Both TeX and DVI viewers inside the web browser

Jim Fowler

Abstract

By using a Pascal compiler which targets WebAssembly, TeX itself can be run inside web browsers. The DVI output is converted to HTML. As a result, both LaTeX and Ti*k*Z are available as interactive input languages for content on the web.

1 Introduction

Many people would like to make technical material (often written in TeX) available on the World Wide Web. Of course, this can be done via web pages, but for mathematical expressions, HTML and MathML produce inferior results. Consequently, many users rely on client-side tools like MathJax [1] to provide beautiful rendering for content in math mode.

There is also a need to go beyond math mode. How might one render a Ti*k*Z [14] picture on the web? In the past, this might have been done with TeX4ht [8] to convert a Ti*k*Z picture to SVG. This article describes the basis of a new method, Ti*k*ZJax [3], which, like MathJax, is client-side, performing its conversions in the client's browser. When the Ti*k*ZJax JavaScript is run, any Ti*k*Z pictures inside `<script type="text/tikz">` tags are converted into SVG images. Ti*k*ZJax is emphatically not a JavaScript reimplementation of Ti*k*Z, but instead works by running ε-TeX itself inside the user's web browser; this copy of TeX is provided to the browser with its memory already loaded with Ti*k*Z.

In short, TeX has been ported to JavaScript. This article describes how we ported TeX to the JavaScript-based environment of web browsers, and how we render the resulting DVI output in HTML. We hope that making TeX itself available in the browser will open up many new possibilities.

2 A Pascal compiler targeting web browsers

TeX was written in an era when computing resources were rather more constrained than today. Many of those constraints have returned within the JavaScript ecosystem, e.g., JavaScript is slower than native code and has limited access to persistent storage.

2.1 Goto is a challenge

To run TeX in a web browser, we initially wrote a Pascal compiler targeting JavaScript. The main challenge is handling `goto` which is used fairly frequently in Knuth's code (especially since the Pascal of that era did not offer an early `return` from procedures and functions), and does not exist as such in JavaScript. However, JavaScript does support labeled loops, labeled breaks, labeled continues, and alongside a trampoline-style device it is possible to emulate in JavaScript the procedure-local `goto`s used in TeX. There are a handful of cases in which a non-local `goto` is used by TeX to terminate the program early, but early termination can also be handled in JavaScript.

Thus, it *is* possible to transpile Pascal to JavaScript. However, it turns out that running TeX inside JavaScript is not particularly efficient!

2.2 WebAssembly

WebAssembly [9] provides a speedier solution. WebAssembly is a binary format for a stack-based virtual machine (like the Java Virtual Machine) which runs inside modern web browsers and is designed as a compilation target for languages beyond JavaScript. There is still no support for `goto`, but the same tricks with labeled loops that make `goto` possible in JavaScript again work in WebAssembly. Our compiler `web2js` [4] digests the dialect of Pascal code that TeX is written in and outputs WebAssembly, which can then be run inside modern web browsers. We chose the "web" in `web2js` to evoke both `WEB` and also the World Wide Web.

WebAssembly, as it is currently implemented in web browsers, does not provide any high-level dynamic memory allocation; it is possible to resize the heap but nothing like `malloc` is provided. Given that TeX also does no dynamic allocation, it's relatively easy to compile TeX to this target.

Since we want to run LaTeX in the browser, it is necessary to use a TeX distribution which supports the ε-TeX extensions. So before feeding the Pascal source code to `web2js`, we `TANGLE` in the change file for ε-TeX. Other change files are needed too. For instance, there is a patch to the Pascal code needed to get the current date and time from JavaScript.

Some additional JavaScript code is needed to support components missing in the browser. For instance, there is no filesystem in the browser, so the Pascal filesystem calls make calls to JavaScript which provides a fake filesystem. The terminal output of TeX can be viewed by opening the "Web Console" in the web browser. Satisfyingly, when it is all working, the TeX banner is visible right there.

