VT-roff: a Terminal for the 21st Century

ABSTRACT

Reports of the demise of `VT-100` are much exaggerated. It lives on (and on) in software, hampering innovation, productivity, and progress wherever it goes. Long after the machine itself and its wondrous and weird supporting cast have departed the scene, its weird features remain: unused, mostly useless, and frequently baffling.

The zombie life of the VT-100 wouldn't matter but for one fact: the long promised GUI environment that would make it obsolete, the flying car of the 1980s, never arrived. Research into better command-line interfaces, to the extent it ever existed, ceased when everyone "knew" it would be replaced by the so-easy-to-use GUI. Yet in 2013, the 35-year old VT-100 still sets the standard, and in emulation remains the tool of choice for millions of computing professionals every day.

Herein described is a *graphical terminal* that affords a very different kind of command-line experience. It subsumes some common features of command-line interaction and breaks free of the character-cell metaphor defined by the VT-100. It bypasses much of the unneeded OS support for nonexistent serial ports and modems, incorporates basic, necessary features currently provided by the "host", and supports graphics equal to any printer or web browser *in the terminal*, not as adjunct programs.

Moreover, it is not a moonshot: it leverages existing technology, *troff*, in a new way. For a small noncommercial investment, it could improve the daily lives of millions of people, and possibly point a new way forward for human interaction with the computer.

26 November 2013

VT-roff: a Terminal for the 21st Century

James K. Lowden

Continuum Analytics

1. Motivation

1.1. The `xterm` Time Warp

Every day, millions of computer professionals spend many hours trapped in a 1978 time warp. That is the year Digital Equipment Corporation introduced the VT-100 ASCII asynchronous terminal. It was among the first to support what would become the ANSI X3.64 standard, sporting a cursor-addressable screen, blinking, bold, and underlined characters, 80 across and 24 high, beautiful green letters on an inky black screen. That functionality is now most commonly provided in software, notably the ubiquitous xterm program, which first appeared in 1984.

Although xterm has been updated continuously since its introduction, and continues to attract both new users and new implementations (with some enhancements), its basic functionality remains unchanged, rooted in the era of punch cards and leaded gasoline. While most software and technology from that era has been superseded, xterm lives on, continuing to emulate now-defunct machines in the name of "compatibility", a zombie haunting the city of Troy.

The problem is not that xterm & friends are ancient. The problem is that they remain the state of the art. The deficiencies of the xterm model are well known. Not only is the basic functionality limited to hardly more than the VT-100 that it emulates, but basic design characteristics of the terminal make it vulnerable to failure during routine use. Because it can be configured through in-band signaling, arbitrary data streams routed to the terminal can easily leave it in an unusable state. For most users, the solution is simply to close the window and start another session, losing whatever work was in progress.

Nor can xterm evolve. Its raison d'etre is emulation; it must adhere to the standard. Because it is ubiquitous, every supporting application expects and requires VT-100 functionality.[1] This network traps programmer and user into continuing the tradition.

It was not supposed to turn out this way. Human factors research pointed the way, as it were, to the WIMP interface: windows, icons, menus, pointer. The character-cell terminal and the primitive command-line interface, with its arcane commands and terse, mysterious messages were deemed anachronisms, preventing ordinary people, unschooled in computer technology, from using the machine. The *graphical user interface* (GUI) would make powerful machines more accessible and their users more productive. Indeed, that *has* happened, of course, as the fortunes of Microsoft and Apple attest.

But for millions who depend on the computer for their daily work, the WIMP interface is not enough. It is what they use for relatively simple, non-repetitive tasks. Contrary to research and universal expectation, nontrivial tasks continue to be most conveniently managed by issuing commands to an interpreter. Work that requires the computer minutes or hours to complete, that needs to be logged, that is repeatedly invoked, sometimes with minor changes to the input: all are commonly executed in the pre-WIMP world of the command-line interface, almost always in a "terminal window" emulating the venerable VT-100.

Far from disappearing, the legion of xterm users is only growing. The arrival of so-called "big data" and the "data science" to cope with it has thrust a whole new group — whose numbers are often too young to remember *The Mary Tyler Moore Show* — into the xterm world. They are a legion of nomads, migrating

[1] In fact, they require xterm functionality, and xterm is now the de facto standard for the thing it emulates.

in time and space as needs dictate, from the near-forgotten world of xterm to the modern GUI. Clearly they didn't arrive at their mode of working by following fashion or advertising. However strange and antediluvian it might be, the xterm environment must afford convenience, human factors research notwithstanding.

The vast on-going experiment that is Linux provides an object lesson: the WIMP interface may be ubiquitous and approachable, but expert users gravitate to what is actually a *simpler* environment, where they can issue complex commands using the entire keyboard. Yes, it's hard to learn and not for everyone. But we shouldn't be *too* surprised. The same can be said, for example, of the violin, or algebraic notation. Complex tasks are often best addressed with tools that seem baffling to the beginner.

If users are voting with their feet, computer science and commercial forces have conspired against them saying, "No, that's wrong, that's old, that's obsolete. You should use the GUI.". A better answer might be for those same entities to respond to the real-world experimental results, and provide a better terminal.

1.2. `xterm` Limitations

Nothing about the limitations of xterm is new or surprising. If anything, the surprise is in how long the list is, and how longstanding.

1.2.1. Graphics

The first obvious missing feature — although not *entirely* absent, as it happens — is the lack of graphics. The xterm program emulates a character-cell terminal. Graphics of any kind, including text with proportional fonts, are displayed in other windows, unconnected to xterm.

This situation is so long-standing that it's considered natural. Experienced users invoke commands in xterm that open a web browser here, an image viewer there, perhaps a spreadsheet, or presentation, or chart. Then the xterm nomad migrates to the graphical application and uses it, separately, until needs dictate that he return to xterm.

