

# Usability

---

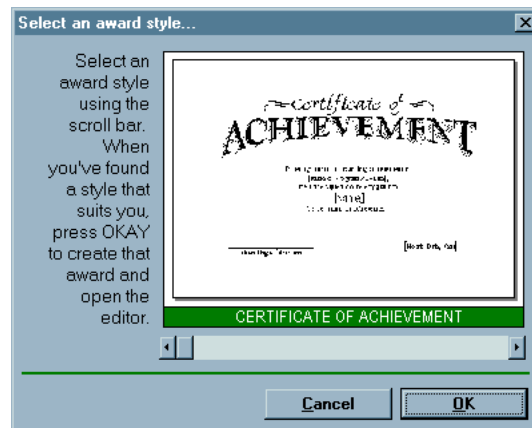
Prof. Rob Miller  
MIT EECS

Spring 2006

6.170 Laboratory in Software Engineering

1

## User Interface Hall of Shame



Source: Interface Hall of Shame

Spring 2006

6.170 Laboratory in Software Engineering

2

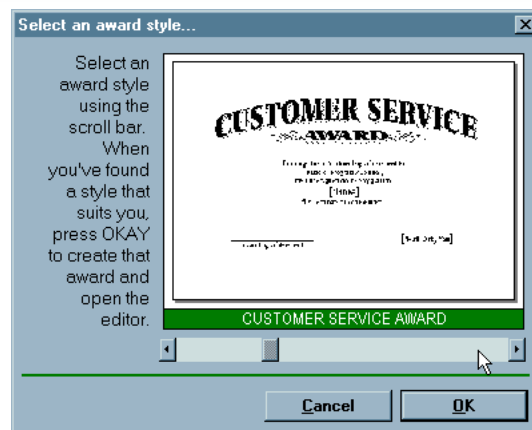
Usability is about creating effective user interfaces (UIs). Slapping a pretty window interface on a program does *not* automatically confer usability on it. This example shows why. This dialog box, which appeared in a program that prints custom award certificates, presents the task of selecting a template for the certificate.

This interface is clearly graphical. It's mouse-driven – no memorizing or typing complicated commands. It's even what-you-see-is-what-you-get (WYSIWYG) – the user gets a preview of the award that will be created. So why isn't it usable?

The first clue that there might be a problem here is the long help message on the left side. Why so much help for a simple selection task? Because the interface is bizarre! The *scrollbar* is used to select an award template. Each position on the scrollbar represents a template, and moving the scrollbar back and forth changes the template shown.

This is a cute but bad use of a scrollbar. Notice that the scrollbar doesn't have any marks on it. How many templates are there? How are they sorted? How far do you have to move the scrollbar to select the next one? You can't even guess from this interface.

## User Interface Hall of Shame



Source: Interface Hall of Shame

Spring 2006

6.170 Laboratory in Software Engineering

3

Normally, a horizontal scrollbar underneath an image (or document, or some other content) is designed for scrolling the content horizontally. A new or infrequent user looking at the window sees the scrollbar, assumes it serves that function, and ignores it. **Inconsistency** with prior experience and other applications tends to trip up new or infrequent users.

Another way to put it is that the horizontal scrollbar is an **affordance** for continuous scrolling, not for discrete selection. We see affordances out in the real world, too; a door knob says “turn me”, a handle says “pull me”. We’ve all seen those apparently-pullable door handles with a little sign that says “Push”; and many of us have had the embarrassing experience of trying to pull on the door before we notice the sign. The help text on this dialog box is filling the same role here.

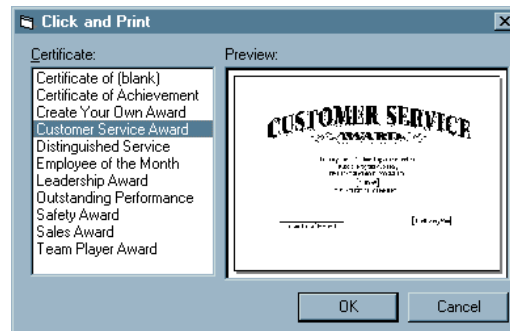
But the dialog doesn’t get any better for frequent users, either. If a frequent user wants a template they’ve used before, how can they find it? Surely they’ll remember that it’s 56% of the way along the scrollbar? This interface provides no **shortcuts** for frequent users. In fact, this interface takes what should be a random access process and transforms it into a linear process. Every user has to look through all the choices, even if they already know which one they want. The computer scientist in you should cringe at that algorithm.