2.3 Why Pascal? Why not C?

There are other approaches to getting TeX to run well in a web browser. An older project, `texlive.js`, achieves this goal via emscripten [15], a C compiler which targets WebAssembly. The resulting website enables client-side creation of a PDF, and so depends on a PDF viewer to see the result. S. Venkatesan [13] discussed this approach and the limitations of PDF output in particular.

2.4 Putting it all together

In the quest for better performance, the same tricks that TEX used historically with format files and memory dumps can be reused in the web browser. The underlying theme is that the ecosystem of a web browser, and its limitations, is more similar to computing in the early 1980s than might have been easily believed.

As with teTEX version 3.0, we do not bother making a special `initex` version and simply allocate a large number of memory cells to a single version of TEX. A program called `initex.js` then loads the initial LATEX format (with only some hyphenation data) and whatever piece of a preamble (e.g., `\usepackage{tikz}`) might be useful for the desired application. Then the WebAssembly heap is dumped to disk, just as would have been done with `virtex` historically. This produces a file, `core.dump.gz`, which is only a couple of megabytes (after compression).

Note that `initex.js` is executed on a machine that already has a complete TEX distribution installed, such as TEX Live. By loading packages and then dumping core on a machine with a complete distribution, it is not necessary to ship much in the way of support files to the browser.

On the browser, both the WebAssembly machine code and `core.dump.gz` are loaded, the dump decompressed, and execution begins again at the beginning of the TEX code but this time with the previously dumped memory already loaded. As described in the TEX82 source code [11, Part 51, Section 1331], when TEX is loaded in such a fashion, the `ready_already` variable is set in such a way as to shortcut the usual initialization, making this browser-based version of TEX ready to receive input very quickly.

3 Rendering DVI in HTML

Running TEX is only half the problem. To build a viewer for the output of TEX, the easiest format to parse is DVI [6, 7]. A DVI file is just a series of commands which change the current position, place characters and rules on the page, change the current font, etc.

Some previous projects make it possible to view DVI files from within web browsers. For instance, `dvihtml` [12] uses DVI specials to appropriately tag pieces of the content so that they can be wrapped by appropriate HTML tags, similar to TEX4ht [8]. Other projects like DVI2SVG [5] translate DVI into SVG with a Java-based tool.

Our new tool is called `dvi2html` [2] and works somewhat differently. For starters, unlike DVI2SVG, our new tool is written in JavaScript (and mostly TypeScript) so it runs in the browser. It is used to read the output of ε-TEX, running in the browser, and output HTML in real-time.

3.1 Fonts

Why wasn't all this done years ago? One significant challenge was the state of "fonts" on the web. Conveniently, it is possible (and relatively easy with CSS) to load server-provided fonts. To support Computer Modern and the like, `dvi2html` presently relies on the BaKoMa TrueType fonts, but given their license, it would be good to generate fonts for the web following MathJax's technique.

It must be mentioned that while fonts can be loaded, the web ecosystem lacks a robust way to query metric information. So we still end up shipping the standard collection of `.tfm` files to the browser, all base64-encoded and placed into a single `.json` file. A significant portion of the code comprising `dvi2html` is designed to parse TEX Font Metric files.

3.2 The challenge of the baseline

But selecting the appropriate typeface is not enough; an HTML viewer for DVI must also position the glyphs in the appropriate positions. This is sadly harder than it ought to be. Although HTML5 supports many methods for positioning text, it does not support positioning text relative to a specified baseline.

A solution to this is available precisely because of the previously loaded metric information. By knowing where the top of the glyph is relative to the baseline, we can use HTML to place the glyph in the correct position.

3.3 Streaming transformation

Instead of a monolithic converter, `dvi2html` is structured as a streaming transformer via asynchronous generator functions. In particular, an input stream is transformed into an object stream of DVI commands. Since many DVI commands come in a variety of lengths (i.e., one-byte, two-byte, three-byte, four-byte versions), this initial transformation collapses the variety of commands in the binary format to a single command.

Armed with a sequence of DVI commands, additional transformations can be applied. For instance, there is some overhead to placing a single glyph on the page in HTML, so one transformer takes sequential SetChar commands from the DVI input and collects them into a single SetText command which can place a sequence of glyphs on the page at once.

