

Font Creation with FontForge*

George Williams

444 Alan Rd.

Santa Barbara, CA 93109, USA

gww@silcom.com

<http://bibliofile.duhs.duke.edu/gww/>

Abstract

FontForge is an open source program which allows the creation and modification of fonts in many standard formats. This article will start with the basic problem of converting a picture of a letter into an outline image (used in most computer fonts). Then I shall describe the automatic creation of accented characters, and how to add ligatures and kerning pairs to a font, and other advanced features. Finally I shall present a few tools for detecting common problems in font design.

Résumé

FontForge est un logiciel libre *Open Source*, permettant la création et la modification de fontes dans plusieurs formats standard. Dans cet article nous allons d'abord présenter le problème de base de la conversion d'une image d'une lettre en une image vectorielle. Ensuite nous parlerons de la création automatique de caractères accentués et de l'adjonction de ligatures et de paires de crénage à une fonte. Enfin, nous présenterons quelques outils qui permettent de détecter des problèmes fréquents du processus de création de fonte.

Introduction

FontForge creates fonts, allows you to edit existing fonts, and can convert from one font format to another.

What is a font? A hundred years ago a font was a collection of little pieces of metal with the same height and one design for each of the letters of the alphabet and some extra symbols like punctuation.

But the world has changed. Fonts are more abstract now; they are described by data in a computer's memory. Three main types of computer fonts are in use today.

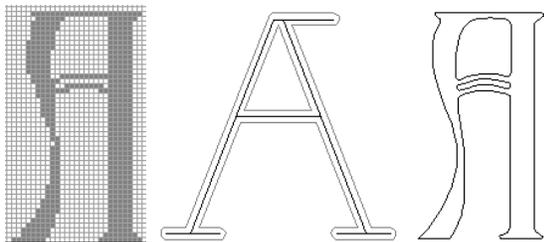
two main disadvantages: there needs to be a different design for each size of the font, and these little pictures end up requiring a large amount of memory.

The other two formats avoid these problems, but often require some reduction in output quality.

A *stroked font* expresses each glyph as a set of stems, with a line drawn down the center of the stem, and then the line is drawn (stroked) with a pen of a certain width.

The final type is an *outline font*. Each glyph is expressed as a set of contours, and the computer darkens the area between the contours. This format is a compromise between the above two: it takes much less space than the bitmap format, but more space than the stroked format, and it can provide better looking glyphs than the stroked format but not as nice as the bitmap format. It makes greater demands on the computer, however, as we shall see when we discuss hints, later on.

FIG. 1: bitmap, stroked and outline fonts



The simplest font type is a *bitmap font*. Each character (actually each glyph) in the font is a tiny little picture of that character expressed on a rectangular grid of pixels. This format can provide the best quality font possible with each glyph perfectly designed, but there are

What is a character? And a glyph? A character is an abstract concept: the letter “A” is a character, while any particular drawing of that character is a glyph. In many cases there is one glyph for each character and one character for each glyph, but not always.

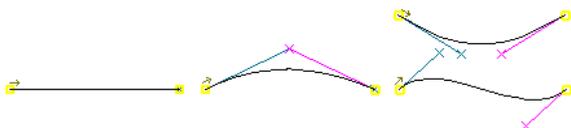
The glyph used for the Latin letter “A” may also be used for the greek letter “Alpha”, while in Arabic writing most Arabic letters have at least four different glyphs (often vastly more) depending on what other letters are around them.

*. Original title: *Font Creation with PfaEdit*.

What is a contour? A contour is just a closed path; each

glyph is composed of contours. Usually this path is composed of several curved segments called splines. Each spline is defined by two end points and either 0, 1 or 2 control points which determine how the spline curves. The more control points a spline has, the more flexible it can be.

FIG. 2: splines with 0, 1 and 2 control points



Font creation

You can create an empty font either by invoking FontForge with the `-new` argument on the command line

```
$ fontforge -new
```

or by invoking the `New` item from the `File` menu. In either case you should end up with a window like this:

FIG. 3: A newly created blank font

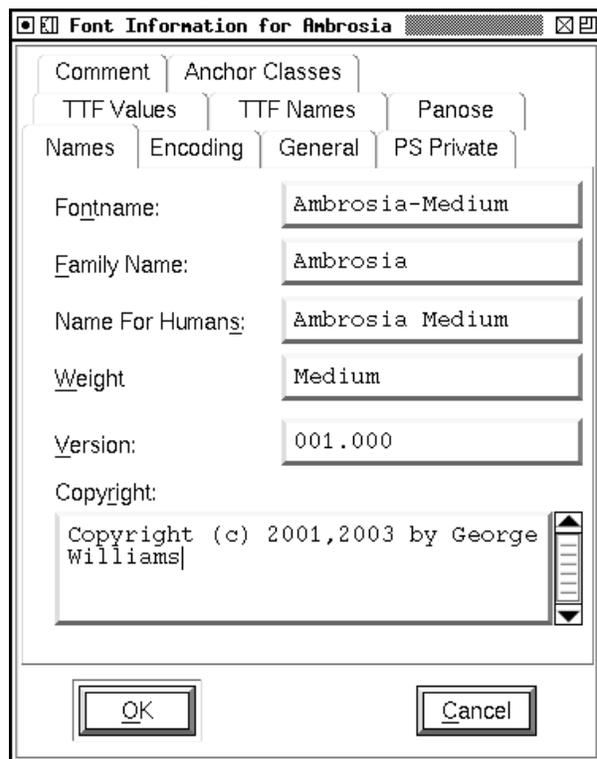
File	Edit	Element	Hints	View	Metric	Window									
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
p	q	r	s	t	u	v	w	x	y	z	{		}	~	

Such a font will have no useful name as yet, and will be encoded with the default encoding (usually Latin1). Use the `Element -> Font Info` menu item to correct these deficiencies. This dialog has several tabbed sub-dialogs; the first one allows you to set the font's various names (see fig. 4):

- the family name (most fonts are part of a family of similar fonts)
- the font name, a name for PostScript, usually containing the family name and any style modifiers
- and finally a name that is meaningful to humans

If you wish to change the encoding (to TeX Base or Adobe Standard perhaps) the `Encoding` tab will present you with a pulldown list of known encodings. If you are making a TrueType font then you should also go to the `General` tab and select an em-size of 2048 (the default coordinate system for TrueType is a little different from that of PostScript).

