Design creativity with maths

Gresham Geometry Lecture -- 26 September, 2002

Harold Thimbleby

Introduction to the 2002/3 Geometry Lecture Series

Gadgets like mobile phones and car navigation systems are often difficult to use. Most of the time we cope, trying to ignore their more confusing and specialised features. Yet occasionally, in some situations, trying to use a complex system can be dangerous and expensive, not just tedious.

A traveller trying to use an automatic ticket machine has to find their destination, ticket class and type of ticket, enter cash, confirm, collect the tickets and change -- all under the pressure of having to catch the train on time as well. People under these circumstances make predictable mistakes, such as leaving their change behind. Such errors can be fixed by changing the design.

Trying to use a mobile phone, or even the radio or navigation system, while driving a car can be so distracting to be dangerous. Ideas like voice control are not going to change the underlying complexity, as anyone who has been frustrated by telephone voice menus will know! Deeper ideas are needed.

A nurse using a syringe pump to provide automatic drug injections is under extreme pressure to "do the right thing" in a distracting environment, yet errors can have unfortunate consequences for patients. Errors caused through ignorance about how to use the equipment may be reduced by better training, but skill-based errors can only be reduced by better design. In fact, most user ignorance can be better dispelled by simplifying designs than by more training!

The 2002/3 Geometry lectures will explore the underlying theories of system design, so that we can see why things are difficult to use, and how they can be made better and easier to use. The slogan for the series of lectures is "*design creativity with maths*."

The maths we cover is simple stuff and surprisingly effective in leading to improved designs, thus helping make systems much easier and more reliable to use. I am not aware that these important mathematical techniques are used by industry, but I hope these lectures will show how easy and useful they are.

The first lecture introduces graphs, mathematical objects that are basically just dots and arrows (and therefore easy to draw and understand). We can imagine web sites to be graphs: the web pages are the dots, and the links between the pages are the arrows. We can also imagine devices like mobile phones to be graphs. Immediately, any phone is like a web site. Which means, more constructively, that we can simulate and test one on the web very easily -- or we can write its user manual as a web site. Such a manual would be complete and correct -- unlike most real mobile phone manuals! We can do lots of other things with graphs, like measuring how long it takes to get across them, and this gives insights into design -- how long would a user take, and how can we redesign things to be easier?

Later lectures will look at how matrices can be used, how codes (e.g., the maths of codes like Morse code) can be used, and how symmetry can be used. The lectures will give plenty of examples that out-perform proprietary products.

Because of their creative and practical design element, all the lectures will appeal strongly to designers and manufacturers, particularly those designing or making highly interactive products such as mobile phones, car radios, and even aircraft cockpits. But the lectures will also appeal to the rest of us, everyday device users -- we who have to put up with using ticket machines, photocopiers and mobile phones... In all lectures, the maths is not difficult and you will go away able to do some device design or analysis of your own. You'll see why these things are hard to use, and you'll wonder why industry does not use maths to make the world easier for us.

Summary of the lecture demonstrations

The first lecture showed how devices (such as mobile phones, video recorderrs, and even games) can be simulated on a computer, and showed how various sorts of manual and description can be generated automatically, for instance to provide a web site documenting how to work the devices. In turn, these sorts of description can be converted into complete and correct user manuals. We also showed how code, as device manufacturers might need to actually build a real, fully-working device can be generated automatically.

For technical people who missed the lecture: the demonstrations showed several finite state automata, how they could be modified and simulated, how their structure allows usability questions to be answered (as well as intelligent help provided), and how they can be used to generate HTML for web-based manuals or Javascript for executable implementations. The FSAs were defined in *Mathematica* code, but the simulations were done in a program I wrote (which parsed the *Mathematica*).

The lecture will be recorded on the Gresham College web site, and you will be able to get the video demonstrations from http://www.gresham.ac.uk under the Geometry lectures.

