Experience with OpenType Font Production

Sivan Toledo*and Zvika Rosenberg†

22nd January 2003

1 Introduction

This article describes the production of a large number of Hebrew OpenType fonts. More specifically, we describe the production of OpenType fonts with TrueType glyph descriptions and with advanced typographic layout features. The project was conducted by MasterFont Studio Rosenberg, a large Hebrew digital font foundry in Tel-Aviv, with assistance from Sivan Toledo from the School of Computer Science in Tel-Aviv University.

Hebrew is a right-to-left script that is used in one of two ways: with or without diacritics. Most Hebrew texts are printed almost without any diacritics, but childern's books, poetry, and bibles are printed with diacritics. Even texts that are printed without diacritics use them occasionally to disambiguate pronounciation and meaning. The diacritics in children's books and poetry are vowels and consonant modifies, which are called *nikud* in Hebrew (meaning "to add points"). Bibles also use a third kind, cantillation marks, which are not treated in this article (but see [4, 5] for a thorough treatment). Hebrew diacritics are quite hard to support in fonts, since there are 15 of them and 27 letters, with up to 3 diacritics per letter. Even though not all the combinations are grammatically possible, there are too many combinations to support them conventiently using precomposed glyphs. In addition, diacritics must be positioned relative to the base letter visually, rather than mathematically. For example, the below-baseline diacritics must be set below the visual axis of the letter [14], not below its mathematical center. This means that ligatures, pair kerning, and simple mathematical centering alone are insufficient for correct placement of diacritics. In addition to diacritics, Hebrew fonts can also benefit from pair kerning, although until recently, pair kerning was not particularly common in Hebrew fonts. For further information about the Hebrew script, see [1, 13, 14].

The initial objective of the project was to convert older fonts into the new OpenType format. This was motivated by the concurrent development of Adobe InDesign 2.0 ME (middle eastern), the first high-end page-layout program to support Hebrew OpenType features. Microsoft's Office XP also supports these features, so InDesign was not the only motivator. But it quickly became evident that the same tools that would be necessary to convert the old fonts to the new format could also be used to streamline the entire font production process at MasterFont. Therefore, the project had two objectives: to convert the old fonts to the new format, and to streamline the production process.

The rest of the paper is organized as follows. The next section provides background on font formats. Section 3 describes how diacritic positioning information is represented in MasterFont's old fonts. Section 4 describes the objectives of the project, and Section 5 describes the software tools that we used to achieve these objectives. The overall production process is described in Section 6, and our conclusions from this development and production experience is described in Section 7.

2 Fonts Formats

This section describes the OpenType font format and other relevant font formats. Unfortunately, the term OpenType is somewhat vague, so this section also establishes a more precise nomeclature.

Type 1, TrueType, and Early Extensions

The first widely-accepted device-indepenent font format was the PostScript Type 1 format [9]. Adobe started selling retail Type 1 fonts in 1986, and published the specification in the late 1980's, after Bitstream figured out how to produce Type 1 fonts (prior to that the specification was proprietary and

*School of Computer Science, Tel-Aviv University, Tel-Aviv 69978, Israel. Email: stoledo@tau.ac.il.

†MasterFont Studio Rosenberg, 159 Yigal Alon Street, Tel-Aviv 67443, Israel. Web address: www.masterfont.co.il.

only Adobe could produce Type 1 fonts). Type 1 fonts consist of two or three main parts: scalable hinted glyph descriptions, metric information, which include glyph dimensions, advance widths, and pair kerning, and possibly bitmaps for screen previewing. Today most systems can rasterize the scalable glyph descriptions so the bitmaps are usually no longer necessary; Adobe distributes freely the Adobe Type Manager which is a Type 1 rasterizer for older Windows and Macintosh computers without a built-in rasterizer. Type 1 fonts are relatively easy to design and produce, and today there are tens of thousands of commercial Type 1 fonts.

The next widely-accepted device-indepenent font format was the TrueType format [6, 2], which was invented by Apple in the late 1980's, as part of a collaboration between Apple and Microsoft, and became the native scalable font format for both Apple Macintosh computers and Windows computers. In a TrueType font, all the data associated with the font, such as glyph descriptions, metric information, administrative informaion (font name, for example), and possibly bitmaps, are placed in a single file, which is organized as a collection of tables. For example, one table contains glyph description, another character-to-glyph mappings, yet another table contains kerning pairs, and so on.

The kinds of data in a TrueType font are almost identical to the kinds of data in a Type 1 font, but they are organized differently and sometimes represented differently. Most importantly, glyph descriptions and hints for low-resolution rasterization are represented differently. In a Type 1 font, glyphs are represented using cubic-spline outlines with declarative hints. In a TrueType font, glyphs are represented using quadratic-spline outlines and a low-level programmable hinting language. Type 1 outlines are easier to design and hint, and most designers use cubic splines when designing fonts. Well-hinted TrueType fonts are hard to produce, but they can achieve pixel-perfect rasterization at low resolutions, something that Type 1 fonts cannot achieve. Virtually all font editors can convert one type to the other. the conversion of TrueType outlines to Type 1 outlines is lossless, but the other direction only produces an approximation. However, the approximation of a Type 1 outline by a TrueType outline can be made arbitrarily accurate and font editors produce approximations that are visually indistinguishable from the original. Hinting information is usually also converted, but not as well.

