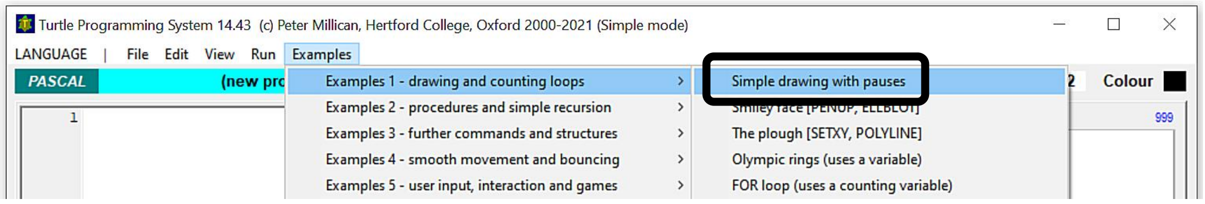
Royal Institution Masterclass

Oxford, June 12, 2021

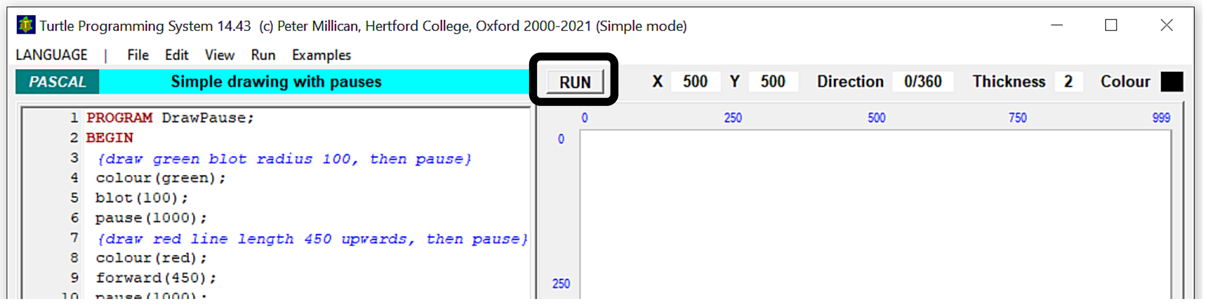
Spirals and Shapes with Turtle Pascal

1. Introducing Turtle Graphics programming

First, click on the "View" menu and select "Simple Mode" to hide the more advanced features of the system. Then click on the "Help" menu, point to "Examples 1 – drawing and counting loops", and select the first program on the list ("Simple drawing with pauses"). This contains simple commands for moving around and drawing on the *Canvas*, which is the square area on the right of the screen. This kind of programming is called *Turtle Graphics* because we imagine the moving and drawing being done by an invisible *Turtle*, which starts off in the middle of the Canvas (pointing "north").


Turtle Programming System 14.43 (c) Peter Millican, Hertford College, Oxford 2000-2021 (Simple mode)
LANGUAGE | File Edit View Run Examples
PASCAL
Examples 1 - drawing and counting loops
Examples 2 - procedures and simple recursion
Examples 3 - further commands and structures
Examples 4 - smooth movement and bouncing
Examples 5 - user input, interaction and games
Simple drawing with pauses
Smiley face [PENUP, ELLBLOT]
The plough [SETXY, POLYLINE]
Olympic rings (uses a variable)
FOR loop (uses a counting variable)
Colour

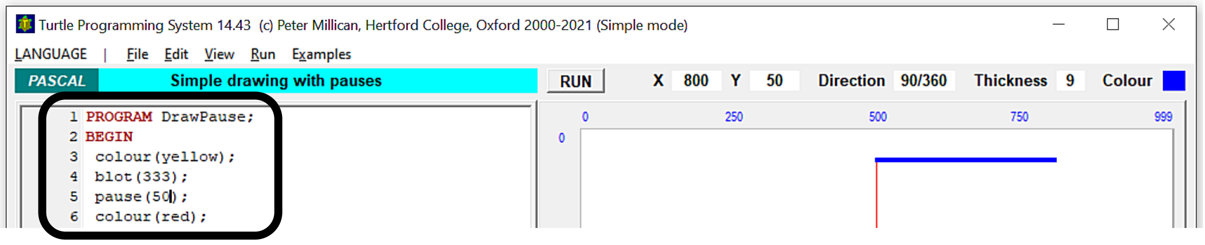
When the program has loaded, click on the **RUN** button and see what happens. Then do it again... Can you work out how the commands are having the effects that you see on the canvas? Notice that the *Turtle* leaves a coloured trail (of some particular colour and thickness) when it moves forward, and that "`pause(1000)`" makes the program pause for 1000 milliseconds (i.e. one second).


