# Step by Step through the "Colouring Cells" Program

## Professor Peter Millican, Hertford College, Oxford

The "Colouring cells" program can be found within the *Turtle System* under the "Examples: user input, interaction and games" sub-menu. It is specially designed to introduce a lot of the concepts that are particularly useful when programming cellular models, focusing particularly on those that are likely to be unfamiliar if you have previously done only traditional "Turtle Graphics" programming.

## The Program, and What it Does

When you first load the program, it will include quite a lot of comments {in curly brackets like this}. If you want to remove the comments, select "Remove any comments" from the "Edit" menu, after which you will see this:

```
PROGRAM ColourCells;
CONST width=10;
      height=15;
VAR x: integer;
    mk: integer;
BEGIN
 canvas(1,1,width,height);
 resolution(width,height);
 for x:=1 to width do
  pixset(x,height,rgb(x));
 repeat
  mk:=detect(\mousekey,5000);
  if mk=1 then
   pixset(?mousex,?mousey,turtc)
  else
  if mk=2 then
   turtc:=pixcol(?mousex,?mousey)
  else
  if mk=4 then
   pixset(?mousex,?mousey,rgb(random(10)+1))
 until mk=\escape
END.
```

When you click on the "RUN" button, you will see a line of ten coloured blocks along the bottom of the Canvas – each of these is in fact a single coloured *pixel*. Now try clicking with the *left* mouse button on some of the white area of the Canvas – each time you do, the relevant pixel will turn black, and this will enable you to see that the entire Canvas has been divided into a grid of 10x15 pixels (with coordinates running from 1 to 10 along the top, and 1 to 15 down the side). Then try clicking with the *right* mouse button on one of the coloured pixels at the bottom – this time, you will see from the "Colour" patch at the right, just above the Canvas, that the *Turtle*'s colour has changed accordingly. And if you now *left-click* elsewhere on the Canvas, you'll see that the relevant pixel now turns to the new colour (rather than black), and you can go on in this way making a coloured pattern on the Canvas. Finally, try clicking with the *middle* mouse button on some pixel. Now, you will find that the pixel turns into one of the first 10 build-in colours, chosen randomly. When you've played with this enough to understand what's happening, press the "Escape" key to finish the program.

## Step by Step through the Program

There is a lot going on in this short program, so let's go in turn through its distinctive features:

```
CONST width=10;
      height=15;
```

- The program starts by defining two *constants*, named "width" and "height". Note carefully how this is done using the CONST keyword. This makes it very easy to change the number of pixels into which the Canvas will be divided – both horizontally and vertically. *Note, however, that any constant definitions have to come at the very beginning of a program, before any variables have been defined*.

```
VAR x: integer;
    mk: integer;
```

- Two integer variables are defined: *x* will be used for counting across the width (i.e. for the relevant cells' *x-coordinate*), and *mk* will be used to identify mouse clicks and key presses.

```
canvas(1,1,width,height);
```

- The command `canvas(1,1,width,height)` defines the Canvas coordinates as starting from the point (1,1) at the top left, with the given width and height. Note for the future, however, that it would be more standard, when creating a cellular automaton, to use `canvas(0,0,width,height)`, so that the x-coordinates run from 0 to width-1, and the y-coordinates from 0 to height-1. (This makes it much easier if the automaton is intended to "wrap around" from bottom to top and right to left.) We'll generally do that in future.

```
resolution(width,height);
```

- The command `resolution(width,height)` defines the Canvas *resolution* – the number of pixels actually used in the Canvas image – as *width* x *height*. In cellular automata, we almost always want the *resolution* dimensions to match the *coordinate* dimensions.

```
for x:=1 to width do
  pixset(x,height,rgb(x));
```

- The command `pixset(x,y,red)` could be used to set the individual pixel at coordinates (*x*,*y*) to the specified colour – in this case, *red*. But *note here that "red" is actually shorthand for a particular colour code number, in fact for 16711680 which is #FF0000 in hexadecimal*.

- The function `rgb(n)` returns the $n^{th}$ built-in colour code: so, for example, `rgb(1)` returns the colour code for green, `rgb(2)` for red, `rgb(3)` for blue, `rgb(4)` for yellow, and so on. Thus the short loop above colours the ten bottom pixels with the first ten build-in colours.

```
repeat
```

- The program now enters a `repeat … until` loop (you'll see the `until` part just above the "END" at the very bottom of the program). This is a very common structure, rather like a FOR loop except that instead of being controlled by the value of a counting variable, it goes round the loop until the final condition (the `until` condition) becomes true. In this case, that condition is *mk*=*escape*, which will become true after the "Escape" key has been pressed.

```
mk:=detect(\mousekey,5000);
```

- This command uses the special `\mousekey` input code – this is one of many that are defined within the system, but it's the only one we need here, and is often the most useful. It signifies detection of *any mouse click or key press* (whichever comes first).
- The command `mk:=detect(\mousekey,5000)` tells the system to wait for up to 5 seconds (i.e. 5,000 milliseconds) to detect either a mouse click or a keypress. If no such event is detected, the variable *mk* will be made equal to 0. If a mouseclick is detected, *mk* will be made equal to 1, 2, or 4 depending on whether it was the left, right or middle mouse button.[1] And if a keypress is detected first, then *mk* will be made equal to a keycode value (the only relevant one here is `\escape`, which happens to be equal to 27).

```
if mk=1 then
 pixset(?mousex,?mousey,turtc)
```

- This deals with a left mouseclick, and here again we see the `pixset` command, but now with parameters `?mousex` and `?mousey` – which give us the coordinates of the pixel where the mouse has been clicked – and `turtc`, which gives us the current *Turtle* colour.

```
else
if mk=2 then
 turtc:=pixcol(?mousex,?mousey)
```

- If the variable *mk* wasn't equal to 1 (i.e. a left mouseclick), then we go on to test for the value 2, which identifies a right mouseclick. But in this case, instead of `pixset`, we see the command `pixcol`, with the now familar parameters `?mousex` and `?mousey` – this returns the colour of the pixel where the mouse has been clicked, and here that colour code is then being assigned to `turtc`, so that the *Turtle* takes on that colour.

```
else
if mk=4 then
 pixset(?mousex,?mousey,rgb(random(10)+1))
```

- Finally, if the variable *mk* was equal to neither 1 nor 2, we get a test for a middle mouseclick, which corresponds to the value 4. This time, the `pixset` command is being used with a random third parameter, to set the pixel where the mouse was clicked to a random choice of the first 10 built-in colours. `random(10)` returns a value between 0 and 9, so we add 1 to get a value between 1 and 10, and then use the `rgb` function to turn that into a colour code.

```
until mk=\escape
```

- We finally get to the end of the `repeat … until` loop, which terminates when the variable *mk* becomes equal to 27 (the numerical value of the code `\escape`), signifying that the "Escape" key has been pressed. At this point, the program ends.

[1] Why are these 1, 2 and 4 rather than 1, 2 and 3? The reason is to do with the bitwise representation of input codes – in various contexts, the left-mouse button is represented by "bit 0", the right-mouse button by "bit 1", and the middle mouse button by "bit 2". The corresponding values are $2^0$, $2^1$, and $2^2$ – i.e. 1, 2, and 4.