FIG. 4: Font name information



Character creation

Once you have done that you are ready to start editing characters; for the sake of example, let's create a capital 'C'. Double click on the entry for "C" in the font view above (fig. 3). You should now have an empty Outline Character window (fig. 5).

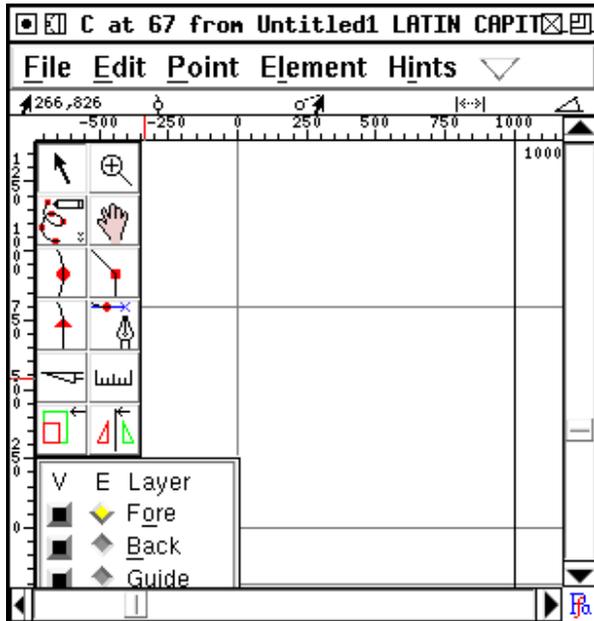
The outline character window contains two palettes smuggled up on the left side of the window. The top palette contains a set of editing tools, and the bottom palette controls which layers of the window are visible or editable.

The foreground layer contains the outline that will become part of the font. The background layer can contain images or line drawings that help you draw this particular character. The guide layer contains lines that are useful on a font-wide basis (such as a line at the x-height). To start with, all layers are empty.

This window also shows the character's internal coordinate system with the x and y axes drawn in light grey. A line representing the character's advance width is drawn in black at the right edge of the window. FontForge assigns a default advance width of one em (in PostScript that will usually be 1000 units) to a new character.

Select the `File -> Import` menu command to import an image of the character you are creating, assuming

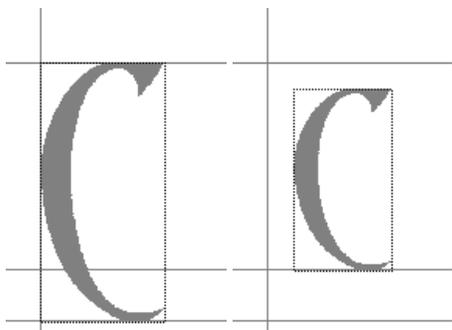
FIG. 5: An empty character



that you have one. It will be scaled so that it is as high as the em-square. In this case that's too big and we must rescale the image (fig. 6) as follows.

Make the background layer editable (by selecting the Back checkbox in the layers palette), move the mouse pointer to one of the edges of the image, hold down the shift key (to constrain the rescale to the same proportion in both dimensions), depress and drag the corner until the image is a reasonable size. Next move the mouse pointer onto the dark part of the image, depress the mouse and drag the image to the correct position.

FIG. 6: Background image

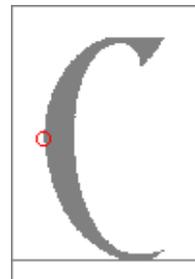


If you have installed the potrace or autotrace program you can invoke **Element -> AutoTrace** to generate an outline from the image; you should probably follow this by **Element -> Add Extrema** and **Element ->**

Simplify. But I suggest you refrain from autotracing, and trace the character yourself (results will be better).

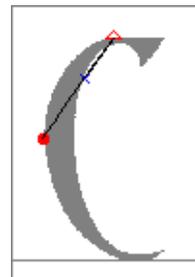
Change the active layer back to foreground (in the layers palette), and select the curve point tool from the tools palette (third from the top of the tools palette, on the left; the icon is a curve running through a circle). Then move the pointer to the edge of the image and add a point. I find that it is best to add points at places where the curve is horizontal or vertical, at corners, or where the curve changes inflection. (A change of inflection occurs in a curve like “S” where the curve changes from being open on the left to being open on the right.) If you follow these rules hinting will work better.

FIG. 7: Tracing 1: beginning



It is best to enter a curve in a clockwise fashion, so the next point should be added up at the top of the image on the flat section. Because the shape becomes flat here, a curve point is not appropriate, rather a tangent point is (the icon on the tools palette is the next one down, with a little triangle). A tangent point makes a nice transition from curves to straight lines because the curve leaves the point with the same slope the line had when it entered.

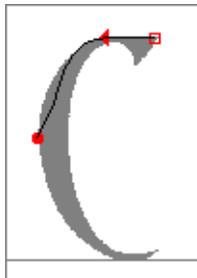
FIG. 8: Tracing 2: tangent point



At the moment this “curve” doesn't match the image at all. Don't worry about that, we'll fix it later; and anyway it will change on its own as we continue. Note that we now have a control point attached to the tangent point (the little blue x). The next point needs to go where the image changes direction abruptly. Neither a curve nor a tangent point is appropriate here, instead

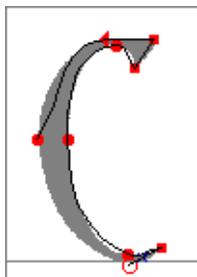
we must use a corner point (the icon on the tools palette with the little square).

FIG. 9: Tracing 3: corner point



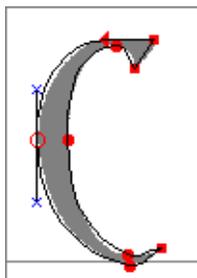
As you can see, the curve now starts to follow the image a bit more closely; we continue adding points until we are ready to close the path.