It is ironic. A standard xterm might be displayed *in* a GUI using a window of 484 dots across and 316 high, but it can't *use* the GUI. Those 152,944 dots are inaccessible to the xterm application, and the xterm user, except to display a fixed-width font in a 80x24 grid. It may be used to launch *but not control* other applications, most of which take advantage of the dot-addressable screen afforded by the GUI.

Actually, the above statement isn't perfectly true. Those pixels *are* addressable in an xterm, just not feasibly. Besides the VT-100, xterm also emulates the Tektronix 4014, which was dot-addressable using a 10-bit scheme, meaning the application sends 4 bytes to encode the on/off values of 5 dots. At 38,400 bit/sec, commonly used in xterm configurations today, it would take

$$\frac{4/5 \cdot 152,944 \cdot 8}{38,400} = 25 \textit{ seconds}$$

to draw a whole screen. The equivalent GUI operation requires a few microseconds.

1.2.2. Characters

In 1978, virtually all the world's computer users spoke English. The VT-100 reflected that fact by using the ASCII character set, which supports most of the characters needed for English (but lacking, for instance, the £ and ¢ symbols). Although in recent years xterm has added support for Unicode, which defines characters for all modern languages, for an xterm user to take advantage of Unicode requires much more than simply starting it in Unicode mode. Because xterm cannot inform an application of the user's preferred character set, both xterm and application must consult a list of variables maintained by the operating system for each process (but useful chiefly to xterm and associated applications).

A typical GUI offers dozens of fonts, many proportional, but xterm, true to its ancestor, uses a fixed-width font. System utilities often arrange output in columns that assume each character is the same width. Although the same information could be arranged more attractively and more *informatively* using proportional, scalable fonts, no standard outlet exists with those features. xterm is the least common demoninator, with the emphasis on *least*.

One common class of typeset documents is frequently viewed in an xterm: the system manual, better known as the *man pages*. Consider

NAME
rm, unlink — remove directory entries

SYNOPSIS
rm [**-dfiPRrvW**] *file* . . .
unlink *file*

DESCRIPTION
The **rm** utility attempts to remove the non-directory type files specified on the command line. If the permissions of the file do not permit writing, and the standard input device is a terminal, the user is prompted (on the standard error output) for confirmation.

This abridged example was taken verbatim from one of 7,602 pages of documentation installed on the system used to prepare the present text. The variety of fonts help the reader distinguish between explanatory, placeholder, and literal text.[2] The typical user never sees the pages presented this way, with proportional and italic fonts, because xterm does not support them.

When the user types *man* he invokes the ancient utility *nroff* to *untypeset* the page. As though tipping its hat to history, nroff eschews even the ANSI control sequence for boldface fonts. Rather, it sends such characters twice, separated by a *backspace* character, thereby "overprinting" the character, as if on a paper teletype. The GNU *less* utility honors these backspace sequences, converting them to ANSI control sequences for xterm to display. Users are sure to be relieved to know that, should it be necessary — or possible! — to connect an DEC Letterwriter 100 to the system, man pages will print just fine, with boldface intact.

The nroff utility served a useful purpose in 1978, when the computer might have taken minutes to print a typeset page, and when users had only character-cell terminals to view the pages with online. A modern computer renders the page faster than it can be downloaded and displayed on a web browser.

The artificial limitations imposed by xterm and how it renders man pages has a pervasive, perverse effect on the technical community. Man pages are easily found as PDF documents on the web, and frequently they are rendered *not* in the most readable way, using a proportional font as presented above, but in a *monospace* font, emulating the nroff output. So accustomed are users to seeing man pages that way, they seem to think that's how they're *supposed* to look, unaware that the very same files can be rendered in proportional fonts with barely a change to the processing command.

1.2.3. Color

The first VT-100 terminals had green characters on a black background. There was no provision to change it. A modern GUI provides millions of colors, and xterm can take advantage of a handful of them. But the application cannot interrogate the terminal to discover its color configuration. Consequently it cannot adapt the colors it uses (or would use) for characters to the user's expressed preferences.

The color problem is perhaps best illustrated by the pedestrian *ls* command, used perhaps hundreds of times a day by an xterm user to display file names. Because in Linux a file's function is independent of its name, ls these days can display different kinds of files in different colors. This speeds the user's ability to distinguish between directories, executables, and plain text files, among others.

Or doesn't. The ls utility cannot discover the background color of an xterm window; there is no technical facility to do so. Consequently the user must choose colors carefully and selectively, lest ls become unusable. If he choses, say, green for directory names, it may be perfectly readable against a white background. Against a green background it will be invisible, something an experienced user can anticipate.

[2] `Monospace` font represents computer input or output. **`Boldface`** indicates something the user types. *`Italics`* is a placeholder; in this case *`file`* is to be replaced with a filename. The remaining text, set in a proportional font, is explanatory, except for appearance of the command name, **rm**, which is boldfaced to make it easy to locate and to signify that it refers to the command, and not the English word. (Although **rm** isn't one of them, some commands are also English words.)

Many combinations are hard to read, and it's not easy to guess what might work and what won't. Which colors work "well enough" is a matter of experimentation.

1.2.4. Complexity

The technology emulated by xterm is surprisingly complex, as might be guessed by perusing its 25,511 word manual. Most of that capability is long obsolete, because it offers "conveniences" that make sense only when the user has a single terminal, with a single 80x24 display, through which to interact with the system. Features such as alternate displays, font sizes, line-drawing characters, a graphical mode, and the ability to emulate long-vanished keyboards are no longer relevant. As an example, xterm includes the ability to switch between 80- and 132-column modes; the sending application need only transmit the proper sequence, and xterm will switch to a narrower font, 132 characters to a line. In a modern system, the feature is never used. If the user wants a wider "terminal", he grabs the window edge and makes it wider.[3]

Curiously, xterm also imposes complexity on the underlying operating system. Just as xterm is emulating a VT-100, so do components of the OS emulate long-missing hardware to which the VT-100 was once connected. Every xterm is supported by *pseudo-terminal* connected to virtual *serial port* with a *line discipline* running at a certain speed, typically these days at 38,400 bits per second. None of this machinery exists anymore, nor ever will, except in museums and in the virtual world behind xterm.