Both Adobe and Apple introduced enhancements to these font formats during the 1990s, but none became widely used. Adobe introduced the Multiple Master font technology [10, 11], which allows users to interpolate between fonts in a family. This enhancement transforms the font family from a discrete set to a continuous spectrum. For example, there aren't just medium and bold fonts, for example, but a continuous weight axis. Apple introduced GX fonts [6] (now called AAT fonts), which are TrueType fonts with additional tables that allow continuous interpolation, like Multiple Master fonts, and which enable high-end typographic features. In particualr, GX fonts can support multiple glyphs for a single character, such as swash, small caps, or old-style figures, they can reorder glyphs, and so on. Neither format became widely accepted by font vendors, probably because the fonts were too hard to produce. No major font foundry besides Adobe produced Multiple Master fonts, and only a few GX fonts were produced by Bitstream and Linotype.

OpenType

OpenType is a new font format [3] that Microsoft and Adobe specified collaboratively. The format was first used, under the name TrueType Open, in the Arabic version of Windows 95, which was introduced in 1996. Adobe later joined the specification effort and the name was changed to OpenType.

OpenType adds three capabilities to the TrueType format: advanced layout features, support for Type 1 glyph descriptions, and digital signatures. The advanced layout features are supported by five new tables: GDEF, that define glyph properties and glyph sets, GSUB, which defines glyph substitution rules, GPOS, which speficies positioning information, JSTF, which provides justification information, and BASE, which provides baseline-adjustment information. These features were designed to support layout of text in complex scripts such as Arabic, Hebrew, Indic, and east Asian scripts (Hebrew is by no means the most complex), and to support high-end typographic features such swash letters and decorative ligatures, old-syle and proportional figures, and small capitals.

The support for Type 1 glyph descriptions is designed to allow lossless conversion of Type 1 fonts to a TrueType-like table-based file format. Such fonts use two new tables, CFF and VORG, to store a losslessly-compressed Type 1 font. Technically, the format of the CFF table corresponds to that of PostScript Type 2 fonts [12], also known as the compact font format, but the essense of the format is a lossless representation of Type 1 glyph

descriptions (outlines and hints). The need to support two kinds of glyph descriptions and hints constained the design of the layout-related tables, in the sense that positioning information in the GPOS table could not be hinted using the normal TrueType or hinting language; a different hinting mechanism is used, which supports both Type 1 and TrueType glyphs.

The support for digitally signing fonts, which uses the DSIG table, is meant to allow users and font-using software to verify the authenticity of fonts. For example, future operating systems might require that certain system fonts are digitally signed by the operating-system's vendor, thereby preventing modified versions of these fonts from being used instead. For independent font vendors, there is currently little benefit in digitally signing fonts, since users currently do not have means to verify these signatures, and since fonts rarely pose security or stability problems to systems.

OpenType fonts with TrueType outlines can be used by any TrueType-capable software, since except for extra tables (which are allowed by the TrueType spefication), the fonts are also valid TrueType fonts. Software such as rasterizers, layout engines, and printer drivers can simply ignore the extra tables. OpenType fonts with Type 1 outlines require special support in rasterizers and printer drivers, support beyond that required to support Type 1 and TrueType fonts. In principle any TrueType-capable layout engine can use any OpenType font, since the metric and layout information are compatible (an exception is pair-kerning information, which can appear in either the 'kern' table or the GSOS table or both in a font with TrueType glyphs, but only in the GPOS table in a font with Type 1 glyphs). To emphasize the fact that OpenType fonts with TrueType glyphs can be used by any TrueType-capable software, whereas special support is required for fonts with Type 1 fonts, files containing the first kind use the `ttf` or `ttc` extension, whereas files containing the second kind use the `otf` extension.

So what is an OpenType font? According to the Microsoft/Adobe specification, any font that conforms to the specification, including both TrueType and Type 1 glyphs, with or without advanced layout tables and digital signatures. This is somewhat confusing, especially since the file icons in Windows do not correspond to the glyph descriptions (Windows 2000 and XP use the 'O' icon for `otf` files and for `ttf` or `ttc` files if the fonts have a DSIG table, and the 'TT' icon for all other `ttf` files). We will use the acronyms OTF for OpenType fonts with Type 1 glyphs, TTF-OTL for fonts with TrueType glyphs and with advanced layout tables, and TTF for fonts with TrueType glyphs but no advanced layout tables.

Microsoft's main interests in the OpenType format lies in supporting complex scripts in Windows, with initial emphasis on Arabic and later on Indic scripts, and in tighter operating-system/font integration using digital signatures. Adobe's main interests in the format lies in exploiting TrueType's advantages, mainly the support for non-latin character encodings and the convenient single-file format, and in providing its fonts and applications easier access to so-called expert glyphs, such as old-style figures and small capitals [8].

Windows and Macintosh Font Files

Differences in the way font files are stored on different platforms cause a few additional difficulties during font production.

On Windows and Unix/Linux systems, a TrueType or OpenType font is stored in a single file. A Type 1 font is represented by up to four files: a `pfb` or `pfa` file that stores the glyph descriptions and a character-to-glyph encoding, an `afm` and/or a `pfm` files that store metric information, as well as the encoding, but no glyphs, and an `inf` file that stores administrative infomation about the font. On Windows system, only the `pfb` and `pfm` files are used.

On Macintosh systems, there are multiple ways to store fonts. The Macintosh operating system supports two byte streams per file. One is called the data fork and the other the resource fork, which was meant to store application resources such as localized strings, icons, and so on. The format of the resource fork allows multiple resources to be stored in a single file. Before MacOS X (that is, before version 10), fonts were always stored in the resource fork of files whose data fork was empty. Fonts and font families are represented by several kinds of resources, such as the FOND resource that describes an entire family (regular, bold, italic, etc.), the SFNT for storing TrueType fonts, the FONT and NFNT for storing bitmap fonts, and so on. The fonts of a family are usually stored in a single file called a *suitecase*, whose resource fork contains multiple resources. The Macintosh also associates a file type (separate from the extension) and a creator code with each file. Font files need to have a specific type to be recognized as such by the Macintosh's operating system.

