Royal Institution Masterclass

Oxford, July 3, 2021

Getting Started with Cellular Automata

1. Familiar *Turtle System* Concepts

From the previous Masterclass worksheet "Spirals and Shapes with Turtle Pascal", you should already be familiar with some fundamental *Turtle System* concepts, including:

- *Turtle Graphics*, invented by Seymour Papert, is a brilliant method for starting programming, based on imagining an invisible *Turtle* that moves around the *Canvas*, drawing as it goes.
- The *Turtle*'s behaviour is defined by a *program*, involving various *keywords* such as PROGRAM, BEGIN, and END, and set running by clicking on the RUN button.
- At any time, the *Turtle* has a particular *position*, recorded by *x* and *y* coordinates (usually on a *Canvas* measuring 1000x1000 units), and a particular *direction*.
- The *Turtle* also has a pen with a particular *colour* and *thickness*, set by the `colour` and `thickness` commands.
- All of these values are shown above the Canvas as the program runs (they are also available as the built-in functions *turtx*, *turty*, *turtd*, *turtc*, and *turtt*).
- The system has 50 named colours (which can be seen in the relevant "QuickHelp 1" panel at the bottom of the Canvas – they start *green*, *red*, *blue*, *yellow*, *violet*, *lime*, *orange*, …).
- Turtle movement and change of direction can be achieved using commands such as `forward`, `back`, `left`, and `right`.
- All of these commands take an *integer* (i.e. whole number) as a *parameter* – for example `forward(300)` or `right(45)` to move forward 300 units or turn right by 45°.
- We can define *variables* that hold values – most often *integer variables*, though they can also be defined to hold *strings* (e.g. words or sentences) and other types (e.g. *Booleans*, *arrays*).
- We can use FOR loops to do things repeatedly, using a variable to keep track of the *iterations* (e.g. "for count := 1 to 100 …" iterates the "count" variable through the values from 1 to 100);
- Shapes can be drawn in various ways, most simply using `circle`, `blot`, `ellipse`, and `ellblot` – these all draw the relevant shape centred on the *Turtle*'s current position.
- We can also draw shapes by moving the *Turtle* through a sequence of points, and then using e.g. `polyline(4)`, or `polygon(4)` to link up the last 4 points in a line or a circuit.
- We can write our own *procedures* – mini-programs with names and parameters that we choose ourselves – which then provide user-defined commands to do particular tasks.
- If we want to use different colours as we iterate some loop, we can use the randomising function `randcol`, e.g. `randcol(20)` randomly chooses from the first 20 system colours.
- We can also generate random behaviour more generally using the `random` function: e.g. `n:=random(1000)` makes variable *n* equal to a random number between 0 and 999.
- The "`if … then …`" or "`if … then … else …`" structure can be used to choose different behaviour depending on the value of some function or variable.
- To treat a number of commands in sequence within a FOR loop or IF structure, use `begin` and `end` to bracket them together – this effectively combines them into a single command.

2. An Illustration, and an Exciting Discovery

Here is an example of a short program that illustrates most of these things – this can be found in the "Examples" menu, under "Examples 2 – procedures and simple recursion": to load it, select "Triangle procedure with limit" from the menu. Note that it includes one new command, `movexy(-100,150)`, which moves the *Turtle* on the *Canvas* 100 units to the left and 150 units down.

```
PROGRAM Triangles;

 Procedure triangle(size: integer);
  Begin
   if size>1 then
    begin
     forward(size);
     right(120);
     forward(size);
     right(120);
     forward(size);
     right(120)
    end
  End;

BEGIN
 movexy(-100,150);
 triangle(256)
END.
```

The exciting discovery comes if we add the surprising command `triangle(size/2)` in turn after each of the `forward(size)` commands (and remember here that you always need a semicolon to separate each pair of successive commands). The resulting program can be found in the same "Examples" menu, labelled "Recursive triangles".

A procedure which "calls itself" in this way is said to be *recursive*, and this is an extremely powerful technique with lots of applications. To see some of the wonderful effects it can produce, you might like to explore some of the early programs in the menu "Examples 9 – self-similarity and chaos".

You could also try adding recursion to programs you have written yourself for the "Spirals and Shapes with Turtle Pascal" worksheet. But if you do this, do notice that *any recursive procedure needs to have a termination condition* – a way of stopping it from recursively calling itself on and on forever. In the example above, this is achieved through the condition:

```
if size>1 then
 begin
  ...
 end
```

which ensures that once *size* reduces to 1, the *triangle* procedure stops doing anything. Thus we progressively get calls of *triangle(256)*, *triangle(128)*, *triangle(64)*, *triangle(32)*, *triangle(16)*, *triangle(8)*, *triangle(4)*, *triangle(2)*, and *triangle(1)* – 8 levels of recursion – but no further.

3. Some Powerful New *Turtle System* Concepts

Now we're going to learn some new concepts, in which the *Turtle* itself will play a much smaller role, and we'll be thinking about the Canvas as a whole, rather than focusing specifically on the *Turtle*'s position on that Canvas. We start with a short program which can be found under "Examples 5 – user input, interaction and games", labelled "Colouring cells". When you first load it, 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.

```
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
  reset(\mousekey);
  mk:=detect(\mousekey,5000);
  if mk=1 then
   pixset(?mousex,?mousey,turtc)
  else
  if mk=2 then
   turtc:=pixcol(?mousex,?mousey)
  else
  if mk=3 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 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.

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

- 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 that *constant definitions can only come at the very beginning of any program*.
- Two integer variables are also defined: *x* will be used for counting across the width (i.e. for the *x-coordinate*), and *mk* will be used to identify mouse clicks and key presses.
- The new 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. We'll do that in future.
- The new 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.
- The following short loop introduces two new commands …

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

- The command `pixset(x,y,red)` is used to set the individual pixel at coordinates (*x*, *y*) to the specified colour – in this case, *red*. *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 colours the ten bottom pixels with the first ten build-in colours.
- The program then 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.
- The following two lines enable us to capture the relevant mouse or keyboard input …

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

- These are using 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. The command `reset(\mousekey)` "resets" it so that it's ready to detect a mouse click or keypress.
- Then 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 3 depending on whether it was the left, right or middle mouse button. 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).
- The following if … then command deals with a left mouseclick …

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

- Here we see the `pixset` command again, 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.
- If the variable *mk* wasn't equal to 1 (i.e. a left mouseclick), then we go on to the following else condition, which identifies a right mouseclick …

```
if mk=1 then
 pixset(?mousex,?mousey,turtc)
else
if mk=2 then
 turtc:=pixcol(?mousex,?mousey)
```

- Now we see another new 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 that colour.
- 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 3 …

```
if mk=1 then
 pixset(?mousex,?mousey,turtc)
else
if mk=2 then
 turtc:=pixcol(?mousex,?mousey)
else
if mk=3 then
 pixset(?mousex,?mousey,rgb(random(10)+1))
```

- 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. Note that `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.

4. Onwards to Cellular Automata

You are now in a position to understand how the *Turtle System* handles the pixel-based manipulation which is crucial for its ability to implement cellular automata so easily. For much more on these, see the documents "Cellular Automata: Modelling Disease, 'Life', and Shell Patterns". Or just have fun exploring the various example programs under the menu "Examples 7 – cellular models"!