Even the help text has usability problems. “Press OKAY”? Where is that? And why does the message have a ragged left margin? You don’t see ragged left too often in newspapers and magazine layout, and there’s a good reason.

On the plus side, the designer of this dialog box at least recognized that there was a problem – hence the help message. But the help message is indicative of a flawed approach to usability. Usability can’t be left until the end of software development, like package artwork or an installer. It can’t be patched here and there with extra messages or more documentation. It must be part of the process, so that usability bugs can be *fixed*, instead of merely patched.

How could this dialog box be redesigned to solve some of these problems?

## Redesigning the Interface



Source: Interface Hall of Shame

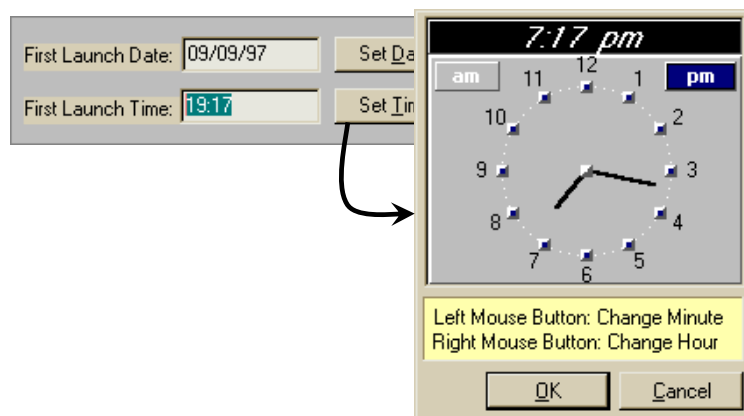
Spring 2006

6.170 Laboratory in Software Engineering

4

Here's one way it might be redesigned. The templates now fill a list box on the left; selecting a template shows its preview on the right. This interface suffers from none of the problems of its predecessor: list boxes clearly afford selection to new or infrequent users; random access is trivial for frequent users. And no help message is needed.

## Another for the Hall of Shame



Source: Interface Hall of Shame

Spring 2006

6.170 Laboratory in Software Engineering

5

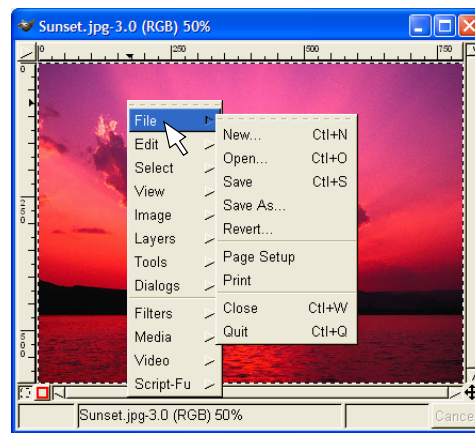
Here's another bizarre interface, taken from a program that launches housekeeping tasks at scheduled intervals. The date and time *look* like editable fields (affordance!), but you can't edit them with the keyboard. Instead, if you want to change the time, you have to click on the Set Time button to bring up a dialog box.

This dialog box displays time differently, using 12-hour time (7:17 pm) where the original dialog used 24-hour time (consistency!). Just to increase the confusion, it also adds a third representation, an analog clock face.

So how is the time actually changed? By clicking mouse buttons: clicking the left mouse button increases the minute by 1 (wrapping around from 59 to 0), and clicking the right mouse button increases the hour. Sound familiar? This designer has managed to turn a sophisticated graphical user interface, full of windows, buttons, and widgets, and controlled by a hundred-key keyboard and two-button mouse, into a **clock radio!**

Perhaps the worst part of this example is that it's not a result of laziness. Somebody went to a lot of effort to draw that clock face with hands. If only they'd spent some of that time thinking about usability instead.

## Hall of Fame or Hall of Shame?



Spring 2006

6.170 Laboratory in Software Engineering

6

Gimp is an open-source image editing program, comparable to Adobe Photoshop. Gimp's designers made a strange choice for its menus. Gimp windows have no menu bar. Instead, all Gimp menus are accessed from a *context menu*, which pops up on right-click.

