

Prettybird: A DSL for Programmatic Font Compilation

ANONYMOUS

ΣPHINX OF BLACK QUARTZ
JUDGE MY VOW

Fig. 1. A pangram of the Latin alphabet using the font "Prettybird Roman", compiled with Prettybird.

Improvements in display technology over time have led to an increase in public interest in the field of digital typography. As a result, the interest in non-standard font usage has grown over time. As the software has become more common, font design has become more accessible to beginners. However, the creation of a new font requires a large amount of time, and achieving proficiency in this art form requires even more. These hurdles hinder the ability of beginners to participate in this form of expression.

In this work we present Prettybird, a domain-specific functional programming language to simplify the early steps of font design. With a simple grammar and reusable functions, Prettybird makes font design more approachable to beginners while reducing the amount of time required for experts to begin a new font. Additionally, Prettybird avoids pitfalls of METAFONT, a previous attempt at font programming. Our anonymous survey shows that both programmers and font designers prefer Prettybird over METAFONT in terms of readability and writability.

CCS Concepts: • **Software and its engineering** → **Domain specific languages**; *Source code generation*; • **Human-centered computing** → *Human computer interaction (HCI)*.

Additional Key Words and Phrases: typefaces, fonts, font design, functional programming

ACM Reference Format:

Anonymous. 2022. Prettybird: A DSL for Programmatic Font Compilation. In . ACM, New York, NY, USA, 12 pages. <https://doi.org/XXXXXXXX.XXXXXXX>

1 INTRODUCTION

Despite the growing interest in non-standard fonts over time, as seen in Fig. 2, font design tools remain largely similar to one another, and almost exclusively paid. Table 1 compares the popularity of common font design software, revealing that that users have the choice between two dominant products: FontLab, a proprietary editor with a large entry fee, or FontForge, a free program with minimal developer support and a small community.

Additionally, interest in non-standard fonts for use in webpages has grown as the internet ages. In the HTTP Archive's 2022 Web Almanac report [12], it was measured that in August, about 83.8% of all desktop sites use web fonts, where a "web font" is defined as a font file hosted by a website, intended to be downloaded and rendered instead of using one of the standard fonts built-in to a

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PLDI '23, June 19–21, 2023, Orlando, FL

© 2022 Association for Computing Machinery.

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...\$15.00

<https://doi.org/XXXXXXXX.XXXXXXX>

Graphical Font Design Software	Lifetime Price	Google Trends RSV at start of year				
		2011	2014	2017	2020	2022
FontLab 8	\$499	68	49	35	25	27
FontForge	\$0	65	49	32	26	25
Fontographer 5	\$259	44	21	8	4	4
FontCreator 14	\$49	32	18	11	6	5
Robofont 4	\$490	0	3	2	2	3

Table 1. Popular graphical font design software mapped to the lifetime price for their base package, and their Google Relative Search Volumes [11]

user’s web browser. This measurement has doubled since April 2014, revealing a large increase in web designers’ preference to use non-standard fonts over time.

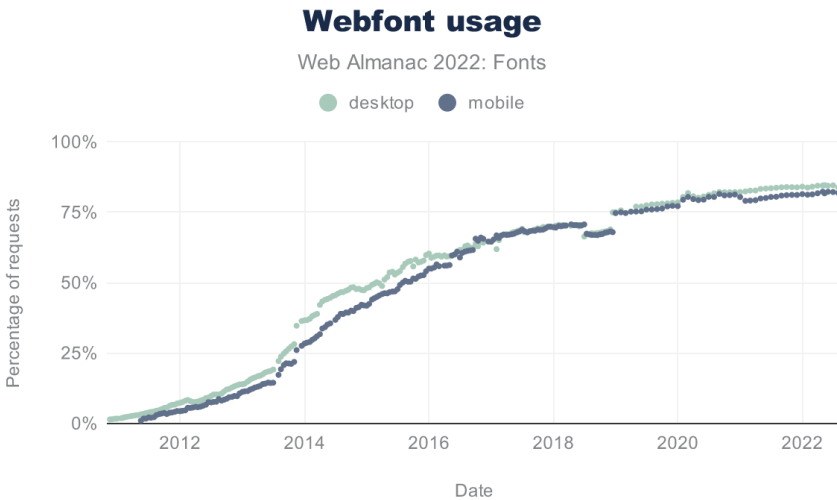


Fig. 2. Webfont usage [12] from Nov 15 2010 to Aug 1 2022.

