ETAP: Experimental Typesetting Algorithms Platform

Didier Verna
EPITA
Research and Development Laboratory
Le Kremlin-Bicêtre, France
didier@lrde.epita.fr

ABSTRACT

We present the early development stages of ETAP, a platform for experimenting with typesetting algorithms. The purpose of this platform is twofold: while its primary objective is to provide building blocks for quickly and easily designing and testing new algorithms (or variations on existing ones), it can also be used as an interactive, real time demonstrator for many features of digital typography, such as kerning, hyphenation, or ligaturing.

CCS CONCEPTS

• **Software and its engineering** → **Application specific development environments**; • **Human-centered computing** → **Heuristic evaluations**; **Information visualization**; • **Applied computing** → *Document preparation.*

KEYWORDS

Typesetting, Paragraph Formatting, Real-Time Interactive Experimentation

ACM Reference Format:
Didier Verna. 2022. ETAP: Experimental Typesetting Algorithms Platform. In *Proceedings of the 15th European Lisp Symposium (ELS'22)*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.5281/zenodo.6334248

1 INTRODUCTION

The world of digital typography is a fascinating one. As an application domain, it combines a strong focus on aesthetics with many complicated technical challenges. On the outside, the concern for aesthetics is everywhere: from the shape of characters and the space between them, to the overall balance of lines, paragraphs, pages, complete documents. On the inside, any kind of formatting algorithm (for example, a paragraph justification one) risks exponential complexity as soon as *some* level of quality is expected.

Defining the notion of (good) typesetting quality is a very complicated and subtle problem, and is out of the scope of this paper. On the other hand, bad typesetting is easily perceived, and hence, rather easy to demonstrate, as it impacts readability and involves such notions as aesthetic disturbance.

Figure 1 exhibits the first four lines of a badly justified paragraph. Notice for example how different the inter-word spacing is between lines 1 (very large) and 4 (very small). Notice also that line 4 is so compressed that it becomes very difficult to separate the words from each other. Finally, within the particular context of the highly compressed fourth line, where the inter-word spacing becomes close to the inter-letter one, the word "knowledge" almost appears written in two words: "know ledge". Even when the casual reader is unaware of all these problems, reading badly typeset documents results in fatigue.

Domain-specific language (DSL) design and implementation is inherently a transverse activity (Ghosh, 2010; Fowler, 2010). It usually requires from the product team knowledge and expertise in

Figure 1: Example of a badly justified paragraph

This project originates in two connected, yet slightly different motivations. The first one is a general interest for the world of digital typography, and the will to "play" and experiment with typesetting algorithms (also in order to get a better understanding of how they work and what they actually do). The second one, more pragmatic, is the need to strengthen an existing lecture on typesetting, with striking illustrations for the various aesthetic challenges that the discipline faces. These two motivations have something in common: they both require a system which must be as real-time and interactive as possible. Suppose you are trying out a new paragraph justification algorithm. Rapid prototyping and experimentation would be made a lot easier with a direct visualization of the results on a sample text (actual contents not so important), and with the ability to interactively tweak such or such parameter from a Graphical User Interface (GUI), while observing the effects in real-time. In a similar vein, in order to illustrate the importance of, say, kerning (inter-letter spacing adjustment), there is nothing like having the ability to visualize a sample text (again, actual contents not so important), and turn kerning on or off by the click of a button.

In general, typesetting experimentation or demonstration is not a very practical thing to do. What You See Is What You Get (WYSIWYG) systems such as Word or Libre Office are usually quite reactive, but their algorithms are not necessarily of the highest quality, and neither are they easily configurable or extensible, let alone replaceable. On the other hand, TEX[6, 7], the obvious competitor, and still a *de facto* standard in terms of typographic quality and customizability, is not a a very interactive system. It works more like a compiled programming language with separate development, compilation, and visualization phases. There was one attempt at providing a GUI for controlling typesetting parameters[2], but it

ELS'22, March 21–22 2022, Porto, Portugal

ACM ISBN 978-2-9557474-6-9.
https://doi.org/10.5281/zenodo.6334248

was limited to global ones and didn't go very far. Many projects attempt to mitigate this by providing more or less interactive and real-time WYSIWYG layers on top of it. However, TEX itself certainly doesn't make it easy to tweak or replace any of its internal typesetting algorithmic components. In fact, the "spaghetti code effect" is a well-known characteristic among the community of TEX hackers.

