

# Digital Typography in a Multiscriptal World

## Towards Fully Programmable Fonts

Kevin Graaf

24 June 2026

# 100% Human-made

Everything in this presentation is 100% human-made.

No “AI.”

No LLMs.

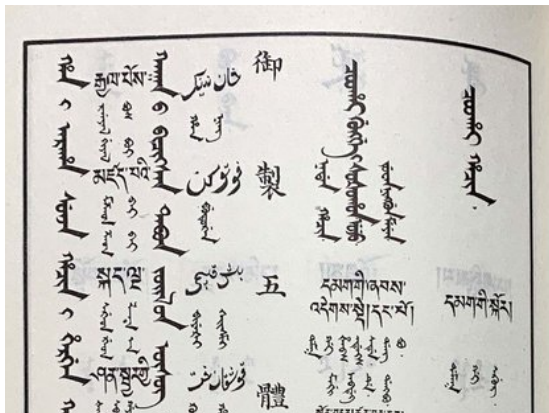
- Computer Scientist/Independent Researcher
- Interested in digital typography and text encoding, especially as they relate to ancient, emerging, and minoritized scripts

# Digital Typography: An Introduction

- Writing system aren't just symbols, rather:
- They are rules by which symbols are composed into text
- Digital typography systems differ in how they represent both the symbols and rules
- I want to argue for an under explored quadrant in this space
- My claim: digital typography is *NOT* a solved problem!

# Multiscriptal Typography

- What about combinations of scripts?
- $n$  writing systems means  $2^n$  combinations, a staggering number!
- 172 Scripts in Unicode!
- Seemingly unbounded degree of complexity
- We must think *systematically* about how writing systems compose
- Typography shouldn't degrade to least common denominator in presence of multiple scripts



Manchu-Chinese-Tibetan-Mongolian-Uyghur  
"Pentaglot" Dictionary

# The Status Quo: TrueType and OpenType

- TrueType/OpenType overwhelmingly dominate digital typography
- Glyph shapes represented as outlines, specified as Bézier splines
- (So-called) “advanced” typography provided by declarative, script-specific shaping models
- The problem?
  - ① Script behavior must be captured ahead of time in centralized set of shaping models,\* big problem for emerging scripts
  - ② Models are sometimes inadequate
  - ③ Implementations typically handle “runs” of text in different scripts (or fonts!) separately, poor interaction
  - ④ No common *abstraction* over script behavior: thus, every individual script is every program’s problem
  - ⑤ Finally: very, very complex

---

\* *Universal Shaping Engine* addresses this to a degree

- METAFONT [3] is *the* major influence on the present work
- Knuth's lesser known companion to T<sub>E</sub>X
- In METAFONT, letterforms are generated *offline*<sup>†</sup> by programs written in language of the same name
- Groundbreaking for its time, but not without important limitations
- Fundamentally based around individual, offline generation of glyphs
- For this reason, I describe METAFONT as “parametric” but not *programmable*
- Minimal support for “advanced” typography (in the OpenType sense)

---

<sup>†</sup>Technically, there are (marginal) ways around this

# The Curse of Declarativity

In CS, we often distinguish between “declarative” and “procedural” languages. Cf. HTML+CSS vs. T<sub>E</sub>X.

## Declarative

- ① “T” and “H” form a ligature when adjacent
- ② Place an acute accent at the anchor point of a vowel

## Procedural

- ① Scan text until “T” is found, if next character is “H” then substitute both for “TH”
- ② Calculate highest point of letter, add distance, draw acute accent

# The Curse of Declarativity (continued)

- Notice that the declarative model must “know” about all pertinent entities
- A declarative model must in some sense “do metaphysics”
- But what if we want to be more theoretically modest?
- Clearly, we’re nowhere close to a “theory of everything” w.r.t. writing
- And anyway, are we really going to trust computer scientists with that?
- More general mechanisms can be—paradoxically—much simpler
- Declarative languages are great for humans! Bad for computers



*Don't multiply entities unnecessarily!<sup>‡</sup>*

---

<sup>‡</sup>Not an actual Ockham quote!