Webpage designers who want to use custom fonts have experienced a reduction in the number of available font design tools over the past 11 years, and the remaining tools are either expensive or outdated. For these reasons, we introduce Prettybird, a domain-specific functional programming language for font design. A successor to METAFONT, Prettybird intends to provide a user experience that is comfortable for users at all skill levels of programming and font design, and also provides higher-level features like recursion to allow for more intricacy and complexity in glyphs. We develop Prettybird with three contributions:

1.1 Language Design

Our first contribution is the design of a language for creating bitmap and scalable outline fonts. Prettybird supplements the font design process for both new and experienced font designers primarily through its ability to define and reuse functions. Function reuse reduces redundancy across similar glyph components, as many glyphs in a font often reuse components such as serifs on capital letters in Roman fonts or tittles above lowercase "i" and "j" glyphs.

Additionally, Prettybird commits to using only four basic functions to build fonts, lowering the barrier to entry for new users even compared to graphical font editors, whose interfaces tend to include many options for experienced font designers.

Font design is a graphical medium and is accordingly dominated by graphical font design software. Text-based font design software like METAFONT has largely been abandoned now that the technical limitations of its design era [1] have been resolved, allowing for graphical font design software to become standard. Prettybird's intended utility is in its revitalization of text-based font design, rather than replacing graphical font design software.

1.2 Compiler Implementation

Our second contribution is the implementation of a Prettybird compiler that produces BDF-, SVG-, and TTF-formatted font files from Prettybird source code. The compiler utilizes the Lark [9] parsing library to parse user input. Once the parse tree has been converted to Prettybird's internal representation, it applies simple post-processing to ensure that user-created fonts are scaled properly by default. This ensures that Prettybird-compiled fonts of point size n are roughly the same height as other fonts of point size n . Finally, the internal representation can be compiled either to a bitmap font or outline font via FontForge's Python scripting API [2]. The specifics of either format are described below:

- *Bitmap Font* - A plaintext file following the Glyph Bitmap Distribution Format (BDF), generated via a custom BDF exporter. These BDF files can then be converted to the Portable Compiled Format (PCF), a format encoding identical information, compressed for faster load speeds. Additionally, bitmap fonts can be converted to scalable bitmap fonts that retain their bitmapped appearance at high resolutions via the FontForge API.
- *Outline Font* - Either a plaintext Scalable Vector Graphics (SVG) file or a binary TrueType Font (TTF) file.

In its current form, the Prettybird compiler suffers from two technical limitations when dealing with outline fonts: overlapping shapes and curve filling. These limitations are not concerns if bitmap output is chosen, as they stem from inconsistencies between SVG rendering and TTF rendering, and bitmap fonts are rendered within the compiler itself.

- *Overlapping Shapes* - The Prettybird compiler treats all outline fonts first as SVG data. This SVG data is passed into FontForge to be converted to TTF data. However, while SVG renderers will draw overlapping shapes on top of one another, TTF renderers will only do so if the contours of the shapes are drawn in the same direction (clockwise or counterclockwise). The SVG format does not allow for contour direction control, so overlapping shapes will cause the intersection of overlapping shapes to be blank rather than filled. This limitation is planned to be resolved in future work as the Prettybird compiler's backend is rewritten to make stronger use of FontForge's scripting API instead of relying on SVG conversions.
- *Curve Filling* - The Prettybird compiler offers the `filled` keyword for its fundamental functions, filling their areas. However, filling shapes is not currently supported for user-defined functions. This limitation is planned to be resolved in future work as filling complex contours with the FontForge API is a simple task.