Whatever the approach, all these systems have one thing in common: they are *production* systems. They target the feature-bloated generation of complete, actual documents. Interactive and real-time experimentation, testing, rapid prototyping, or demonstration is a different goal, and it is the niche that ETAP tries to occupy. It is not meant to become a complete typesetting system, although it could very well turn out to be a Petri dish for one [11, 12]. Rather, it attempts to provide low-level data structures and building blocks for experimenting with new typesetting algorithms (or variations on existing ones), with interactive and real-time parametrization and visualization, and hopefully in a near future, quality assertion and analysis (for some definition of "quality").

Section 2 describes the project and its current features. Section 3 gives a brief overview of the underlying implementation. Finally, Section 4 concludes and Section 5 details some general directions for future work.

2 CURRENT FEATURES

Figure 2 provides a screenshot of ETAP's GUI, which is currently implemented in LispWorks[1] CAPI[2]. The platform currently focuses on paragraph formatting algorithms. The interface can be described as having four main areas.

2.1 Area 1: Text Editor

Area 1 is a simple text editor (a CAPI `editor-pane`) which lets you adjust the textual contents of the typeset paragraph. Any change in the text is automatically and continuously propagated to the paragraph view (area 4).

2.2 Area 2: Global Options and Features

Area 2 provides control over some global options, features, and visual clues, also tracked continuously and in real-time.

The paragraph disposition pane lets one choose between justification and various ragged formatting. Note that some combinations of algorithm / disposition don't actually make much sense, but still, these parameters are considered sufficiently orthogonal to be separate in the GUI.

The features pane lets one toggle kerning (inter-letter spacing), ligatures (character fusion, *e.g.* ff for ff), and hyphenation (word splitting) on or off. Kerning and ligature information is provided by the font in use. The hyphenation implementation is that of TEX, itself based on Liang's thesis[9]. The language (hence the hyphenation patterns set) is currently hard-coded to English.

The clues pane allows one to choose what is actually displayed in the paragraph view (area 4): the characters themselves, but also different kinds of bounding boxes, plus the hyphenation points, and the underfull / overfull boxes. In Figure 2, the paragraph view exhibits individual characters, hyphenation points, and underfull boxes.

Finally, there is a slider to set the desired paragraph width.

2.3 Area 3: Algorithms

This is where you select a specific paragraph formatting algorithm. Depending on the chosen one, this area also displays algorithm-dependent variants, options, or other adjustable parameters. Again, any choice of algorithm or any modification of its parametrization is automatically and continuously reflected in the final paragraph view.

A complete description of the currently available algorithms is out of the scope of this paper, but each one is implemented in its own file, and there is always an explanatory comment at the top. Here, we only provide a quick overview of the five algorithms currently implemented.

2.3.1 Fixed. The "fixed" algorithm uses only the *natural*, constant, inter-word spacing (a value provided by the font in use). Hence, it can practically never justify properly. Lines are created sequentially, without look-ahead or backtracking: there are no paragraph-wide considerations.

2.3.2 Fit. As the name suggests, this is an implementation of the so-called First, Best, and Last Fit classical algorithms. Those ones make full use of elastic inter-word spacing ("glue" in TEX terms) when attempting to justify lines. The acceptable range of inter-word spacing is also an information provided by the font in use. By nature of the *-Fit algorithms, lines are also created sequentially here, without look-ahead or backtracking: there are no paragraph-wide considerations.

2.3.3 Barnett. This one is an implementation of a justification algorithm from Michael Barnett[1], originally published in 1965. In short, this algorithm behaves more or less as a combination of different *-Fit policies, while favoring overfull lines when no perfect solution is found.

2.3.4 Duncan. This one is an implementation of a justification algorithm from C. J. Duncan [5], originally published in 1963. In short, this algorithm searches for an acceptable breaking solution while minimizing hyphenation.

2.3.5 Knuth-Plass. Finally, the fifth and last one is the TEX one, *a.k.a.* the famous "Knuth-Plass" algorithm[8]. This is the one visible in Figure 2, and you can see that it has TEX's full set of adjustable parameters available.

