computers are bad

You are receiving this facsimile because you signed up for fax delivery of this newsletter. To stop delivery, contact Computers Are Bad by email or fax.

https://computer.rip - me@computer.rip - fax: +1 (505) 926-5492

2021-05-24 dialogs not taken

Note: I put up a YouTube video about a minor aerial lift disaster in northern New Mexico. You can see it here: https://www.youtube.com/watch?v=1NDc760fxbY.

Note 2: I have begrudgingly started using Twitter to ramble about the things I spend my day on. It's hard to say how long this will last. https://twitter.com/jcrawfordor.

When we look back on the history of the graphical user interface, perhaps one of the most important innovations in the history of computing, we tend to think of a timeline like this: Xerox, Apple, Microsoft, whatever we're doing today.

Of course that has the correct general contours. The GUI as a concept, and the specific interaction paradigms we are familiar with today, formed in their first productized version at the Xerox Palo Alto Research Center (XPARC). Their production version, the Alto, was never offered as a commercial product but was nonetheless widely known and very influential. Apple's early machines, particularly the Lisa and Macintosh, featured a design heavily inspired by the work at XPARC. Later, Microsoft released Windows 3.11 for Workgroups, the first one that was cool, which was heavily inspired by Apple's work.

In reality, though, the history of the GUI is a tangled one full of controversy ("inspired" in the previous paragraph is a euphemism for "they all sued each other") and false starts. Most serious efforts at GUIs amounted to nothing, and the few that survive to the modern age are not necessarily the ones that were most competitive when originally introduced. Much like I have perennially said about networking, the world of GUIs has so ossified into three major branches (MacOS-like, Windows-like, and whatever the hell Gnome 3 is trying to be [1]) that it's hard to imagine other options.

Well, that's what we're about to do: imagine a different world, a world where it's around the '80s and there are multiple competing GUIs. Most importantly, there are more competing GUIs than there are operating systems, because multiple independent software vendors (ISVs) took on the development of GUIs on top of operating systems like CP/M and DOS. The complexities of a GUI, such as de facto requiring multi-tasking, required that these GUIs substantially blur the line between "operating system" and "application" in a way that only an early PC programmer could love.

And that is why I love these.

Before we embark on a scenic tour of the graveyard of abandoned GUIs, we need to talk a bit about the GUI as a *concept*. This is important to comprehend the precedents for

the GUI of today, and thus the reason that Apple did not prevail in their lawsuit against Microsoft (and perhaps Xerox did not prevail in their lawsuit against Apple, although this is an iffier claim as the Xerox vs. Apple case did not get the same examination as Apple vs. Microsoft).

What *is* a GUI?

I believe that a fundamental challenge to nearly all discussions about GUIs is that the term “GUI” is actually somewhat ill defined. In an attempt to resolve this, I will present some terminology that might be a better fit for this discussion than “GUI” and “TUI” or “graphical” and “command-line.” In doing so I will try to keep my terminology in line with that used in the academic study of human-computer interaction, but despite the best efforts of one of my former advisors I am not an HCI scholar so I will probably reinvent terminology at least once.

The first thing we should observe is that the distinction between “graphics” and “text” is not actually especially important. I mean, it is very important, but it actually does not fundamentally define the interface. In my experience people rarely think about it this way, but it ought to be obvious: libraries such as newt can be used to create “gui-esque” programs in text mode (think of the old Debian installer as an example), while there are graphical programs that behave very much like textmode ones (think of text editors). Emacs is a good example of software which blurs this line; emacs simultaneously has traits of “TUI” and “GUI” and is often preferred in graphical mode as a result.

To navigate this confusion, I use the terms “graphics mode” and “text mode” to refer strictly to the technical output mechanism--whether raster data or text is sent to the video adapter. Think about it like the legacy VGA modes: the selection of graphics or text mode is important to user experience and imposes constraints on interface design, but does not fundamentally determine the type of interface that will be presented.

What does? Well, that’s a difficult question to answer, in part because of the panoply of approaches to GUIs. Industry and researchers in HCI tend to use certain useful classifications, though. The first, and perhaps most important, is that of a functional UI versus an object-oriented UI. Do not get too tangled in thinking of these as related to functional programming or OO programming, as the interface paradigm is not necessarily coupled to the implementation.

A functional user interface is one that primarily emphasis, well, functions. A command interpreter, such as a shell, is a very functional interface in that the primary element of interaction is, well, functions, with data existing in the context of those functions. On the other hand, a modern word processor is an object oriented interface. The primary element of interaction is not functions but the *data* (i.e. objects), the available functions are presented in the context of data.