Overview of these notes

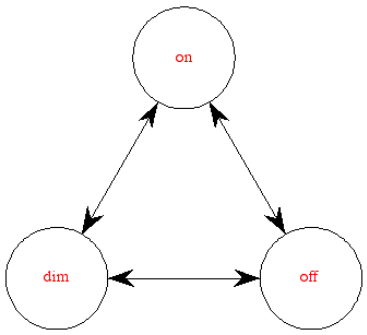
These notes show how *Mathematica* can analyse descriptions of interactive devices, by writing text, drawing pictures, and by doing numerical analyses. *Mathematica* is a sort of mathematician's word processor: all the text and pictures here were created in it, and *all* the information shown about devices, whether pictorial, numerical or textual, was worked out by *Mathematica* from definitions of the devices. The results have not been touched up: everything is automatic, and the same sorts of results could be worked out for other devices as desired -- for instance, if a new device was being designed. The descriptions of devices used here are ones that we demonstrated working in the lecture. Exactly the same definitions can also be used for generating user manuals, intelligent help or for building complete systems.

These brief notes don't exhaust all the possibilities, of course.

Since some people reading these notes will not be interested in any details of how *Mathematica* works; the *Mathematica* instructions themselves have not been printed. (Anyone who wants the *Mathematica* code can email Harold Thimbleby for it.)

A very simple device

The first, and simplest, device we'll consider is a simple light bulb, with three states: off, dim and fully on. This can be drawn as a 'transition diagram' with three circles, one for each of the states, and with arrows between them showing how one could change the state. You can think of the diagram as a game board: when you press a button, you move along the right arrow to a new circle (for clarity I haven't written down arrow names). As it happens, this bulb allows any state to go to any other state directly, so every line is a double-headed arrow -- but this is rarely the case with more complex devices.



Most device descriptions are quite big, but the light bulb is simple enough so that we can show it in its entirety. You can see below how *Mathematica* has got names for the states, descriptions of how it works, and how to draw it on screen for working simulations.

```
modelType → Light bulb
indicatorLoc → {120, 45, 218, 70}
buttonLocs → {{170, 131}, {169, 85}, {170, 108}}
buttons → {Off, On, Dim}
labels → {dim, off, on}
fsm → {{2, 3, 0}, {0, 3, 1}, {2, 0, 1}}
indicators → {{Dim}, {Off}, {On}}
startState → 2
manualRange → 3
manual[1] → {in dim light, in the dark, in bright light}
manual[2] → {are in dim light, are in the dark, are in bright light}
manual[3] → {{dim}, {dark}, {bright}}
notes → {Null, Simple light bulb, with dim mode, Null}
action → press pressed pressing
beeping → never
positions → 0,0,0,0
bulbpicture.eps
Graph → -Graph:<6, 3, Directed>-
```

Running an animation of a device

Mathematica can take the sort of description shown above and create an animation. The animation works, just like the device is intended to. We could use it to test the device out on users, or to see whether it works as we intended. *Mathematica* is fully programmable and all sorts of realistic features could be provided. Here, we just give a simple animation that shows a row of buttons underneath an indicator panel. For fun, the indicator panel is in green, rather like a LED display.

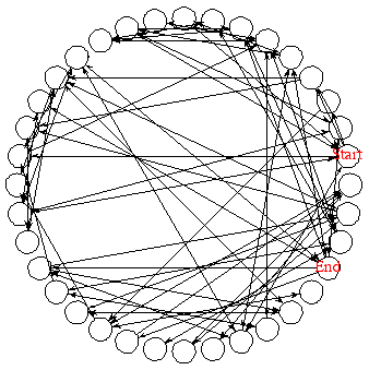
Dim

Off On Dim

At least if you were reading this text in a *Mathematica* document, *Mathematica* could run the simulation. You can click the buttons and they work, which *Mathematica* shows by changing the text in the display above. The light bulb picture happens to be a static picture, but if we wanted to spend more time programming *Mathematica* we could get it animated too. On paper, of course nothing will work!

The wolf, goat and cabbage problem

In the lecture, one of the fun examples was the "wolf, goat and cabbage" problem. The problem requires that the goat is never left alone with the cabbage, and for the wolf never to be left alone with the goat -- in either case, something will get eated! Like a device, it has states (which correspond to various combinations of wolf, goat and so on being on each side of the river) and it has actions (which correspond to the canoe carrying one or more things across the river). We first show this problem as a simple graph:



In this diagram, the problem has effectively been changed to finding a route, following arrows from one circle to another, starting at the circle labelled "Start" and going on to the finish of the game at the state "End".