# The Dubsar Programmable Typography Engine



- After the Sumerian word for scribe (lit. “tablet-writer”)
- My claim: fonts are *functions* from abstract text to concrete forms
- My bigger claim: only *programmable* typography has the generality required to handle *all* categories of human writing
- A Dubsar font is a program in the language of the Dubsar VM bytecode (i.e. a basic programming language)
- A font's code is constructed such that it fully embodies the “logic” of a writing system in a self-contained manner
- All “advanced” typographic features fall out as modes of use of this single, general mechanism
- *Drastically* simpler: 2500 lines of Standard ML vs. hundreds of thousands (millions?) of lines for avg text handling/rendering stack

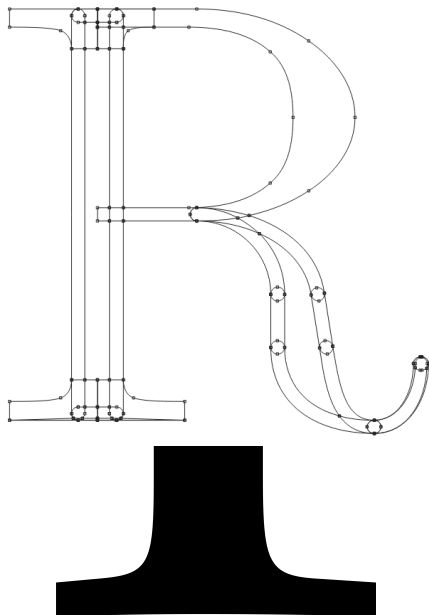
# The Big Ideas

- Every glyph is a function
- Each of these functions can access *and modify* the graphical state
- Glyph-functions can compute optimal layout and spacing (e.g. kerning) dynamically, no kerning tables needed
- Naturally allows for multiscriptal typography: just compute!
- Fonts also provide everything needed for proper UI, e.g. cursor placements
- All scripts share a common *abstraction*
- Mitigates the “pay to play” problem: programs don’t need to “know” anything specific about any writing system they support

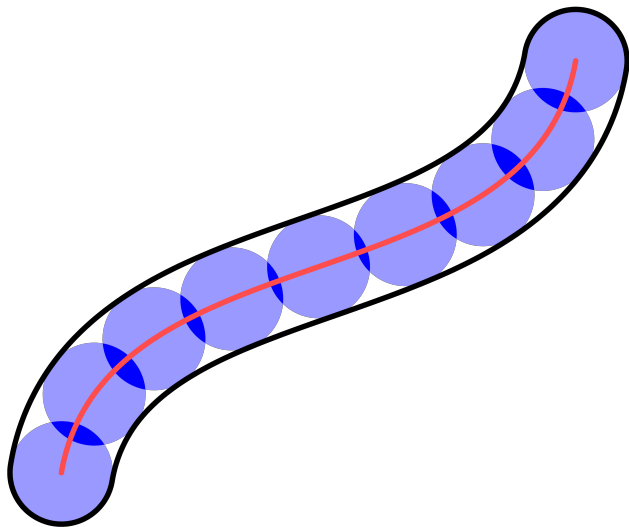
- Instead of outlines, Dubsar employs a stroke-based graphics model
- Strokes largely mitigate need for manual hinting
- Can be much more compact in terms of storage
- Arguably far more natural for CJK glyphs
- A more “semantic” representation of glyphs
- Typified by METAFONT
- In the METAFONT style, a virtual pen is stroked along some path

# Limitations of Stroke Fonts

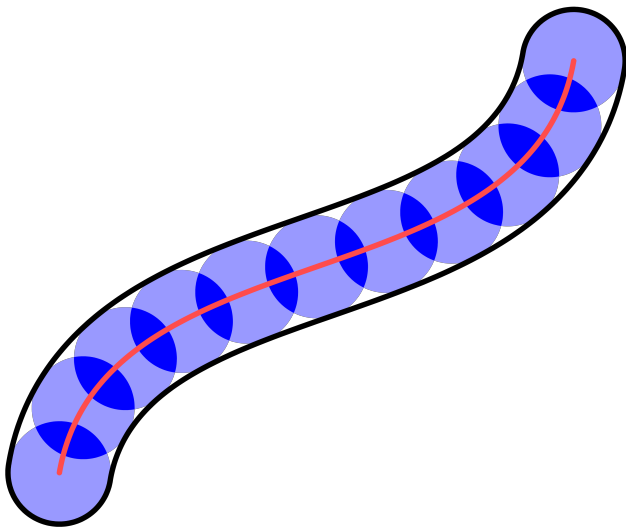
- Many glyphs are not simple unions of strokes! (e.g. any kind of smooth join)
- Computer Modern itself is more like a baroque outline font
- Pen-based strokes are inefficient to rasterize, exceptionally difficult to convert to outlines
- Why?
  - 1 Must “stamp” pen envelope along stroke path at every pixel-sized increment, significant overdraw!
  - 2 Offset curves of  $n$ th-order Bézier curves are usually not themselves  $n$ th-order Béziers!