"Prettybird Roman", the font used in Fig. 1 and Fig. 6 experiences both of these issues. The font file was manually modified in FontForge after being compiled from its source code in order to fill in serif curves and remove overlapping intersections.

1.3 Study on Preference of Prettybird over METAFONT

Our final contribution is a study on user preferences between Prettybird and METAFONT. Respondents self-identified their proficiency in programming and font design, and were then asked about the readability and writability of code samples of the two languages that produce similar glyphs. All four demographics listed above overwhelmingly prefer Prettybird over METAFONT in terms of readability and writability for samples utilizing vectors and Bezier curves.

2 BACKGROUND

In this section we describe the definitions and workflows of font design, as well as definitions necessary to describe the internal functionality of the Prettybird compiler.

2.1 Characters and Fonts

In font design, a *character* is a symbol representing a letter or number, or some other category of mark such as an arithmetic operation. A character could be implemented as many different *glyphs*, which are representations of a character in different styles, such as bolded or italicized.



Fig. 3. A collection of glyphs for the lowercase character "a", in the Times New Roman font, in the styles normal, bolded, italicized, and bolded-italicized respectively.

A *typeface* defines a set of design principles for what characters should look like, including their shapes, stroke thicknesses, or whether or not they have *serifs* (a small stylistic stroke or mark near the end of a larger stroke of a glyph). A typeface could be implemented as many different *fonts*, which are interpretations of typefaces under some style, such as bolded, italicized, or thin.



Fig. 4. Roboto Thin and Roboto Black, two fonts for the typeface Roboto.

2.2 Hinting

Font hinting is the practice of including instructions in font data regarding how to render glyphs at lower resolutions, such that the glyphs are still legible and match the style of the font. Many common methods for hinting exist, however the most common, used by TrueType and many other modern formats, is grid-fit hinting. In grid-fit hinting, a font rasterizer will morph the provided outline of a glyph such that its distinctive characteristics remain distinctive at small sizes.

Manual hinting is also commonly used, especially in glyphs that require more detail, such as script fonts or fonts with characters that require more strokes.



Fig. 5. The string "WATER" in the font Times New Roman, with overlapping glyph areas highlighted.

2.3 Kerning

Font kerning is the practice of adjusting the space in between pairs of glyphs in a font to achieve a more visually appealing result. Popular fonts such as Futura and Helvetica contain large amounts of manual kerning data to ensure that the fonts are maximally legible.

Optical kerning is an existing method of automatic kerning that uses outline information to compute the optimal spacing between pairs of characters. Automatic optical kerning is offered by most font design software, however still requires user setup and intervention throughout the process.

2.4 Existing Font Programming Approaches

Prettybird is designed as a successor to METAFONT, a previous approach to font programming first published by Knuth in 1979 [6] and later refined in 1986 [7][8]. METAFONT was initially designed as a "system for the design of alphabets suited to raster-based devices that print or display text" [7], however since its creation multiple methods for converting the bitmap font files generated by METAFONT into scalable outline fonts have been published [5][10].

3 MOTIVATING EXAMPLE

Prettybird's offering of functions provides a feature not offered in many graphical font editors: reusable font components such as serifs. The code example in Fig. 9 defines two reusable serif functions that are applied to the uppercase "I" and "T" glyphs. Serif fonts make heavy reuse of serifs, so functionality to reuse these shapes widely across multiple glyphs greatly reduces the time spent adding serifs to symbols.

As a proof of concept, we designed "Prettybird Roman", the font used in Fig. 1 using these serif functions, as well as a bubble function used for the round components in the glyphs B, C, D, G, P, and R. Code samples can be found in Appendix A.



Fig. 6. The glyphs generated by Fig. 9

4 PRETTYBIRD

Prettybird is a high-level, domain-specific functional programming language designed to simplify the creation of new fonts. This section describes the definitions of operators and operands, as well as the language's functional behavior.