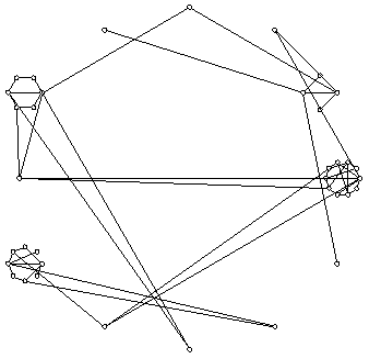
There are some sets of circles which you can get to, and once there you can move around in the set freely, but if you take a way out of the set you can't get back: you can get stuck if you make wrong decisions and get into one of these sets. The sets of circles are called *strongly connected components*, and *Mathematica* can easily find them. The problem has 12 strongly connected components, some with 4 states, some with 8, and quite a few (8, in fact) with only 1 state.

One component is when the cabbage and goat have both been eaten. The component is everything that can be done transporting the wolf or the man alone across the river, but from any of these states it isn't possible to go back to a state where the eaten cabbage or goat exists again. Another component is when you have everything. Another strongly connected component includes the start state, and every state with all objects -- cabbage, goat and wolf -- present one side or the other of the river. Since, by symmetry, getting the objects to one side of the river is the same as getting them to the other side, the start and end states must be in the same strongly connected component.