Storing fonts in multiple resources within a single file, and storing the font data in the resource fork rather than the data fork creates problems when

fonts are moved between platforms. To address this issue, Apple introduced new ways to store fonts in MacOS X. First, suitecases can now be placed in the data fork, using a format called a `dfont` file (after its extension). This format is essentially a traditional Mac font file, but in the data rather than the resouce fork. Second, MacOS X also allows Windows-style packaging of TrueType and OTF files in a single file in which the font data is stored in the data fork with no header or trailer.

3 Diacritic-Positioning Information in Older Fonts

Masterfont's existing fonts used three mechanisms for positioning diacritics. These mechanisms were developed over a long period of time—each additional mechanism was designed to correct deficiencies in previous ones.

The first mechanism uses a number of precomposed glyphs. These are used for *dagesh*, a dot that appears inside letters, and which must be carefully positioned to prevent overstriking the letter itself. Such combinations are also used for *shin/sin* dots, for diacritics attached to a final letter (only two combinations are used in modern Hebrew), and a few other cases. There are Unicode code points for these glyphs, as well as code points in the MacHebrew 8-bit encoding, and they are used by both Macintosh and Windows systems.

On the Macintosh, precomposed glyphs are also used to place diacritics on particularly difficult letter: *resh*, *dalet*, and *qof*. The first two are wide letters that have a single vertical stem at the right, under which below-baseline diacritics should be placed. Incorrect placement under these letters looks awful. The *qof* is the only non-final letter with a descender, so special attention is required to prevent the below-baseline diacritics from colliding with the descender. These glyphs do not have Unicode code points, but they are used in all the Mac's Hebrew fonts, including the fonts bundled by Apple with the operating system. These glyphs appear to have been added by Apple quite early on. They do fix the most serious placement cases, but they do not fix all the placement problems. Furthermore, this glyph set appears to have been designed by somebody without much expertise in Hebrew, since some of the precomposed glyphs can never appear in a text (e.g., *dalet* with a *hataf-patah*).

Since the precomposed glyphs are insufficient for correctly positioning all the combinations, MasterFont developed a few years ago a positioning aid for the Macintosh. This software, called *Mapik-Ve-Day*, is a Macintosh system extension that selects a diacritic glyph according to the base letter. Fonts designed to work with this software have three sets of diacritics: one for narrow letters, one for wide letters, and one for medium-width letter. The diacritics in all sets usually have the same shapes, and all have zero advanced widths, but each set has different sidebearings. The three *qamats* glyphs, for example, all have the same shape and zero width, and all print to the right of the insertion point (so they appear below the preceding letter in a right-to-left text run), but the amount of right shifting is different. The software essentially divides the letters into three categories, each of which has its own set of diacritics.

These two mechanisms, the precomposed glyphs and the three sets of diacritics of different widths, essentially mimic hot-led diacritics.

These two mechanisms allow fairly good control over positioning, provided the font is designed for this system in mind. In particular, special attention must be given to the left sidebearings of base letters, to ensure that the diacritics from the appropriate category are well placed under them.

To allow more precise control over diacritic positioning, MasterFont began to use explicit positioning information in the fonts. These are stored in the kerning table of the font, even though they are not real kerning pairs, since the insersion point should not move after the positioning adjustment. For example, a kerning pair *alef-qamats* with value 35 means that the *qamats* glyph must be shifted relative to the *alef* glyph by 35 font units. The insertion point remains exactly after (to the left of) the *alef*, as if no adjustment took place. This information is not used by the Macintosh operating system, but a some Adobe applications can use it.

MasterFont currently has 86 fonts with this level of diacritic positioing, which is essentially the set of fonts designed for book work. In addition, there are many more display fonts with pair kerning information but without elaborate diacritic support.

4 Objectives

Now that the setting has been described, we can enumerate the objectives of this project in more detail.

First and foremost, we wanted to convert the diacritic positioning information in the existing fonts to OpenType advanced-layout tables. We wanted

to do this without specifying any additional positioning information. This restriction was motivated by several reasons:

- Efficiency; there was no point in doing extra design work on fonts that already produce perfecly-positioned texts. For example, there is no point in specifying positioning information for a combination represented by a carefully designed-precomposed glyph.
- Correctness; by not specifying new positioning information, the chances of introducing new positioning bugs are minizied.
- Continued support for older applications; we wanted to be able to produce additional new fonts that would be able to work not only in OpenType applications, but also in older applications. By building new fonts according to the old specification and automatically converting them to OpenType, the foundry would be able to offer customers either an OpenType font or an old-syle font for legacy applications.

Second, we wanted to automatically produce fonts for both the Windows and Macintosh platforms (Windows fonts also work on Unix and Linux). More specifically, we wanted to produce the fonts for the two platforms using the same source fonts, rather than generate a Windows font and convert it, and then generate a Mac font and convert it. This meant that the glyph sets for Windows and Mac fonts had to be merged, since previously the foundry used different glyph sets for each platform. Furthermore, if possible, we wanted to have identical *finished* fonts, except for the suitcase wrappers. It turned out to be possible.

