

Converting Outline Fonts to OpenDX Font Format

J. P. Hagon

Physics Centre, School of Natural Sciences
The University, Newcastle upon Tyne, NE1 7RU
United Kingdom*

A systematic procedure for converting industry standard outline fonts to OpenDX fonts is described in detail. Currently, the procedure has been successful in converting both Postscript Type 1 and TrueType fonts to native DX format. For the vast majority of fonts tested, the method works cleanly, without need for subsequent modification of the generated files. However, when failures do occur, these can often be repaired by a small amount of manual tweaking.

I. INTRODUCTION

OpenDX (henceforth referred to as *DX*) is a general purpose data visualisation system similar to *Khoros*, *IDL*, *AVS*, *Amira* and others. As its name implies it is open source software and therefore freely available. It was formerly a highly-regarded (and fairly expensive) product from IBM known as *Data Explorer*. IBM released the *Data Explorer* source code into the public domain under a special licence in 1999.

DX has many advantages over other similar systems in that it has an extremely versatile data model and an excellent visual programming interface. One of its weaknesses however, has been the lack of good-quality fonts available in its own native format. A small number of commercial fonts have been produced for *DX* in the past but there is just one outline (or ‘area’ font in *DX* terminology) that comes with the standard publicly available versions.

II. DX FONT FORMAT

There are two types of font supported by *DX* — ‘line’ fonts and ‘area’ fonts. The former are similar to the fonts that were common on pen-plotter output devices some years ago. Such fonts are still useful for screen display where hardcopy quality is not important since they can be rendered very quickly. They are not our concern here and will not be discussed further. ‘Area’ fonts rely on filled polygons and are therefore capable of much higher quality than line fonts. Unfortunately there is just one such font supplied with the standard *DX* release — the excellent *Pitman* monospaced font.

A. Area Font Structure

Most outline fonts are fairly simple in concept — inner and outer boundary lines (often defined in terms of cubic splines) are used to define an area to be filled. The spline defining the inner outline is opposite in direction (clockwise/anti-clockwise) to a spline defining an outer boundary. Postscript and TrueType fonts have opposite conventions in this regard.

Things are more complicated with a *DX* area font. Firstly, polygons rather than splines are used to define the outlines. Secondly, areas to be filled in are not defined with clockwise/anti-clockwise polygons, instead the required area must be *triangulated* to create an *area mesh*. These concepts are illustrated in Figure 1.

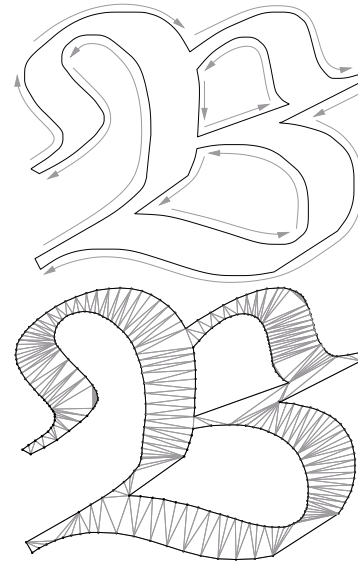


FIG. 1: Comparing methodologies for Postscript/TrueType and *DX* fonts. The letter \mathfrak{B} of the AMS Euler Bold Fraktur font as represented in Postscript form (upper figure) and *DX* form (lower). In the upper diagram there are three boundaries defined, one going clockwise and the others (defining the inner boundaries) going anti-clockwise. The lower diagram shows how a filled area is represented in a *DX* font using a triangulated mesh bounded by the same outlines as in the upper diagram.

In a *DX* font file, the boundary polygons are defined through a set of positions and the connections defining the triangulated mesh are defined as a set of integer triples, each integer referring to a particular position. For example, a simple hyphen (which is essentially just a rectangle) might be defined in a *DX* font file as shown in Figure 2. *DX* fonts have exactly 256 entries, making them equivalent to *8-bit* fonts commonly used today. There is no flexibility in the format to allow for larger (or smaller) fonts. The official description of the font format can be found in the *DX User's Guide*¹.

III. THE FONT CONVERSION METHOD

In order to get from, say, a Type 1 Postscript outline to a *DX* font in the form illustrated in Figure 2 requires roughly the following steps:

```

object "positions_hyphen" class array type
float rank 1 shape 3 items 4 data follows
0.276 0.187 0.0
0.011 0.187 0.0
0.011 0.245 0.0
0.276 0.245 0.0

attribute "dep" string "positions"
#
object "connections_hyphen" class array type
int rank 1 shape 3 items 2 data follows
2 1 0
0 3 2

attribute "ref" string "positions"
attribute "element type" string "triangles"
attribute "dep" string "connections"
#
object "hyphen" class field
component "positions" value "positions_hyphen"
component "connections" value "connections_hyphen"
attribute "name" string "hyphen"
attribute "char width" number 0.333
attribute "series position" number 45.000000

```

FIG. 2: An entry for the hyphen character from a native *DX* font file. Note the ‘char width’ and ‘series position’ attributes.