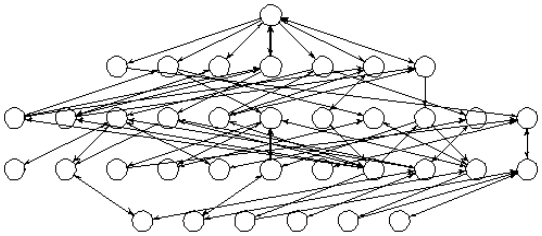
Below, we've used *Mathematica* to summarise the four states of one of the strongly connected components, this one with only the man and wolf present.

```
man ~~~~~~ wolf
~~~~~~ man, wolf
wolf ~~~~~~ man
man, wolf ~~~~~~
```

It's fun to get *Mathematica* to draw a transition diagram with the strongly connected components pulled apart to make them clearer:

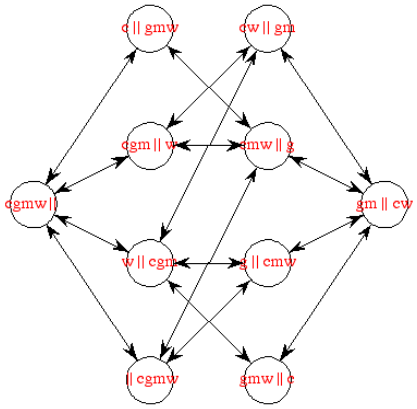


While interesting, this drawing probably doesn't help solve the problem. I've redrawn the same graph in a different way below. Each row in this new diagram of states is the same distance from the start state, which is at the top: so, for example, the second row shows all 7 possible states that can be reached in one canoe trip across the river.



Whereas the previous diagram didn't really help us, in this one an optimal solution to the problem is represented as a path down the diagram that only goes only from top to bottom: if it ever turned back upwards (or even went left or right without going down) in the diagram, it would be going to a state that would have been easier to reach on a shorter route more directly from the top. We'll use exactly the same form of graph drawing to look at a video recorder later in these notes: again, the diagram allows one to read off good ways of using a device -- and, conversely, the shape of the diagram gives a designer a good idea of how efficiently users can work the device. (Because we've drawn a rather small, dense graph, there isn't enough space to show the names of the state of the wolf, goat and cabbage in each state.)

The problem is tricky because sometimes cabbages or goats get eaten, and there is then no going back. In the diagram, some arrows are one way: if a canoe trip is taken that corresponds to one of these arrows, it is a one way trip in terms of the states that can be reached. If the cabbage gets eaten by the goat, no states with cabbages are accessible any longer. (If something like a DVD player was like this, it would be very tedious to use! Of course, video recorders are sometimes like this when you accidentally record over a favourite programme.) Well, we can use *Mathematica* to automatically find the states that cannot be got out of, delete them, and hence make a simpler version of the problem where nothing can go wrong. We make a new set of easy states, and then draw the new graph. This one has a pleasing, elegant structure: and we have derived it completely automatically, simply by deleting the nasty states in the original problem. All we have to do is define what we mean by nasty (in this case, delete all states that are not in the strongly connected component containing the start state) and then run *Mathematica* to fix the design. One might want to do similar things with actual devices: make it impossible for the user to get stuck with them. If we decide what criteria we want, we can redesign automatically.



The big wolf, goat, cabbage graph was a bit obscure without any state names, so we've defined some names for the states. The states are named symbolically, so that they are easier to understand: thus, "cwm||" with nothing after the || river symbol means that the cabbage, wolf and man are all together on the left side of the river -- and the goat must have been eaten, since it is on neither side of the river! As *Mathematica* arranged this diagram, the further right one goes the harder it is to get there from the starting point, with everything on the left of the river. Interestingly, the goal of the problem, with everything on the right of the river is the hardest state to get to: it's shown on the far right.

A JVC video recorder

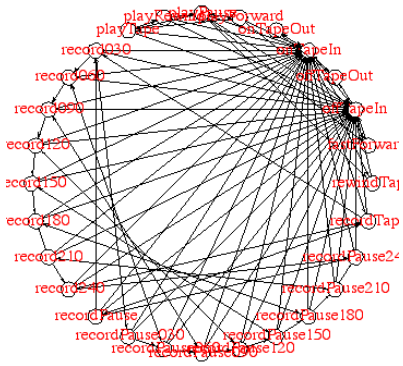
Now we look at a JVC video recorder, the JVC HR-D540EK.

What would this device look like? As with the light bulb, we can ask *Mathematica* to make a simulation that can be used.

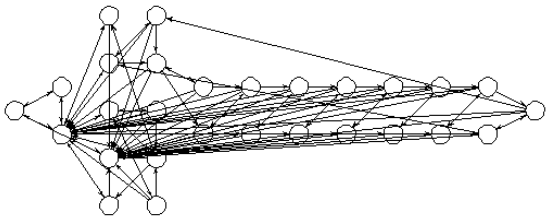
Play | Operate | Forward | Rewind | Pause | Record | Stop / Eject | Tape in

At least the photograph is more realistic than the light bulb's! The big black bar would show the video recorder's state, but as it has started off, nothing is lit up to see.

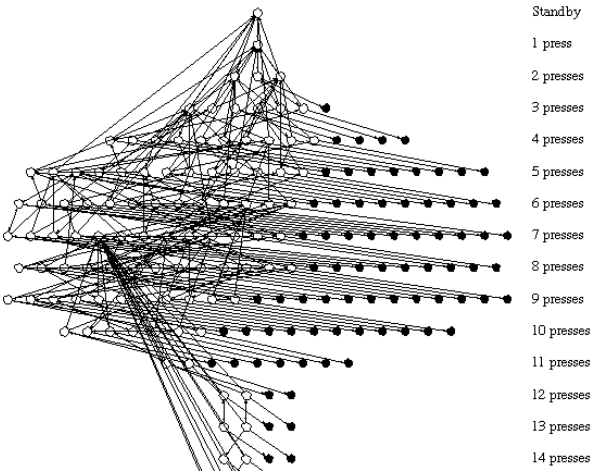
We next show this device as a circular graph. The advantage of a circular embedding is that no two lines are ever drawn on top of each other, so we can be certain that we are looking at everything. It helps if we write the names of the states too, but there are so many that the drawing gets quite messy! Even though the diagram is messy (and we haven't put the states in a useful order around the perimeter), you can see little facts like the two states *on-with-tape-in* and *off-with-tape-in* are very easy to get to from almost anywhere -- you can clearly see the cluster of arrow heads hitting each of these states from almost every other state.

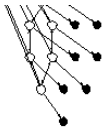


An alternative way of drawing the graph (like one way we drew the wolf, goat cabbage problem) is to 'rank' the states, so that the machine off (with its tape out) is shown at the far left, and each column of states is the same distance from off. Thus the further right you go in the drawing, the harder things are to do -- at least if you start from off. The long tail of states, each increasingly hard to get to, makes this JVC device hard to use. (We've drawn the graph going left-to-right, rather than top-to-bottom as with the wolf problem, because it fits on paper better this way.)



Below is a similar graph (now rotated to go top down, rather than right to left) for a Nokia mobile phone, the 5110.





It's interesting to look at the average length of best paths between any two states. (For the lecture notes, we've hidden the simple *Mathematica* code that works out this number; we've just shown the result.)

