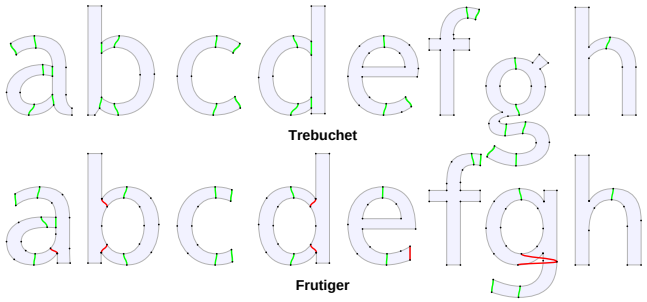

Figure 5 The top row shows characters from the font Trebuchet. The typographer has used CVT entry 87 to control the height of round, black features in lowercase letters, indicated by the green links between control points. The bottom row shows Frutiger, along with the uses of CVT entry 87 as transferred from Trebuchet by our autohinter. Red lines indicate where hints were automatically discarded because the natural distance between the points was too different from the value in the CVT table.

glyph	references to CVT entry 87					
‘a’	75*	143	143	156	156	164
‘b’	111*	113*	156	156		
‘c’	156	156	160	172		
‘d’	111*	113*	156	156		
‘e’	156	156	193*			
‘f’	155	156				
‘g’	45*	45*	111*	156	156	178
‘h’	156					

Table 1 Some of the references to CVT entry 87 when translating Trebuchet hints to Frutiger. For each pair of points whose vertical distance is constrained by this CVT entry, the natural distance between the points in the Frutiger glyphs is listed. The value given to entry 87 is the median of these natural distances, 156 units in this case. The starred values are outliers.

The solution comes from recognizing that the major reason the CVT is used is to take a set of distances that are *approximately* the same in the outline, and force them to be *exactly* the same number of pixels in the rendered bitmap. Since the goal is to provide this consistency while changing the outlines as little as possible, the CVT entry will generally contain some average value, which is close to all the distances it is going to be used to constrain. We can look at all the uses of a particular CVT entry to estimate what its value should be.

Let’s look at how this works on our ‘a’ character. The person hinting Trebuchet chose to use CVT entry 87 to represent the height of round, black features in lowercase characters. Accordingly, most of the lowercase letters that have round parts reference CVT 87, as we see in the top row of Figure 5. The ‘a’ glyph alone uses entry 87 six times—that is, there are six pairs of knots in the ‘a’ whose distance is constrained by CVT entry 87.

Table 1 shows the “natural” distances between each of these pairs in Trebuchet for characters ‘a’ through ‘h’. One pair of points in the ‘a’ is 75 units apart vertically in the unhinted outline, another is 143 units, and so on. To determine the overall value to place in the CVT entry, we take the median of all these individual guesses, which in this case is 156 units. The starred numbers in the listing indicate those uses of the CVT entry where the natural outline distance differs by more than 20% from the median value. We label these *outliers*, and we remove (comment out) the hints corresponding to these uses during the translation process, as they usually represent cases where the shape of the target character differs enough from that of the source character that the CVT constraint is inappropriate. These commented-out constraints correspond to the red lines in the lower row of Figure 5. Note that Trebuchet has a so-called *spectacle g*, while Frutiger has a *multi-story g.* In this case, it is likely that two the forms of the ‘g’ require entirely different hinting strategies, since many of the hints of the source ‘g’ are simply not appropriate for the target character shape. These inappropriate

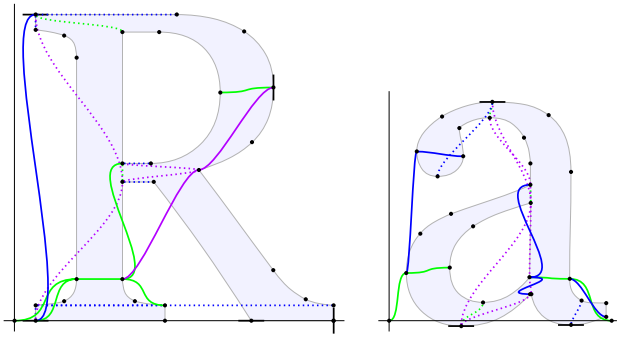

Figure 6 A visualization of the VTT Talk hints created by a professional hinter for two characters of Georgia Roman. *Link* constraints are shown in green, *dist* constraints in blue, and *interpolate* constraints in purple. Solid lines indicate x-direction constraints, while dashed lines indicate y constraints. *Anchors* are indicated with small "wings" on the anchored knot.