- ① Obtain the boundary points corresponding to all inner and outer lines for each character in a font.
- ② Triangulate the appropriate regions and obtain a set of connections for each character.
- ③ Output positions, connections and width information for each character in *DX* font format.

To perform this task, use will be made of three software packages, all of which are available in the public domain. The packages are *fontforge*², *pstoedit*⁴ and *Triangle*⁵. A brief description of each package now follows along with an explanation of its contribution to the *DX* font conversion process.

fontforge

This remarkable application, written by George Williams, is an outline font editor capable of creating and editing both Postscript and TrueType fonts. It is similar to commercial font editors such as *Fontlab* or *Fontographer* and provides much of the same functionality. It is available for multiple platforms and can be compiled from source if required. Further details may be obtained from the *fontforge* web site².

For all its many features, only limited use was made of *fontforge* in the *DX* font production procedure. In particular it was used to obtain the following vital font information:

- The official name of the font.
- The name, ascii code and widths of each font character. This is stored temporarily in one file for each font.
- An Encapsulated Postscript (eps) rendering of each character in the font for subsequent processing by *pstoedit*.

The above procedure can be easily automated thanks to *fontforge*'s own scripting language! *fontforge* was also used to convert the input font (whether Type 1 or TrueType) into a common internal format allowing both types of format to be treated in a similar fashion.

pstoedit

This is a well-known and very useful public domain package written by Wolfgang Glunz which converts Postscript (and PDF) files into a variety of vector formats⁴.

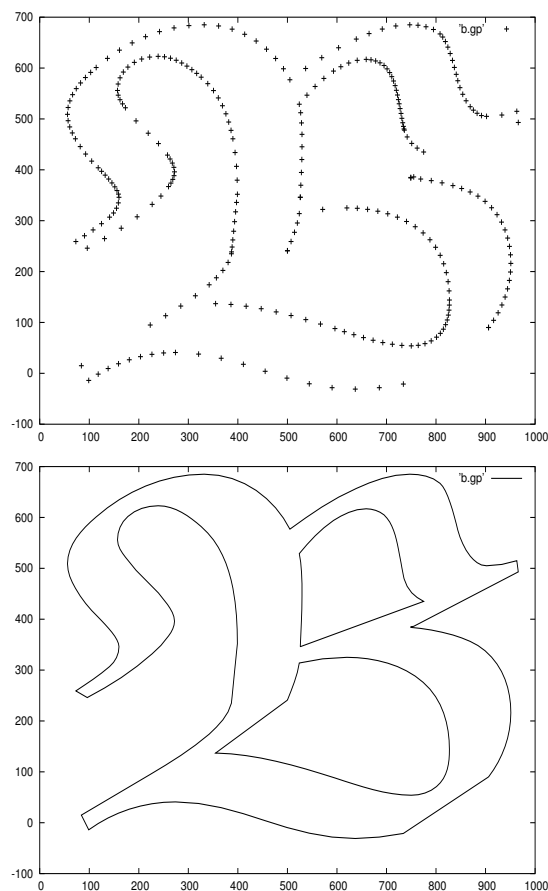


FIG. 3: *gnuplot* rendering of the Euler Fraktur \mathfrak{B} . The upper diagram shows a points plot, the lower shows a line plot (each point is connected to the next point (as ordered in the input file created by *pstoedit*).

pstoedit was used to extract the boundary point information for each character by converting the eps files generated by *fontforge* into *gnuplot*³ commands. The *gnuplot* driver was chosen because its output is in a very convenient form for subsequent processing — the boundary points being returned as a column of (x, y) pairs. When a full closed curve was completed, this was indicated by a blank line and the next set of points was started (if there was more than a single closed curve for a particular character). The generated output file can then be loaded into *gnuplot* and viewed via the *gnuplot* command

plot <file> or (if you want to see the points joined up) alternatively plot <file> with lines. Some *gnuplot* output for the Euler Fraktur character discussed earlier is shown in Figure 3.

Note that we are now clearly on the right track, the remaining task is to add the connection information. For this we use the *Triangle* package.