FIG. 10: Tracing 4: continuing



We close the path just by adding a new point on top of the old start point.

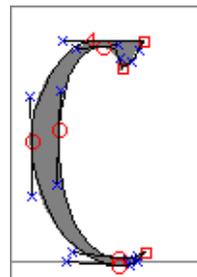
FIG. 11: Tracing 5: closing the curve



Now we want to make the curve track the image more closely; to do this, we must adjust the control points (the blue “x”es). To make all the control points visible select the pointer tool and double-click on the curve. Then move the control points around until the curve looks right.

Finally we set the advance width. Again with the pointer tool, move the mouse to the width line on the

FIG. 12: Tracing 6: make it look right



right edge of the screen, depress and drag the line back to a reasonable location.

And we are done with this character.

Navigating to characters

The font view provides one way of navigating around the characters in a font. Simply scroll around it until you find the character you need and then double click to open a window looking at that character.

Typing a character will move to that character.

But some fonts are huge (Chinese, Japanese and Korean fonts have thousands or even tens of thousands of characters) and scrolling around the font view is an inefficient way of finding your character. `View -> Goto` provides a simple dialog which will allow you to move directly to any character for which you know the name (or encoding). If your font is a Unicode font, then this dialog will also allow you to find characters by block name (e.g. there is a pull-down list from which you may select Hebrew rather than Alef).

The simplest way to navigate is just to go to the next or previous glyph. And `View -> Next Glyph` and `View -> Prev Glyph` will do exactly that.

Loading background images better

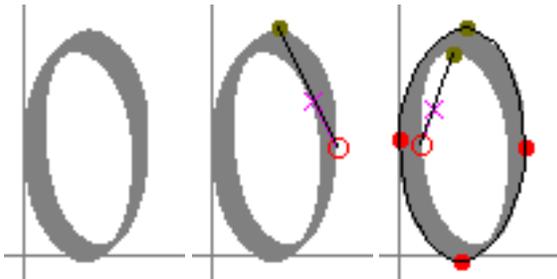
In the background image of the previous example the bitmap of the letter filled the canvas of the image (with no white borders around it). When FontForge imported the image it needed to be scaled once in the program. But usually when you create the image of the letter you have some idea of how much white space there should be around it. If your images are exactly one em high then FontForge will automatically scale them to be the right size. So in the following examples all the images have exactly the right amount of white space around them to fit perfectly in an em.

FontForge also has the ability to import an entire bitmap font (for example a “pk” or “gf” font produced by METAFONT or the “bdf” format developed by Adobe for bitmaps) to provide properly scaled background images for all characters in a font.

Creating the letter “o” — consistent directions

Let us turn our attention to the letter “o” which has a hole (or counter) in the middle. Open the outline view for the letter “o” and import a background image into it.

FIG. 13: Tracing o



Notice that the first outline is drawn clockwise and the second counter-clockwise. This change in drawing direction is important. Both PostScript and TrueType require that the outer boundary of a character be drawn in a certain direction (they happen to be opposite from each other, which is a mild annoyance); within FontForge all outer boundaries must be drawn clockwise, while all inner boundaries must be drawn counter-clockwise.

If you fail to alternate directions between outer and inner boundaries you may get results like the one on the left: . If you fail to draw the outer contour in a clockwise fashion the errors are more subtle, but will generally result in a less pleasing result when the character is rasterized.

Technical and confusing: the exact behavior of rasterizers varies. Early PostScript rasterizers used a “non-zero winding number rule” while more recent ones use an “even-odd” rule. TrueType uses the “non-zero” rule. The example given above is for the “non-zero” rule. The “even-odd” rule would fill the “o” correctly no matter which way the paths were drawn (though there would probably be subtle problems with hinting).

To determine whether a pixel should be set using the even-odd rule, draw a line from that pixel to infinity (in any direction) and count the number of contour crossings. If this number is even the pixel is not filled. If the number is odd the pixel is filled. Using the non-zero winding number rule, the same line is drawn, contour crossings in a clockwise direction add 1 to the crossing count, while counter-clockwise contours subtract 1. If the result is 0 the pixel is not filled; any other result will fill it.

The command `Element -> Correct Direction` looks at each selected contour, figures out whether it qualifies as an outer or inner contour and reverses the drawing direction when the contour is drawn incorrectly.

Creating letters with consistent stem widths, serifs and heights

Many Latin (and Greek and Cyrillic, LGC for short) fonts have serifs, that is, special terminators at the end of stems. And in almost all LGC fonts there should only be a small number of stem widths (i.e. the vertical stem of “l” and “i” should probably be the same).

FontForge does not have a good way to enforce consistency, but it does have various commands to help you check for it, and to find discrepancies.

Let us start with the letter “l” and go through the familiar process of importing a bitmap and defining its outline.

FIG. 14: Beginning “l”



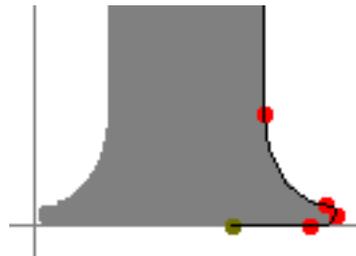
Use the magnify tool to examine the bottom serif, and note that it is symmetric left to right.

FIG. 15: Magnified “l”



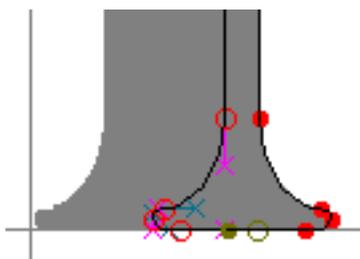
Trace the outline of the right half of the serif:

FIG. 16: Half traced “l”



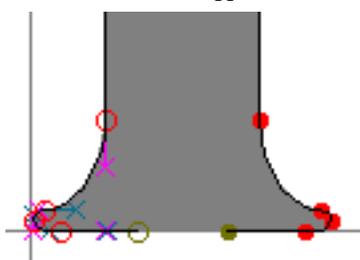
Select the outline, invoke `Edit -> Copy`, then `Edit -> Paste`; then invoke `Element -> Transform` and select `Flip` (from the pull down list) and check `Horizontal`.