Two remaining choices had to be made, concerning the glyph-description format and the Mac-packaging format. After some deliberation, we decided to produce TTF-OTL fonts rather than OTF fonts. Producing OTF fonts has one advantage, namely that the original Type 1 outlines and hints remain intact, whereas automatic conversion is used when producing TTF fonts from the same source files. We felt that given the accuracy of the conversion, this had essentially no impact on high resolution output from printers or imagesetters. We also felt that modern on-screen rasterizers that use antialiasing (font smoothing) produce acceptable output from automatically-hinted and unhinted TrueType outlines. Therefore, we felt that OTF's advantage over TTF-OTL is not particularly significant. On the other hand, on pre-2000 Windows and pre-X Macs, OTF fonts require the Adobe Type Manager utility, which is free but nonetheless requires downloading and installing by the end user. Also, some applications, most importantly Microsoft's PowerPoint, cannot use OTF fonts even on systems that do support them (this is true for both PowerPoint 2000 and PowerPoint 2002). Finally, one of the tools we used in the production of the fonts, Microsoft's VOLT, provides much better support for TTF-OTL fonts than for OTF fonts; this is described more fully in the next section. Given the clear disadvantages and only slight advantage of OTF fonts, we decided to produce TTF-OTL fonts.

5 Tools

The production of the fonts uses a number of software tools, which we describe in this section. We also describe alternatives to some of the tools and explain our choice of tools.

Assembly of the Advanced Typographic Tables

The tool that we used to construct the advanced typographic tables in the fonts is Microsoft's Visual OpenType Layout Tool (VOLT). Given a font, we automatically construct input files for VOLT, and use it to construct a derivative font with appropriate GDEF, GPOS, and GSUB tables.

Adobe's OpenType Font Development Kit (FDK) can perform a similar function. However, when we started the project, the FDK had incomplete support for the GPOS table, which meant that we could not use it to convert our Hebrew fonts. The FDK is also OTF-oriented, in that it cannot accept TrueType glyphs as input, where as we wanted a TTF-to-TTF-OTL conversion, in order not to modify the glyphs or their hints. These restrictions appear to still hold.

In addition to VOLT, microsoft also distributes command-line tools that can construct advanced typographic tables, but they are quite old and we did not experiment with them.

Versions 4.0 and 4.5 of FontLab can also produce OpenType fonts with advanced typographic tables. But our understanding has been that the OpenType support in FontLab is based on Adobe's FDK and hence does not support the required GPOS mechanisms, so we did not explore this approach.

Since we used VOLT, we would like to comment on its advantages and disadvantages. VOLT is a

visual tool, designed to allow a font designer to specify visually glyph substitutions and positionings. Together with the sample fonts, it is a superb tool for understanding how OpenType layout features work and for prototyping and developing OpenType layout support. In addition to visual design, VOLT allows the user to export and import the structure of the layout tables or parts of the tables into text files with special syntax. Since we had to convert many fonts, we opted for automatically generating these files, which are called VOLT project files (`vtp`) rather than for visually constructing each font. On the other hand, VOLT does not have command-line options that can be used to open a font and a VOLT project file and ship an assembled TTF-OTL font. This means that even if the project files are constructed automatically en mass, they cannot be processed in batch mode—each font must be opened using a menu command, the corresponding project file must be imported using another menu command, and the output font must be shipped using yet another command.

Data Extraction and Project File Construction

To produce the input file for VOLT, we had to convert the data in the initial TTF fonts into VOLT project files. The format of the VOLT project files is complex and undocumented, but quite simple to figure out by comparing the file to the font structure in the graphical user interface.

We performed the conversion using two custom-written programs. The first program, which is written in C, reads a TTF font and outputs an `afm`-like text file, which maps glyph indices to PostScript glyph names, and which documents all the kerning pairs (glyphs in TrueType fonts are stored in an array and are refered to by VOLT using their indices).

The second program, which is written in the AWK language, transforms this information into a VOLT project file. The transformation consists of:

- Mapping each glyph index to a glyph name and Unicode code point(s).
- Defining glyph groups.
- Defining glyph substitution lookups, for exmaple to map normal diacritics to wide or narrow ones, to map glyph sequences to precomposed glyphs, and in one case to decompose a precomposed sequence depending on the preceding glyphs (this supports two meaning and vocalizations of the unicode *vav+holam* sequence, which can stand for a single long vowel or a consonant followed by a short vowel; there is a single Unicode sequence for both cases, but they should be printed and vocalized differently).
- Defining diacritic positioning information.
- Defining kerning pairs.

We wrote the AWK program by developing a prototype font visually in VOLT, exporting the project file, and then writing the program so that given the input file, it generates a similar project file. We then loaded the resulting project file into VOLT and inspected the resulting OpenType structure. When we discovered problems, we modified the AWK program to correct them and so on. This methodology exploits the two strengths of VOLT: the ability to design and inspect OpenType layout tables visually, and the ability to accept automatically-generated input.

Minor Modifications of Other TrueType Tables

We gradually discovered that other TrueType tables had to be modified to ensure that the fonts perform properly. Here are the main modifications that we perform on the fonts:

- We add code-page-support and unicode-range information to the fonts. This information is used by Windows and Windows applications to determine which scripts a font supports.
- While we use VOLT to construct the unicode character-to-glyph mapping table (a subtable of the 'cmap' table), we later add an 8-bit Hebrew encoding table to support older Mac applications. This encoding subtable must be added even if the original TrueType font has one, since VOLT regenerates the 'cmap' table, and to the best of our knowledge, cannot emit this particular subtable.
- We remove the 'kern' table, which contains diacritic-positioining data that are not true kerning pairs, as described above.
- We add a 'gasp' table to specify that the fonts should to be antialiased at all sizes.
- We make minor modifications to the 'name' table, which contains copyright, version, URL, and other textual information for the end user.