This is certainly inconsistent with other applications, and new users are likely to stumble trying to find, for example, the File menu, which never appears on a context menu in other applications. (I certainly stumbled as a new user of Gimp.) But Gimp's designers were probably thinking about expert users when they made this decision. A context menu should be faster to invoke, since you don't have to move the mouse up to the menu bar. A context menu can be popped up anywhere. So it should be faster. Right?

Wrong. With Gimp's design, as soon as the mouse hovers over a choice on the context menu (like File or Edit), the submenu immediately pops up to the right. That means, if I want to reach an option on the File menu, I have to move my mouse carefully to the right, staying within the File choice, until it reaches the File submenu. If my mouse ever strays into the Edit item, the File menu I'm aiming for vanishes, replaced by the Edit menu. So if I want to select File/Quit, I can't just drag my mouse in a straight line from File to Quit – I have to drive into the File menu, turn 90 degrees and then drive down to Quit! **Cascading submenus** are actually slower to use than a menu bar.

Gimp's designers made a choice without fully considering how it interacted with human capabilities.

## Hall of Shame or Hall of Fame?

---



Spring 2006

6.170 Laboratory in Software Engineering

7

Finally, we have the much-reviled Paperclip.

Clippy was a well-intentioned effort to solve a real usability problem. Users don't read the manual, don't use the online help, and don't know how to find the answers to their problems. Clippy tries to suggest answers to the problem it thinks you're having.

Unfortunately it's often wrong, often intrusive, and often annoying. The subjective quality of your interface matters too.

## The User Interface Is Important

---

- User interface strongly affects perception of software
  - Usable software sells better
  - Unusable web sites are abandoned
- Perception is sometimes superficial
  - Users blame themselves for UI failings
  - People who make buying decisions are not always end-users

Spring 2006

6.170 Laboratory in Software Engineering

8

So what? Why should we care about usability? After all, human beings are capable of extraordinary learning and adaptation. Even the worst interface can be fixed by a man page, right?

Putting aside the essential inhumanity of this position, there are some practical reasons why we should care about the user interfaces of our software. Usability strongly affects how software is perceived, because the user interface is the means by which the software presents itself to the world. “Ease of use” ratings appear in magazine reviews, affect word-of-mouth recommendations, and influence buying decisions. A Harris poll (reported in the Wall Street Journal 11/8/05) found that ease of use (61%), customer service (58%), and no-hassle installation (57%) are the most important factors US adults consider when purchasing a new technology product.

Usable software sells. Conversely, unusable software doesn’t sell. If a web site is so unusable that shoppers can’t find what they want, or can’t make it through the checkout process, then they will go somewhere else.

Unfortunately, a user’s perception of software usability is often superficial. An attractive user interface may seem “user friendly” even if it’s not really usable. Part of that is because users often blame themselves for errors they make, even if the errors could have been prevented by better interface design. (“Oops, I missed the File menu again! How stupid of me.”) So usability is a little different from other important attributes of software, like reliability, performance, or security. If the program is slow, or crashes, or gets hacked, we know who to blame. If it’s unusable, but not fatally so, the usability problems may go unreported.

## User Interfaces Are Hard to Design

- You are not the user
  - Most software engineering is about communicating with other programmers
  - UI is about communicating with users
- The user is always right
  - Consistent problems are the system's fault
- ...but the user is not always right
  - Users aren't designers

Spring 2006

6.170 Laboratory in Software Engineering

9

Unfortunately, user interfaces are not easy to design. You (the developer) are not a typical user. You know far more about your application than any user will. You can try to imagine being your mother, or your grandma, but it doesn't help much. It's very hard to *forget* things you know.

This is how usability is different from everything else you learn about software engineering. Specifications, assertions, and object models are all about communicating with other *programmers*, who are probably a lot like us. Usability is about communicating with other *users*, who are probably not like us.

The user is always right. Don't blame the user for what goes wrong. If users consistently make mistakes with some part of your interface, take it as a sign that your *interface* is wrong, not that the users are dumb. This lesson can be very hard for a software designer to swallow!