FIG. 17: Pasted “l”



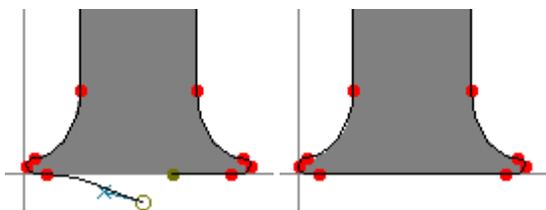
Drag the flipped serif over to the left until it snugles up against the left edge of the character:

FIG. 18: Dragged “l”



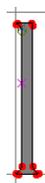
Deselect the path, and select one end point and drag it until it is on top of the end point of the other half.

FIG. 19: Joining “l”



Finish off the character.

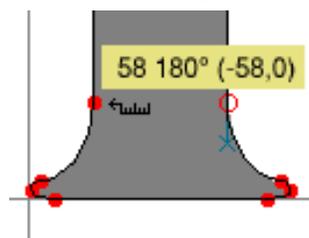
FIG. 20: Finished “l”



But there are two more things we should do. First let’s measure the stem width, and second let’s mark the height of the “l”.

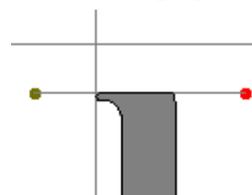
Select the ruler tool from the tool palette, and drag it from one edge of the stem to the other. A little window pops up showing the width is 58 units, the drag direction is 180 degrees, and the drag was -58 units horizontally, and 0 units vertically.

FIG. 21: Measuring stem width



Go to the layers palette and select the Guide radio box (this makes the guide layer editable). Then draw a line at the top of the “l”. This line will be visible in all characters and marks the ascent height of this font.

FIG. 22: Making a guideline

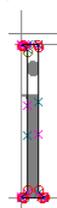


The “i” glyph looks very much like a short “l” with a dot on top. So let’s copy the “l” into the “i”; this will automatically give us the right stem width and the correct advance width

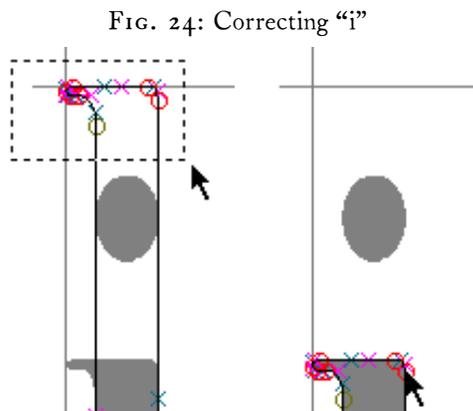
The copy may be done either from the font view (by selecting the square with the “l” in it and pressing Edit -> Copy) or from the outline view (by Edit -> Select All and Edit -> Copy). Similarly, the Paste may be done either in the font view (by selecting the “i” square and pressing Edit -> Paste) or the outline view (by opening the “i” character and pressing Edit -> Paste).

Now, import the “i” image, and copy and paste the “l” glyph.

FIG. 23: Import “i”

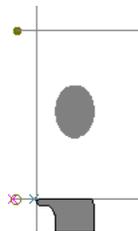


Select the top serif of the outline of the `l` and drag it down to the right height:



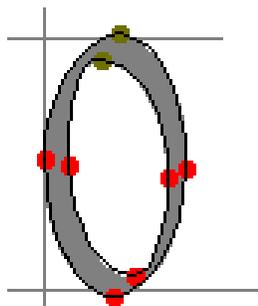
Go to the guide layer and add a line at the x-height:

FIG. 25: Making another guideline



Looking briefly back at the “`o`” we built before, you may notice that it reaches a little above the guideline we put in to mark the x-height (and a little below the baseline). This is called the overshoot or o-correction, and is an attempt to remedy an optical illusion. A curve actually needs to rise about 3% of its diameter above the x-height for it to appear on the x-height.

FIG. 26: Comparing “`o`” to guidelines



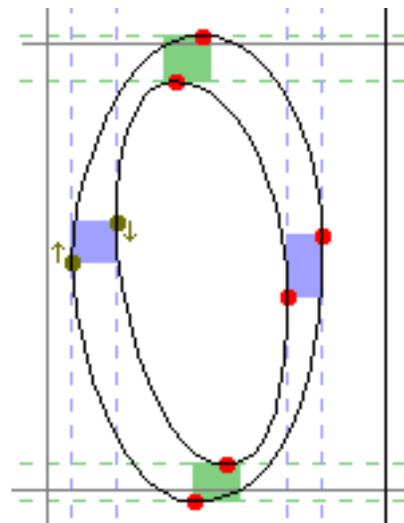
Continuing in this manner we can produce all the base glyphs of a font.

Hints

At small point sizes on display screens, computers often have a hard time figuring out how to convert a glyph’s outline into a pleasing bitmap to display. The font designer can help the computer out here by providing what are called *hints*.

Basically every horizontal and vertical stem in the font should be hinted. FontForge has a command `Element -> Autohint` which should do this automatically. Or you can create hints manually—the easiest way is to select two points on opposite sides of a stem and then invoke `Hints -> Add HHint` or `Hints -> Add VHint` respectively for horizontal or vertical stems.

FIG. 27: “`o`” with hints



Accented letters

Latin, Greek and Cyrillic all have a large complement of accented characters. FontForge provides several ways to build accented characters out of base characters.

The most obvious mechanism is simple copy and paste: Copy the letter “`A`” and Paste it to “`Ã`”; then Copy the tilde accent and Paste it Into “`Ã`”. (N.B. Paste Into is subtly different from Paste. Paste clears out the character before pasting, while Paste Into merges the clipboard into the character, retaining the old contents.) Then open up “`Ã`” and position the accent so that it appears properly centered over the `A`.