In a way, this dichotomy actually captures the “GUI vs TUI” debate better than the actual difference between graphics and text mode. Text mode applications are usually, but not always, functional, while graphical applications are usually, but not always, object oriented. If you’ve worked with enough special-purpose software, say in the sciences, you’ve likely encountered a graphical program which was actually functional rather than object oriented, and found it to be a frustrating mess.

This has a lot to do with the discovery and hiding of functionality and data. Functional interfaces tend to be either highly constrained (e.g. they are only capable of a few things) or require that the bulk of functionality be hidden, as in the case of a typical shell where users are expected to know the available functions rather than being offered them by the interface. Graphical software which attempts to offer a broad swath of functionality, in a functional paradigm, will have a tendency to overwhelm users.

Consider the case of Microsoft Word. I had previously asserted that word processors are usually an example of the object oriented interface. In practice, virtually all software actually presents a blend of the two paradigms. In the case of Word, the interface is mostly object-oriented, but there is a need to present a large set of commands. Traditionally this has been done by the means of drop-down menus, which date back nearly to the genesis of raster computer displays. This is part of the model or toolkit often called WIMP, meaning Windows, Icons, Menus, Pointer. A very large portion of graphics mode software is WIMP, and the WIMP concept today is exemplified by many GUI development toolkits which are highly WIMP-centric (Windows Forms, Tk, etc).

If you used Office 2003 or earlier, you will no doubt remember the immense volume of functionality present in the menu bar. This is an example of the feature or option overload that functional interfaces tend to present unless functionality is carefully hidden. It makes an especially good example because of Microsoft's choice in 2007 to introduce the "ribbon" interface. This was a remarkably controversial decision (for the same reason that any change to any software ever is controversial), but at its core it appears to have been an effort by Microsoft to improve discoverability of the Office interface through contextual hiding of the menus. Essentially, the ribbon extends the object oriented aspect of the interface to the upper window chrome, which had traditionally been a bastion of functional (menu-driven) design.

Menu-driven is another useful term here, although I tend to prefer "guided" as a term instead (this is a term of my own invention). Guided interfaces are those that accommodate novice or infrequent users by clearly expressing the available options. Very frequently this is by means of graphical menus, but there are numerous other options, one of which we'll talk about shortly. The most extreme form of a guided interface is the wizard, which despite being broadly lambasted for Microsoft's particularly aggressive use in earlier Windows versions has survived in a great deal of contexts. A much more relaxed form would be the "(Y/n)" type hints often shown in textmode applications. "Abort, Retry, Fail?," if you think about it, is a menu [2]. This guidance is obviously closely related to discoverability, basically in the sense that non-guided interfaces make little to no attempt at discoverability (e.g. the traditional shell).

Another useful term is the direct manipulation interface. Direct manipulation is a more generalized form of WYSIWYG (What You See Is What You Get). Direct manipulation interfaces are those that allow the user to make changes and immediately see the results. Commonly this is done by means of an interface metaphor in which the user directly manipulates the object/data in a fashion that is intuitive due to its relation to physical space [3]. For example, resizing objects using corner drag handles. Direct manipulation interfaces are not necessarily WYSIWYG. For example, a great deal of early graphics and word processing software enabled direct manipulation but did not attempt to show "final" output until so commanded (WordPerfect, for example).

This has been sort of a grab basket of terminology and has not necessarily answered

the original question (what *is* a GUI?). This is partially a result of my innate tendency to ramble but partially a result of the real complexity of the question. Interfaces generally exist somewhere on a spectrum from functional to object-oriented, from guided to un-guided, and in a way from text mode to graphics mode (consider e.g. the use of Curses box drawing to replicate dialogs in text mode).

My underlying contention, to review, is this: when people talk about "GUI vs TUI," they are usually referring not to the video mode (raster or text) but actually to the interface paradigm, which tends to be functional or object oriented, respectively, and unguided or guided, respectively. Popular perceptions of the GUI vs. TUI dichotomy, even among technical professionals, are often more a result of the computing culture (e.g. the dominance of the Apple-esque WIMP model) than technical capabilities or limitations of the two. What I am saying is that the difference between GUI and TUI is a cultural construct.

Interface Standardization: CUA

In explaining this concept that the "GUI vs TUI" dichotomy is deeper than the actual video mode, I often reach out to a historic example that will be particularly useful here because of its importance in the history of the GUI--and especially the history of the GUIs we use today. That is IBM Common User Access, CUA.

CUA is is not an especially early event in GUI history but it's a formative one, and it's useful for our purposes because it was published at a time--1987--when there were still plenty of text-only terminals in use. As a result, CUA bridges the text and raster universes.