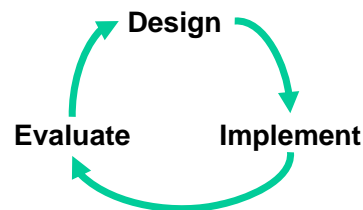
Unfortunately, the user is not always right. Users aren't oracles. They don't always know what they want, or what would help them. In a study conducted in the 1950s, people were asked whether they would prefer lighter telephone handsets, and on average, they said they were happy with the handsets they had (which at the time were made rather heavy for durability). Yet an actual test of telephone handsets, identical except for weight, revealed that people preferred the handsets that were about half the weight that was normal at the time. (Klemmer, *Ergonomics*, Ablex, 1989, pp 197-201).

Users aren't designers, either, and shouldn't be forced to fill that role. It's easy to say, "Yeah, the interface is bad, but users can customize it however they want it." There are two problems with this statement: (1) most users don't, and (2) user customizations may be even worse! One study of command abbreviations found that users made twice as many errors with their *own* command abbreviations than with a carefully-designed set (Grudin & Barnard, "When does an abbreviation become a word?", CHI '85). So customization isn't the silver bullet.

## Iterative Design

---

- UI development is an iterative process



- Do you want the design cycle to be internal, or visible to your customers?

Spring 2006

6.170 Laboratory in Software Engineering

10

So user interface development is inherently **risky**. We don't (yet) have an easy way to predict whether a UI design will succeed.

**Iterative design** offers a way to manage the inherent risk in user interface design. In iterative design, the software is refined by repeated trips around a design cycle: first imagining it (design), then realizing it physically (implementation), then testing it (evaluation).

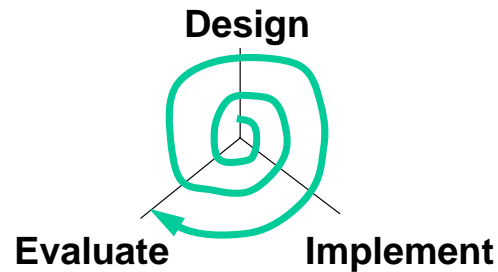
Unfortunately, many commercial UI projects inflict iterative design on their paying customers. They design a bad user interface, implement it, and release it. Evaluation then takes place in the marketplace, as hapless customers buy their product and complain about it. Then they iterate the design process on version 2.

On the other hand, if you keep all your design iterations in-house, you may never release anything! It's very costly to do every iteration of a design with a high-quality implementation in a language like Java or C++ -- especially if you discover you have to throw away all that code because the design was bad.

## Spiral Model

---

- Use throw-away prototypes and cheap evaluation for early iterations



Spring 2006

6.170 Laboratory in Software Engineering

11

The **spiral model** offers a way out of the dilemma. We build room for several iterations into our design process, and we do it by making the early iterations as cheap as possible.

The radial dimension of the spiral model corresponds to the **cost** of the iteration step – or, equivalently, its **fidelity** or **accuracy**. For example, an early implementation might be a paper sketch or mockup. It's low-fidelity, only a pale shadow of what it would look and behave like as interactive software. But it's incredibly cheap to make, and we can evaluate it by showing it to users and asking them questions about it.

## Usability Defined

---

- Usability: how well users can use the system's functionality
- Dimensions of usability
  - Learnability: is it easy to learn?
  - Efficiency: once learned, is it fast to use?
  - Memorability: is it easy to remember what you learned?
  - Errors: are errors few and recoverable?
  - Satisfaction: is it enjoyable to use?

Spring 2006

6.170 Laboratory in Software Engineering

12

The property we're concerned with here, **usability**, is more precise than just how "good" the system is. A system can be good or bad in many ways. If important requirements are unsatisfied by the system, that's probably a deficiency in functionality, not in usability. If the system is very expensive or crashes frequently, those problems certainly detract from the user's experience, but we don't need user testing to tell us that.

More narrowly defined, usability measures how well users can use the system's functionality. Usability has several dimensions: learnability, efficiency, memorability, error rate/severity, and subjective satisfaction.