Triangle

This excellent program⁵ is the work of Jonathan Shewchuk. It produces a triangulated mesh, given a set of input points and constrained segments — i.e. the boundary outlines of each font character.

```
#
# Vertices, dimension, attributes, boundary markers
#
286 2 0 0
#
# Vertex no., x, y
#
0 0.906 0.09
1 0.734 -0.021
.
.
284 0.528219 0.394703
285 0.527145 0.369736
#
# Segments, boundary markers
#
286 0 0
#
0 0 1
1 1 2
2 2 3
.
.
192 192 193
193 193 0
194 194 195
.
.
243 243 244
244 244 194
245 245 246
.
.
284 284 285
285 285 245
#
# Holes
#
2
#
0 0.640961 0.323633
1 0.6685625 0.6155
```

FIG. 4: A *Triangle* ‘.poly’ file showing how vertices, segments and holes are set up. Note the termination segments which close each polyline and the two hole coordinates.

Triangle is a very efficient program and made the task of triangulation easy and relatively simple. The input required is a simple text file (referred to as a ‘.poly’ file — see Figure 4) with entries supplied for:

- A list of vertices — these are the nodes which form the boundary outlines for each character. They take the form of (x, y) pairs.
- A list of segments, i.e. the connection information needed to construct the boundary polygon. These are a list of integer pairs corresponding to the vertices mentioned previously. Since all the vertices are correctly ordered, this list can be easily generated and since all polygons are of the simple closed form, the last entry for a given polyline will be of the form $(n + m - 1, n)$ where n is the starting vertex and m is the number of points in a given closed polyline.

- A list of ‘holes’ if any. These are points which lie within regions inside certain polylines which are not to be triangulated. In the case of the Fraktur \mathfrak{B} , it is clear that there are two interior polygons which enclose regions which are not to be triangulated. By specifying a hole point anywhere in these regions, *Triangle* is instructed not to triangulate that region. *Triangle* produces a set of triangulated elements connecting polyline vertices from this input and stores the elements in a ‘.ele’ file.

Vertices and segments are essentially provided by *pstoedit* but holes need to be calculated explicitly. As mentioned previously, the sense of a polygon (whether it is clockwise or anti-clockwise) determines whether it should be filled or not. If it is not to be filled then a hole coordinate must be placed somewhere within the polygon.

In the Type 1 Postscript format an anti-clockwise polygon is one forming an inner boundary and therefore containing a hole; for TrueType it’s the other way round. A simple algorithm exists⁶ for determining if a polygon is clockwise. For a closed polygon with n points $(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})$, calculate the quantity:

$$\mathcal{A} = \frac{1}{2} \sum_{i=0}^{n-1} (x_i y_{i+1} - x_{i+1} y_i) \quad \text{with } (x_n, y_n) \equiv (x_0, y_0)$$

If $\mathcal{A} > 0$ then the polygon is anti-clockwise, otherwise it is clockwise. Once a hole polygon has been identified, any point within it serves as a hole point for *Triangle*.

It’s not as trivial as you might think to construct an interior point of a closed simple polygon. The following algorithm was used⁷:

- ① Identify a convex vertex v .
- ② For each other vertex q do:
 - (a) If q is inside avb , compute distance to v (orthogonal to ab). Where a and b are adjacent vertices to q .
 - (b) Save point q if distance d is a new minimum.
- ③ If no point is inside, return midpoint of ab , or centroid of avb .
- ④ Else if some point inside, qv is internal: return its midpoint.

Application of this algorithm usually results in a hole point being set very close to a boundary segment — so close, that to the naked eye the point often seems to lie *on* the segment.

IV. PUTTING IT ALL TOGETHER

Two perl scripts have been written to automate the above procedure. The scripts are *g2poly* and *font2dx*. Eventually these will be combined into a single script. *gpoly* converts a *gnuplot* input file (generated by *pstoedit*) into a ‘.poly’ file suitable for input into *Triangle*. Everything else is handled within the *font2dx* script which calls *g2poly*. *font2dx* can optionally reencode a font — there is a choice of several common encoding schemes.

The general form of a *font2dx* command is:

font2dx [OPTION]... FILENAME

where FILENAME is a Postscript Type 1 or TrueType font file. At present, the following options are available:

