GUI : GRAPHICAL USER INTERFACE

"Graphical user interfaces" have evolved from nothing within the last ten years or so, to the delight of many and the consternation of a few (who are not necessarily obstructive traditionalists – they include people with disabilities which make it difficult for them to see the full two-dimensional screen, or to manipulate a mouse, for whom the great leap forward to GUIs was in practice a great leap backward). We are stuck with the abbreviation GUI, which probably pleases no one : the traditionalists would prefer to write it in full, while the graphics freaks would rather use an icon.

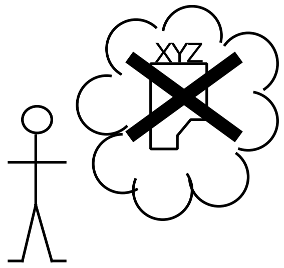
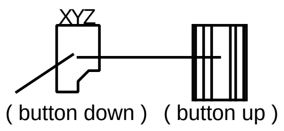
WHAT'S THE DIFFERENCE ?

It is very difficult precisely to identify *the* essential difference by looking at different available GUI systems, because they differ from the older systems in several respects. In the Macintosh system, a completely new interface is put on top of a rather traditional operating system, and you can run MS-DOS or Unix through a GUI if you want. This does emphasise that we are talking of the *interface* only, not the facilities provided by the underlying software.

That in turn has encouraged a conscious separation of systems into basic operating system and user interface, with a clearly defined internal interface between the two. This has a number of advantages : for example, it is easier to adapt the user interface to continuing changes in technology, and it is easier to provide different interfaces for different languages, which has always been a particular problem where the languages use a different character set (Russian), and might not read from left to right (Arabic, Chinese).

THERE IS NO DIFFERENCE.

Of course, that isn't strictly true, though from the computer's point of view it's very nearly true. We do not really intend to deceive you, but it's worth making the point that the essential function of the interface hasn't changed. To continue the theme of languages, a graphical interface is merely one in which we use a graphical language instead of its verbal equivalent. The object of the exercise is to convert an idea in our heads into information in the computer system, or vice versa. In either case, the same problem must be solved : our way of understanding things isn't the same as the computer's (insofar as it can be said to "understand" anything), so there is a significant translation task to be undertaken. Consider this simple example.

Agent	Action	Example	
Person	Think of something to do		
Person	Formalise the instruction	Delete XYZ	
Person	Plan an action, using the vocabulary	"I must type 'DELETE XYZ'."	"I must move the pointer to the XYZ icon, press the button, move to the 'Trash' icon, and release the button."
Person	Issue the interaction	Type : `DELETE XYZ`	

System	Receive the interaction	Sequence of keyboard interrupts and ASCII code characters.	Sequence of mouse movement and button interrupts.
System	Analyse the input	Table look-up for DELETE; file table reference for XYZ.	Sequence of operations identifies type of action; screen map reference to identify XYZ and 'trash'.
System	Synthesise the instruction	It means : "Remove XYZ from file table; Reclaim disc space."	
System	Do it.	Do so.	

Clearly, there are differences in detail between the two methods, but there's no difference in principle. Notice particularly that at the computer end the difference is tiny. The implications for the system are not so tiny, as it's a great deal harder to maintain the screen map than to maintain the file table (and you have to maintain a file table anyway), but once that's done the process is straightforward.

The real difference, and the important difference, is for the people. Unless you have never used anything but a graphics interface, you will be well aware that the experience is very different from that of using a purely textual interface, and that's why the GUI systems have become popular.

WIMP INTERFACES.

It's convenient to discuss the effect of GUI by decomposing another acronym, WIMP, into its components, to see how they (more or less) separately contribute to the results.

Windows share out the screen into areas associated with different items – usually, though not invariably, activities in progress – within the computer. This gives you an easy way to keep track of several things at once, and (in some sense) to shift the terminal from one activity to another by selecting different windows. The particular contribution of the windows is the parallel display. Icons and mice aren't necessary, though you do need some sort of pointer to identify the active window.