If all these things worked well together, perhaps the unnecessary complexity wouldn't be so bad. Unfortunately they don't, and cumulative effect is to inhibit progress and disuade programmers from stepping into the tar pit. As, Simon Tatham, author of *PuTTY*, another VT-100 terminal emulator, says on his web site,

> *Once upon a time, I had a vision that maybe it would be possible to take the Linux telnet implementation, and the* xterm *front end, and hack them together to form a terminal-emulator-and-telnet client which should then be relatively easy to port to Win32. So I tried it. It was an utter failure: I have* never *found two more mutually hostile pieces of software. So I re-implemented a terminal emulator, and Telnet, from scratch....*

The old VT-100 provided in its day a measure of user interaction by enabling the application to place characters anywhere on the screen, to change their appearance (including, later, their color). This capability gave rise to the so-called *full-screen editor*, of which *vi* and *emacs* are two examples.[4] It is also largely vestigial today: vi and emacs both long ago developed GUI versions (although the character-cell versions remain useful). Other cursor-addressing applications are mostly forgotten, or have moved to the GUI. The major force that drove that move was the richer environment afforded by the GUI, but another factor was the complexity of writing cursor-addressing applications and the limited functionality such applications could offer.

For the most part, xterm users today invoke programs from a command-line shell. Cursor-addressability is mostly used to move along that command line by character or word, or to jump to the beginning or end of the line.

1.2.5. Fragility

In a VT-100 (and still in xterm today) all information arrives at the terminal from the system via the (now virtual) serial port as a stream of bytes. Meta-information — to control appearance, place characters at arbitrary positions, or to turn features on and off — arrive over the same port as part of the same information stream. This interlacing of data with metadata is known technically as *in-band signaling*. More commonly the control sequences are referred to as "escape codes" because the first character is the ASCII *escape* character. Today of course GUI applications achieve such effects and much more through *out-of-band* signaling: the application controls the window through function calls to the GUI. Information to be displayed in the window carries the data only, never metadata.

[3] On the system used to compose this paper, the maximum terminal size was unimaginable in 1978: 318 characters wide and 88 high, with a dot-addressable display containing 2,261,760 pixels, 1920 across by 1178 high. The processor supporting it is perhaps 9 orders of magnitude faster than the Intel 8080 used int VT-100.

[4] Nowadays, of course, these applications run in terminals that are themselves windows. Hardly anyone calls them "full screen" anymore, and fewer still remember the *line editors* they replaced.

Because most of the complex features that the control sequences can affect are unused, the sequences themselves are, for the most part, unwanted. The user does not want the incoming data stream — i.e., the data to be *displayed*, or so he thought — to move the cursor, or change the character attributes, or alter the font size. Most especially he doesn't want the meaning of the *backspace* and *return* keys to be changed. As a baseline for functionality, the user would prefer that whatever information arrives at the port, the configuration of the xterm displaying it remain unchanged. Unfortunately, that cannot be the case for xterm if it is to stay true to its purpose of emulating a VT-100 terminal.

With some frequency — not every day, perhaps, but a few times a month, even for an experienced user — an attempt to view a file ends in tragedy because it contained data that xterm interpreted as a control sequence. In most cases the cause is pilot error; the experienced user learns never to send data of uncertain provenance to the terminal. Looked at another way, though, it's failure by design: why should sending arbitrary data to the screen cause the system to become unusable?

It's worth exploring that question a moment. How likely is it to happen, after all?

A VT-100 control sequence begins with the ANSI *escape* character, a single byte whose value is 27. A sequence of `27 91 2 74`, for instance, clears the screen. Other sequences turn features on and off, or change the meaning of keys.

A byte can hold up to 256 different possible values, of which 27 is one. The probability of a randomly selected single byte in a randomly selected file having that value is 1/256, or 0.39%. As the size of the file sent to the terminal grows, the probability of one of those bytes being 27 approaches 1.0, as does the probability that the ensuing happenstance sequence will be "interpreted" and acted on by the terminal. Meanwhile, the probability of that being the user's intention or a desirable outcome approaches zero.

1.2.6. Functionality: Not

A few basic functions make the command-line environment easier to use:

- Editing the command line: moving to the beginning or end of the line, moving by word or by character, and cutting and pasting portions of the command
- Listing and searching through previously issued commands
- Displaying file contents or program output, and scrolling the text up and down

Fundamental as these are, they are not provided by xterm. All xterm does is send keystrokes and display characters. It is up to the program interpreting those keystrokes to do something useful with them.

The first two functions are typically provided by the keystroke-processing library *readline* or, sometimes *editline*. Simple programs that do not use these libraries leave the user out in the cold. Including them isn't especially difficult,[5] but their very necessity is a throwback to the non-programability of the VT-100. *All* functionality, even moving the cursor along a single line, is provided by the host application.

Because the terminal sends keystrokes and the receiving program processes them, even cursor movement requires configuration. As with color, the division of function involves the user in arcane detail from which he has absolutely nothing to gain.

An example may help illustrate. The readline program has a *move by word* feature.[6] Usually this is mapped to two key combinations: *control-arrow* (left or right) and, because emacs uses them, *alt-b* and *alt-f* (mnemonically, *before* and *after*). But what is *alt*? The VT-100 didn't have an *alt* key and the present text will spare the reader further arcane details. When the user presses *alt* on his keyboard, xterm sends a control sequence representing the key. The sequence is *configurable*, of course, this time by the X server itself, which defines the keyboard. Because the defaults for readline, xterm, and the X server were set by people who almost certainly don't know each other, it's not uncommon for the X meaning of *alt* not to be what readline expects. Then, when the user types,

[5] readline does present some portability issues, however.