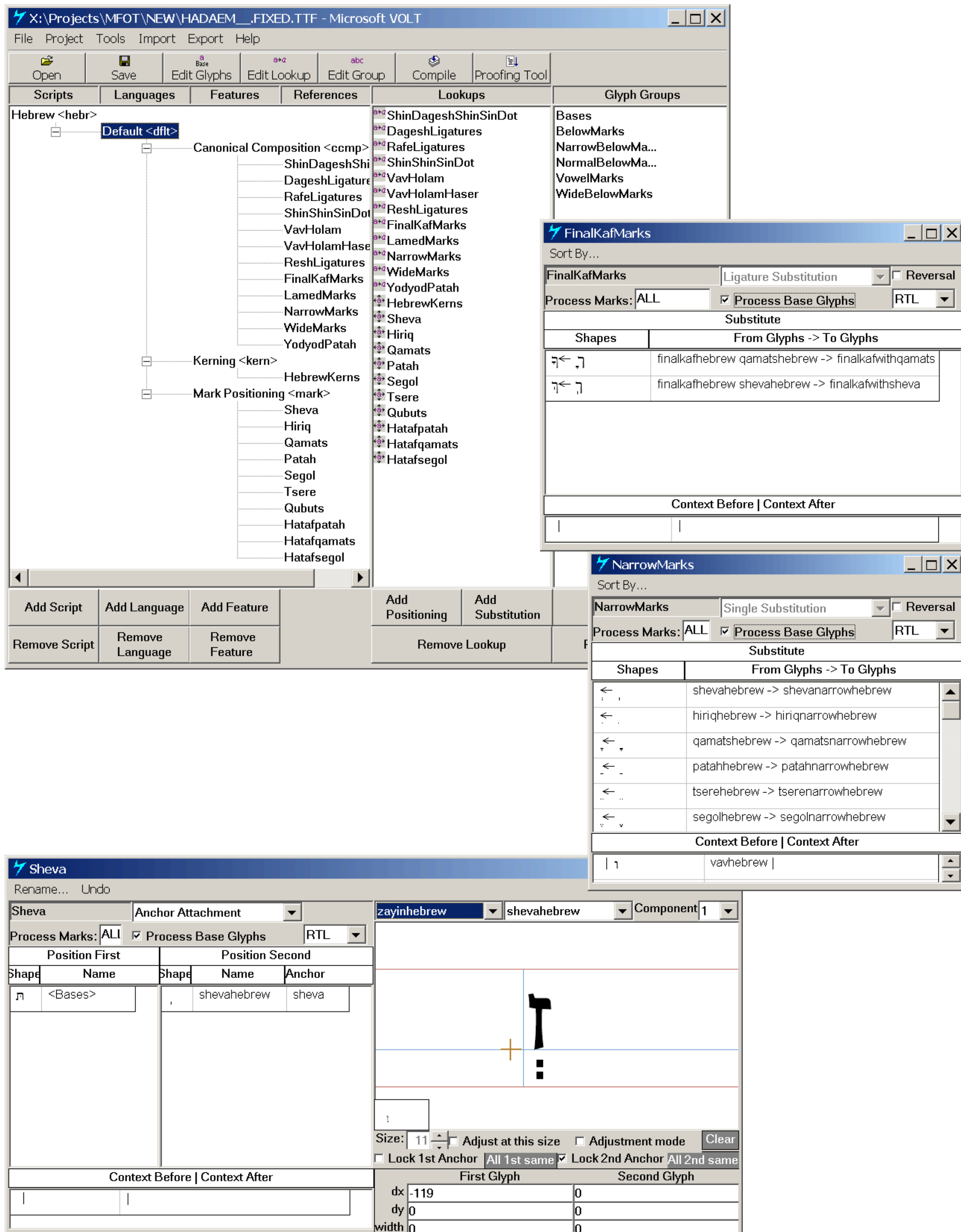

Figure 1: One of the OpenType fonts in VOLT along with three types of OpenType lookups that we use to position diacritics: anchor positioning, single contextual substitution, and ligature substitution. Our fonts also use pair-kerning adjustments and more complex contextual substitutions.

Some of the modifications are done with a specially-written C program. The other modifications are performed by converting the appropriate table to XML format using Apple's `ftxdumperfuser` command-line tool for MacOS X, modifying the XML file using AWK and SED programs, and then fusing the resulting XML-formatted table to the font file, again using `ftxdumperfuser`.

We perform some of the modification using a C program and some using the Apple tool because when we started the project, the Apple Font Tools for MacOS X had not yet been released. We therefore wrote a custom C program to perform the modifications we found necessary. As testing progressed and we discovered more required or desired modifications, we switched to using the Apple tool, which was easier than extending the C program. One problem with the XML format that Apple uses to describe font tables is that simple text processing tools like AWK and SED do not understand XML's hierarchical structure. But for relatively simple modifications, the combination of `ftxdumperfuser` and AWK or SED scripts works.

There is another tools that can perform TrueType-to-XML and XML-to-TrueType conversions, Just van Rossum's TTX[1]. We experimented with this tool, which is written in the Python language, but we could not install it successfully. The trouble was that TTX uses a number of other Python packages, for example, to perform numerical calculations, which did not install propely on our systems. But assuming that these issues in TTX are easily solvable, it is certainly an alternative to the Apple tools, and at least in principle, it is a multi-platform tool that runs on Windows, MacOS, and Linux.

General-Purpose Font Editors

Although no general-purpose font editor was used for the actual conversion, we did use two font editors to produce the input font files and to inspect fonts that seemed to have problems. We used Fontographer to produce the input files to the conversion process, and we used PfaEdit to inspect fonts.

Machintosh Suitcase Generation

We used the `ufond` command-line program from George Williams' FONDU[2] package to package TTF-OTL fonts into suitcases, and we used command-line tools from the MacOS X developer kit to tag the suitcases with the required file type and creator code.