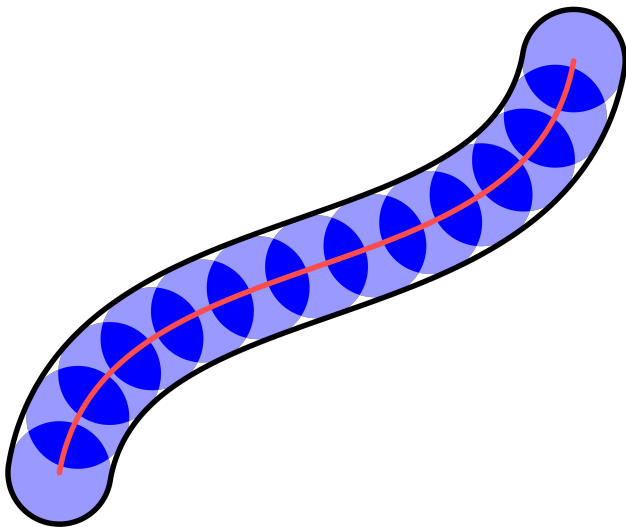
## Pen-based Stroke Font Overdraw



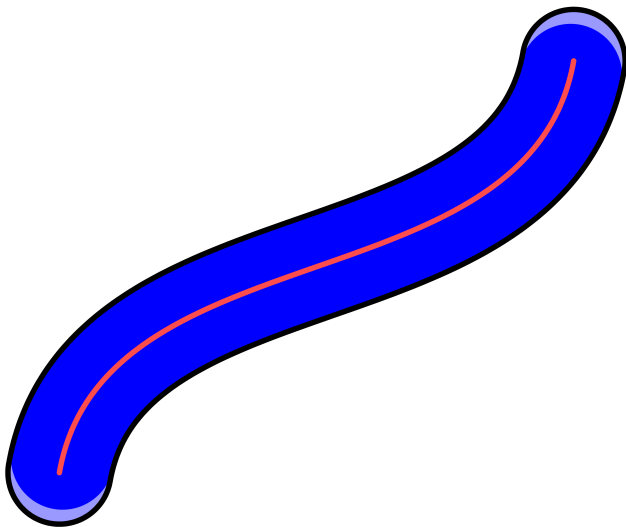
## Pen-based Stroke Font Overdraw



## Pen-based Stroke Font Overdraw



# Pen-based Stroke Font Overdraw



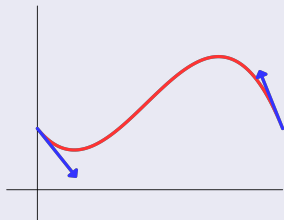
# Profile-Based Graphics Model

- Luckily, there's a better way!
- Profile-based models instead specify the center of a stroke and a varying width, according to mathematical functions
- Kind of like an infinitely thin pen nib with three degrees of freedom: push left, push right, twist
- Not unknown in the literature [1, 2, 4], but obscure
- Far more efficient to rasterize, no overdraw!
- To be precise: A Dubstar stroke is composed of a centerline, specified by 2D cubic Bézier spline, and a profile. The profile is defined at every point along the centerline by a left and right distance from the centerline, at some angle (possibly relative to its tangent), each specified by a 1D cubic Hermite spline.

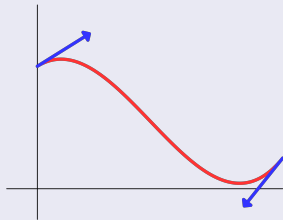
- Previous profile-based stroke models have to treat end caps specially
- My contribution: a rotational degree of freedom
- Rotation can be absolute *or* relative to tangent
- Specified in terms of Hermite splines, just like left and right profiles

# Components of a Stroke

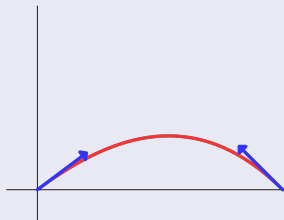
Left Width (1-D Hermite)



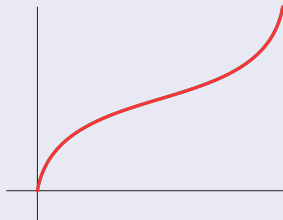
Right Width (1-D Hermite)