Turtle Programming System 14.43 (c) Peter Millican, Hertford College, Oxford 2000-2021 (Simple mode)
LANGUAGE | File Edit View Run Examples
PASCAL
Simple drawing with pauses
RUN

X 500 Y 500 Direction 0/360 Thickness 2 Colour

```
PROGRAM DrawPause;
BEGIN
 {draw green blot radius 100, then pause}
 colour(green);
 blot(100);
 pause(1000);
 {draw red line length 450 upwards, then pause}
 colour(red);
 forward(450);
```

The example programs contain some *comments* (between curly brackets, and shown in blue italics) – when you've read them, they can be removed using "Remove any comments" from the "Edit" menu. Then make some simple changes to the program, like changing the colours, blot sizes, distances, angles, or pause times. Then **RUN** the program again, and repeat...


Turtle Programming System 14.43 (c) Peter Millican, Hertford College, Oxford 2000-2021 (Simple mode)
LANGUAGE | File Edit View Run Examples
PASCAL
Simple drawing with pauses
RUN

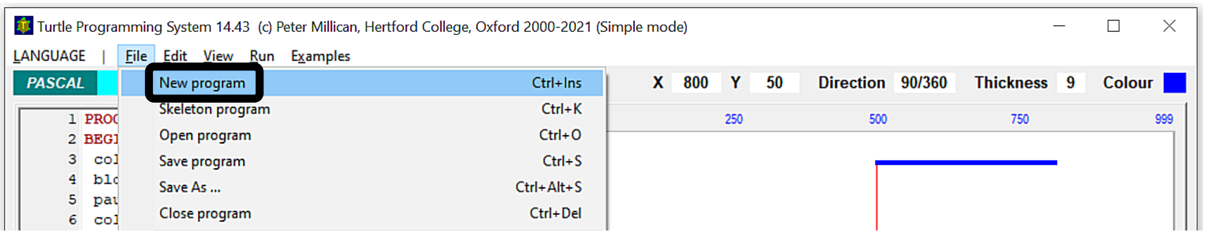
X 800 Y 50 Direction 90/360 Thickness 9 Colour

```
PROGRAM DrawPause;
BEGIN
 colour(yellow);
 blot(333);
 pause(50);
 colour(red);
```

At the end of this handout, you'll find a section called "The Program". *Read this through* (because it explains some of the basics of Turtle Graphics) and ask if there's anything you don't understand.

2. FOR loops and a spiral pattern

Click on the "File" menu, and then on "New program" (the first entry). Alternatively, you can choose "Skeleton program" (the second entry), which gives you a very short program with **PROGRAM**, **BEGIN**, and **END.**, and a few example commands already written in.


Turtle Programming System 14.43 (c) Peter Millican, Hertford College, Oxford 2000-2021 (Simple mode)
LANGUAGE | File Edit View Run Examples
PASCAL
New program Ctrl+Ins
Skeleton program Ctrl+K
Open program Ctrl+O
Save program Ctrl+S
Save As ... Ctrl+Alt+S
Close program Ctrl+Del
X 800 Y 50 Direction 90/360 Thickness 9 Colour
250 500 750 999

Type in the following program (or edit the skeleton program accordingly), and then click **RUN** to see what it does.

```
PROGRAM SpiralA;
VAR count: integer;
BEGIN
  for count:=1 to 130 do
    begin
      forward(count);
      right(10);
    end
END.
```

The line "VAR count: integer" defines an integer *variable* called "count". You can think of this as creating a box labelled "count" which is able to contain just one *integer* (i.e. a whole number, either positive, zero, or negative). After "BEGIN", the main program involves a "for loop":

```
  for count:=1 to 130 do
    begin
      forward(count);
      right(10);
    end;
```

This in turn puts all the integers from 1 to 130 into the `count` variable, each time performing the commands between "begin" and "end" (this is called a *loop*, because it goes "round and round" that sequence of commands). So these commands – "`forward(count); right(10)`" – get performed 130 times, first with *count* equal to 1, then 2, then 3, … and finally 130.

What these commands do is first to move the *Turtle* forward by *count* units (i.e. by 1 unit, then 2 units, then 3 units, … and eventually by 130 units), and then (each time) turn the *Turtle* right by 10 degrees. So the *Turtle* moves round a spiral path, getting faster and faster as it goes. If you add a *blot* command to the end of the loop, you'll be able to see more clearly the *Turtle*'s stopping places:

```
  for count:=1 to 130 do
    begin
      forward(count);
      right(10);
      blot(5);
    end;
```

(If you want to stop the *Turtle* from automatically drawing a line as it moves "forward" from one stopping place to another, put the command "`penup;`" immediately after "**BEGIN**", before the FOR loop starts. This lifts its *pen* off the Canvas so that it can move without drawing. To put the *pen* back down on the Canvas at any point, use "`pendown;`".) Having identified the stopping places, you can now create a simple pattern at each of them, for example a red blot with a black circle around it.

```
PROGRAM SpiralB;
VAR count: integer;
BEGIN
  for count:=1 to 130 do
    begin
      forward(count);
      right(10);
      colour(red);     ┐   These four lines draw a red blot surrounded
      blot(200);       ├   by a black circle (without moving the Turtle).
      colour(black);   │   You might like to play with different patterns
      circle(200);     ┘   by changing the colours and blot/circle sizes.
    end
END.
```