This mechanism is not particularly efficient; if you change the shape of the letter “`A`” you will need to regenerate all the accented characters built from it. To alleviate this, FontForge has the concept of a Reference to a character. Thus, you can Copy a Reference to “`A`”, and Paste it, then Copy a Reference to tilde and Paste it Into, and then again adjust the position of the accent over the `A`.

Then if you change the shape of the A the shape of the A in “ \tilde{A} ” will be updated automatically — as will the width of “ \tilde{A} ”.

But FontForge knows that “ \tilde{A} ” is built out of “A” and the tilde accent, and it can easily create your accented characters itself by placing the references in “ \tilde{A} ” and then positioning the accent over the “A”. (Unicode provides a database which lists the components of every accented character (in Unicode)). Select “ \tilde{A} ”, then apply **Element -> Build -> Build Accented** and FontForge will create the character by pasting references to the two components and positioning them appropriately.

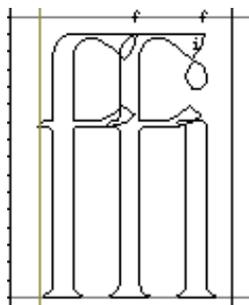
FontForge has a heuristic for positioning accents — most accents are centered over the highest point of the character — but sometimes this will produce bad results (e.g. if the one of the two stems of “u” is slightly taller than the other the accent will be placed over that stem, rather than being centered over the character), so you should be prepared to look at your accented characters after creating them. You may need to adjust one or two (or you may need to redesign your base characters slightly).

Ligatures

One of the great drawbacks of the standard Type 1 fonts from Adobe is that none of them come with “ff” ligatures. Lovers of fine typography tend to object to this. FontForge can help you overcome this flaw (whether it is legal to do so is a matter you must settle by reading the license agreement for your font). FontForge cannot create a nice ligature for you, but what it can do is put all the components of the ligature into the character with **Element -> Build -> Build Composite**. This makes it slightly easier (at least in Latin) to design a ligature.

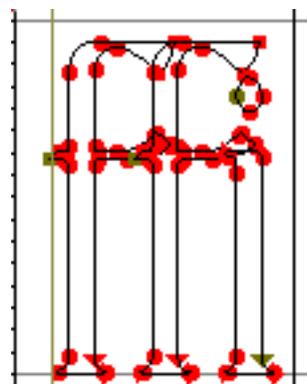
Use the **Element -> Char Info** dialog to name the character; we’ll use “ffi” as an example. This is a standard name and FontForge recognizes it as a ligature consisting of f, f and i. Apply **Element -> Default ATT -> Common Ligatures** so that FontForge will store the fact that it is a ligature. Then use **Element -> Build -> Build Composite** to insert references to the ligature components.

FIG. 28: ffi made of references



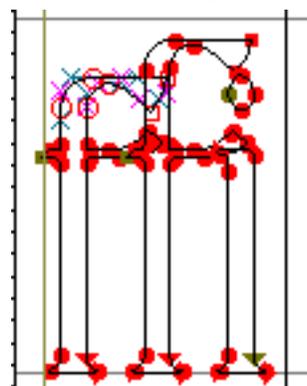
Use **Edit -> Unlink References** to turn the references into a set of contours.

FIG. 29: ffi without references



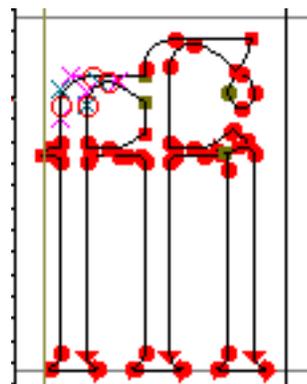
Adjust the components so that they will look better together. Here the stem of the first f has been lowered.

FIG. 30: ffi adjusted



Use **Element -> Remove Overlap** to clean up the character.

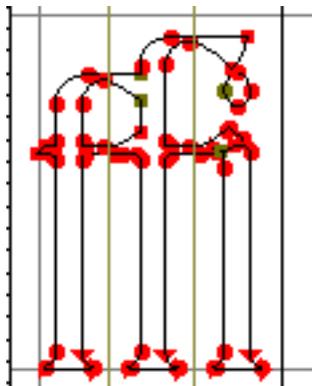
FIG. 31: ffi cleaned up



Some word processors will allow the text editing caret to be placed inside a ligature (with a caret position between each component of the ligature). This means that the user of that word processor does not need to know s/he is dealing with a ligature and sees behavior very similar to what s/he would see if the components were present. But for the word processor to be able to do this, it must have some information from the font designer giving the appropriate locations of caret positions.

As soon as FontForge notices that a character is a ligature it will insert enough caret location lines into it to fit between the ligature's components. FontForge places these on the origin, and if you leave them there FontForge will ignore them. But once you have built your ligature you might want to move the pointer tool over to the origin line, press the button and move the caret lines to their correct locations. (Only AAT and OpenType support this).

FIG. 32: ffi with ligature carets



Metrics

Once you have created all your glyphs, you should presumably examine them to see how they look together. There are three commands designed for this:

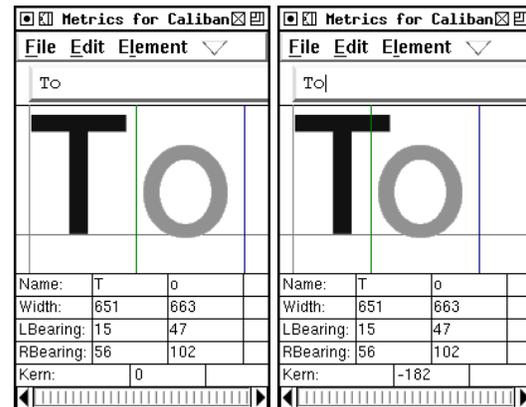
- `Windows -> New Metrics View` opens a window which displays several glyphs at a very large size. You can change the advance width of each glyph here to make a more pleasing image.
- `File -> Print` prints a sample text using the font, or all the glyphs of the font, or several glyphs one per page, or several glyphs at a waterfall of point sizes.
- `File -> Display` opens a dialog which allows you to display a sample text in this (or indeed several) fonts.