We also used a script to automatically scan an entire directory with TTF-OTL fonts and classify them into families. The classification is done using the family name field of the font. Once the classification is made, the script invokes `ufond` to package the fonts of the family into a suitcase.

6 The Production Process

The final production process consists of four stages. The *prevolt* stage is performed by a script that processes all the input TTF fonts in a given directory. For each input file, the script generates a modified TTF file and a volt project file. The modification to the input files that we perform at this stage involve fixing the 'post' table to overcome an apparent Fontographer bug, upgrading the version of the 'OS/2' table and adding to it Unicode-range and code-page information, and removing the 'kern' table. The volt project file is produced by invoking the C program that generates an `afm`-like text file with glyph infomation and kerning and diacritic-positioning information, and converting that file to a volt project file using a custom AWK program.

In the second stage, each modified TTF file is opened in VOLT, the corresponding project file is imported, and the resulting font is shipped out. This requires only three menu commands in VOLT and clicking 'okay' a couple of times, but it is still the most time consuming stage of the production process.

The next stage, the *postvolt* stage, processes all the files shipped from VOLT in a given directory. A script invokes `ftxdumperfuser` a few times on each font, to add a MacHebrew encoding subtable, to fix the 'name' table, and to add a 'gasp' table. Although some of the processing can be moved from the *prevolt* to the *postvolt* or vice versa, some processing must be done before VOLT can process the font and some processing must be done after VOLT ships the font. Hence, a prevolt and a postvolt stages are necessary.

The final *suitcases* stage again uses a script to process an entire directory. The script first extracts the family name from each font file. Next, the script finds all the font files that belong to a single family, and invokes `ufond` on them to create a suitcase. The suitcase is created by `ufond` in the data fork of a file. We then move the font data to the resource fork of the font file, and tag it with appropriate type

[1] `http://www.letterror.com/code/ttx/`
[2] `http://fondu.sourceforge.net`

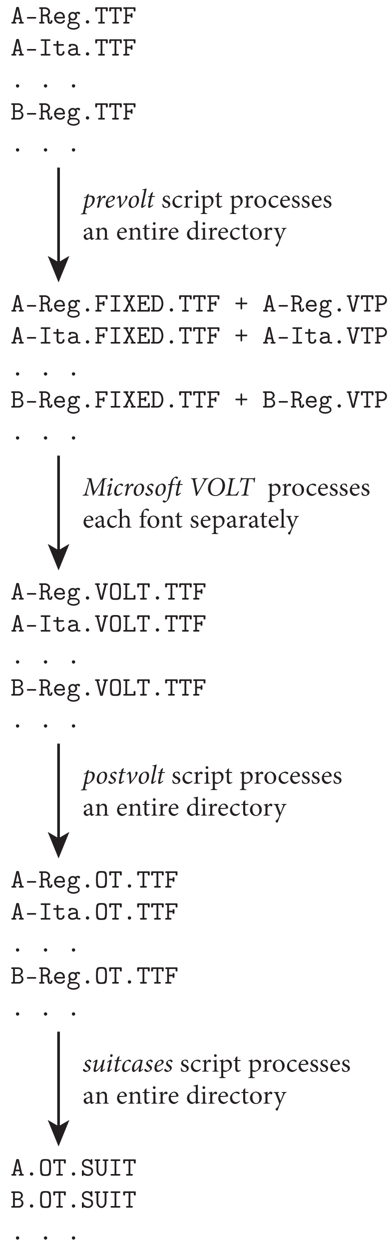

Figure 2: An overview of the production process. The figures shows the input and output files for each stage and how that stage operates. The *postvolt* stage produces the final font files for Windows and MacOS X, and the *suitcases* stage produces the final font files for any Mac operating system, X or older.

Ligature substitution:	שֶׁלָּךְ
Anchor positioning:	זְמָן
Contextual substitution:	וְגָם
	מָצוֹת vs. מִצְוֹת
Insensitivity to mark orderings:	
dalet+dagesh+qamats	גָּל
dalet+qamats+dagesh	גָּל

Figure 3: Hebrew diacritic positioning using OpenType features. The first three lines show how OpenType features utilize the mechanisms of older fonts. These examples correspond exactly to the examples shown in Figure 1. The fourth line shows a new application of contextual substitution that yields correctly-positioned glyphs for two identical unicode sequences with different grammatical meanings and different vocalizations. The last two lines show that the OpenType features have been encoded in a way that prints all valid diacritic sequences in the same way, as mandated by Unicode. The sample was prepared using Adobe InDesign 2.0 ME.

and creator codes. The automatic recognition of the font files that belong to each family minimizes the chances for errors that could occur if a family-configuration file was written by hand.

Due to the use of `ftxdumperfuser`, the *prevolt* and *postvolt* stages must be performed on a MacOS X machine. The VOLT stage must be performed on a Windows machine. The *suitcases* stage is also performed on a MacOS X, in order to move the font data to the resource fork and in order to assign type and creator tags; non-Macintosh operating systems do not support these file-system features.

בְּרֵאשִׁית, בָּרָא אֱלֹהִים, אֵת הַשָּׁמַיִם, וְאֵת
הָאָרֶץ. וְהָאָרֶץ, הָיְתָה תֹהוּ וָבֹהוּ, וְחֹשֶׁךְ, עַל פְּנֵי
תְהוֹם; וְרוּחַ אֱלֹהִים, מְרַחֶפֶת עַל פְּנֵי הַמָּיִם.
וַיֹּאמֶר אֱלֹהִים, יְהִי אוֹר; וַיְהִי אוֹר. וַיַּרְא אֱלֹהִים
אֶת הָאוֹר, כִּי טוֹב; וַיַּבְדֵּל אֱלֹהִים, בֵּין הָאוֹר וּבֵין
הַחֹשֶׁךְ. וַיִּקְרָא אֱלֹהִים לָאוֹר יוֹם, וְלַחֹשֶׁךְ קָרָא
לָיְלָה; וַיְהִי עֶרֶב וַיְהִי בֹקֶר, יוֹם אֶחָד.