- noclean Don't clean up intermediate files (usually there are *hundreds* of these!) — by default these files are deleted leaving just the generated and original fonts.
- scale=<integer> Attempt a rescaling of the font. I originally used this to understand the geometry of TrueType fonts. Basically it attempts to correctly scale a font in cases where the default scale factor of 1000 fails. I find that for professional Postscript Type 1 fonts I never need to do anything here.
- negate Reverse the normal convention for inner and outer closed polygons. Again this is a tweak for non Postscript fonts. It is sometimes useful for testing, but most of the time you shouldn't need it.
- flat=<number> Set *psedit* 'flat' parameter. The default is 1.0 and the acceptable range of values is [0.2–100.0]. This parameter controls how accurately curves in fonts are approximated by polylines. Higher numbers give rougher approximations.
- enc=<encoding> Change the font encoding. Choose from: compacted, original, AdobeStandardEncoding, iso8859-1, isolatin1, latin1, iso8859-2, latin2, iso8859-3, latin3, iso8859-4, latin4, iso8859-5, iso8859-6, iso8859-7, iso8859-8, iso8859-9, iso8859-10, isothai, iso8859-13, iso8859-14, iso8859-15, latin0, koi8-r, jis201, jisx0201, win, mac, symbol, wansung, big5, johab, jis208, jisx0208, jis212, jisx0212, sjis, gh2312, gb2312packed, unicode, iso10646-1, TeX-Base-Encoding.
If the encoding is not one of these, then an encoding file is looked for, with the assumed name <encoding>.enc. Hence, many of the standard encoding schemes in \TeX can also be used.
- help Print usage information and help.

When converting an outline font to *DX* format using *font2dx* it is recommended that the result be checked by using modified versions of the *FontPreview* or *FontLayout* sample networks which come as part of the standard *DX* distribution. *DX* is very fussy about its fonts and if there's the slightest thing wrong, it will complain.

Unfortunately, if you use the *Caption* module (which is used by the previously mentioned sample networks), you will not be told where in the file the problem occurred — it is then better to debug your *DX* fonts with the *Text* module which, if there is a problem, will report the line number in the font file the problem started. For most well constructed fonts, there should be no problems and you will end up with a useable *DX* font. The **Troubleshooting** section below deals with some common issues.

font2dx will process only the first 256 character glyphs in a font. Modern fonts often have many more than this. If there is a 'hidden' glyph not in one of the first 256 slots, then you could try manually reencoding the font with a tool

such as *fontforge* prior to running *font2dx*. One useful encoding for European languages is the TeX-Base-Encoding. This encoding maximises (loosely) the number of diacritical characters in the first 256 slots — often fonts have these characters but the font designer has placed them in slots outside the normal range of accessibility. Many standard encodings, including TeX-Base-Encoding, can be set using *font2dx* via the --enc=<encoding> option mentioned earlier.

A further issue is that *DX* font characters have a width attribute, but no explicitly defined height. This is an issue if you want to use \TeX/\LaTeX to typeset *DX* annotations via the *DX-fontutils* suite because \TeX can use characters which have *zero* width. In such cases, it is impossible to correctly scale such characters unless there is a corresponding height (so that scaling ratios can be calculated). *font2dx* therefore adds a *char height* attribute, equivalent to *(height) + (depth)* enabling proper scaling even for zero width characters.

Finally, note that *font2dx* outputs fonts in 'dx text follows' format. It is straightforward to convert them to 'dx binary' format (and save a considerable amount of space in doing so).

A. Quality Issues

It may be worth experimenting with the --flat option to optimize font quality. The default value for this parameter is 1 and this seems to produce very good quality fonts, i.e. unless the fonts are greatly enlarged, it is almost impossible to detect the polygonal character of the outlines. In fact, a value of 10 produces pretty decent results for most text fonts tested. Figure 5 illustrates the effect of the --flat parameter for the URW Times-Roman font.

FIG. 5: *DX* rendering of URW Times-Roman for different flat parameters: flat = 100 (top); flat = 10 (middle); flat = 1 (bottom). Even a flat value of 100 produces recognisable text albeit in effectively a different font!

For exceptionally fine and detailed fonts a flat parameter of less than 1 may prove necessary. Consider, the WebOMints-GD ornament font, a sample of which is shown in Figure 6. Even here, it seems that a flat value of 1 is more than adequate, and even a value of 10 produces a surprisingly good result. Clearly though, the quality has totally broken down with a flat value of 100 producing an almost unrecognisable image.

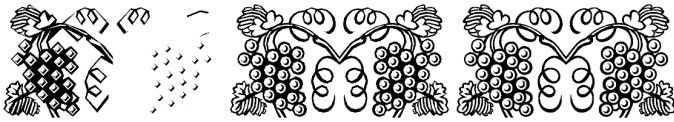


FIG. 6: *DX* rendering of WebOmints-GD for different flat parameters: flat = 100 (left); flat = 10 (middle); flat = 1 (right). In this case the loss of quality for a flat value of 100 is fatal — the glyph is totally unrecognisable.

There is a trade-off between font size and quality with smaller flat parameters leading to larger file sizes as might be expected. However it is generally true that as flat parameters get very large, the space saved is much less than the enormous loss in quality. This is illustrated in Table I.