[6] In fact, every readline feature can be mapped to any key. A great many Linux users have spent a day or more discovering how.

```
$ ls *.txt
```

and presses alt-b, instead of moving the cursor back before the *****, he sees

```
$ ls *.txtâ
```

This happens because the control sequence sent by xterm is not the one readline recognizes as meaning "more the cursor left one word"; in fact, xterm hasn't sent an actual ANSI control sequence, one that leads off with the ASCII escape character. Not to worry, because xterm has a feature called *meta sends escape* that maps what the X server has deemed the *meta* character to send an ASCII escape. This feature can be invoked through an xterm menu. To make it permanent, the user who knows the X property grammar simply makes the appropriate entry for xterm in the `~/.Xresources` file, and configures the X server to update its property database (by running xrdb) when it starts. He's also well advised to make sure the terminal and readlines are similarly configured on all the machines he uses. Frequently, he decides just to use *control-arrow*.

The last function, displaying and scrolling text, involves an odd ritual so engrained in the user experience that it *seems* natural, but isn't. If a command produces more output than can be displayed on the screen — that is, in the xterm window — the text scrolls off the top of the screen to make room for the next line at the bottom. xterm supports a limited capture feature: the user can set the number of lines to be buffered off-screen, to be scrolled down to view. The contents of this scrollback buffer cannot be searched or archived, except by copying and pasting the text.[7]

A better facility for displaying files and capturing output is a *pager*. Most programs that present information for the user to peruse do not include a built-in pager. By convention, the application discovers the user's configured pager by consulting the `PAGER` environment variable, the value of which is usually the name of a text-display utility. In the event no such variable has been established, most programs will invoke the system pager, often `/usr/bin/more`.

Because the xterm scrollback buffer is essentially featureless, the novice user quickly learns the virtue of capturing text in the pager. Because that's not done automatically, he becomes accustomed to tacking `| more` (or similar) onto the end of any command likely to produce voluminous output. Thus does a basic feature of the display become one the user learns to provide for himself.

Very few users use more than one pager. In fact, very few pagers exist and one, *GNU less*, is ubiquitous. The lack of variety is an indication that pager functionality is not something users feel the need to customize. The continued active development of GNU less, on the other hand, demonstrates its usefulness.

1.2.7. `xterm`: Unsuited to the Purpose

The modern command-line interface — exemplified by xterm and emulated by many others — has ossified. It suffers from being *good enough* for most purposes; its features tolerated because they can usually be avoided. The layers of emulation and the design goal of emulation inhibit innovation, because changing anything means changing a great many others, with only speculative benefits and near-certain resistance from those who maintain and use them. The xterm utility is a vehicle through which other work is done, not an end to itself. As such, it is crucial that it remain *unchanged*. Although the problems are long-standing and widely recognized, the path out of the morass is not.

Vendors are uninterested, for if the xterm user community has voted with its feet, it is not voting with its wallet. Widespread adoption of any plausible successor would require the cooperation of a variety of actors, many of whom have no particular interest in xterm *per se*. To the extent that an alternative required changes to existing software, such change is extremely unlikely to be forthcoming unless the source code were freely available. An open-source implementation would both encourage widespread distribution (as part of Linux) and guarantee the program's continued availability into the future. Desirable as such a new platform might be to users, it offers no profit incentive to commercial software vendors.

It is a market ripe for disruption that no one can disrupt.

[7] And *only* the text. No formatting information is preserved.

Many of the same constraints apply to the free software community. A new and better terminal, while not nearly as complex as many successful free software projects, is too large for one person to undertake in his spare time. Without a reference implementation, the intrepid programmer has little hope of attracting attention, much less persuading any like-minded fellows to join him. In short, something radically new offers no model for a community to form around.

The best hope for change would come from a large, publicly minded entity interested in improving the workaday experience of millions of people in exchange for their mostly silent thanks. And, of course, for the satisfaction of a job well done.

2. Features of a New Terminal

Consider in broad strokes what a highly functional command-line facility might look like.

- handle all cursor movement
- capture program output automatically, infinitely, and facilitate searching/saving it
- support *job control* in some form, to run jobs in the background and capture their output *modified shell* ask the simplified shell for help with tab-completion
- provide a dot-addressable canvas for applications

In short, something of a cross between a web browser (local cursor, infinite capture, multi-session) and a printer (dot-addressable canvas). Or an interactive PDF document, perhaps. This section describes how each of these features simplifies the entire system and improves the user experience.

2.1. Local Cursor Control

Local cursor control simplifies the system by removing the need for several pieces of software. It improves the user experience by returning control to one program, the local terminal.

On the system used to prepare this paper may be found no less than 85 programs that interpret the keyboard in some way. Many are, for all intents and purposes, obsolete. Some, such as *bash*, are shells. A few others, such as *top* (which shows processes consuming CPU time) make use of xterm's character-cell addressability. None has any need to control the cursor. They control the cursor to afford the user certain conveniences commensurate with a 1970s system.

By removing cursor control from the host program, several pieces of supporting software fall by the wayside:

readline Instead of managing individual keystrokes, the application accepts fully formed input from the terminal. readline also keeps a history of commands and provides features for the user to scroll back or search through them. One of the complexities readline confronts is to keep a separate history for different client applications for the user. Another is that the user may — in principle, at least — have any kind of terminal and keyboard known to the system,[8] and the user may map any key to any readline function. Unfortunately this feature offers less than it might seem: if the user *does* connect with another type of terminal, readline will *not* detect it. It is up to the user to configure his login scripts to load different readline options for different terminals, or to harmonize among the terminals the keystrokes sent to readline.

termcap The mirror of readline, termcap provides a uniform set of functions to address the terminal, allowing a program to clear the screen, for instance, without regard to the kind of terminal attached or the particular control sequence used for that purpose.

ncurses The little-loved ncurses program provides a similar set of functions for keyboard management, allowing a program to move the cursor anywhere on the screen in response to the keys pressed. The new ncurses program is rare indeed. Most such work migrated long ago the a GUI or web browser.