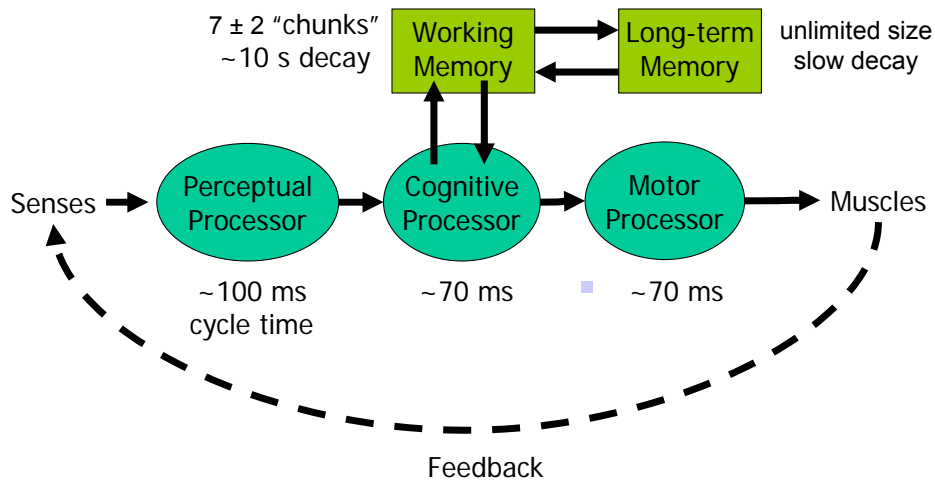
Notice that we can **quantify** all these measures of usability. Just as we can say algorithm X is faster than algorithm Y on some workload, we can say that interface X is more learnable, or more efficient, or more memorable than interface Y for some set of tasks and some class of users.

## Outline

---

- Rest of today: human capabilities
  - Perception
  - Motor skills
  - Memory
  - Color vision
- Next time: usability engineering

## Human Information Processing



Spring 2006

6.170 Laboratory in Software Engineering

14

Just as it helps to understand the properties of the computer system you're programming for – its processor speed, memory size, hard disk, operating system, and the interaction between these components – it's important for us to understand some of the properties of the **human** that we're designing for. Needless to say, there's far more to this topic than we can cover in one lecture, so we'll just hit some highlights that are particularly worth knowing when you're designing a user interface.

Here's a simple abstraction of the human cognitive system. Just as a computer has memory and processor, so does our model of a human. Actually, the model has several different kinds of memory, and several different processors.

The **perceptual processor** takes the sensory input and attempts to recognize things in it: letters, words, phonemes, icons, faces, etc.

The **cognitive processor** makes comparisons and decisions. The cognitive processor does most of the work that we think of as "thinking".

The **motor processor** receives an action from the cognitive processor and instructs the muscles to execute it. There's an implicit **feedback** loop here: the effect of the action (either on the position of your body or on the state of the world) can be observed by your senses, and used to correct the motion in a continuous process.

The model has two memories: a **working memory**, which is small and short-lived, and **long-term memory**, which is huge and shows little decay.

Note that this model isn't meant to reflect the anatomy of your nervous system. There probably isn't a single area in your brain corresponding to the perceptual processor, for example. But it's a useful abstraction nevertheless. One reason it's useful is that we can assign **numerical parameters** to its components, to derive rough-and-ready estimates of how a user might behave.

## Perceptual Fusion

- Two stimuli within the same cycle of the perceptual processor ( $T_p \sim 100\text{ms}$ ) appear **fused**
- Consequences
  - 10 frames/sec is enough to perceive a moving picture
  - Computer response  $< 100\text{ ms}$  feels instantaneous
  - Causality is strongly influenced by fusion

Spring 2006

6.170 Laboratory in Software Engineering

15

One interesting effect of the cycle time of the perceptual processor is **perceptual fusion**. Here's an intuition for how fusion works. Every cycle, the perceptual processor grabs a frame (snaps a picture). Two events occurring less than the cycle time apart are likely to appear in the same frame. If the events are similar – e.g., Mickey Mouse appearing in one position, and then a short time later in another position – then the events tend to *fuse* into a single perceived event – a single Mickey Mouse, in motion.

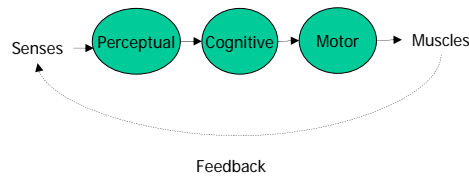
Perceptual fusion is responsible for the way we perceive a sequence of movie frames as a moving picture, so the parameters of the perceptual processor give us a lower bound on the frame rate for believable animation. 10 frames per second is good enough, but 20 frames per second is better ( $T_p$  may be as fast as 50 ms for the quickest humans and for the most favorable conditions).