```
3.86905
```

This means that if you know how to use this device perfectly (and few people do!), on average to do anything will take you almost 4 button presses. With such bad averages, it's interesting to know what the hardest operations are:

```
fast forward → pause recording, but stop in 240 minutes
off, with tape in → pause recording, but stop in 240 minutes
off, with tape out → pause recording, but stop in 240 minutes
on, with no tape → pause recording, but stop in 240 minutes
play a tape fast forward → pause recording, but stop in 240 minutes
pause playing a tape → pause recording, but stop in 240 minutes
play a tape fast backward → pause recording, but stop in 240 minutes
play a tape → pause recording, but stop in 240 minutes
rewind a tape → pause recording, but stop in 240 minutes
```

So, the hardest operations are all ones that end up with the video recorder doing something in 240 minutes -- in fact, the state at the extreme right of the last diagram we drew.

The JVC has 8 buttons and 28 states, so if the buttons were used optimally to make doing anything as brief as possible, the average would be less, namely about $\log_8 28$, which is about 1.6. This is a lot less than what the 3.8 the JVC achieves. Another way of calculating this is to see that 1 state (namely, the one you start from) can be reached with zero button presses; 8 states can be reached with 1 press, which leaves 19 states that can be reached in 2 presses. The exact average cost is therefore $(0\times1+1\times8+2\times19)/28=1.64$. Of course, we might get some data that says users tend to do some things more often than others, and we ought to weight these actions more than ones that aren't used much -- we'll look at these sorts of "real world" issues in the second lecture in the series.

```
1.60245
```

We can conclude the JVC was not designed to minimise button presses to do things, regardless of other concerns. One would therefore have expected some other advantage for the JVC design decisions, such as the buttons more often meaning the same things, like `[PLAY]` always meaning play. Let's check this idea out next.

Some buttons have names like `[OPERATE]` and `[PLAY]` that seem to have helpful names. We can look at the design of the system and find out how likely buttons are to do things. We've taken a very simple approach here, but we can see for example, that the `[OPERATE]` button makes the JVC device on 44% of the time -- other times, most of the time, `[OPERATE]` makes the device *in*operative (off)!

For the JVC HR - D540EK VCR

`Play` when it does something, always achieves: {on, tape in}

`Operate` when it does something, mostly achieves: on (44.444444444444443 % of the time)

`Forward` when it does something, always achieves: {fast forward, on, tape in}

`Rewind` when it does something, always achieves: {on, rewind, tape in}

`Pause` when it does something, always achieves: {on, pause, tape in}

`Record` when it does something, always achieves: {on, tape in}

`Stop / Eject` when it does something, mostly achieves: on (44.444444444444443 % of the time)

`Tape in` when it does something, always achieves: {on, tape in}

What this means is that when the `[PLAY]` button does something it will leave the video with the on and tape in lights on. (Of course, if the video was off, `[PLAY]` would do nothing.) That's not very surprising, but some of the other buttons' meanings are.

When buttons are pressed on a device, it should give feedback that something has happened. Do some buttons not give decent feedback? The 'ambiguous' actions are shown below; for instance if the JVC model is on with a tape in and you make it go fast forward, it won't tell you anything has happened.

```
on, with tape in -> fast forward
fast forward -> on, with tape in
play a tape fast forward -> on, with tape in
play a tape fast backward -> on, with tape in
rewind a tape -> on, with tape in
on, with tape in -> rewind a tape
```

It looks like the video should have had indicators for fast forward and rewind states. It doesn't.

Rather than carry on writing special *Mathematica* code for each idea we have, we'll write a single function that prints out some interesting facts about any device.

Here is the information for the simple light bulb.The first time we use the *Mathematica* function, we'll ask it to explain what everything means, but to save space below we won't print this explanation again.

```
Model: Light bulb
   -- the model type.

Number of states 3
   -- how many things can be done with this device?

Number of edges 6, which is 100.% complete
   -- In a complete graph, you can do anything in one step, so if this figure is 100%, the device cannot be made faster to use.

Probability a button does nothing 0.333333 = 33.3333% of the time
   -- chance a random button press does nothing.

Average recovery overhead for a random press 0.666667 (excluding the press)
   -- if you make a random press, how hard on average is it to get back? Compare this figure with the mean cost.

Average cost to get anywhere after a random press 0.666667 (excluding the press)
   -- Your random press can give you a bit more information, but has it made your task harder? Compare this figure with the mean cost between states.

Average number of button presses to get from anywhere to anywhere 1.
   -- the average cost of doing anything from anywhere, the mean cost (taking everything as equally likely).