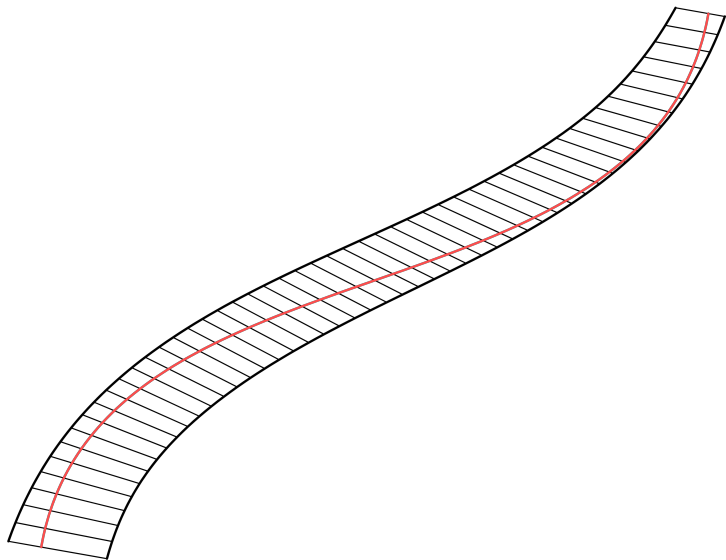
Rotation (1-D Hermite)



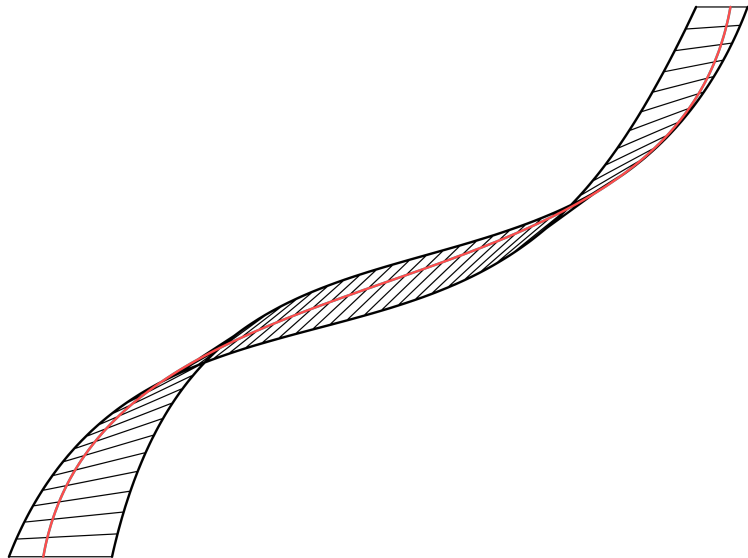
Centerline (2-D Bézier)



## Putting It All Together (Relative Rotation)



## Putting It All Together (Independent Rotation)



# Why Programmable Typography?

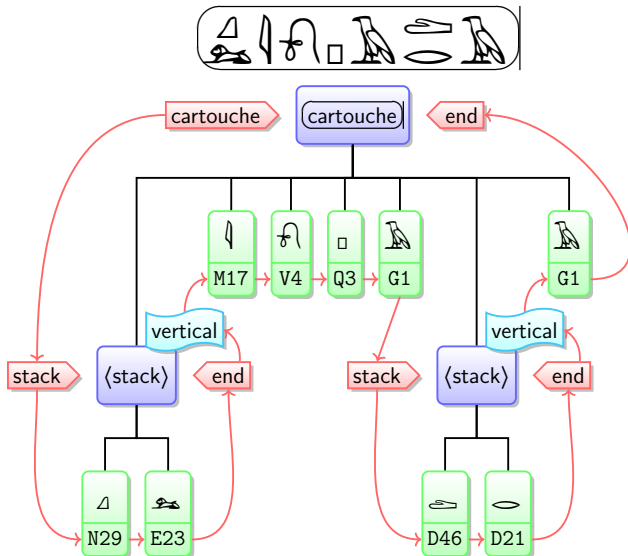
- Programmable typography is all the same less complex and more general
- A programming language can specify any advanced behavior that can be imagined
- No need for centralization, provide a general mechanism once
- For the skeptics: TrueType *already* has an (insecure!) virtual machine for hinting!
- OpenType is already Turing Complete!
- Unite all aspects of typography (hinting, kerning, shaping, etc) under one feature
- Fonts are properly understood as functions from abstract text to concrete visual forms

- *A stack-based virtual machine*
- Unlike TrueType, inherently safe
- All font-programs halt (cf. eBPF)
- Big Idea: every endpoint can dynamically inspect and alter the graphical context
- Every “scribeme” (graphical component “drawn together” e.g. ligature) is its own function