[8] and long forgotten by man

pty The pty driver is probably best described by its man page:

> *The pty driver provides support for a device-pair termed a pseudo terminal. A pseudo terminal is a pair of character devices, a master device and a slave device. The slave device provides to a process an interface identical to that described in tty(4). However, whereas all other devices which provide the interface described in tty(4) have a hardware device of some sort behind them, the slave device has, instead, another process manipulating it through the master half of the pseudo terminal.*

The pty is responsible for handling various aspects of communication with the terminal. For example, a program may accept one line at a time, known as *cooked mode* or, if as in the case of a text editor (or *top*), it may put the line in *raw mode* to process each keystroke. It also handles a primitive stop/start feature, whereby the user may press a special key, usually *control-s*, to stop the computer from sending output to the terminal. This permitted the user to read the fast-appearing output before it disappeared off the top of the screen, handy for those occasions when he hadn't thought of the pager beforehand. Saves paper, too.

It may be noted that *control-s* is used for a variety of purposes in different applications, usually in support of *search* or *save*. The unwary user who presses control-s in the shell soon finds it mysteriously unresponsive. The experienced user recognizes his "mistake" and presses control-q to resume communication. Everyone else just abandons the window as unusable.

Simple applications — those with no logical use for keyboard control, or that, like top, update a static display, can dispense with the entire supporting cast of terminal-control software. They can simply read the input prepared at and sent from the user's terminal, and send output to it as though to a printer. This mode of interaction is in fact very familiar; it describes interaction with a web browser.

2.2. Local Pager

The computer supporting xterm today has vast resources far in excess of that required to buffer output from a command-line program, even if that output is millions of lines.[9] The model for a local pager is the existing pager, exemplified by GNU *less*, and simplified, shorn of its concern for terminal variety and line-discipline issues. Retained would be its search and scroll features, and its ability to save the information it captures.

Something missing from *less*, because it's only a pager, after all, is the capacity to invoke other programs or "follow a link" in a general sense. As shown later, this could be a feature of a local pager.

2.3. Local Job Control

In an xterm connected to a shell that supports *job control*, the user can run multiple processes at once in the same shell. Faced with a (perhaps unexpectedly) long-running command, the user sends a *suspend* signal, usually by pressing *control-z*. This places the active process — often one that is unresponsive, predictably or not — in the "background". The user is presented with a new prompt, and may choose to leave the task suspended, continue it in the background, or kill it.

While support for this feature depends on the shell, what does *not* depend on the shell is where the output goes. The terminal has a single connection open to the shell; all input and output travel over that link. Background tasks may write to it, and will, if their output was not redirected when the task was started.

It's hard to see much utility in job control in an environment in which the terminal manages command history and paging. Instead of a *suspend* signal, *control-z* could be trapped by the terminal. The running job would continue to run, its output captured in a local buffer to be viewed later, and meanwhile a

[9] If for some reason the sending program *does* exceed the capacity of the receiving terminal to deal with it, the terminal always has the option of not receiving it, that is, of not reading what the sender is sending. The effect is to block the channel at the sender's end, precisely the same as with *control-s* but much more efficiently, and under program control.

fresh connection opened, providing a new prompt. Whether the "background" tasks are managed in "tabs", as in a web browser, or some other way could vary by implementation.

2.4. Tab Completion

One very helpful feature provided by most shells today goes under the rubric of *tab completion*. Using GNU *bash* as an example, the user types part of a command or filename, and presses tab. The shell, which is processing keystrokes one at a time, can usually infer what the incomplete word refers to. If the partial text matches a unique word, shell provides it and advances the cursor. If not, nothing seems to happen, unless the next character is also a tab, in which event the shell prints a list of choices. Often with a few more keystrokes the user can form a unique substring, press *tab* again, and move on.

In an xterm the shell processes the command-line a keystroke at a time. A terminal with local cursor control would send entire commands, fully formed, a line (or more) at a time. What then becomes of tab completion? Should the terminal do it itself? And, who needs the shell anymore anyway, if the terminal has taken over its work? Doesn't the terminal become the shell?

In the presence of a terminal such as this document describes, the shell is diminished and simplified, but not obsolete. In the xterm world, the shell is really two components: a command processor and a user interface. Indeed, some Linux systems distributions use two shells: *bash* for interactive use, and *dash* for script processing.[10] A terminal with more local capability takes over many user-interface features, but the shell retains the function of command-line processing (and script processing).

Because the terminal is not actually interpreting the command line, it *cannot* support tab completion. The shell *defines* both the command-line syntax and the locations it will search for files and executables. Tab completion therefore is a user-interface feature the shell would need to support, albeit not in the xterm way.

If the terminal normally sends complete commands to the newly simplified shell, how might it request suggestions from the shell, and provide them to the user?

The question requires careful consideration. While several solutions are possible, one must take care not to impose new syntax that might conflict with existing use. Because there are millions of shell scripts and shell users, the window for new syntax is nearly closed. Nevertheless, shell authors and POSIX mavens may find a way.

A different approach would be to use a second connection. The terminal would send whole commands over the usual channel, and use a second connection, perhaps called the *metachannel* for help of various kinds. The terminal could send, say,

```
PARTIAL: open documen→
```

(where the → symbol represents the ASCII TAB character). The shell could reply with a list on the same channel for the terminal to display. This mode of interaction over a second channel opens a wealth of possibilities unavailable to xterm.