4.1 Language Overview

Prettybird is intended to be a standalone language, with each program corresponding to a compiled font. Each program consists of a set of glyph declarations. Glyph declarations consist of a **base**

Rule	Description
program : (character function_declaration)+	Declares a font consisting of characters and function declarations
character : "char" identifier "{" statement* "}"	Declares a character consisting of operations on glyph space
statement : "base" base_statement	Defines the appearance of the glyph's space before operations are performed on it
"steps" "{" (update_mode [atom function_call])* "}"	Defines the set of operations performed on the glyph's space
update_mode : ("draw" "erase") "filled"?	Determines whether an atom will fill a pixel in glyph space, or clear it, and whether the atom will operate on all points within its boundary
atom : "vector" "(" point "," point ")"	Defines an operation on glyph space given a set of inputs
"rectangle" "(" point "," point ")"	
"ellipse" "(" point "," point ")"	
"bezier" "(" point "," point "," point ")"	
point : "(" number "," number ")"	Designates a point in space

Table 2. Fundamental subset of Prettybird grammar

declaration, describing the size and appearance of the glyph space before any functions are applied to it, and a **steps** declaration, describing an ordered list of functions to apply to the glyph space. The grammar for this behavior is described in Table 2. Prettybird utilizes an immutable number system consisting of number and point data operated on by a set of built-in, fundamental functions called atoms.

4.2 Number Data

Number data is represented either as a signed floating-point value, or a pair of signed floating-point values also referred to as a point. These values are strictly immutable; data manipulation is accomplished through recursion in user-defined functions. The language implements a simple mathematics system with plus, minus, multiply, divide, exponent, and modulus operators. Additionally, the logical bitwise operators and, or, and xor are implemented for extended utility. Operators take either numbers or points as their operands, and are type-agnostic. Operating on two numbers trivially provides a single number result. Operating on two pairs maps the operation between each item in the pair to an output point, e.g.

$$(a, b) * (c, d) = (a * c, b * d),$$

and operating between a number and a pair will broadcast the number to a duplicate pair and perform a pairwise operation, e.g.

$$a - (b, c) = (a, a) - (b, c) = (a - b, a - c).$$

4.3 Atoms

An atom, or atomic operation, denotes a primitive operation that modifies the contents of a glyph space given a set of inputs. We chose vectors, rectangles, ellipses, and Bezier curves as atoms for

the language because they make up the set of fundamental building blocks of most fonts. A short description of each atom follows:

- **vector(point, point)** - Applies a vector pattern to glyph space from the first point to the second point.
- **rectangle(point, point)** - Applies a rectangle pattern to glyph space, using the input points as the top-left corner and the bottom-right corner of the rectangle.
- **ellipse(point, point)** - Applies an ellipse pattern to glyph space, using the input points as the top-left corner and the bottom-right corner of the rectangle in which the ellipse is inlaid.
- **bezier(point, point, point)** - Applies a Bezier curve pattern to glyph space using the input points as the curve's two endpoints, and the control point.

In the language's implementation, there are additional provided operations such as point, circle, square, and spline that can be derived from **vector**, **ellipse**, **rectangle**, and **bezier**. For an improved user experience, some overloaded atoms have been included in the compiler implementation to provide alternative ways to apply these atoms to the glyph space, such as **ellipse(point, number, number)**, which applies an ellipse pattern given a center, width, and height.

4.4 Functional Behavior

Prettybird functions are defined with an identifier and a set of parameters, followed by a set of statements. Each function call spawns a blank copy of glyph space to operate on. Each copy of glyph space is added to a glyph space stack for the current character being operated on. When a function terminates, it applies its own glyph space to the glyph space underneath it in the stack via a binary "OR" operation, essentially overlaying the above space onto the below space. Fig. 7 illustrates this behavior, with **left_bar** as a function that draws a vector filling the leftmost column, **bottom_bar** as a function that draws a vector filling the bottommost row, and **corner_dot** as a function that draws a point filling the top-right corner.