'flat' parameter	100	10	1
URW Times	181717	254580	543254
WebOmints-GD	417797	709443	2036248

TABLE I: Font file sizes (in bytes) for different flat parameters in the case of URW Times and WebOmints-GD.

V. TROUBLESHOOTING

There's an excellent chance that *font2dx* will produce a clean font first time — it has done so for me in the vast majority of cases. However, some fonts can be troublesome and there are basically three things which can go wrong:

- ① There is a bug in *font2dx* or *g2poly* (e.g. wrong calculation of holes). If you find such bugs, please let me know. These bugs will usually manifest themselves via a *cannot open font file* error in *DX*. You will need to use the *Text* module to find out where in the font file the problem has occurred.
- ② There is a bug (or some geometric idiosyncrasy) in the actual font itself. One common fault is that the direction of a boundary is incorrect. Many modern font rasterizers can detect this and autocorrect, so the original font designer may be unaware of the problem. *font2dx* expects a completely consistent use of boundary polylines, so this occasionally causes problems.

A solution is to edit the font in a font editor such as *fontforge* and change the problem boundary lines so that they are consistent through all the characters in the font — remember, for Postscript, outer boundaries are *clockwise* and inner boundaries *anti-clockwise* and vice-versa for TrueType. *fontforge* makes this job relatively painless, as it recognises these inconsistencies and can autocorrect them if required.

- ③ There is a problem with the output from one of the support programs, *fontforge*, *pstoedit* or *Triangle*. It is testimony to the quality of these programs that, in the course

of testing over one hundred fonts, I have found nothing that can be traced to a problem or bug in any of them.

TrueType, OpenType and other non-Postscript Fonts

font2dx can also handle TrueType and OpenType fonts (and probably other font types supported by *fontforge*). Generally, it seems that OpenType files (at least the ones present on my Windows XP system) have scale factors of 2048 so the `--scale=2048` option should be used when dealing with these. If you get the scaling wrong, that should be apparent when looking at the finished *DX* font using, say, the *FontPreview* network. TrueType fonts, on the other hand, are a bit more unpredictable, sometimes you need to scale, sometimes not.

Missing Glyphs and/or Inverse Images

This is usually an indication of inconsistent inner/outer boundary directions. In this case, the option `--negate` often repairs the damage. However, if the problem is random, you may need to resort to using *fontforge* or a similar editor to tidy things up.

A. Case History: Adobe *BrushScript*

Adobe have deservedly won an excellent reputation for the quality of their fonts and BrushScript is no exception. However when processed with *font2dx* and tested with the *FontPreview* network, the following display is shown:

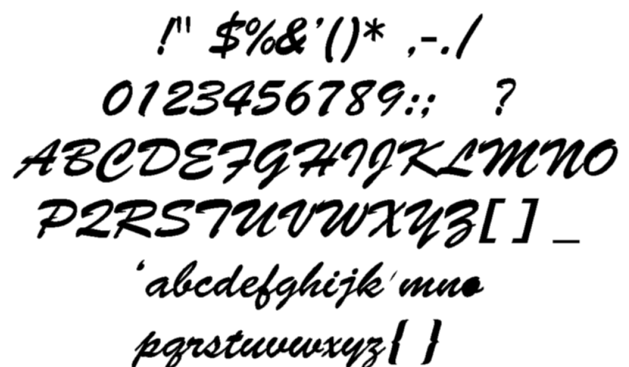


FIG. 7: *FontPreview* rendering of Adobe BrushScript. Note that there is no lowercase 'l' and the lowercase 'o' has lost its middle. The characters should look like this — *l o*. This illustrates that you have to be careful when checking fonts. A cursory glance, and you may think everything is OK, but you should try to check every character.

It is useful in problem cases to examine the various files constructed by *font2dx* during the course of a conversion. Normally these files are deleted automatically at the end of a run but you can retain them by specifying the `--noclean` option

when running *font2dx*. *font2dx* creates 6 files for each character so you can end up with hundreds of files in a typical run. In the case of the character ‘a’ of the BrushScript font, these files are listed in Table II:

file	description
<code>o_BrushScript.eps</code>	Encapsulated Postscript file created by <i>fontforge</i> .
<code>o_BrushScript.gp</code>	<i>gnuplot</i> file created by <i>pstoedit</i> .
<code>o_BrushScript.poly</code>	‘Poly’ file created by <i>g2poly</i> for initial input into <i>Triangle</i> .
<code>o_BrushScript.1.poly</code>	modified ‘Poly’ file created by <i>Triangle</i> .
<code>o_BrushScript.1.node</code>	‘Node’ file created by <i>Triangle</i> containing polygon points.
<code>o_BrushScript.1.ele</code>	‘Ele’ elements file created by <i>Triangle</i> containing triangulation elements.