2.5. Dot-addressable

By dropping the character-cell model, we free the terminal to avail itself of the underlying GUI of which it is a part. Rather than simply displaying ASCII characters as they're sent by the program, and intercepting a few control sequences for primitive positioning and simple character attributes, the terminal can present the window it occupies to the program as an open canvas on which to draw.[11] In effect, the terminal becomes a virtual printer. When the application is done drawing, it presents a cursor and a prompt, and resumes command-line processing.

[10] *dash* has been shown to be about twice as fast at script execution as *bash*. It achieves efficiency partly by eschewing features that make interactive user more convenient.

[11] Veterans of the early days of GUIs may recall the promise of *Display Postscript*, which once promised to be the way programs rendered text and graphics on the screen (and lingers on today in OS X). When it was proposed in the late 1980s, workstations were far slower than modern machines, had relatively few fonts, and lower resolution displays. These shortcomings detracted from Postscript's otherwise fine qualities for the purpose of drawing on-screen graphics.

3. Introducing VT-roff

A VT-roff terminal would support the most widely distributed, and nearly forgotten, document preparation system in existence: *troff*, and its GNU implementation, *groff*.

Output to the terminal is of two kinds:

1. Plain text, encoded as UTF-8, where the sending application (e.g., *ls*) assumes a monospace font.
2. Formatted text and graphics, as a virtual troff printer.

At right is the device-independent language sometimes called "ditroff" that Brian Kernighan invented in 1982 to serve as an intermediary between *troff input*, with which a user prepares a document (usually with the aid of a macro package), and *device input*, the langage of the printer or other device.

The *troff* program emits this primitive move-here-print-that language, sending it to a so-called *post-processor*, which uses whatever mechanism the device supplies to render the output. VT-roff would be such a post-processor, and the underlying graphics library often, but not necessarily, X. Not only is it simple to implement in terms of X, it's already been done, quite successfully, 20 years ago. The groff project includes *gxditview*, intended for viewing troff documents natively on an X11 system.

Let us now consider how VT-roff compares with xterm based on the analysis in the previous section.

Output Language

s*n*	size in points
f*n*	font as number from 1 to *n*
c*x*	ASCII character *x*
C*xy*	character \(*xy*; terminate *xy* by white space
H*n*	go to absolute horizontal position *n*. (*n* > 0)
V*n*	go to absolute vertical position *n* (down is positive)
h*n*	go n units horizontally (to the right; *n* > 0)
v*n*	go n units vertically (down; *n* > 0)
nnc	move right *nn*, then print *c*
n*b a*	end of line (information only — no action needed) *b* = space before line, *a* = after
w	paddable word space (information only — no action needed)
p*n*	new page *n* begins — set V to 0
x ...\n	device control functions
D ...\n	drawing functions (graphics)

Device Control

x i	init
x T s	name of typesetter is *s*
x r *n h v*	resolution is *n*/inch, *h* = minimum horizontal motion, *v* = minimum vertical
x p	pause (can restart)
x s	stop — done forever
x t	generate trailer
x f *n s*	font position *n* contains font *s*
x H *n*	set character height to *n*
x S *n*	set slant to *n*

Drawing Functions

Dl *dh dv*	draw line from current position by *dh dv*
Dc *d*	draw circle of diameter *d* with left side here
De *d1 d2*	draw ellipse of diameters *d1 d2*
Da *dh1 dv1 dh2 dv2*	draw arc from current position to *dh*1 + *dh*2*dv*1 + *dv*2, center at *dh1 dv1* from current position
D~ *dh1 dv1 dh2 dv2 ...*	draw B-spline from current position to *dh1 dv1*, then to *dh2 dv2*, then to ...

3.1. VT-roff Capabilities

3.1.1. Informational Coherency

Because it is cursor-addressable, input and output on an xterm are interspersed: The user may enter information anywhere on the screen (wherever the application has placed the cursor) and the system's output may likewise appear anywhere. It is a system without history. Many utilities clear the screen at some point, erasing what came before, regardless of how it was created or the user's wishes. Processes running "in the background" under job control can also write to the terminal, sometimes with the effect of inserting the output of one command in the midst of output from another on the same line, or in the middle of the user's input. The result can be difficult to interpret even for the experienced user.

Always Forward

VT-roff adheres to the policy: *output follows input*. Input and output both appear at the cursor and nowhere else. VT-roff provides no cursor control to the application; there is no "up". Consequently history is preserved; prior commands and their results can reviewed in simple, linear form.[12]

On pressing *Enter*, the terminal sends the command to the system, and moves the cursor to the beginning of the next line. Output commences from that position.

Combining Text and Graphics: `stdgrph`

VT-roff accepts output from the system over two ports, one for text, the other for graphics.

The text port, as ever, is connected to standard output. VT-roff interprets the text literally: it emulates a simple dumb terminal, printing the characters sequentially in monospace font, honoring tabs, carriage returns, and newlines. In short, it behaves like xterm, minus cursor control. Readers of a certain age may recognize its resemblance to a typewriter.

The second port, here designated *standard graphics*, accepts ditroff-formatted text and graphics.

An application that prepares its own ditroff sequence can simply open a connection to the standard graphics port and begin writing. The `x init` sequence puts the VT-roff implementation into "graphics mode" and indicates to the user that something graphical (or at least not just plain text) is about to displayed. One example of such an application might be groff itself, with a new post-processor that simply redirects the ditroff output to the graphics port.

Applications that rely on troff or its preprocessors to generate ditroff need only invoke troff using the VT-troff post-processor.

Perhaps the simplest solution is to have a shell, when hosted by VT-roff, open a fourth file descriptor. The application could simply write ditroff directly to *filedes 3*.

By using two ports, VT-roff avoids the problems that accompany in-band signaling: *any* valid UTF-8 values output to the primary port are reflected int the terminal, and *no* values presented there affect the terminal's configuration. Nothing is erased. The meaning of the keys is unaffected; the terminal remains fully functional. Extra "garbage" (perhaps graphics instructions printed literally) is at least readable, if inscrutable. For convenience a VT-roff implementation could choose to provide the user with the ability to select and erase displayed information to remove unwanted output.