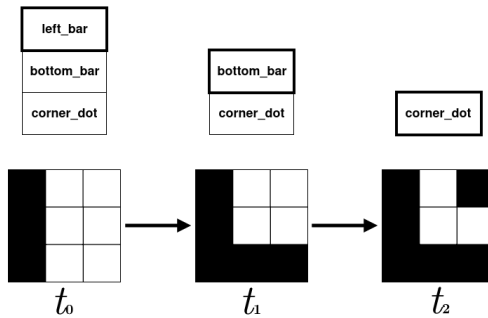


Fig. 7. Function stack execution over time.

The motivation behind the glyph space stack is ease of use; because functions spawn their own glyph space, functions may be written without the concern of overwriting a character's glyph space with "temporary" data. For example, a function to apply a parabolic pattern to glyph space might first draw an ellipse to its local space, then erase an arc of the ellipse to produce the desired shape. Without the worry of overwriting a character's base glyph space, the bottom arc of the ellipse may be erased without the concern of erasing other important data.

Rule	Description
function_declaration : "define" identifier function_parameter_list : "{" (update_mode [atom function_call])* "}"	Declares a function given a function identifier and a list of operations on glyph space
function_parameter_list : "(" function_parameters* ")" function_parameters : identifier "," function_parameters identifier	
function_call : identifier "(" [function_call_parameters] ")" function_call_parameters : value "," function_call_parameters value	Defines a list of identifiers to be used as function parameter names
value : point number identifier	
	Calls a function given a set of values

Table 3. Extended grammar for functions

In the interest of a simple grammar, user-defined functions do not specify the type of their arguments. However, atoms implicitly do specify the type of their arguments due to their implementation in the Prettybird compiler. Because the language only supports two datatypes, number and point, this application of the duck typing paradigm[3] is better suited for new programmers while largely avoiding the issue of propagating incorrect data types through multiple function calls that other duck-typed languages experience.

4.5 Hinting and Kerning

By default, the Prettybird compiler applies automatic hinting via FontForge. An advantage of font programming languages over standard font editing practice is the notion that a font designer’s intentions are described via the language, rather than obscured by the shape of the glyph itself [7]. We can provide enhanced hinting due to the fact that users are already manually hinting their fonts as they type out Prettybird commands, however we save this development for future work.

Additionally, the compiler applies rudimentary automatic kerning to compiled fonts, ensuring that characters have a minimum empty width between each other such that they can be easily distinguished.

5 EVALUATION

We evaluate Prettybird’s readability and writability through a self-reporting survey comparing near-equivalent glyphs generated by Prettybird and METAFONT code samples.

We received 63 anonymous responses to the survey and asked respondents to self-report how much time they have spent programming, how much time they have spent designing fonts, and their familiarity with Bezier curves on a scale of 1 ("Not familiar at all") to 5 ("Extremely familiar"). From the results, 58 respondents were self-described as programmers (having written "any kind of code"), 10 respondents were self-described as font designers, and 22 respondents were self-described as familiar with Bezier curves.

In addition to self-reporting their familiarity with these topics, respondents were shown the two capital "I" glyphs in Fig. 10; one was generated from an example in The METAFONTbook [7], and one was generated by Prettybird. These code samples were not labeled with the name

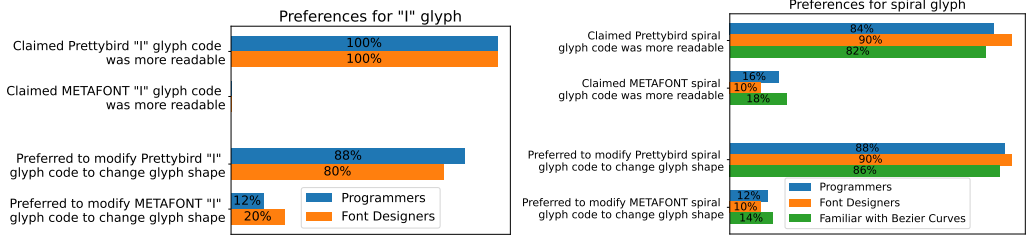


Fig. 8. Preferences between Prettybird and METAFONT for the capital "I" and spiral glyphs.

of the language and did not utilize syntax highlighting. The METAFONT sample used the same formatting as it was presented in the original text.