Number of single button presses with direct (length 1) undo 6 = 100.%
```

```
   -- how often, if you make a mistake, can it be undone directly with one button press?

Average undo cost for single button press 1.
   -- If you make a single button press mistake, on average what does it cost to recover?

Over-run errors do not happen on this device.
   -- An over-run error occurs when a button is pressed once too often and goes to another state. On this device, pressing a button twice always leaves you in
```

Over-run errors don't really happen on light bulbs, or at least not this one. Imagine a typical gadget with nasty rubber keys that you aren't sure you've pressed hard enough. Suppose you press `[OFF]` and the device has not gone off. Maybe the device is slow; maybe the lights tend to stay on for a bit; or maybe you didn't press `[OFF]` hard enough, and until you press it properly it isn't going to switch off. So you press it again. If, in fact, this is the second press of `[OFF]`, we will call it an over-run error. Maybe an over-run of `[OFF]` will switch this device back on again? On our simple light bulb, since `[OFF]` only switches the bulb off, switching it off when it is off keeps it off -- and the same for all the other buttons. Pressing `[DIM]` when the bulb is dim keeps it dim; pressing it twice still keeps it dim. Pressing `[ON]` when the bulb is on keeps it on; pressing it twice still keeps it on. Hence the summary information above says over-run errors do not happen.

As we can see below, the JVC device has some curious properties. If we have an over-run error (e.g., we wanted to get to the video to play a tape, but we pressed `[PLAY]` once too often, perhaps because we didn't notice when the device got where we wanted it -- perhaps it is too slow or doesn't provide decent feedback), then on the JVC it takes 2.3 presses on average to get back to where we wanted to be (or 3.3 including the error). On the other hand, to get from anywhere to anywhere takes on average 3.9 presses: so an over-run error is practically the same as getting completely lost -- an over-run error puts you about as far away on average from where you want to be as you can be. On the other hand (we have 3 hands?!), if you make a completely random button press, it only takes 1.8 presses to recover (on average) -- or 2.8 including the error. But this is *easier* than an over-run error! There are three main reasons for this: (*i*) some random presses do nothing, and therefore cost nothing to recover from; (*ii*) most random presses don't get you as far away as an over-run; (*iii*) if a button worked to get you to this state, it is likely to work to get you away from it (in other words, over-run errors are likely).

```
Model: JVC HR-D540EK VCR
Number of states 28
Number of edges 106, which is 14.0212% complete
Probability a button does nothing 0.526786 = 52.6786% of the time
Average recovery overhead for a random press 1.78125 (excluding the press)
Average cost to get anywhere after a random press 3.91964 (excluding the press)
Average number of button presses to get from anywhere to anywhere 3.86905
Number of single button presses with direct (length 1) undo 38 = 35.8491%
Average undo cost for single button press 3.76415
Average recovery length from single over-run 2.30986
Maximum over-run length length 9
```

The joke about three hands, feeble as it was, reminds me that the remote control for this video recorder is *completely* different from the unit itself. We haven't space to show it here, but it's very obvious from any drawing of the transition diagram. Making it different doubles the learning the user has to do to make good use of the device, and almost doubles the size of the user manual.

If you make a random press you may find out more about the device. It's a tempting thing to do: you walk up to something. What does it do? The only way to find out is to press a button and see what happens. On the JVC, if you press a button at random you may have made it harder (by a bit) to get anywhere. This is because sometimes it is much harder to get back to where you were. But the difference isn't much, and if you can find out something useful about where you are (what the device is doing) by pressing a button, on the JVC this could is a good strategy. Sometimes pressing a button does nothing. For example, on the JVC if you press `[PLAY]` when it is playing, nothing happens. Suppose we modify the JVC so that a user can tell if a button will do something. For example, each button might have a little light that comes on only if the button works. These useful buttons would be easy to find in the dark. Now, if we press a button at random it will always do something. How does this change the numbers?