TABLE II: Files created for the character ‘a’ in processing the Adobe BrushScript font by *font2dx*. Usually these files are deleted unless the `--noclean` option is used. The files have a general name of the form `<char name>_.<suffix>`.

There is a useful utility called *showme* which is part of the *Triangle* package. Typing ‘`showme o_BrushScript.1`’ will produce the window shown on the left side of Figure 8. By default, *showme* will produce a view of connections (or elements). Clicking on the ‘poly’ button gives a view of segments and holes.

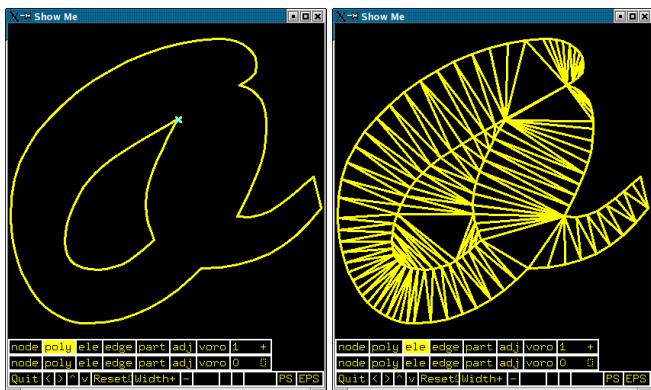


FIG. 8: (Left) Output of *showme* for character ‘a’ showing segments and holes. (Right) Looking at connections for the same character.

Note that the *showme* control panel has two rows, one labelled ‘1’ and the other labelled ‘0’. The latter row corresponds to the original (0th) ‘.poly’ file created by *g2poly*. Since it contains only polyline information, clicking on the ‘node’, or ‘ele’ buttons will produce nothing.

Clearly there is something wrong with the connection information. Moving in closely to the area around the cross marking the ‘hole’ we notice some spurious points (Figure 9).

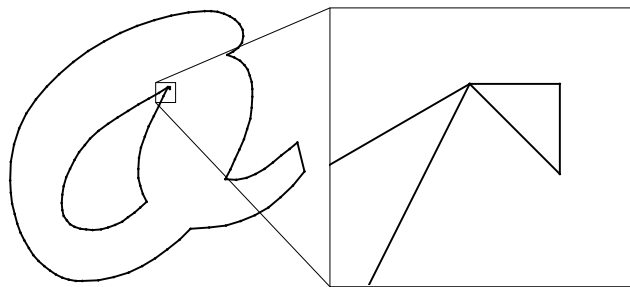


FIG. 9: Zooming in to the area around the ‘hole’ position. It looks as if there are some ‘extra’ points (probably three) which shouldn’t be there.

We can obtain approximate (x, y) coordinates by clicking with the middle mouse button. We find that the suspicious points are around $x = 0.215, y = 0.270$. We have two choices:

- Tweak the coordinates in the *gnuplot* file produced by *pstoedit*. This is the most efficient way because we don’t have to worry about connections — *g2poly* will recalculate them.
- We can edit the original font (which is almost certainly the source of the problem) using (say) *fontforge*.

First, let’s look at the former possibility. An examination of the file `o_BrushScript.gp` reveals the following subset of positions with the pair of columns on the left occurring in the middle of the file and the pair at the right occurring at the end of the file.

```

.      .      .      .
.      .      .      .
216    270    .      .
216    268    .      .
214    270    .      .
200.699 262.311    214    270
186.531 254.234    216    270
.      .
.      .

```

These units refer to the original Postscript font coordinates which are 1000 times larger than *DX* font coordinates (*DX* fonts have a normalized ‘size’ of 1.0, Postscript/TrueType fonts are normalized to 1000). There are three points clustered together which are almost identical and to arouse our suspicions are integral rather than floating point. We will delete two of these and see what happens. Further down the file we also notice a pair of similarly positioned integer (x, y) pairs — we delete one of these also. Next, we type:

```
g2poly o_BrushScript.gp 1 1000 > o_BrushScript.poly
```

to create a new ‘.poly’ file — note the extra arguments ‘1’ and ‘1000’. The first argument tells *g2poly* that the Postscript convention is to be used for border outlines; use ‘-1’ for TrueType fonts. The second argument is the scaling parameter (always 1000 for Postscript fonts). Then we run *Triangle* as follows:

```
triangle o_BrushScript.poly
```