The context is this: by the late '80s, IBM considers itself a major player in the world of personal computers, in addition to mainframes and mid/minis. Across these domains existed a variety of operating systems with a variety of software. This is true even of the PC, as at this point in time IBM is simultaneously supporting OS/2 and Windows (2). While graphical interfaces clearly existed for these systems, this was still an early era for raster displays, and for the most part IBM still felt text mode to be more important (it was the only option available on their cash cow mainframes and minis).

Across operating systems and applications there was a tremendous degree of inconsistency in basic commands and interactions. This was an issue in both graphical and textual software but was especially clear in text mode where constraints of the display meant that user guidance was usually relatively minimal. We can still clearly see this on Unix-like operating systems where many popular programs are ported from historical operating systems with varying input conventions (to the extent they had reliable conventions), and few efforts have been made to standardize. Consider the classic problem of exiting vim or emacs: each requires a completely different approach with no guidance. This used to be the case with essentially all software.

CUA aimed to solve this problem by establishing uniform keyboard commands and interface conventions across software on all IBM platforms. CUA was developed to function completely in text mode, which will be somewhat surprising considering the range of things it standardized.

The most often discussed component of CUA is its keyboard shortcuts. Through a somewhat indirect route (considering the failure of the close IBM/Microsoft

collaboration), CUA has been highly influential on Windows software. Many of the well-known Windows keyboard commands come from CUA originally. For example, F1 for help, F3 for search, F5 to refresh. This is not limited just to the F keys, and the bulk of common Windows shortcuts originated with CUA. There are, of course, exceptions, with copy and paste being major ones: CUA defined Shift+Delete and Shift+Insert for cut and paste, for example. Microsoft made a decision early on to adopt the Apple shortcuts instead, and those are the Ctrl+C/Ctrl+V/Ctrl+X we are familiar with today. They have been begrudgingly adopted by almost every computing environment with the exception of terminals and Xorg (but are then re-implemented by most GUI toolkits).

The keyboard, though, is old hat for text mode applications. CUA went a great deal further by also standardizing a set of interactions which are very much GUI by modern standards. For example, the dialog box with conventional options of “OK” and “OK/Cancel” come from CUA, along with the ubiquitous menu sequence of File first and Help last.

While being graphical by modern standards these concepts of drop-down menus and dialog boxes were widely implemented in text mode by IBM. From a Linux perspective, this is rarely seen and would likely be a bit surprising. Why is that?

I contend that there is a significant and early differentiation between IBM and UNIX interfaces that remains highly influential today. While today the dichotomy is widely viewed as philosophical, at the time it was far more practical.

UNIX was developed inside of AT&T as a research project and then spread primarily through universities and research organizations. Because it was viewed primarily as a research operating system, UNIX was often run on whatever hardware was available. The PDP-11, for example, was very common. Early on, most of these systems were equipped with teletypewriters and not video terminals. Even as video terminals became common, there were a wide variety in use with remarkably little standardization, which made exploiting the power and flexibility of the video terminal very difficult. The result is that, for a large and important portion of its history, UNIX software was built under the assumption that the terminal was completely line-oriented... that is, no escape codes, no curses.

IBM, on the other hand, had complete control of the hardware in use. IBM operating systems and software were virtually always run on both machines and terminals that were leased from IBM as part of a package deal. There was relatively little fragmentation of hardware capabilities and software developers could safely take full advantage of whatever terminal was standard with the computer the software was built for (and it was common for software to require a particular terminal).

For this reason, IBM terminal support has always been more sophisticated than UNIX terminal support. At the root was a major difference in philosophy. IBM made extensive use of *block* terminals, rather than *character* terminals. For a block terminal, the computer would send a full “screen” to the terminal. The terminal operated independently, allowing the user to edit the screen, until the user triggered a submit action (typically by pressing enter) which caused the terminal to send the entire screen back to the computer and await a new screen to display.

This mechanism made it very easy to implement “form” interfaces that required minimal computer support, which is one of the reasons that IBM mainframes were particularly prized for the ability to support a very large number of terminals. In later block terminals such as the important 3270, the computer could inform the terminal of the

location of editable fields and even specify basic form validation criteria, all as part of the screen sent to the terminal for display.

Ultimately, the block terminal concept is far more like the modern web browser than what we usually think of as a terminal. Although the business logic is all in the mainframe, much of the interface/interaction logic actually runs locally in the terminal. Because the entire screen was sent to the terminal each time, it was uniformly possible to update any point on the screen, which was not something which could be assumed for a large portion of UNIX's rise to dominance [4].

As a result, the IBM terminal model was much more amenable to user guidance than the UNIX model. Even when displaying a simple command shell, IBM terminals could provide user guidance at the top or bottom of the screen (and it was standard to do so, often with a key to toggle the amount of guidance displayed to gain more screen space as desired). UNIX shells do not do so, primarily for the simple reason that the shells were developed when most machines were not capable of placing text at the top or bottom of the screen while still being able to accept user input at the prompt.