2.4 Area 4: Paragraph View

Area 4 is where the typeset paragraph is eventually rendered, depending on the selected algorithm and options, and along with the various visual clues selected in area 2. In the screenshot from Figure 2, the orange triangles indicate the hyphenation points, and the rectangles at the end of lines 2 and 9 denote the underfull lines (that is, the lines that are too short to be justified). Overfull lines would be indicated, as in TEX, by the same rectangles, only filled in with orange.

[1] http://www.lispworks.com/
[2] http://www.lispworks.com/products/capi.html

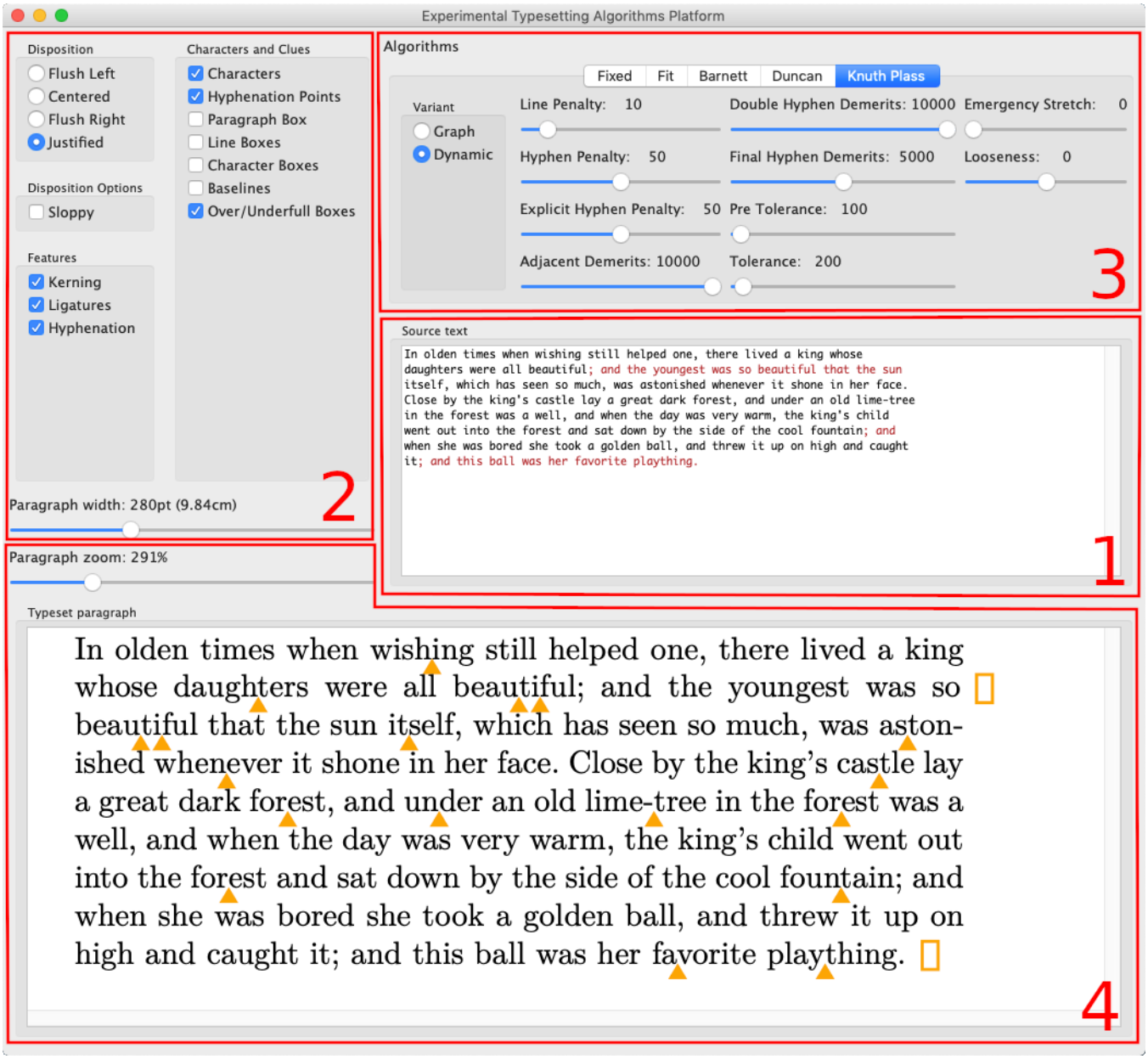

Figure 2: ETAP's graphical user interface

There is a slider for zooming in / out the view. Note that the zooming facility is the only GUI component that doesn't retrigger the typesetting engine. It operates at the window level.