Figure 4: The first few phrases of Genesis, typeset in Adobe InDesign 2.0 ME using an OpenType version of Henri Fridlaender's *Hadassah* typeface.

7 Conclusions

This project led us to several conclusions concerning OpenType and font production in general.

Reflections on Font Production

Fonts are atypical objects that present special challenges to their manufecturers. The uniqueness of fonts is caused by a combination of factors that is not present similar products.

First, fonts are highly complex data objects. While Type 1 fonts are fairly simple, TrueType, OpenType, and other advanced font formats are complex, with hundreds of individual fields that client software uses.

Second, fonts, especially those produced by independent foundries, are supposed to work with a variety of operating systems and software applications. Since different clients access different parts of font data, a font that tests perfectly with a given set of clients might fail on another client, as we have often discovered during the production process. To compound the problem, font-file specifications specify the format and intent of data fields, but they specify only loosely the behavior of client data. This is often done on purpose, to accomodate the behavior of older existing clients, but it makes it almost impossible to create working fonts without extensive testing. Let us give an example: are the glyph names in a TrueType or OpenType font significant, or are they present only to allow printer drivers to download the font into a printer in a consistent manner? Presumably, applications should use the encoding table in the 'cmap' table to find glyphs, not the PostScript names of the glyphs. But if some application uses glyph names instead, a font with variant names would not work in that application (we note that even Microsoft- and Apple-supplied fonts use names for the Hebrew glyphs that are inconsistent with the Adobe Glyph List).

Third, font data, such as glyph outlines, hints, and metric information has a very long life, but font files have a shorter life span. The outlines and metrics of widely-used fonts by foundries such as Linotype, Bitstream, or Agfa-Monotype are probably more than 20 years old, clearly an old age in the digital world. On the other hand, font files need to be recreated from this data once in a while. Font data that might have been used to create pre-PostScript fonts, were used again in the mid 1980s to create PostScript Type 3 fonts, then PostScript Type 1 fonts, then TrueType fonts. The TrueType font might have been upgraded once with better hints, then again to add the Euro symbol, then again to add non-latin glyphs for supporting additional scripts. In between, special versions might have been produced for bundling with operating systems or software packages. In the non-latin font market, special versions might have been produced to work with various layout programs. And today, the same font data is used to produce TTF-OTL and/or OTF fonts. Even a relatively young foundry like MasterFont has font data going back about 15 years.

Forth, font foundries are typically small businesses or small business units [7]. This limits their ability to invest in custom programming to automate font production. This again is a particular problem in the non-latin font market, where fonts are more specialized.

The longevity of font data from which new font files must be occasionally produced, together with the fact that many foundries accumulate hundered or thousands of fonts, should imply that automation should be widely employed in font production. That is, producing a new font by launching a font editor, loading an older version, making the required changes, and generating a new font, is inefficient when a large number of fonts must be produced. Manual production is also prone to errors and hence requires more testing than an automatic conversion that processes all the fonts in exactly the same way. However, the size of most independent foundries preclude large investments in custom automation (that is, paying programmers to automate font production).

This problem has been addressed recently in three ways:

- Microsoft, Adobe, and Apple, who need to encourage font production, release free font-production tools (Adobe's interest in indepedent font production stems from its role as a vendor of application, not from its role as a font foundry). Microsoft has released VOLT and a large number of other tools for hinting, adding digital signatures, editing strings in a font, and so on. Adobe has released the FDK, and Apple has released a number of font tools. To a large extent, the three companies release the tools that they use themselves to produce fonts. Still, the effort in publicly releasing and supporting these tools is significant, and the wide availability of these tools does help indepenent foundries. In some cases the tools are not ideal for automated production. For example, the inability of VOLT to process a font

without invoking the graphical user interface slows down production.

- General-purpose font editors now include scripting languages. Specifically, FontLab uses Python and PfaEdit it's own scripting language. These tools will reduce the cost of automation and hence of font production, especially if scripts can be applied to a font from the command line (this is true for PfaEdit, and perhaps also for FontLab). Obviously, this approach requires programming, but at least the programming environment is the familiar font editor. One remaining problem in this approach is that these font editors read the font data into their own in-memory data structures and then generate a new font, which can have side effects on tables that are not supposed to be modified by a script. FontLab is careful not to modify data it does not need to (for example, not to modify TrueType glyphs unless glyphs are edited), but PfaEdit regenerates TrueType glyphs whenever it outputs a font file.

- Individuals produce powerful free tools that can assist in automating font production. These include TTX and PfaEdit. One problem with using these tools in commercial font production is that they tend to be less stable. One might expect that they would not be as well supported as commercial tools, but that is not always true: George Williams, for example, the creator of PfaEdit, provides excellent and prompt support.

Font production requires careful management over time of source files, deliverables, and software. Source files are used to produce fonts over many years, and they need to be updated by adding glyphs, metric infomation, and so on. When the source files are updates or when deliverables in new formats are needed, deliverables are produced again. It is beneficial to automate new production cycles as much as possible, especially when many fonts are involved. Since foundries often cannot invest significant funds in custom programming, standard font-production tools must support automation and re-production. The scripting support in FontLab is a step in this direction, and essentially any command-line tool is usable in script-driven batch processing. But other tools, like VOLT, do not support automated production and re-production. The importance of automation and the requirements of multiple production cycles may be evident to large font foundries, but are usually not evident to smaller foundries.