Arbitration and Type-ahead

Because VT-roff produces one display from three input streams (keyboard, standard output, and standard graphics) it must in some sense arbitrate between them when more than one is sending data at a time. It would be a shame for a a graph or typeset graphical output to be disturbed by an accidental keystroke or warning message from the sending application. Better would be to let each operation run to completion unless the user explicitly request it be interrupted.

xterm does not arbitrate; if the user types something while output is being displayed the typed characters appear amidst the output, producing a confusing result. Experienced users exploit this feature by "typing ahead", queuing keystrokes until the command-processor reads them.

To ensure a coherent display, while the graphics port is active, it is given priority. Text output from the system is given secondary priority; any messages produced "alongside" the graph are printed subsequently.[13] To facilitate type-ahead (which remains useful), VT-roff could capture the "next" command in a sub-window of some sort that provides a temporary (but fully functional) place to prepare the next command.

[12] This always-forward policy applies to the cursor, which exists only in text mode. Graphics mode has no cursor, and ditroff commands perforce may draw up and down as desired, permitting periodic redrawing based on time or user input.

[13] A VT-roff implementation should enable the user to terminate the graphic output if so desired.

3.1.2. VT-roff Formatted Text Graphics Support

Implemented as a troff post-processor, VT-roff is as capable of producing typeset text and graphics as any printer, subject to display resolution.[14] Existing documents, notably man pages, become more readable without alteration. The troff tools, installed on most Linux systems, are available for mathematics, tables, vector drawings, and graphs. Depending on need and sophistication, programs wishing to address the screen graphically could send ditroff directly, or invoke troff tools.

Some potential features — new capabilities afforded by implementing VT-roff as a ditroff post-processor, should an implementation choose to support them — stretch the meaning of a *terminal* as generally understood. For example, strange-but-useful interactive programs based on ncurses, such as *top*, could be re-implemented in terms of ditroff by "overprinting the page". Similarly, hypertext links — already supported in the GNU troff PDF post-processor — could be displayed and used. Not only would that pave the way to hyperlinked man pages, but simple filters could use the terminal's underlying graphical features. Among other uses, relatively simple programs might display image files and PDF documents, or implement a web browser.

3.1.3. `s/Complex/Simple/g`

VT-roff trades the accidental complexity of VT-100 cursor addresability for simple dumb-terminal text and equally simple printer graphics. It presents its graphics and formatted text functionality as a new-old thing:

- a language consisting of 27 directives, 11 for text output, 5 for graphics, and 9 for device-control.
- The higher-level troff input language on which troff macro packages are built
- various pre-processor packages with domain-specific languages to describe tables, equations, pictures, and graphs

These languages provide different levels of abstraction from the underlying GUI, and afford the programmer different levels of control for the display of high quality text and graphics. Furthermore, by using *existing* tools, the same output can readily be redirected to a printer or PDF file.

At the same time VT-roff removes the need for several tools regularly employed (sometimes unwittingly) by user, and the need to make those various tools interact smoothly. Rather than seeking out the readline configuration file or guessing how to coordinate ls colorizing with xterm colors, the user configures VT-roff, once and for all.

3.1.4. `s/Fragile/Robust/g`

By discarding xterm and its VT-100 emulation, we close the door on myriad problems, not least of which is the fragility of the terminal, the way it's subject to failure when presented with an incoherent data stream. The language contains no commands to affect the keyboard, to start and stop output, nothing that could send the terminal off to never-never land, as so often happens with xterm. Notably absent, in fact, is any notion of conventional terminal control, including clearing the screen.

While a terminal prepared to interpret this input would not be able by itself to render, say, a PDF file, neither is it apt become corrupted in the attempt, because the ditroff language lacks any feature to alter or configure the VT-roff terminal.

4. Similar Work

The VT-roff terminal includes the following features:

- command-line editing
- integrated pager

[14] Laser printers have much higher resolution than computer displays, typically 1200 dpi and 96 dpi, respectively.

- replacement for job-control features
- simple, dumb-terminal display of text
- Mixed “dumb” text and printer-like graphics in the same window as an infinite, integrated, scrolling buffer

To the author’s knowledge, nothing quite like it has been attempted.

Character-cell terminals such as xterm emulates disappeared decades ago. Graphical *terminals* never replaced them. Rather, they were replaced by entire GUIs, of which xterm was supposed to be an increasingly diminishing part.

GUIs themselves do, therefore, in some sense represent similar work, a giant experiment in supplanting the character-cell terminal. Observation suggests they will continue to demonstrably fail one class of users.

The web browser can also be seen as similar work. The user issues a “command” to the browser (in some sense) by entering a URL or filling in a form, and the result combines text and graphics in one window. The browser also in effect offers job-control and the ability to browse command & output history.

A web browser includes features not contemplated for VT-roff, such support for audio and video, or just handing control of a portion of the window to another application. On the other hand, the web browser is poorly integrated (often intentionally) with the system it runs on.

The author is not aware of research using the web browser to replace or integrate with a command-line shell.

Bell Labs produced a better terminal for *Plan 9*, the operating system it designed as a replacement for UNIX. That terminal, known as *8½* in Plan9 and as *9term* in its UNIX variant, includes local cursor control and integrated pager. Like VT-roff, 9term uses UTF-8 text, but 9term has no provision for graphics.

The input and output captured in 9term is not strictly linear, as in VT-roff: the user can move text around the buffer and submit the edited text to the system. Output is then inserted in the buffer at the cursor.[15] 9term provides richer local editing features than VT-roff, which deserve consideration.

5. Example Applications

It may be useful to consider in concrete terms how the existence of a graphical terminal might affect the design of existing applications.

5.1. R

Many applications, of course, integrate text and graphics. Perhaps the *R Project* is as good an example as any. Of necessity — because there is no graphical terminal — R has two kinds of windows: text, for command-line input and output, and graphics for output.