The paragraph is currently rendered with a hard-wired font: Latin Modern Roman 10pt, Extended Cork encoding. The font and encoding descriptions have been copied from a MacTEX (TEXlive) distribution[3]. In particular, the font information (notably including kerning and ligatures) is read from its original TEX Font Metrics (TFM) file thanks to a library also developed by the author of this paper[4].

3 IMPLEMENTATION

In this section, we provide a brief overview of ETAP's implementation. The design of the internals is heavily inspired from that of TEX, but it also fits a different set of objectives, most importantly being a experimentation platform rather than a production system.

[3]https://www.tug.org/mactex/

[4]https://github.com/didierverna/tfm

3.1 Basic Data Structures

ETAP provides five basic data structures. Characters are directly represented by their corresponding `character-metrics` structure from the TFM library. Then, there are classes for kerns (fixed, possibly negative, space between characters), and break points (discretionaries and glues). Discretionaries and glues follow TEX's jargon and design.

A *discretionary* represents a potential break point with different material to typeset, depending on whether the break actually occurs or not. For example, the sequence of characters `ffi` is reified as a discretionary specifying that without a break, the ligature ffi may be used, and in case of breaking the line, the first line ends with `f-` and the next one begins with `fi` (or the ligature fi). A simple hyphenation point simply states that in case of breaking a line, the first one ends with a dash, and nothing else happens otherwise.

A *glue* represents an elastic space between words, with a natural width, plus specific amounts of shrinkability and stretchability (again, those values are provided by the font in use).

3.2 The Lineup

The paragraph text retrieved from Area 1 of the GUI is processed into a so-called *lineup*. A lineup is essentially a vector of objects to typeset. The paragraph text is trimmed from consecutive blanks. It is then sliced into words, possibly hyphenated (in which case discretionaries are added). After that, ligatures are handled if requested (which may lead to the creation of new discretionaries, or the modification of existing ones). Kerns are then inserted at the appropriate places, again, if requested. Finally, an infinitely stretchable glue is appended at the end of the lineup.

3.3 The Lines

Each algorithm's entry point is implemented as a method on a generic function called `create-lines`. The algorithms receive a lineup, a paragraph width, a disposition, and set of algorithm-specific options specified in the GUI. They compute their own view on where exactly the lineup should be broken into lines, and they return the lines in question.

A *line* is essentially a sequence of characters, each one with a specific horizontal placement with respect to the beginning of the line. This placement is computed out of the characters widths, the kerns, and the glue present in the lineup, and of course, the desired line's length. Characters placed at a specific horizontal position are called *pinned characters*.

3.4 The Paragraph

Finally, the resulting *paragraph* is created and passed to the GUI for rendering. There is in fact not much left to do to generate it. Each line computed by the selected algorithm is positioned both horizontally and vertically, relative to the paragraph's top-left corner. Such a fully placed line is called a *pinned line*. The horizontal position of each line depends on the selected paragraph disposition (centered, flushed, or justified). Vertically, the lines are simply spaced by a currently hard-wired constant (the "line skip" in TEX's jargon).

4 CONCLUSION

ETAP is currently in an "early prototype" development state[5]. The internals are not stabilized, there is no decent documentation, the code has *not* been carefully crafted, and no concern for optimization or general performance has entered the picture yet.

Despite all this, the project already works surprisingly well. The GUI runs very smoothly in real-time, and it has been used successfully several times already to support lectures or conferences on typesetting. The observable reactions in the audience, facing the real-time effects of kerning, hyphenation, or ligaturing, for example, is a testimony to the pertinence of this approach for increasing the general awareness of the technical challenges involved in digital typography.

One of the most important advantages in using Common Lisp [10] for this project is the ease of development and the concision of the resulting code. The program (excluding the TFM library and a large font description file) is currently just under 3000 lines of code. The GUI code and the typesetting building blocks take around 25% of that each, and the other half of the code is devoted to the algorithms implementations. The Knuth-Plass algorithm itself, for which we actually provide two different implementations (see Section 5.2), takes less than 500 lines (granted, the whole of TEX isn't there obviously; user-level macros, mathematics, *etc.*).