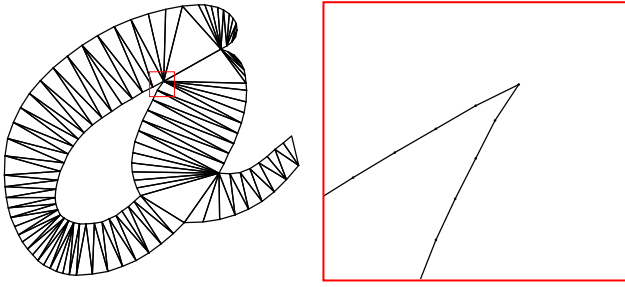


FIG. 10: The modified BrushScript ‘a’ — success!

to create new connections. Checking the new connections file with *showme* (type `showme o_BrushScript.poly` gives us the corrected character as shown in Figure 10.

Alternatively, we may choose to fix the original font by examining the defective characters using a font editor such as *fontforge*. The bogus points clearly originate in the original font as Figures 11 shows. It is beyond the scope of this docu-

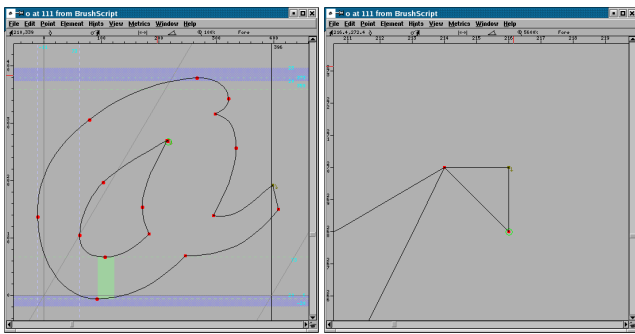


FIG. 11: Using *fontforge* to investigate character flaws. Here, the letter ‘a’ from Adobe BrushScript is examined (left). Zooming in to the suspected problem area reveals several spurious points which can be deleted (right).

ment to explain the intricacies of font editing software such as *fontforge* but all that is needed here (probably) is the deletion of these apparently bogus points. Now all that remains is to perform a similar investigative procedure for the other flawed characters — in this case ‘c’. I think it’s clear what a useful tool *fontforge* is — even if you never intend to use it as a font designing tool!

B. Case History: Architect

This is a free Tekton-like font, widely available⁹ on the Web. The particular version we discuss here, is intended for Apple Mac systems and can be obtained from:

<http://www.erik.co.uk/font/sans.html>

An initial run with *font2dx* produces the following *FontPreview* output in *DX*.

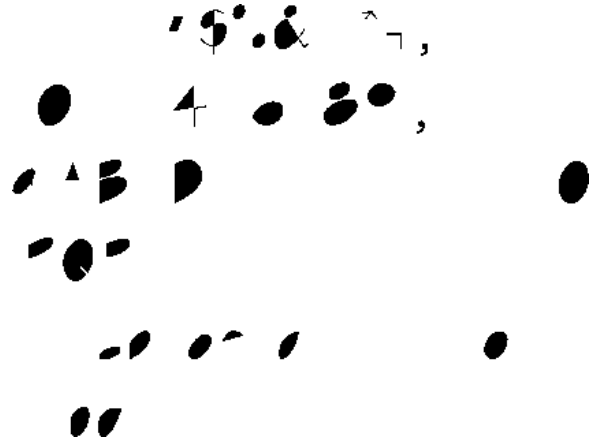


FIG. 12: *FontPreview* rendering of *Architect*. Disaster! Virtually every character is wrong.

This is so bad that it’s actually good! When you see something like this, it often means that the inner and outer loops have been interpreted the wrong way round. The `--negate` option can be used to correct this. So, re-running *font2dx* with `--negate` gives us the following from *FontPreview*:



FIG. 13: *FontPreview* rendering of *Architect* after applying `--negate`. We’re getting there!

There are now problems with just 5 characters — ‘&’, ‘\$’, ‘4’, ‘6’ and ‘9’. We need to investigate each of these individually. To do this, we need to rerun *font2dx* with `--negate` as before, but also with `--noclean` so that we may examine each character.

To begin with, we use the *showme* utility to look at the character ‘4’. Typing `showme four_Architect.poly` produces the display shown in Figure 14.

We need to tell *Triangle* that there should be a hole in the centre of the character to stop it from triangulating that region. The overlapping polygons indicated by the red circle prevent *font2dx* from calculating the hole correctly.