Respondents were asked which program was easier to read and which program they would prefer to modify if they wanted to make changes to how the glyphs looked. 100% of respondents preferred Prettybird for readability, and between 80% and 88% of respondents across all demographics preferred Prettybird for writability as seen in Fig. 8.

A similar question was asked of the set of spiral glyphs in Fig. 11, again one generated from a METAFONT sample [4], and one generated by Prettybird. This example contrasts with the capital "I" glyph example, as both "I" glyphs are roughly identical, however the METAFONT spiral glyph uses cubic Bezier curves while the Prettybird spiral uses quadratic Bezier curves. Again, Prettybird was ranked easier to read and modify among those who self-reported as familiar with Bezier curves, as shown in Fig. 8. Both the readability and modifiability preferences for the spiral glyph code are also matched among the programmer and font designer demographics.

These results describe a preference among both programmers and font designers of Prettybird’s grammar and program structure for glyphs that use the common `vector` and `bezier` commands. The size of the margins of this preference indicate that a possible reason for the decline in use of METAFONT was the complexity of its programs rather than a lack of utility of font programming as a whole.

6 CONCLUSION AND FUTURE WORK

We propose Prettybird as an accessible and powerful functional programming language for programmatic font design. Prettybird offers a simplistic grammar and common font design utilities to provide a modern mode of font design.

Prettybird provides basic building blocks for font design, however its utility is limited due to its current implementation. Future work on Prettybird will involve integrating it with a live graphical preview or a font editor such as FontForge to combine the best aspects of graphical and text-based font design.

Although the design principles of METAFONT and Prettybird differ, some features such as a pen tool could provide additional utility for many users. We plan to provide users the ability to use glyph subspaces generated by function calls as a kind of brush, allowing them to stamp this shape repeatedly or use it to draw variable-width curves.

Finally, we plan to enhance support for kerning. This will require an expansion to the current set of the language’s grammar. Better automatic kerning can be accomplished with modifications to the compiler backend, however we seek to implement complete control over kerning without requiring users to exit the language.

REFERENCES

- [1] Nelson H. F. Beebe. 2005. The design of TEX and METAFONT: A retrospective.
- [2] The FontForge Project Contributors. 2004. FontForge Open Source Font editor. <https://fontforge.org/en-US/>
- [3] Python Software Foundation. 2022. Glossary. <https://docs.python.org/3/glossary.html#term-duck-typing>
- [4] Jeremy Gibbons. 1970. Dotted and Dashed Lines in METAFONT. (02 1970).
- [5] Boguslaw Jackowski, Janusz M. Nowacki, and Piotr M. Strzelczyk. 2001. MetaType 1 : a METAPOST-based engine for generating Type 1 fonts.
- [6] Donald Knuth. 1979. *TEX and METAFONT*. American Mathematical Society.
- [7] Donald Ervin Knuth. 1990. *Computers and typesetting*. Vol. C - The METAFONTbook. Addison Wesley.
- [8] Donald Ervin Knuth. 1990. *Computers and typesetting*. Vol. D - METAFONT: The Program. Addison Wesley.
- [9] Lark. 2018. Lark. <https://github.com/lark-parser/lark>
- [10] Han-Wen Nienhuys. 2011. mftrace. <http://lilypond.org/mftrace/>
- [11] Alessandro Rovetta. 2021. Reliability of Google Trends: Analysis of the Limits and Potential of Web Inforeveillance During COVID-19 Pandemic and for Future Research. *Frontiers in Research Metrics and Analytics* 6 (2021). <https://doi.org/10.3389/frma.2021.670226>
- [12] Bram Stein. 2022. Webfont usage. *Fonts | 2022 | The Web Almanac by HTTP Archive* (Sep 2022). <https://almanac.httparchive.org/en/2022/fonts#fig-1>