The real benefit, though, to stream transformations is that the various transformations can be composed, with new transformations plugged in as desired. For instance, a package like `xcolor` will generate `\special`s with push color and pop color commands, and these can be processed by a single stream transformer which understands these color commands. Another composable transformer knows about raw SVG data and can appropriately emit such code into the generated HTML.

Finally, this sort of design will make it possible to compose new transformers for hitherto unimagined `\special`s. Most interestingly, such `\special`s could facilitate additional interactivity on the web in future versions.

4 Some next steps

The tools for running TEX itself inside a browser are useful for more than Ti*k*ZJax. For instance, these same tools make a "live LATEX editor" possible in which a user can edit LATEX source in a web page and view the resulting DVI without installing software and without relying on a cloud-based LATEX compilation service.

The Ximera platform provides `\answer` which creates answer blanks within mathematical expressions. For instance, `1 + 3 = \answer{4}` creates an equation in which the right-hand-side is an answer blank. It would be wonderful to add `\answer` to a copy of LATEX running in the browser.

Additional extensions to TEX itself are possible, like a hypothetical jsTEX which would extend TEX with the ability to execute JavaScript code, akin to LuaTEX [10]. The reader can imagine additional applications of this platform.

References

[1] D. Cervone. MathJax: A platform for mathematics on the Web. *Notices of the AMS* 59(2):312–316, 2012. `ams.org/notices/201202/rtx120200312p.pdf`

[2] J. Fowler. dvi2html. `github.com/kisonecat/dvi2html`, 2019.

[3] J. Fowler. Ti*k*Zjax. `github.com/kisonecat/tikzjax`, 2019.

[4] J. Fowler. web2js. `github.com/kisonecat/web2js`, 2019.

[5] A. Frischauf and P. Libbrecht. DVI2SVG: Using LATEX layout on the Web. *TUGboat* 27(2):197–201, 2006. `tug.org/TUGboat/tb27-2/tb87frischauf.pdf`

[6] D. Fuchs. The format of TEX's DVI files. *TUGboat* 1(1):17–19, Oct. 1980. `tug.org/TUGboat/tb01-1/tb01fuchs.pdf`

[7] D. Fuchs. Erratum: The format of TEX's DVI files. *TUGboat* 2(1):11–11, Feb. 1981. `tug.org/TUGboat/tb02-1/tb02fuchszab.pdf`

[8] E. M. Gurari. TEX4ht: HTML production. *TUGboat* 25(1):39–47, 2004. `tug.org/TUGboat/tb25-1/gurari.pdf`

[9] A. Haas, A. Rossberg, et al. Bringing the web up to speed with WebAssembly. *ACM SIGPLAN Notices* 52(6):185–200, 2017.

[10] T. Hoekwater. LuaTEX. *TUGboat* 28(3):312–313, 2007. `tug.org/TUGboat/tb28-3/tb90hoekwater-luatex.pdf`

[11] D. E. Knuth. *TEX82*. Stanford University, Stanford, CA, USA, 1982.

[12] M. D. Sofka. TEX to HTML translation via tagged DVI files. *TUGboat* 19(2):214–222, June 1998. `tug.org/TUGboat/tb19-2/tb59sofka.pdf`

[13] S. K. Venkatesan. TEX as a three-stage rocket: Cookie-cutter page breaking. *TUGboat* 36(2):145–148, 2015. `tug.org/TUGboat/tb36-2/tb113venkatesan.pdf`

[14] Z. Walczak. Graphics in LATEX using Ti*k*Z. *TUGboat* 29(1):176–179, 2008. `tug.org/TUGboat/tb29-1/tb91walczak.pdf`

[15] A. Zakai. Emscripten: An LLVM-to-JavaScript compiler. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, pp. 301–312. ACM, 2011.

◇ Jim Fowler
100 Math Tower, 231 W 18th Ave
Columbus, Ohio 43212
USA
`fowler (at) math dot osu dot edu`
`http://kisonecat.com/`