Kerning

Even in fonts with the most carefully designed metrics

there are liable to be some character combinations which look ugly. Some combinations are fixed by building ligatures, but most are best approached by kerning the inter-character spacing for that particular pair.

FIG. 33: kerning in the Metrics view



In the above example the left image shows the un-kerned text, the right shows the kerned text. To create a kerned pair, select the two glyphs, then use `Windows -> New Metrics View` and move the mouse to the right-most character of the pair and click on it, the line (normally the horizontal advance) between the two should go green (and becomes the kerned advance). Drag this line around until the spacing looks nice.

Checking a font for common problems

After you have finished making all the characters in your font, you should check it for inconsistencies. FontForge has a command, `Element -> Find Problems`, which is designed to find many common problems (as you might guess).

Simply select all the characters in the font and then bring up the `Find Problems` dialog. Be warned though: Not everything it reports as a problem is a real problem, some may be an element of the font's design that FontForge does not expect.

The dialog can search for many types of problems:

- Stems which are close to but not exactly some standard value.
- Points which are close to but not exactly some standard height.
- Paths which are almost but not quite vertical or horizontal.
- Control points which are in unlikely places.
- Points which are almost but not quite on a hint.
- and more ...

I find it best just to check for a few similar problems at a time; switching between different types of problems can be distracting.

Generating a font

The penultimate² stage of font creation is generating a font. N.B.: The File -> Save command in FontForge will produce a format that is only understood by FontForge and is not useful in the real world.

You should use File -> Generate to convert your font into one of the standard font formats. FontForge presents what looks like a vast array of font formats, but in reality there are just several variants on a few basic font formats: PostScript Type 1, TrueType, OpenType (and for CJK fonts, also CID-keyed fonts) and SVG.

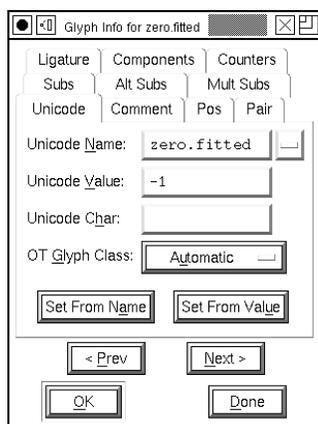
OpenType advanced typography

In OpenType and Apple's Advanced Typography fonts it is possible for the font to know about certain common glyph transformations and provide information about them to a word processor using that font (which presumably could then allow the user access to them).

Simple substitutions Suppose that we had a font with several sets of digits: monospaced digits, proportional digits and lower case (old style) digits. One of these styles would be chosen to represent the digits by default (say the monospaced digits). Then we could link the default glyphs to their variant forms.

First we should name each glyph appropriately (proportional digits should be named "zero.fitted", "one.fitted", and so forth, while oldstyle digits should be named "zero.oldstyle", "one.oldstyle", and so forth. Use the Element -> Glyph Info command to name them.

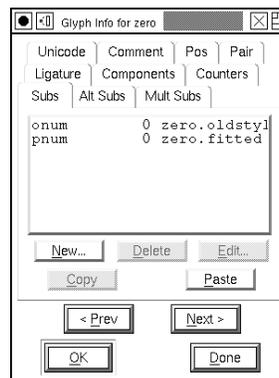
FIG. 34: Naming a glyph



2. The final stage of font creation would be installing the font. This depends on what type of computer you use and I shan't attempt to describe all the possibilities here.

To link the glyphs together we invoke Element -> Glyph Info again, this time on the default glyph and select the Subs tab. This provides a list of all simple substitutions defined for this glyph.

FIG. 35: Providing substitutions



Pressing the [New] button will allow you to add a substitution:

FIG. 36: Editing substitutions



Each substitution must contain: the name of a glyph to which it is to be mapped, a four-character OpenType tag used to identify this mapping, and a script and language in which this substitution is active. There is a pull-down menu which you can use to find standard tags for some common substitutions (the tag for oldstyle digits is 'onum'). This substitution is for use in the Latin script and for any language; again there is a pull-down menu to help choose this correctly.

Contextual substitutions OpenType and Apple also provide contextual substitutions. These are substitutions which only take place in a given context and are essential for typesetting Indic and Arabic scripts.

In OpenType a context is specified by a set of patterns that are tested against the glyph stream of a document. If a pattern matches, then any substitutions it defines will be applied.

Instead of an Indic example, let us take something I'm more familiar with, the problem of typesetting a

Latin script font where the letters “b”, “o”, “v”, and “w” join their following letter near the x-height, while all other letters join near the baseline.

Thus we need two variants for each glyph, one that joins (on the left) at the baseline (the default variant) and one which joins at the x-height. Let us call this second set of letters the “high” letters and name them “a.high”, “b.high”, and so forth.

FIG. 37: Incorrect & correct script joins

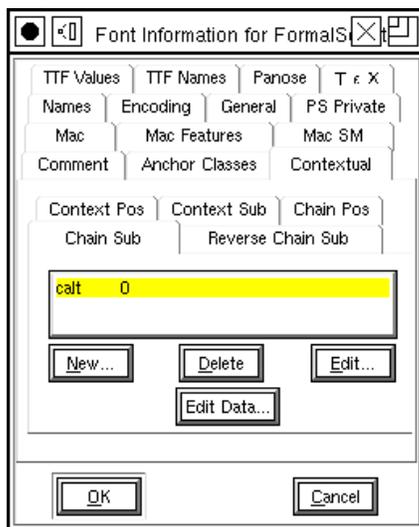


We divide the set of possible glyphs into three classes: the letters “boww”, all other letters, and all other glyphs. We need to create two patterns; the first will match a glyph in the “boww” class followed by a glyph in the “boww” class, while the second will match a glyph in the “boww” class followed by any other letter. If either of these matches the second glyph should be transformed into its high variant.

The first thing we must do is create a simple substitution mapping for each low letter to its high variant. Let us call this substitution by the four character OpenType tag “high”. We use Element -> Glyph Info as before except that here we use the special “script/language” called “—Nested—” (an option in the pull-down menu).