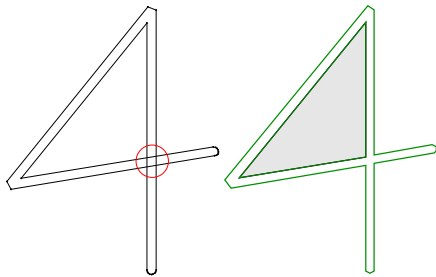


FIG. 14: *showme* rendering of character ‘4’ from the *Architect* font. The problem is the overlapping polygon indicated by the red circle. *font2dx* cannot deal with these at present, so manual tweaking is required. I understand that overlapping polygons are generally not considered to be good practice in font design — the right hand figure shows how it should have been done, i.e. with *two* non-overlapping polygons rather than a single overlapping polygon.

All that is required is that we supply *Triangle* with the coordinates of any point in the required hole region (the shaded area on the right hand side). To do this, simply click with the middle mouse button in the hole region and *showme* will produce the coordinates for you on the screen. The coordinates then need to be added to the ‘Holes’ section of the appropriate file, in this case the file named `four_Architect.poly`. You should have something like this:

```
#
# Holes
#
1
#
0 0.2628 0.3628
```

near the bottom of the file. Previously the number of holes was incorrectly set to zero. We change this to ‘1’ and add the coordinates we obtained by clicking the middle mouse button somewhere in the hole region. Each hole point is indexed, starting at zero, so we must add an integer ‘0’ before the x, y coordinate pair.

We next manually run *Triangle* on the modified `.poly` file:

```
triangle four_Architect.poly
```

This produces the all-important `.ele` file containing the triangulated elements which should now be correct. To check, we can run *showme* again to look at the connections (click on the button labelled ‘ele’ in the *upper* row of buttons). It turns out that the other characters all have the same problem of overlapping polygons. The procedure in each case is similar to that described except in the case of the ‘&’ and ‘\$’ characters, we

need to add two holes not one. Having fixed all the characters, we then re-run *font2dx* with the `--negate` and `--noclean` options as before.

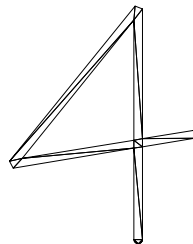


FIG. 15: *showme* rendering of character ‘4’ using the corrected `.poly` file.

It is important to remember to use `--noclean` — if you don’t the changes made to the `.poly` files will be lost! For this reason it’s a good idea to save your edited `.poly` files just in case!

Incidentally, I found that there are several versions of the *Architect* font available on the web. Most of them have problems, but not necessarily the same as those outlined here. For example, one version did not require the use of `--negate` but did need rescaling using `--scale=1000`. Another version had problems with a different set of characters.

VI. SUMMARY

font2dx has been tested on many fonts and has produced a clean font (i.e. one that did not require further manual adjustments) in the majority of cases (your mileage may vary!). Where there were flaws, such as in Adobe BrushScript, a workaround was always possible with a little effort.

There are almost certainly going to be some fonts which fail to convert — please let me know if *font2dx* fails completely on a particular font. To get an idea of the quality of fonts produced by *font2dx*, a sample set of fonts, originally produced in Postscript Type 1 format by URW and converted to *DX* format by a preliminary version of *font2dx* is available for download from the OpenDX web site⁸.

Acknowledgments

First, to George Williams, Wolfgang Glunz and Jonathan Shewchuk for making publicly available their fine packages, *fontforge*, *pstoedit* and *Triangle* which have made this work possible. Also, to the developers, past and present, of *DX* — thanks for the wonderful work you’ve done and are continuing to do. There are many other open source packages I’ve used in the making of this software, in particular the scripting language, *perl*, the plotting software, *gnuplot* and the $\text{\TeX}/\text{\LaTeX}$ typesetting system which produced this documentation. Thanks finally, to David Thompson of VIS. Inc. for his encouragement and for sorting out a bug which, without his timely help, I’d probably still be searching for.

* Electronic address: Jerry.Hagon@ncl.ac.uk

¹ *IBM Data Explorer User’s Guide*, pp307–312, 1997.

² G. Williams, *fontforge*, <http://fontforge.sourceforge.net>. *fontforge* was formerly known as *pfaedit*.

³ *Gnuplot* is a widely-used function plotting program, see <http://www.gnuplot.info> for details.

⁴ W. Glunz, *pstoedit*, <http://www.pstoedit.net>.

⁵ J. Shewchuk, *Triangle*, <http://www.cs.cmu.edu/~quake/triangle.html>.

⁶ P. Bourke, *Determining whether or not a polygon (2D) has its vertices ordered clockwise or counterclockwise*, <http://astronomy.swin.edu.au/~pbourke/geometry/clockwise>, 1998.

⁷ *Comp.Graphics.Algorithms FAQ*, §2 Subject 2.06.

⁸ <http://www.opendx.org>.

⁹ In several slightly different forms and a variety of formats.