source font	target font	success rate (%)	review & cleanup (min.)	manual hinting (min.)	savings (%)
Sylfaen Sans	Sylfaen Sans Bold	84%	5.9	9.4	37%
Georgia Italic	Georgia Bold Italic	86%	6.7	7.9	15%
Georgia Roman	Georgia Bold	93%	4.6	7.1	35%
Georgia Roman	Bodoni	78%	3.3	3.3	0%
Georgia Roman	Calisto	74%	3.0	4.3	30%
Georgia Roman	Perpetua	76%	1.2	2.7	56%
Georgia Roman	Revival	82%	1.3	2.3	43%

Table 2 Times for hinting a sample of representative characters, both starting with the autohinted font and starting with no hints at all.

hints are automatically discarded by the outlier mechanism. Only *link* constraints, which reference the CVT, are eliminated. Other types of hints do not refer to the CVT, and so are never discarded as long as there are matches for the points they constrain.

4 Results

Our program takes two TrueType fonts as input: a source font, from which the hints are transferred; and a target font, which is hinted by the program. The program takes under a minute to match the outlines, translate the hints, and create the new CVT for a 256-character font. Once the target font is hinted, it still needs to be reviewed by hand and corrected by an experienced typographer. Even minor errors in the translated hints or CVT can take a considerable amount of time to identify and correct, so the translation has to be highly accurate in order to be useful.

Figure 7 shows how the set of manually-defined hints for two glyphs from Georgia Roman, 'R' and 'a', have been automatically transferred to five different fonts. Figures 8 and 9 compare the unhinted versions of Sylfaen Sans Bold and Georgia Bold, respectively, to the versions hinted automatically, at 16, 17, and 19 ppem, the most commonly used on-screen sizes. In these examples it is clear that most of the objectionable artifacts in the unhinted versions have already been corrected by the automatic hinting. Note, for instance, the improved 'O' shapes and the much more uniform stem weights in both fonts. Still, the autohinted versions are not perfect; note for instance where the bowl of the Georgia Bold 'b' has narrowed unacceptably, especially at lower sizes. Imperfections like these will need to be corrected by hand.

We evaluated our method by using the program to transfer hints between three pairs of fonts within the same family (Sylfaen Sans Bold from Sylfaen Sans, Georgia Bold from Georgia, and Georgia Bold Italic from Georgia Italic) as well as four target fonts from a source font of a different font family (Bodoni, Calisto, Perpetua, and Revival—all from Georgia). Table 2 summarizes the results of