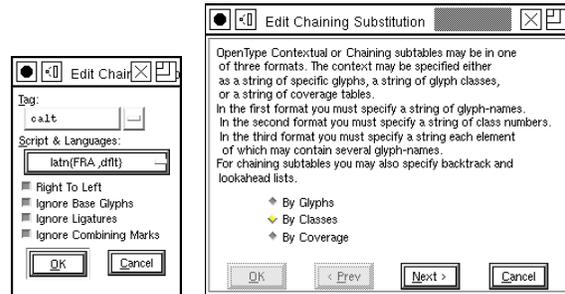
The tricky part is defining the context. This is done with the Contextual tab in the Element -> Font Info dialog, revealing five different types of contextual behavior. We are interested in contextual chaining substitutions.

FIG. 38: Font Info showing Contextual Subs



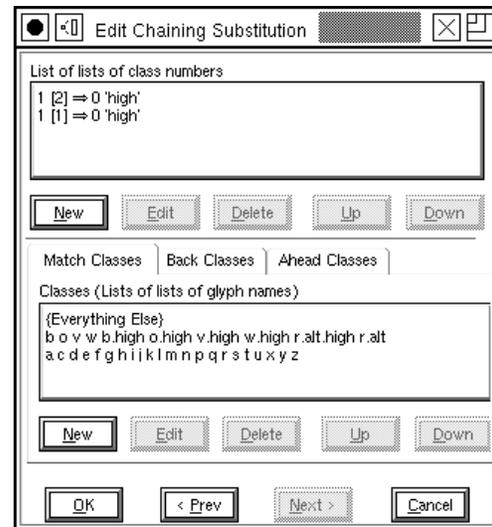
You can add a new entry by pressing the [New] button. This brings up a series of dialogs. The first requests a four character OpenType tag and a script/language, much as we saw earlier. The next dialog allows you to specify the overall format of the substitution.

FIG. 39: Tag & Script dialog and format of contextual chaining substitution



The next dialog finally shows something interesting. At the top are a series of patterns to match and substitutions that will be applied if the string matches. Underneath that are the glyph classes that this substitution uses.

FIG. 40: Overview of the contextual chaining substitution



A contextual chaining dialog divides the glyph stream into three categories: those glyphs before the current glyph (these are called backtracking glyphs), the current glyph itself (you may specify more than one, and this (these) glyph(s) may have simple substitutions applied to them), and finally glyphs after the current glyph (these are called lookahead glyphs).

Each category of glyphs may divide glyphs into a different set of classes, but in this example we use the

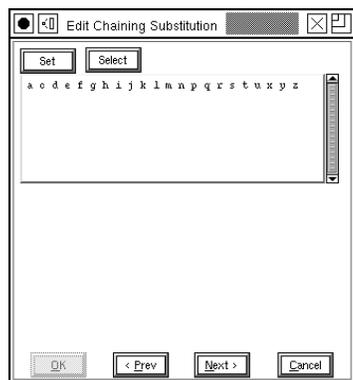
same classes for all categories (this makes it easier to convert the substitution to Apple’s format).

The first line (in the “Lists of lists” field) should be read thus: If a backtracking glyph in class 1 is followed by a match glyph in class 2, then location 0 in the match string (that is, the first glyph) should have the simple substitution ‘high’ applied to it. If you look at the glyph class definitions you will see that class 1 includes those glyphs which must be followed by a high variant, so this seems reasonable.

The second line is similar except that it matches glyphs in class 1. Looking at the class definitions we see that classes 1 & 2 include all the letters, so these two lines mean that if any letter follows one of “bovw” then that letter should be converted to its ‘high’ variant.

To edit a glyph class, double click on it. To create a new one press the [New] button (under the class list).

FIG. 41: Editing glyph classes



This produces another dialog showing all the names of all the glyphs in the current class. Pressing the [Select] button will set the selection in the font window to match the glyphs in the class, while the [Set] button will do the reverse and set the class to the selection in the font window. These provide a shortcut to typing in a lot of glyph names. Pressing the [Next] button defines the class and returns to the overview dialog.

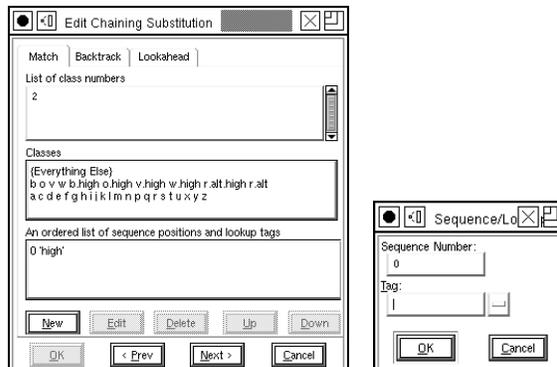
To edit a pattern double click on it, or to create a new one press the [New] button (under “Lists of lists”).

Again the pattern string is divided into three categories, those glyphs before the current one, the current one itself, and any glyphs after the current one. You choose which category of the pattern you are editing with the tabs at the top of the dialog. Underneath these is the subset of the pattern that falls within the current category, the classes defined for this category, and finally the substitutions for the current glyph(s). Clicking on one of the classes will add the class number to the pattern.

To edit a substitution double click on it, or to create a new one press the [New] button (under “An ordered

list ...”). The sequence number specifies which glyph among the current glyphs should be modified, and the tag specifies a four character substitution name.

FIG. 42: Adding matches and substitutions



Apple advanced typography

Some of Apple’s typographic features can be readily converted into equivalent OpenType features, while others cannot be.

Non-contextual ligatures, kerning and substitutions can generally be converted from one format to another. Apple uses a different naming convention and defines a different set of features, but as long as a feature of these types is named in both systems, conversion is possible.

Contextual substitutions Apple specifies a context with a finite state machine, which is essentially a tiny program that looks at the glyph stream and decides what substitutions to apply.

Each state machine has a set of glyph class definitions (just as in the OpenType example), and a set of states. The process begins in state 0 at the start of the glyph stream. The computer determines what class the current glyph is in and then looks at the current state to see how it will behave when given input from that class. The behavior includes the ability to change to a different state, advancing the input to the next glyph, applying a substitution to either the current glyph or a previous one (the “marked” glyph).