Perceptual fusion also gives an upper bound on good computer response time. If a computer responds to a user's action within  $T_p$  time, its response feels instantaneous with the action itself. Systems with that kind of response time tend to feel like extensions of the user's body. If you used a text editor that took longer than  $T_p$  response time to display each keystroke, you would notice.

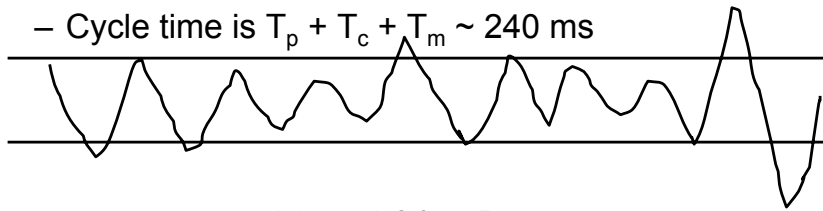
Fusion also strongly affects our perception of causality. If one event is closely followed by another – e.g., pressing a key and seeing a change in the screen – and the interval separating the events is less than  $T_p$ , then we are more inclined to believe that the first event caused the second.

## Motor Processing

- Open-loop control
  - Motor processor runs a program by itself
  - Cycle time is  $T_m \sim 70$  ms



- Closed-loop control
  - Muscle movements (or their effect on the world) are perceived and compared with desired result
  - Cycle time is  $T_p + T_c + T_m \sim 240$  ms



Spring 2006

6.170 Laboratory in Software Engineering

16

The motor processor can operate in two ways. It can run autonomously, repeatedly issuing the same instructions to the muscles. This is “open-loop” control; the motor processor receives no feedback from the perceptual system about whether its instructions are correct. With open loop control, the maximum rate of operation is just  $T_m$ .

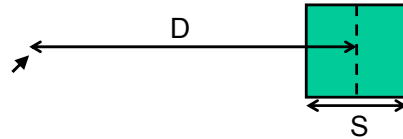
The other way is “closed-loop” control, which has a complete feedback loop. The perceptual system looks at what the motor processor did, and the cognitive system makes a decision about how to correct the movement, and then the motor system issues a new instruction. At best, the feedback loop needs one cycle of each processor to run, or  $T_p + T_c + T_m \sim 240$  ms.

Here’s a simple but interesting experiment that you can try: take a sheet of lined paper and scribble a sawtooth wave back and forth between two lines, going as fast as you can but trying to hit the lines exactly on every peak and trough. Do it for 5 seconds. The frequency of the sawtooth carrier wave is dictated by open-loop control, so you can use it to derive your  $T_m$ . The frequency of the wave’s **envelope**, the corrections you had to make to get your scribble back to the lines, is closed-loop control. You can use that to derive your value of  $T_p + T_c$ .

## Pointing Tasks: Fitts's Law

---

- How long does it take to reach a target?

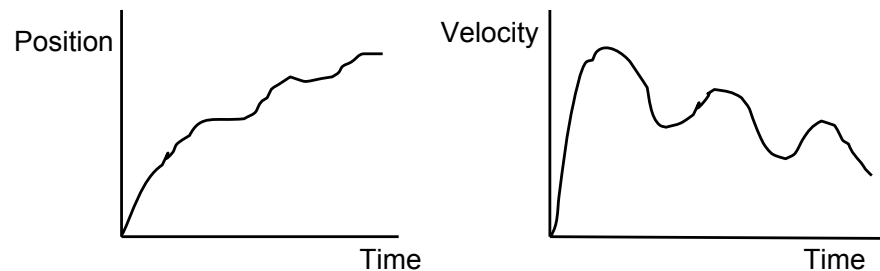


- Moving mouse to target on screen
- Moving finger to key on keyboard
- Moving hand between keyboard and mouse

Let's consider a common motor task in user interfaces: pointing at a target of a certain size at a certain distance away (within arm's length, of course). The time it takes to do this task is governed by a relationship called **Fitts's Law**. It's a fundamental law of the human sensory-motor system, which has been replicated by numerous studies. Fitts's Law applies equally well to using a mouse to point at a target on a screen, putting your finger on a keyboard key, or moving your hand between keyboard and mouse.