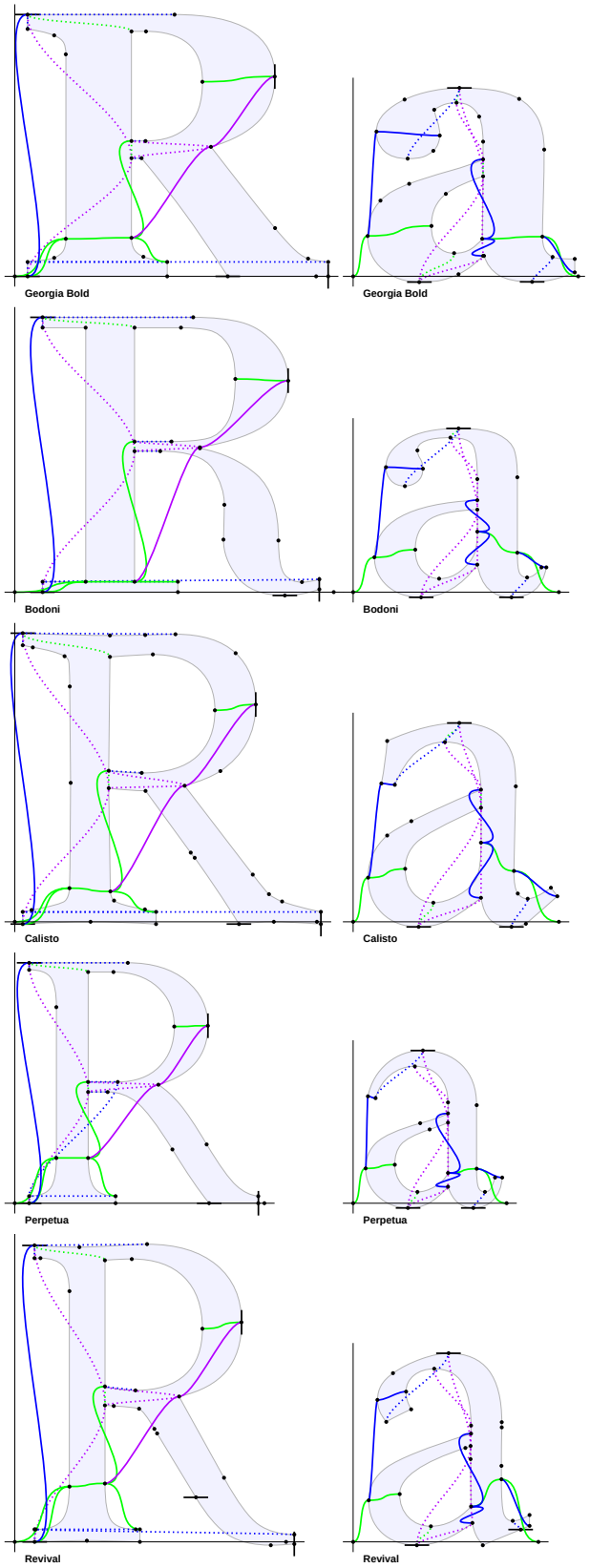

Figure 7 Visualization of hints transferred from the Georgia Roman of Figure 6 to five other fonts.

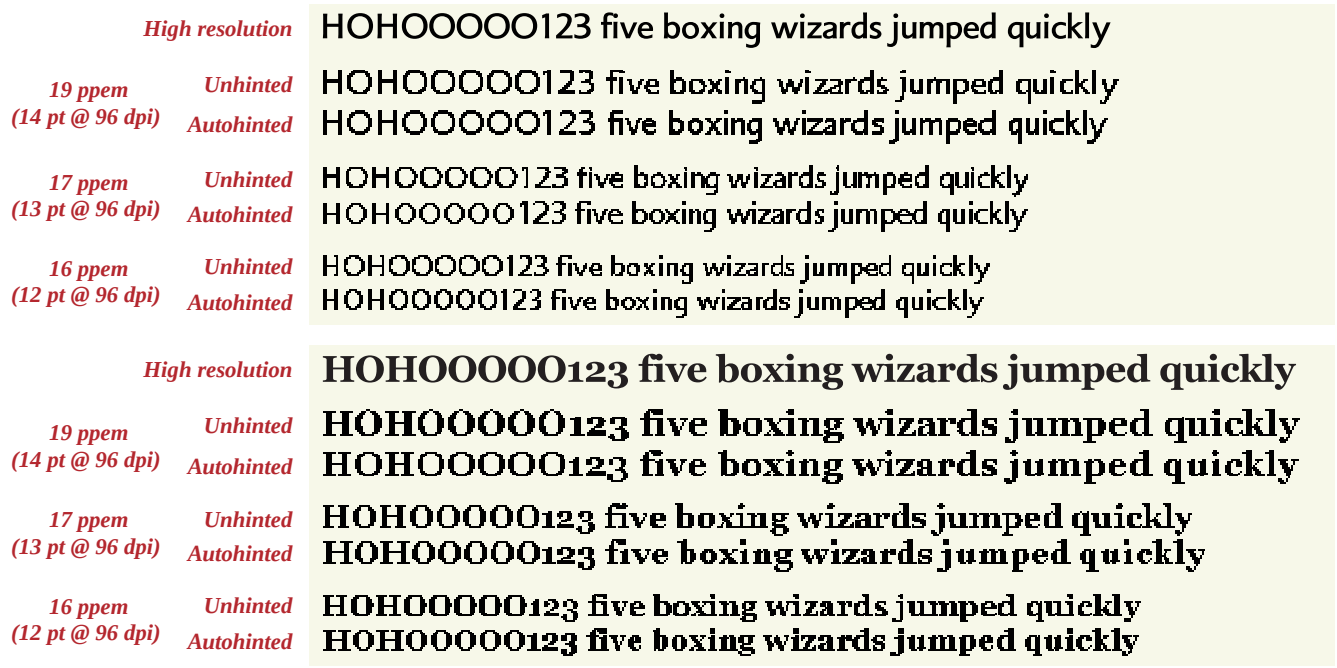

Figure 8 Sylfaen Sans Bold (top), and Georgia Bold (bottom), each autohinted by transferring hints from the corresponding roman typeface.

these tests. In each case, just the alphanumeric glyphs were hinted.