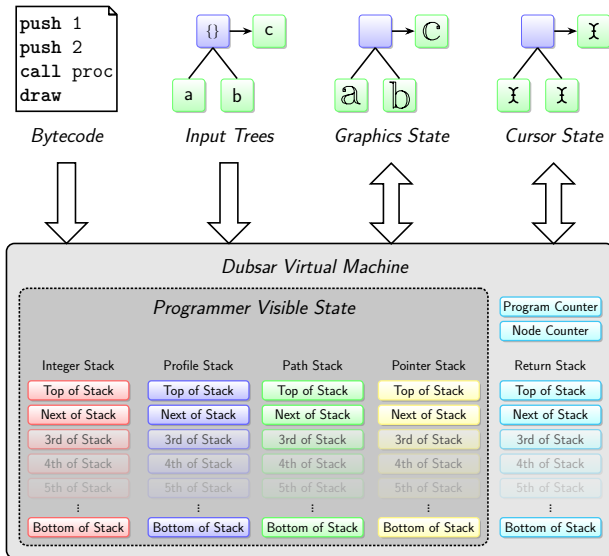
# Pattern Matching

- Glyph code is attached to a pattern
- The implementation scans input from beginning, dispatches corresponding code where matches are found
- Think regular expressions (but generalized to trees) or pattern matching in SML/OCaml/Haskell
- Can be compiled to very efficient code

# Structured Input



# Virtual Machine Block Diagram



# How does Dubsar handle...

- Hinting? —Current DPI can be accessed by font code
- Ligature Substitution? —Pattern matching or modification of graphical state
- Shaping? —Pattern matching + more advanced graphical “introspection” as with ligatures
- Kerning? —Compute distance from previous glyph, place accordingly
- Multiscriptal Typography? —No need for special treatment!

- Fonts written in programming language, DubsarScript
- This code is compiled to the virtual machine
- I'm interested in GUI tools, but this will require further research

# A Taste of DubsarScript

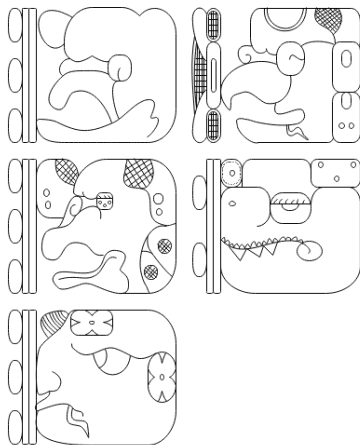


```
function wedge (rot: number, size: number)
  tip: path := (0, 5) lineto (4, 5) lineto (2, 4) close
  len: path := (2, 4) lineto (2, -5)
  wdg: path := tip .. len
  return scale(rotate(wdg, rot), size)
end

glyph <DINGIR :full>
  draw wedge(0, 1.0)
  draw wedge(-45, 0.8)
  draw wedge(-90, 1.0)
  draw wedge(-135, 0.8)
end
```

# Case Study: Maya

- I have successfully developed a font for the Maya script (pictured right)
- Allows for correct placement of glyphs into quadrats (blocks)
- However, *also* provides intuitive, interactive text input/editing
- A testament to Dubsar's flexibility
- And, I believe, the first time this has been accomplished!
- Crucially, Dubsar has no Maya-specific features or functionality



There's a lifetime of work to be done here...

- Prepare code for a public release, lots of boring engineering work
- Add enough glyph coverage to typeset original Maya texts
- Develop fonts for other hieroglyphic scripts (esp. Egyptian)
- Integrate Dubsar into everyday application software

# References

- [1] Elena J Jakubiak, Ronald N Perry, and Sarah F Frisken. “An improved representation for stroke-based fonts”. In: *Proceedings of ACM SIGGRAPH*. Vol. 4. 2006, p. 2.
- [2] R. Victor Klassen. “Variable width splines: a possible font representation?” In: *Electronic Publishing* 6.3 (1993), pp. 183–194.
- [3] Donald E. Knuth. *The METAFONT book*. United States: Addison-Wesley Longman Publishing Co., Inc., 1989. ISBN: 0201134446.
- [4] Paul Murrell. *Variable-Width Lines in R*. May 2017. URL: <https://www.stat.auckland.ac.nz/~paul/Reports/VWline/vwline-intro/power-curve.html>.

# Questions?

email: `kevingraaf@kbg.sh`

website: `https://kbg.sh`

code: `https://fossil.kbg.sh`