You can construct an effective, though possibly limited (because of the 24×80 grid), window system with a character terminal provided that its characters are addressable. (The old "glass teletypes", which only let you write serially to the last line of the display, won't do.) There is debate on whether it's better to let windows overlap, or to keep them all in full view (a tiled screen).

Icons, like windows, associate different areas of the screen with different items, but in this case the items are objects rather than activities. Some signal sent when a pointer is within such an area is interpreted as selecting the associated object. No keyboard skills are required. This works reasonably well with only a few icons, but can become exceedingly confusing if there are many – and, as icons cannot be ordered in any useful sense, there is no general way of reducing the confusion. Other serious criticisms are that they can be hidden under other icons, or off screen, and that it's very hard to make effective new icons.

Icons are based on the principle that a picture is worth a thousand words. Assuming that the principle is true, icons are therefore potentially great time-savers – but only if the thousand words are the words you want. Apart from that, their major attribute seems to be that designing new icons gives GUI programmers something to do when on a real text-based system they would be wasting their time working out stupid acronyms. We will allow that it might be just that we haven't yet met a system where icons are used effectively.

Menus are lists of items, usually, though not necessarily, presented in words, from which one item is to be selected. The principle behind the use of menus is that, at any point in the execution of a programme at which one must make a choice between different ways to proceed, the programme certainly knows what possibilities are available, while the person using it might not. That being so, it is more sensible for the programme to display the list and request a choice than for the person to rely on (possibly inadequately informed) memory.

Pointers are symbols which appear on the computer screen. A pointer is not a part of the information displayed by any of the running programmes; instead, it is displayed by the operating system, and identifies a currently significant position on the screen. Pointers commonly go with **Mice**, or equivalent devices. Generally, a mouse is anything except the standard keyboard that people can use to move a pointer around. A mouse is conventionally equipped with one or more buttons, which can be used to send signals to the computer. No one knows what the "best" number of buttons is. It is fairly difficult to separate mice from pointers, as they're very much a single system – you don't know where to move the mouse if you can't see the pointer. (That's another case where the input and output parts of the terminal must be closely coordinated.)

(Some claim that the M stands for ***Mice*** *– or that the P stands for* ***Pull-down menus****, or* ***Pop-up menus****, depending on their background. Others add an S, standing for* ***Selection****.)*

HOW TO USE WIMP.

Given that machinery, there are lots of ways in which its components could be combined together to make a coherent system, but there does seem to be consensus on how to do it. The screen is divided into a number of windows, each of which is associated with some activity in the computer. Several windows may be associated with one programme; it's up to the programme to make sense of what happens. At any time, some window is active, which means that any keyboard transaction, and any mouse transaction with the pointer lying within the active window, is directed to the programme owning the active window. Other programmes which are running can change their windows without affecting the active window. Merely moving the pointer to lie within another window has no effect on the computer's "real" work, but something has to keep track of the pointer's position. A special signal – invariably (?) a click on a mouse button – is used to switch the active window to the window in which the pointer is lying.

The most important point about that description has nothing to do with any of its details; it is that managing the correct functioning of these facilities is *all* the responsibility of the software which drives the terminal. This brings us back to the user interface management system which we introduced in the chapter *USING TERMINALS*. Provided that a certain, small, number of defined signals can be exchanged with the other programmes using the interface, we can implement this component in any way we like. If someone presents us with a superholographic three-dimensional display, or contrariwise we are reduced to using a set of not-too-clever character terminals, one for each process, we can manage somehow. In other words, we can construct a user interface manager which looks after the screen, and provides a service to programmes which wish to use the screen. Indeed, it is more accurate to say that we *must* provide such a manager, because only in this way can we guarantee that different programmes will use their parts of the screen consistently and without interfering with others.