A CODE SAMPLES

```

define horizontal_serif(attach_point, face_direction) {
  draw vector(attach_point - (10, 0),
              attach_point + (10, 0))
  draw bezier(attach_point - (10, 0),
              attach_point - (10, 0) + (7, face_direction * 4),
              attach_point - (10, 0) + (7, 0))
  draw bezier(attach_point + (10, 0),
              attach_point + (10, face_direction * 4) - (7, 0),
              attach_point + (10, 0) - (7, 0))
}

define vertical_serif(attach_point, face_direction) {
  draw vector(attach_point,
              attach_point + (0, 10))
  draw bezier(attach_point + (0, 10),
              attach_point + (face_direction * 10, 3),
              attach_point + (0, 3))
}

char I {
  base { blank(48, 72) }

  steps {
    draw filled rectangle((21, 12), (27, 52))

    horizontal_serif((24, 12), 1)
    horizontal_serif((24, 52), -1)
  }
}

char T {
  base { blank(48, 72) }

  steps {
    draw filled rectangle((21, 12), (27, 52))
    draw filled rectangle((6, 9), (42, 12))

    horizontal_serif((24, 52), -1)
    vertical_serif((6, 9), 1)
    vertical_serif((42, 9), -1)
  }
}

```

Fig. 9. Definitions for a full horizontal serif and a half vertical serif, used to create "I" and "T" glyphs

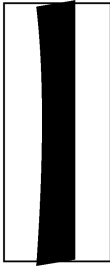
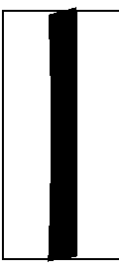
Prettybird Code Sample	METAFONT Code Sample
<pre>char I { base { blank(48, 72) } steps { draw vector((24, 6), (24, 42)) draw bezier((18, 7), (18, 43), (20, 24)) draw vector((18, 7), (24, 6)) draw vector((18, 43), (24, 42)) } }</pre>	<pre>mode_setup; em#:=10pt#; cap#:=7pt#; thin#:=1/3pt#; thick#:=5/6pt#; o#:=1/5pt#; define_pixels(em,cap); define_blacker_pixels(thin,thick); define_corrected_pixels(o); curve_sidebar=round 1/18em; def test_I(expr code,trial_stem,trial_width) = stem#:=trial_stem*pt#; define_blacker_pixels(stem); beginchar(code,trial_width*em#,cap#,0); "The letter I"; penpos1(stem,15); penpos2(stem,12); penpos3(stem,10); x1=x2=x3=.5w; y1=h; y2=.55h; y3=0; x2l:=1/6[x2l,x2]; penstroke z1e..z2e{down}..z3e; penlabels(1,2,3); endchar; enddef;</pre>
	

Fig. 10. Code samples and generated capital "I" glyphs presented in the survey

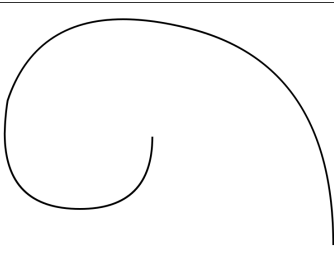
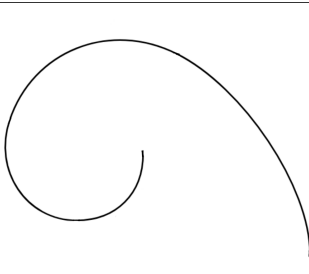
Prettybird Code Sample	METAFONT Code Sample
<pre>char a { base { blank(90, 70) } steps { draw bezier((90, 0), (50, 60), (90, 50)) draw bezier((50, 60), (0, 40), (10, 70)) draw bezier((0, 40), (20, 10), ((1 - 2) * 5, 10)) draw bezier((20, 10), (40, 30), (40, 10)) } }</pre>	<pre>p := (90,0) .. controls (90,20) and (70,50) .. (50,60) .. controls (30,70) and (7,61) .. (0,40) .. controls (-5,25) and (5,10) .. (20,10) .. controls (32,10) and (40,18) .. (40,30); draw p</pre>
	

Fig. 11. Code samples and generated spiral glyphs presented in the survey