```
Average cost to get anywhere after a working random press 4.04582 (excluding the press)
   -- Your random press will give you a bit more information, but has it made your task any easier?
```

It's worse! So, on the JVC you're better off not playing with the buttons "to see what they do". But you're only better off if you know what it is doing, and that would require the indicator lights to tell you what it was doing. We've already seen they are inadequate. So, on the JVC the user is in a quandary: you can't always tell what state it is in, and experimenting to find out makes any task harder. Of course, to be fair, once you've experimented and found out where you are, you can now use the JVC properly, which you can't do when you don't know what it is doing.

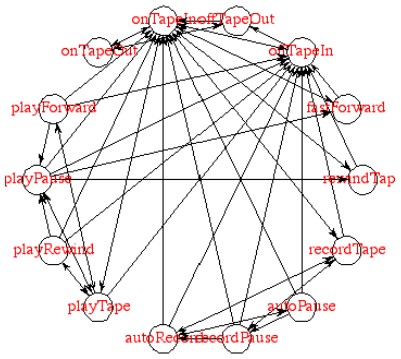
A Philips video recorder

Now compare the JVC with a Philips VCR. The Philips VR502 looks better in this sort of analysis, though it has fewer states (it doesn't have as many features).

```
Model: Philips VR502
Number of states 18
Number of edges 143, which is 46.732% complete
Probability a button does nothing 0.272727 = 27.2727% of the time
Average recovery overhead for a random press 1.46465 (excluding the press)
Average cost to get anywhere after a random press 1.98457 (excluding the press)
Average number of button presses to get from anywhere to anywhere 2.04902
Number of single button presses with direct (length 1) undo 79 = 55.2448%
Average undo cost for single button press 2.01389
Average recovery length from single over-run 2.25
Maximum over-run length length 6
```

Simplifying the JVC video recorder

If the Philips is easier (in the sense we've explored) then can we redesign the JVC to make it easier? We've already pointed out how the 'tail' of states on the JVC make many things harder. Let's delete them and see what happens.




```
Model: Reduced JVC HR-D540EK VCR
Number of states 14
Number of edges 50, which is 27.4725% complete
Probability a button does nothing 0.571429 = 57.1429% of the time
Average recovery overhead for a random press 0.723214 (excluding the press)
Average cost to get anywhere after a random press 1.94133 (excluding the press)
Average number of button presses to get from anywhere to anywhere 2.07692
Number of single button presses with direct (length 1) undo 25 = 50.%
Average undo cost for single button press 1.6875
Average recovery length from single over-run 1.
Maximum over-run length length 1
```

Wolf and goats again

Finally, we look at the wolf, goat and cabbage problem again for comparison. Many of the figures are infinity because once something has gone wrong (e.g., the goat has been eaten) there is nothing you can do: the average of impossible and anything else is still impossible on average! Unlike the interactive devices (like video recorders) we've analysed, pressing buttons at random to try and better understand what's going on does not help.

```
Model: Wolf, goat & cabbage problem
Number of states 36
Number of edges 94, which is 7.46032% complete
Probability a button does nothing 0.626984 = 62.6984% of the time
Average recovery overhead for a random press Infinity (excluding the press)
Average cost to get anywhere after a random press Infinity (excluding the press)
Average number of button presses to get from anywhere to anywhere Infinity
Number of single button presses with direct (length 1) undo 70 = 74.4681%
Average undo cost for single button press Infinity
Average recovery length from single over-run Infinity
Maximum over-run length length Infinity
```

The rather high probability (62%) that a button does nothing really means in this case that if you shut your eyes and wished "I want to take the goat and the wolf across the river" then about 62% of the time you couldn't do it, for instance because the goat was on the other side, or was eaten, or something else had gone wrong you hadn't noticed with your eyes shut.

What about other benefits?

If *Mathematica* can use the definitions to run full interactive simulations of the devices, surely it -- or, rather, mathematics more generally -- can do more? Yes, it can. Later lectures will develop the ideas further, and will show how mathematics can be used constructively to help improve designs. With tools like *Mathematica*, the mathematics becomes quite easy to use; in fact, the mathematics is quite straight forward and could easily be embedded inside familiar design tools. The ideas would then move from the research field into industry.

Conclusions

Mathematics can underpin the design process

Mathematics is very creative and helps improve design

Mathematics is easy to use with the right tools
(doesn't have to be Mathematica!)

Mathematics can be used for simulation, manuals, design and manufacture

---

About the author

Harold Thimbleby is Gresham Professor of Geometry and Director of the UCL Interaction Centre, UCLIC. He is one of the first Royal Society-Wolfson Research Merit Award Holders. More details on his lecture series, plus many background papers and references are available at http://www.uclic.ucl.ac.uk/usr/harold/gresham

Email him at h.thimbleby@ucl.ac.uk

Converted by *Mathematica* (October 3, 2002)