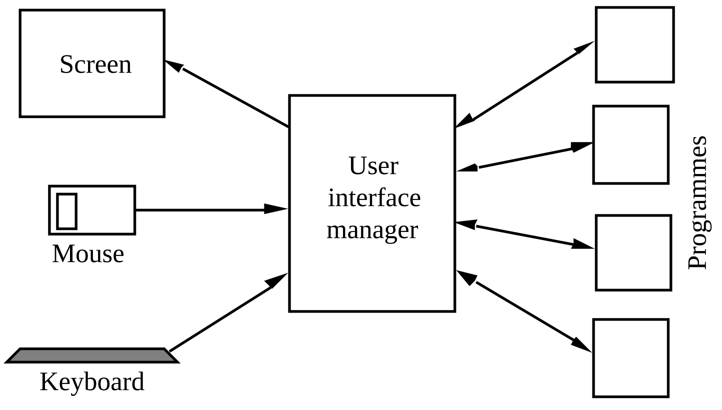
WHAT THE USER INTERFACE MANAGER SHOULD DO.

Like any server, it must provide facilities which help people to do the things they want to do, and also provide safeguards for the system and other people as required. For a single-user machine, the safeguards might be less important; we might like to protect the system itself from harm (or we might take the view that if you programmed it, you deserve the consequences), while on a shared system we might be much more careful. Generally, though, we define a set of reasonable interface transactions which we believe should be available to a programme, and define signals between programme and interface which correspond to these transactions. Typically, we'd provide procedures accessible to the programmes which would look after the details of managing these signals. After that, it's up to us to implement the interface in any way we like.

What are the "reasonable interface transactions" ? They must permit programmes, and the operating system, to display material on the screen, and receive input from the terminal input devices. If we only want to run teletype-like displays, that's probably sufficient; if we want to use graphics, then we'll also need ways of specifying window sizes, and coordinates within windows, and of displaying things at prescribed positions and of reading the position of the pointer. That's about the minimum we need, though there are many other possibilities. For example : what about the window position ? We can either let the programme specify it, or leave it to the interface manager to find a place. Another possibility is provision for input from, and output to, disc files, as we suggested for command files and terminal logs. So far as we know, there is no GUI management system (or graphical UIMS) which makes such provision – but if our arguments are correct, it's only by implementing such a coordinated manager that we will derive the full potential benefit from the GUI scripting languages now appearing. It will be interesting to see what develops. Finally, as a general principle, if we want to make sure that the interface is managed properly, then as much as possible of the work should be done by the interface manager – particularly administrative details, like keeping track of changes to obscured portions of windows and displaying them again when the obscuring object is removed.

The interface must also look after some less obvious tasks, which don't normally concern people writing programmes. It must keep track of the currently active window so that it can send keyboard input to the right activity, and it must keep track of the pointer position so that a mouse button signal can be handled appropriately. To do so, it must receive the signals from the mouse, or other pointer control device, and interpret these appropriately. A traditional mouse or a { trackerball or trackball, depending on where you are } produces a stream of signals each denoting an incremental movement forward, backward, left, or right; joysticks produce a signal which can be interpreted as a direction, and sometimes a speed, of motion; touchpads produce signals which give the current position of contact on the pad. It is obviously not difficult to convert any of these into a common form, which can then be used to adjust the pointer position.

Here we see the terminal as a selection device, as the destination of the signal might be the current active programme, or the operating system. (Recall the dual function of the terminal.) In a multiprogramming system, an operating system transaction might direct a signal to a different programme. We can think of the whole system as an evolved form of the primitive system we proposed earlier; the diagram below illustrates the point.



It might be helpful to summarise all this activity by regarding each window on the screen as a different peripheral device, owned by a single programme; operating systems have been handling such systems for a long time. Then the job of the GUI is to implement all those on a single terminal. Provided that the GUI can provide and accept essentially the sort of signals which would have been used with the set of separate devices, then the system will work. This emphasises the separate nature of the GUI.

It *isn't* the operating system's job to worry about what is in the windows. It's the programme's business to decide what's on the screen, then, once decided, it's the operating system's business to get it there. Similarly, while we want consistency in the interpretation of input to a programme, only the programme knows how to do it, so the operating system is restricted to passing on low level information – "That was a double-click" rather than "That was a select operation". While there really is no hard-and-fast line, the principle is that the operating system should provide general facilities, while the programmes look after specific requirements.

QUESTIONS.

Just how does a GUI make operating system operations easier and faster ? – or harder and slower ? (Ask yourself *why* it works, not just what happens.)