5 FUTURE WORK

In addition to improving the general state of the project (essentially meaning stabilizing the internals and providing accurate and up to date documentation), we currently envision two major directions for future work.

5.1 Direction 1: Experimentation

One of the very first, and already achieved goal of this project was to make it easy to experiment with typesetting algorithms, by either creating new ones, extending or modifying existing ones, and quickly visualizing the results. In a near future, we intend to use ETAP to do research on known typesetting problems such as rivers detection, or experiment with new features or extensions, notably to the Knuth-Plass algorithm. Some people, for instance, prefer different kinds of placement for end-of-line hyphens in justified paragraphs.

5.2 Direction 2: Analysis

Because typography is not a technical question only, but also an aesthetic one, a very difficult problem, when experimenting with typesetting algorithms, is how to evaluate the quality of the results. Of course, the ability to directly visualize a typeset paragraph, as in this project, is a tremendous help, but it is surely not enough.

In fact, we can come up with mathematical formulas representing some measure of typesetting quality (for example, taking into account the amount of stretching or shrinking of lines, compared to their natural width), and this is in fact precisely what the Knuth-Plass algorithm attempts to optimize, paragraph-wide (the so-called *badness*).

[5] https://github.com/didierverna/etap

With a platform such as ETAP, it becomes very easy to instrument the underlying data structures to keep track of quality measurement (badness, demerits from TEX, or anything else one may think of), and perform statistical analysis afterward.

Here lies the second most important advantage in using Common Lisp for this project. Its interactive nature makes it effortless to bypass the GUI altogether, and run the typesetting algorithms, without visualization, from the Read-Eval-Print Loop (REPL) or through a batch script.

In a near future, it is hence also our intention to collect large empirical data on the quality of typesetting (for example, using TEX's notion of quality) and perform statistical and comparative analysis between different algorithms, or algorithm implementations. For example, we can easily run the five algorithms currently implemented on the same paragraph, for many different widths, and compare the resulting data. The original Knuth-Plass algorithm uses a *dynamic programming*[3, 4] optimization technique for cutting through the (potentially very large) graph of break possibilities. ETAP already provides an unoptimized (and slow) variant implementation of it, working on the full graph. It would be interesting to collect statistical data from both these implementations, in order to get a concrete idea of the impact of TEX's optimization on the actual quality of the typesetting.

REFERENCES

[1] Michael P. Barnett. *Computer Typesetting: Experiments and Prospects.* MIT Press, January 2000.

[2] Kaveh Bazargan. Batch commander: a graphical user interface for TEX. *TUGBoat*, 26(1):74–80, 2005.

[3] Richard Bellman. The theory of dynamic programming. *Bulletin of the American Mathematical Society*, 60(6):503–516, 1954. doi: 10.1090/S0002-9904-1954-09848-8.

[4] Richard Bellman. *Dynamic Programming.* Princeton University Press, 2003.

[5] C.J. Duncan, J. Eve, L. Molyneux, E.S. Page, and M.G. Robson. Computer typesetting: an evaluation of the problems. *Printing Technology*, 7:133–151, 1963.

[6] Donald E. Knuth. *The TEXbook.* Addison-Wesley, 1984.

[7] Donald E. Knuth. *TEX: the Program*, volume B of *Computers and Typesetting.* Addison-Wesley, January 1986.

[8] Donald E. Knuth and Michael F. Plass. Breaking paragraphs into lines. *Software: Practice and Experience*, 11(11):1119–1184, 1981. doi: 10.1002/spe.4380111102.

[9] Franklin Mark Liang. *Word Hy-Phen-a-Tion by Com-Put-Er.* PhD thesis, Stanford, CA, USA, 1983.

[10] ANSI. American National Standard: Programming Language – Common Lisp. ANSI X3.226:1994 (R1999), 1994.

[11] Didier Verna. Star TEX: the next generation. In Barbara Beeton and Karl Berry, editors, *TUGboat*, volume 33. TEX Users Group, 2012.

[12] Didier Verna. TiCL: the prototype (Star TEX: the next generation, season 2). In Barbara Beeton and Karl Berry, editors, *TUGboat*, volume 34. TEX Users Group, 2013.