## Analytical Derivation of Fitts's Law

- Moving your hand to a target is closed-loop control
- Each cycle covers remaining distance  $D$  with error  $\epsilon D$



Spring 2006

6.170 Laboratory in Software Engineering

18

We can derive Fitts's Law by appealing to the human information processing model. Fitts's Law relies on closed-loop control. In each cycle, your motor system instructs your hand to move the entire remaining distance  $D$ . The accuracy of that motion is proportional to the distance moved, so your hand gets within some error  $\epsilon D$  of the target (possibly undershooting, possibly overshooting). Your perceptual and cognitive processors perceive where your hand arrived and compare it to the target, and then your motor system issues a correction to move the remaining distance  $\epsilon D$  – which it does, but again with proportional error, so your hand is now within  $\epsilon^2 D$ . This process repeats, with the error decreasing geometrically, until  $n$  iterations have brought your hand within the target – i.e.,  $\epsilon^n D \leq S$ . Solving for  $n$ , and letting the total time  $T = n (T_p + T_c + T_m)$ , we get:

$$T = a + b \log (D/S)$$

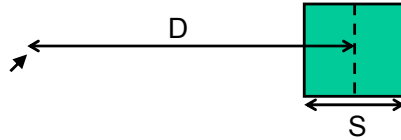
where  $a$  is the reaction time for getting your hand moving, and  $b = - (T_p + T_c + T_m) / \log \epsilon$ .

The graphs above show the typical trajectory of a person's hand, demonstrating this correction cycle in action. The position-time graph shows an alternating sequence of movements and plateaus; each one corresponds to one cycle. The velocity-time graph shows the same effect, and emphasizes that hand velocity of each subsequent cycle is smaller, since the motor processor must achieve more precision on each iteration.

## Fitts's Law

---

- $T = RT + MT = a + b \log (D/S)$



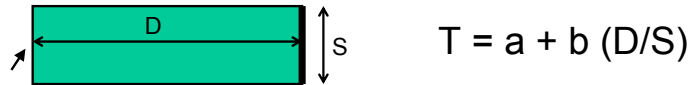
- $\log(D/S)$  is the *index of difficulty* of the pointing task

Fitts's Law has some interesting implications. The edge of the screen stops the mouse pointer, so you don't need a correcting cycle to hit it. Essentially, the edge of the screen acts like a target with *infinite* size. So edge-of-screen real estate is precious. The Macintosh menu bar, positioned at the top of the screen, is faster to use than a Windows menu bar (which, even when a window is maximized, is displaced by the title bar).

So if you put controls at the edges of the screen, they should be active all the way to the edge to take advantage of this effect. Don't put an unclickable margin beside them.

## Path Steering Tasks

- Fitts's Law applies only if path to target is **unconstrained**
- But the task is much harder if path is constrained to a tunnel



- This is why cascading menus are slow!

Spring 2006

6.170 Laboratory in Software Engineering

20

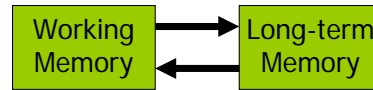
We can also see why cascading submenus like Gimp's are hard to use, because of the correction cycles the user is forced to spend getting the mouse pointer carefully over into the submenu. Because the user must keep the mouse inside the menu **tunnel**, they must move it slowly enough so that the error of each cycle ( $\epsilon d$  where  $d$  is the distance moved in that cycle) is always less than  $S$ . Thus the distance of each cycle is  $d \leq S/\epsilon$ , and so the total number of cycles is proportional to  $D/S$ . That's a lot slower than the  $\log(D/S)$  in Fitts's Law, which applies to unconstrained pointing – it's **exponentially** slower!

Gimp offers the worst possible behavior here, by making the submenu disappear as soon as the mouse pointer exits the tunnel. Microsoft Windows does it a little better – you have to hover over a choice for about half a second before the submenu appears, so if you veer off course briefly, you won't lose your target. But now we know a reason that this solution isn't ideal: it exceeds  $T_p$ , so it destroys perceptual fusion and our sense of causality. And you still have to make that right-angle turn to get into the menu.

Apple Macintosh does even better: when a submenu opens, there's a triangular zone, spreading from the mouse to the submenu, in which the mouse pointer can move without losing the submenu. The user can point straight to the submenu without unusual corrections, and without even noticing that there might be a problem.