The "success rate" column gives the percentage of these 62 glyphs in which the transferred hints basically worked. More specifically, for a "successful" glyph, the overall appearance of the glyph conformed to the original outline at high sizes (38 ppem and above) without any stretching or distortion, whereas below 38 ppem there might be some cleaning up to do, but no major reshaping or rethinking of the hints. If a glyph did not conform to its original outline at high sizes or required major reshaping at low sizes, then it was considered "unsuccessful." As can be seen from the table, the hinter had a fairly high success rate by this measure, especially when hinting characters within the same font family.

The next column gives an estimate of the number of minutes required for an experienced typographer to review the results of the autohinter and clean up any problems in the transferred hints. The figures in this column were estimated by performing this process on some 3 to 11 representative glyphs in the target font. These same glyphs were also manually hinted by the same typographer and the times required reported in the following column. Finally, the rightmost column provides an estimate of the overall time savings provided by the example-based hinter.

Note that the very high success rate of the hinter translates into a more moderate overall time savings, since even a perfectly-hinted font requires time to review, and since a few small problems in the hints can be time-consuming to correct. Still, these savings are significant, considering that a full font of 256 characters can take on the order of 20–40 hours for a skilled professional to produce.

5 Conclusion

We have adapted the earlier work of Hersch and Betrisey on automatic hinting through shape matching to create a useful production tool for hinting TrueType fonts. Instead of using hand-created templates for each character to be hinted, we use an existing, hinted font as the template, allowing the hints of one font to be transferred to another. This translation process includes estimation of the control value table entries used to unify feature sizes across a font. The matching algorithm, while simple, works well for a wide variety of character shapes, including serifed and italic fonts.

The hint transfer itself is somewhat less successful, owing primarily to the different strategies used in hinting different styles of characters (e.g., modern serif vs. oldstyle serif). The technique is already quite useful for transferring hints between members of the same family (a roman to a bold, for instance). We expect that transferring hints between fonts of different families will become more and more practical as more fonts are hinted with the VTT tool, so that the hinter has a larger selection of source fonts to choose from and can pick one that is more similar to the target font.

An important advantage of our approach over previous autohinters is that it preserves the hand-crafted hinting strategy, built by a professional typographer, in the newly hinted font. Thus, the translated hints provide a good starting point and generally require only minor cleanup and adjustment. With time, we expect this work to evolve into a highly practical tool for speeding the creation of production-quality digital fonts.

Acknowledgements

We are indebted to Michael Duggan, Greg Hitchcock, and Beat Stamm of the Microsoft eBooks group for the many long discussions about hinting, VTT, TrueType, and typography in general.

References

[1] Adobe Systems, Inc. *Adobe Type 1 Font Format*, March 1990.

[2] Sten F. Andler. Automatic generation of gridfitting hints for rasterization of outline fonts or graphics. In *Proceedings of the International Conference on Electronic Publishing, Document Manipulation, and Typography*, pages 221–234, September 1990.

[3] Apple Computer, Inc. *The TrueType Font Format Specification*, 1990. Version 1.0.

[4] Claude Bétrisey. *Génération Automatique de Contraintes pour Caractères Typographiques à l'Aide d'un Modèle Topologique*. PhD thesis, École Polytechnique Fédérale de Lausanne, 1993.

[5] Roger D. Hersch. Character generation under grid constraints. In *Proceedings of SIGGRAPH 87*, pages 243–252, July 1987.

[6] Roger D. Hersch and Claude Bétrisey. Model-based matching and hinting of fonts. In *Proceedings of SIGGRAPH 91*, pages 71–80, July 1991.

[7] Beat Stamm. Visual TrueType: A graphical method for authoring font intelligence. In R. D. Hersch, J. André, and H. Brown, editors, *Electronic Publishing, Artistic Imaging, and Digital Typography*, pages 77–92, March/April 1998.