Like any such application, R is more or less forced into dealing with windows — which are orthogonal to its goal of displaying graphs — by the lack of a graphical terminal. Also in keeping with similar applications, it doesn’t do a particularly good job of associating input with output for later review. Nor is there any graphical output *stream*, such as ditroff, that could be captured and shared. Preserving and sharing the graphs is an entirely separate effort.

R is made more complex than absolutely necessary as a multi-window application. Not only must it manage the windows but, because window management varies by operating system, that part of the R system is duplicated, tailored to each operating-system.[16] A graphical terminal would present R with OS-independent graphical facilities: ditroff at the lowest level and, at a much higher level, *grap*, a troff preprocessor that converts e.g. time-series data into a graphs.

[15] known in 9term as the *output point*.

[16] This is one reason so many applications are now written as *web* applications. Browsers vary, but less than GUIs do.

5.2. SQLite

Another class of applications that would benefit from a graphical terminal produce tabular output. Perhaps the prime example would be SQL query editors, of which SQLite is just one example. Like R, most SQL DBMSs offer a GUI tool for composing SQL queries and viewing the results. They suffer from the same problems, too: lack of input-output connection, no linear history, window management and attendent complexity orthogonal to the application domain.

Tabular output — which interactive SQL applications share with R — benefit more than might be apparent at first blush from a dot-addressable canvas. The ability to draw lines between columns and rows, to shade alternating (groups of) rows, to put headers in a larger font, or to otherwise highlight particular regions. In VT-roff, the integrated pager would support scrolling and searching just as with any other text.

Tabular output is effected most readily in troff with the *tbl* preprocessor, the output of which is passed to troff, which in turn could write to VT-roff. While that would be a perfectly fine version 1.0, the terminal-as-printer motif does fall a little short in terms of user experience: the user cannot resize the columns of the tabular output, or re-sort the table by a particular column, or indeed affect the output in any way. The simple ditroff language has no notion of "row" and "column", much less of a table, or sorting one.

The experience could be greatly enhanced if VT-roff had its own, simpler version of tbl devoted to regular, grid-like output that the terminal could capture and manipulate. Such a feature is outside the scope of this paper, however.

5.3. Documentation

The state of Linux documentation is one of disarray, using a variety of technologies. The system itself is documented as man pages, which are troff documents. Some application also supply man pages, but many use HTML or PDF documents instead. Some GNU projects use *info* files, a system once promoted by the GNU project as a replacement for man pages.

This variety is partly the product of desire on the part of both writers and readers for good-looking, readable documents, something xterm and nroff decidedly do not provide. VT-roff would rectify that situation immediately, simply by supporting formatted text.

A higher level of functionality would include support for hyperlinks. In troff, hyperlinks are supported two post-processors: *grohtml* for HTML, and *gropdf* for direct PDF output.[17] Extensions to the *device control* portion of ditroff let the troff input supply hyperlink data to the post-processor, which can embed them in the output document such that the displaying application recognizes them.

In VT-roff, the terminal itself is the displaying application. If it supported the same extensions that gropdf does, the user would be able to browse within and between documents. As long as the linked-to document is troff input on a local filesystem, VT-roff would need very little logic to support it directly. Links could be added to man pages, making them as convenient to use as any other system, and much easier to print than HTML.

6. Conclusion

6.1. Capable and Simple

VT-roff incorporates the most important aspects of command-line work, and simplifies both their user affordance and their implementation.

The VT-roff user does not coordinate the colors, fonts, encoding, and keystroke definitions of readline, less, and xterm on the various systems he uses. He cannot crash the terminal by sending it binary data. Output is never lost off the top of the screen, but is uniformly preserved, where it can be scrolled, searched, and saved. Long-running jobs can be parked, their output segregated to another buffer, as appropriate for a concurrent process to a linear-output model. Text and graphics can be interspersed, their relationship maintained by the forward-only output model. With the addition of simple utilities modelled on *cat*, he can

[17] Until recently, PDF documents were not directly supported by groff. groff produced Postscript files, and a separate utility converted that to PDF.

display a variety of files right in the terminal.

The VT-roff programmer has a simple, 11-command language for text output. He need not involve himself in windowing systems and their proprietary functions. Or, should he prefer, the higher-level functionality of troff and its preprocessors are also available, and entirely compatible.

The system supporting VT-roff can eventually shed its character terminal baggage, the software supporting and emulating long-obsolete equipment. troff itself on the other hand, much disused at present, gets a shot in the arm as a newly central part of the display technology. The effort that has gone into creating troff is not lost or reinvented. Efforts devoted to attractive formatted text presentation, somewhat dispersed across various projects because of the lack of a graphical terminal, can be centralized in one place.

A graphical terminal represents a new way for users to interact with command-line applications. In some ways it mimics features outside an xterm environment. In other ways it incorporates them.

6.2. troff Leverage

The troff system is mature, fast, and small. It has been and continues to be used to prepare manuscripts of all lengths, including books. The present text, all of it, renders on an ordinary laptop in one second. The entire suite of troff tools is measured in megabytes, a rounding error on modern systems.

troff has been part of the X environment for its whole existence, and ditroff has already been mapped onto X primitives. Using troff to define a graphical X terminal, while novel, is nonetheless both a natural chose and an efficient use of existing technologies.

6.3. VT-roff Potential

The problems of xterm, although widely acknowledged, present no commercial opportunity for software vendors. Only a freely available implementation, an example for others to explore and follow, and perhaps emulate, would have the standing to move users and programmers away from the tried-and-true, but obsolete-and-restricted xterm terminal.

A graphical terminal represents a real advantage to users and programmers alike. Arguably, VT-roff represents research into a new way the for user to use the computer. At the very least it represents an alternative that hasn't been tried. As the VT-100 passes its 35^{th} birthday, it's about time someone did.