Reflections on OpenType

Will the OpenType format, and in particular the advanced typographic layout, succeed, or will it fail like Multiple Master fonts and GX fonts? This question is important for font vendors, application developers, and font consumers, who all need to decide whether or not to invest in OpenType. There are several reasons to believe that OpenType layout will become successful and widespead. First, both Adobe and Microsoft are pushing this technology. In particular, both companies provide font producers with free production tools, Adobe has converted almost its entire typeface library to OpenType (which was not the case with Multiple Masters), and Microsoft has provided application developers with both tools to utilize OpenType layout features (through the Uniscribe and OTLS libraries) and with assistance with developing layout engines. Both companies already produce major applications that exploit OpenType's layout features, namely Office XP and InDesign. Other companies are also developing OpenType layout engines, such as IBM's ICU open-source engine. Second, the technology is essential for presenting text in some scripts, such as Indic scripts, and essential for presenting Arabic and Hebrew text with diacritics (Arabic and Hebrew without diacritics can be set without OpenType layout features). In particular, Microsoft has used OpenType layout in its Arabic fonts and operating systems for years now. Third, the OpenType format has been designed to make font design easy, which was not the case for Multiple Masters and GX. The structure of OpenType's layout support is declerative. For example, the font designer can declare that a certain glyph substitution only occurs in a certain glyph context (preceding and following glyphs). The algorithmic question of how to recognize the context is left to the layout engine. In contrast, GX's structure is procedural/programmable, and the font designer must specify a finite-state machine that the layout engine uses to detect the context. In general, declarative font technologies like outline fonts and Type 1 hinting have been more successful than programmable technologies like TrueType hinting, GX state machines, or metafont. (TrueType hinting is unsuccessful in the sense that the cost of hinting causes most fonts to remain unhinted or automaticaly hinted.) The obvious reason is that most font designers are graphic designers by training, rather

than programmers.

One must acknowledge, however, a few problems related to OpenType layout. First, the main benefit that OpenType brings to the Latin-script market are alternate glyphs, especially small caps, old-style figures, proportional figures, and swashes. But these glyphs have been available for years, although in separate small-caps, old-style-figures, and expert fonts. Using these separate fonts is not much harder than using OpenType layout features: instead of marking a run of text or a character style with OpenType features, the document designer or graphic designer marks the text or style with an alternate font. This makes it a bit harder to switch font families, but the difference is not that dramatic. It is interesting to note that Adobe did not go all the way in its efforts to make OpenType layout useful. For example, using OpenType features to automatically select optically-scaled glyphs would have made optically-scaled fonts easier to use, but Adobe kept optically-scaled glyphs in separate fonts, probably in order to support different pricing for fonts with and without optical scaling.

Acknowledgements

Thanks to Microsoft's Paul Nelson for answering numerous OpenType-related questions on the VOLT-community's message board, on the OpenType mailing list, and via private email exachanges. Thanks to WinSoft's Pascal Rubini for answering questions regarding the behavior of InDesign 2.0's Hebrew layout engine, and regarding the Hebrew-diacritic-handling mechanisms of older applications. Thanks for George Williams for making PfaEdit available. Some of the historical information concerning font formats was gathered from replies to a question that Adam Twardoch posted on the OpenType mailing list.

References

[1] The Unicode Consortium. *The Unicode Standard Version 3.0*, 2000. Parts of the standard are available online at `http://www.unicode.org`.

[2] Microsoft Corporation. *TrueType 1.0 Font Files, Technical Specification Revision 1.66*, 1995. Available online from `http://www.microsoft.com/typography`.

[3] Microsoft Corporation. *OpenType Specifications Version 1.4*, 2002. Available online from `http://www.microsoft.com/typography`.

[4] Yannis Haralambous. Typesetting the holy bible in hebrew, with TEX. *TUGboat*, 15(3):174–191, 1994. Also appeared in the *Proceedings of EuroTEX 1994*, Gdańsk.

[5] Yannis Haralambous. "tiqwah": a typesetting system for biblical hebrew, based on TEX. *Bible et Informatique*, 4:445–470, 1995.

[6] Apple Computer Inc. *TrueType Reference Manual*, 1999. Available online from `http://fonts.apple.com`.

[7] Emily King. *New Faces: Type Design in the First Decade of Device-Independent Digital Typesetting*. PhD thesis, Kingston University, 1999. Available online at `http://www.typotheque.com` under 'Articles'.

[8] Thomas W. Phinney. TrueType, PostScript Type 1, & OpenType: What's the difference? PDF document available online at `http://www.font.to/downloads/TT_PS_OT.pdf`, 2001.

[9] Adobe Systems. *Adobe Type 1 Font Format*, 1990. Avilable online from `http://partners.adobe.com/asn/developer/technotes/main.html`.

[10] Adobe Systems. *Type 1 Font Format Supplement*, 1994. Technical Specification #5015, Avilable online from `http://partners.adobe.com/asn/developer/technotes/main.html`.

[11] Adobe Systems. *Designing Multiple Master Typefaces*, 1997. Technical Specification #5091, Avilable online from `http://partners.adobe.com/asn/developer/technotes/main.html`.

[12] Adobe Systems. *The Type 2 Charstring Format*, 2000. Technical Specification #5177, Avilable online from `http://partners.adobe.com/asn/developer/technotes/main.html`.

[13] Sivan Toledo. A simple technique for typesetting hebrew with vowel points. *TUGBoat*, 20(1):15–19, 1999.

[14] Ada Yardeni. *The Book of the Hebrew Script: History, Palaeography, Script Styles, Calligraphy and Design*. Carta, Jerusalem, 1997.