Using the same example of a Latin script font ...

We again need a simple substitution to convert each letter into its high alternate. The process is the same as it was for OpenType, and indeed we can use the same substitution.

Again we divide the glyphs into three classes (Apple gives us some extra classes whether we want them or not, but conceptually we use the same three classes as in the OpenType example). We want a state machine with two states (again Apple gives us an extra state for free, but we shall ignore that); one is the start state (the base state,

where nothing changes), and the other is the state where we've just read a glyph from the "bovw" class.

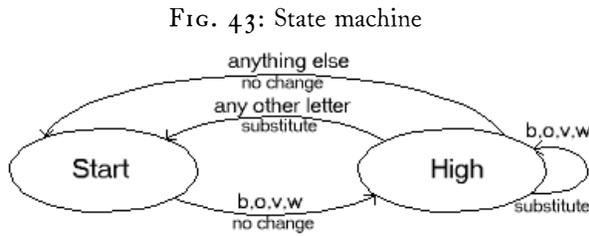
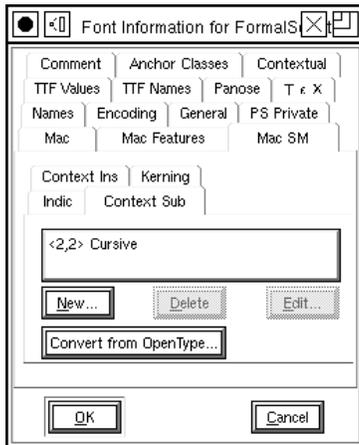


FIG. 43: State machine

Again we use the Element -> Font Info dialog and the Mac SM tag to look at the contextual substitutions available. Again there are several types of contextual behavior, and we are interested in contextual substitutions.

FIG. 44: Font Info showing State Machines



Double clicking on a state machine, or pressing the [New] button provides an overview of the given state machine. At the top of the dialog we see a field specifying the feature/setting of the machine; this is Apple's equivalent of the OpenType four-character tag. Under this is a set of class definitions, and at the bottom is a representation of the state machine itself. See fig. 45.

Double clicking on a class brings up a dialog similar to that used in OpenType:

FIG. 46: Editing Apple glyph classes

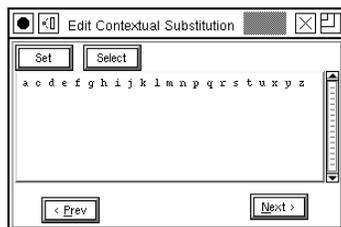
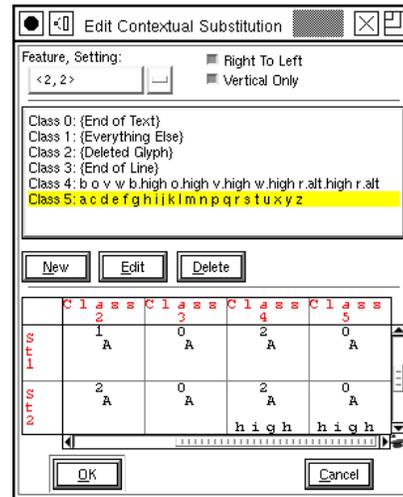


FIG. 45: Overview of a State Machine

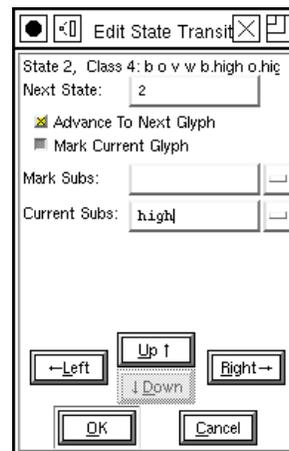


Clicking on a transition in the state machine (there is a transition for each state/class combination) produces a transition dialog.

This controls how the state machine behaves when it is in a given state and receives a glyph in a given class. In this example it is in state 2 (which means it has already read a "bovw" glyph), and it has received a glyph in class 4 (which is another "bovw" glyph). In this case the next state will be state 2 again (we will have just read a new "bovw" glyph), read another glyph and apply the "high" substitution to the current glyph.

At the bottom of the dialog is a series of buttons that allow you to navigate through the transitions of the state machine.

FIG. 47: Transition dialog



Pressing [OK] many times will extract you from this chain of dialogs and add a new state machine to your font.

Appendix: Additional features

FontForge provides many more features, further descriptions of which may be found at

<http://fontforge.sf.net/overview.html>

Here is a list of some of the more useful of them:

- Users may edit characters composed of either third order Bézier splines (for PostScript fonts) or second order Béziars (for TrueType fonts) and may convert from one format to another.
- FontForge will retain both PostScript and TrueType hints, and can automatically hint PostScript fonts.
- FontForge allows you to modify most features of OpenType's GSUB, GPOS and GDEF tables, and most features of Apple's morx, kern, lcar and prop tables. Moreover it can often convert from one format to another.
- FontForge has support for Apple's font formats. It can read and generate Apple font files both on and off a Macintosh. It can generate the FOND resource needed for the Mac to place a set of fonts together as one family.
- FontForge allows you to manipulate bitmap fonts as well as outline fonts. It has support for many formats of bitmap fonts (including TrueType's embedded bitmaps — both the format prescribed by Apple and that specified by Microsoft).
- FontForge can interpolate between two fonts (subject to certain constraints) to yield a third font between the two (or even beyond). For instance given a “Regular” and a “Bold” variant it could produce a “DemiBold” or even a “Black” variant.
- FontForge also has a command (which often fails miserably) which attempts to change the weight of a font.
- FontForge can automatically guess at widths for characters, and even produce kerning pairs automatically.
- FontForge has some support for fonts with vertical metrics (in Japanese, Chinese and Korean fonts), and some support for right to left fonts (Arabic, Hebrew, Linear-B, etc.).
- FontForge has some support for PostScript (and pdf) Type 3 fonts, allowing for glyphs with different strokes and fill and images.
- FontForge has a scripting language which allows batch processing of many fonts at once.