## Memory

- Working memory
  - Small:  $7 \pm 2$  "chunks"
  - Short-lived: ~10 sec
  - Maintenance rehearsal fends off decay (but costs attention)
- Long-term memory
  - Practically infinite in size and duration
  - Elaborative rehearsal transfers chunks to long-term memory



Spring 2006

6.170 Laboratory in Software Engineering

21

Working memory is where you do your conscious thinking. In terms of the MHP, working memory is where the cognitive processor gets its operands and drops its results. The currently favored model in cognitive science holds that working memory is not actually a separate place in the brain, but rather a pattern of **activation** of elements in the long-term memory.

A famous result is that the capacity of working memory is roughly  $7 \pm 2$  things. That's pretty small! A good interface won't put heavy demands on the user's working memory.

Working memory decays in tens of seconds. Maintenance rehearsal – repeating the items to yourself – fends off this decay, but maintenance rehearsal requires attention. So if distractions can destroy working memory.

Long-term memory is probably the least understood part of human cognition. It contains the mass of our memories. Its capacity is huge, and it exhibits little decay. Long-term memories are apparently not intentionally erased; they just become inaccessible.

Maintenance rehearsal (repetition) appears to be useless for moving information into into long-term memory. Instead, the mechanism seems to be **elaborative rehearsal**, which seeks to make connections with existing chunks. Elaborative rehearsal lies behind the power of mnemonic techniques like associating things you need to remember with familiar places, like rooms in your childhood home. Elaborative rehearsal requires attention resources too.

## Chunking

---

- “Chunk” = unit of perception or memory
- Chunking depends on presentation and existing knowledge

M W R C A A O L I B M F B I B

MWR CAA OLI BMF BIB

BMW RCA AOL IBM FBI

- 3-4 digit chunking is ideal for encoding unrelated digits

The elements of perception and memory are called **chunks**. In one sense, chunks are defined symbols; in another sense, a chunk represents the activation of past experience.

Our ability to form chunks in working memory depends strongly on how the information is presented – a sequence of individual letters tend to be chunked as letters, but a sequence of three-letter groups tend to be chunked as groups. It also depends on what we already know. If the three letter groups are well-known TLAs (three-letter acronyms) with well-established chunks in long-term memory, we are better able to retain them in working memory.

Chunking is illustrated well by a famous study of chess players. Novices and chess masters were asked to study chess board configurations and recreate them from memory. The novices could only remember the positions of a few pieces. Masters, on the other hand, could remember entire boards, but only when the pieces were arranged in *legal* configurations. When the pieces were arranged randomly, masters were no better than novices. The ability of a master to remember board configurations derives from their ability to **chunk** the board, recognizing patterns from their past experience of playing and studying games.

## Color Blindness

---

- 8% of males can't distinguish red-green



normal vision

red-green deficient

- Blue-yellow color blindness also exists, but is rarer

Spring 2006

6.170 Laboratory in Software Engineering

23

Color deficiency (“color blindness”) affects a significant fraction of human beings. An overwhelming number of them are male. Since color blindness affects so many people, it is essential to take it into account when you are deciding how to use color in a user interface. Don’t depend solely on color distinctions, particularly red-green distinctions, for conveying information. Microsoft Office applications fail in this respect: red wavy underlines indicate spelling errors, while identical green wavy underlines indicate grammar errors.

Traffic lights are another source of problems. How do red-green color-blind people know whether the light is green or red? Fortunately, for US traffic lights, there’s a spatial cue: red is always above (or to the left of) green. For some kinds of red-green color deficiency, the red light also looks darker than the green light.

## Chromatic Aberration

---

- Lens can't focus blue and red at same time
- So blue-on-red text looks fuzzy and hurts to read



Spring 2006

6.170 Laboratory in Software Engineering

24

Chromatic aberration is another important problem. The refractive index of your eye's lens varies with the wavelength of the light passing through it; just like a prism, different wavelengths are bent at different angles. So your eye needs to focus differently on red features than it does on blue features.

As a result, an edge between widely-separated wavelengths – like blue and red – simply can't be focused. It always looks a little fuzzy. So blue-on-red or red-on-blue text is painful to read, and should be avoided at all costs.

Apple's ForceQuit tool in Mac OS X, which allows users to shut down misbehaving applications, unfortunately falls into this trap. In its dialog, unresponding applications are helpfully displayed in red. But the selection is a blue highlight. The result is incredibly hard to read.

## Next Time: Usability Engineering

---

- Design heuristics
- Low-fidelity prototypes
- User testing