Of course curses capabilities are now ubiquitous through the magic of every software terminal pretending to be a particularly popular video terminal from 1983. Newer software like tmux usually relied on this from the start, and older mainstays like vi have had support added. But the underlying concept of the line-oriented shell ossified before this happened, and "modern" terminals like zsh and fish have made only relatively minor inroads in the form of much more interactive assisted tab completion.

IBM software, on the other hand, has been offering on-screen menus and guidance since before the C programming language. Well prior to CUA it was typical for IBM software to use interactive menus where the user selects an option, hierarchical/nested menus, and common commands via F keys which were listed at the bottom of the screen for the user's convenience.

While many IBM operating systems and software packages do offer a command line, it's often oriented more towards power users and typical functions were all accessible by a guided menu system. Most IBM software, especially by the '80s, provided an extensive online help facility where pressing F1 retrieved context-aware guidance on filling out a particular form or field. Indeed, the CUA concept of an interactive help system where the user presses a Help icon and then clicks on a GUI element to get a popup explanation--formerly common in Windows software--was a direct descendent of the IBM mainframe online help.

The point I intend to illustrate here is not that IBM mainframes were surprisingly sophisticated and cool, although that is true (IBM had many problems but for the most part the engineering was not one of them).

My point is that the modern dichotomy, debate, even religious war between the GUI and TUI actually predates GUIs. It is not a debate over graphical vs text display, it is a debate over more fundamental UI paradigms. It is a fight of guided but less flexible interfaces versus unguided but more powerful ones. It is a fight of functional interfaces versus object oriented ones. And perhaps most importantly, it is a competition of "code-like" interfaces versus direct manipulation.

What's more, and here is where I swerve more into the hot take lane, the victory of the text-mode, line-oriented, shell interface in computer science and engineering is not a result of some inherent elegance or power. It is an artifact of the history of computing.

Most decisions in computing, at least most meaningful ones, are not by design. They are by coincidence, simultaneously haphazard but also inevitable in consideration of the decades of work up to that point. Abstraction, it turns out, is freeing, but also confining. Freeing in that it spares the programmer thinking about the underlying work, but confining in that it pervasively, if sometimes subtly, steers all of us in a direction set in the '60s when our chosen platform's lineage began.

This is as true of the GUI as anything else, and so it should be no surprise that IBM's achievements were highly influential, but simultaneously UNIX's limitations were highly influential. For how much time is spent discussing the philosophical advantages of interfaces, I don't think it's a stretch to say that the schism in modern computing, between the terminal and everything else, is a resonating echo of IBM's decision to lease equipment and AT&T's decision to make UNIX widely available to academic users.

Some old GUIs

Now that we've established that the history of the GUI is philosophically complex and rooted in things set in motion before we were born, I'd like to take a look at some of the forgotten branches of the GUI family tree: GUI implementations that were influential, technically impressive, or just weird. I've already gone on more than enough for one evening, though, so keep an eye out for part 2 of... several.

[1] It is going to take an enormous amount of self-discipline to avoid turning all of this into one long screed about Gnome 3, perhaps the only software I have ever truly hated. Oh, that and all web browsers.

[2] Menus in text mode applications are interesting due to the surprising lack of widespread agreement on how to implement them. There are many, many variations across commonly used software from limited shells with tab completion to what I call the "CS Freshman Special," presenting a list of numbered options and prompting the user to enter the number of their choice. The inconsistency of these text mode menus get at exactly the problem IBM was trying to solve with CUA, but then I'm spoiling the end for you.

[3] This is a somewhat more academic topic than I usually verge into, but it could be argued and often is that graphical software is intrinsically metaphorical. That is, it is always structured around an "interface metaphor" as an aid to users in understanding the possible interactions. The most basic interface metaphor might be that of the button, which traditionally had a simple 3D raised appearance as an affordance to suggest that it can be pressed down. This is all part of the puzzle of what differentiates "GUI" from "TUI": graphical applications are usually, but not always, based in metaphor. Textmode applications usually aren't, if nothing else due to the constraints of text, but it does happen.

[4] This did lead to some unusual interaction designs that would probably not be repeated today. For example, in many IBM text editors an entire line would be deleted (analogous to vim's "dd") by typing one or more "d"s over the line number in the left margin and then submitting. The screen was returned with that line removed. This was more or less a workaround for the fact that the terminal understood each line of the text document to be a form field, and so there was some jankiness around adding/removing lines. Scrolling similarly required round trips to the computer.