3. Procedures and parameters

The *Turtle System* comes with several built-in commands for drawing shapes, like `blot` and `circle`, but you can also specify your own commands to make more interesting or complex shapes. To do this, you write a mini-program called a *procedure*, and give it the name of the command you want to define. The following program contains a procedure that defines a command called "filledcircle", which draws a red blot of size 200 surrounded by a black circle of size 200:

```
PROGRAM SpiralC;
VAR count: integer;

  Procedure filledcircle;  ┐
  Begin                    │
    colour(red);           │
    blot(200);             ├  here is the procedure, defining
    colour(black);         │  a new command "filledcircle"
    circle(200);           │
  End;                     ┘

BEGIN
  for count:=1 to 130 do
    begin
      forward(count);
      right(10);
      filledcircle;  ◄────────  this line calls the procedure
    end
END.
```

In fact, this program (SpiralC) draws exactly the same as the previous program (SpiralB) – the four lines drawing the red blot surrounded by a black circle have simply been taken out and put inside the “filledcircle” procedure. This may seem a bit pointless so far, but procedures become much more valuable when they are called multiple times in a program, and especially when they are defined with *parameters*, which enable them to be flexible (in the same way as most of the built-in commands). For example, you can change the “filledcircle” procedure in the last program (SpiralC) as follows:

```
Procedure filledcircle(size: integer);
Begin
  colour(red);
  blot(size);
  colour(black);
  circle(size);
End;
```

Now the procedure is able to draw filled circles of *varying* sizes, so when you call it, you must specify the size you want. To get the previous effect, you need to change the calling line to:

```
filledcircle(200);
```

When this calls the procedure, it “posts in” the value 200, making *size* equal to 200 – so yet again, the program does exactly the same thing as before. But now, if you change the value (e.g. to 150, or 240, or whatever), you’ll see that the size of the filled circles changes accordingly. You can also set the size of the circles to whatever is the current value of *count*, so they get bigger as the spiral grows:

```
filledcircle(count);
```

or else to *half* the current value of count, which gives a very different effect by exposing lines drawn as the *Turtle* moves forward (unless you have added the command “`penup;`” as explained earlier):

```
filledcircle(count/2);
```

In what follows, we’ll only be using this procedure with just a single parameter (i.e. *size*), but it’s worth noting that you can also define a procedure to have two or more parameters, for example:

```
Procedure filledcircle(size,fillcolour: integer);
Begin
  colour(fillcolour);
  blot(size);
  colour(black);
  circle(size);
End;
```

Now the procedure is able to draw filled circles of different sizes *and* different fill colours, so when you call it, you must specify both the size and the colour, for example:

```
filledcircle(200,red);
```

Here is a version of the procedure which has three parameters, and specifies also the border colour:

```
Procedure filledcircle(size,fcolour,bcolour: integer);
Begin
  colour(fcolour);
  blot(size);
  colour(bcolour);
  circle(size);
End;
```

4. Random colours, positions, and sizes

To make the program more colourful, however, it is easiest to use the function `randcol(n)`, which randomly sets the *Turtle*'s colour to one of the first *n* colours defined in the system.[1] So to make the "filledcircle" procedure multi-coloured, you could replace "`colour(red)`;" with "`randcol`(30);" so that it chooses one of 30 colours for the blot instead of printing a red blot every time.

So far, the program has just been drawing filled circles in a spiral pattern, but now you have the "filledcircle" procedure, you might want to try using it differently. Suppose, for example, that you want to put filled circles at random points on the Canvas. To do this, you can use the command `setxy(x,y)` to position the Turtle on the Canvas at coordinates (*x*, *y*), and the function `random(n)` which generates a random number between *0* and *n*–1 inclusive. Since the standard Canvas dimensions are 1000x1000, with coordinates 0 to 999 inclusive, the following command will position the Turtle randomly on the Canvas:

```
setxy(random(1000),random(1000));
```

Then to draw a filled circle with a random size between 20 and 50 (inclusive) at the new position, you could use:

```
filledcircle(20+random(31));
```

If you replace the body of your earlier FOR loop with these instructions, and count up to 2000 instead of only 130, you get:

```
PROGRAM ScatterA;
VAR count: integer;

  Procedure filledcircle(size: integer);
  Begin
    randcol(30);
    blot(size);
    colour(black);
    circle(size);
  End;

BEGIN
  for count:=1 to 2000 do
    begin
      setxy(random(1000),random(1000));
      filledcircle(20+random(31));
    end
END.
```

This will display 2000 filled circles, randomly placed, randomly sized (between 20 and 50) and randomly coloured, which should cover nearly all of the Canvas.[2]

[1] The 50 defined colours can be seen by clicking on the "QuickHelp 1" tab below the Canvas, and choosing "Operators/Colours" at the right hand side. The "QuickHelp 2" tab lists the built-in commands.

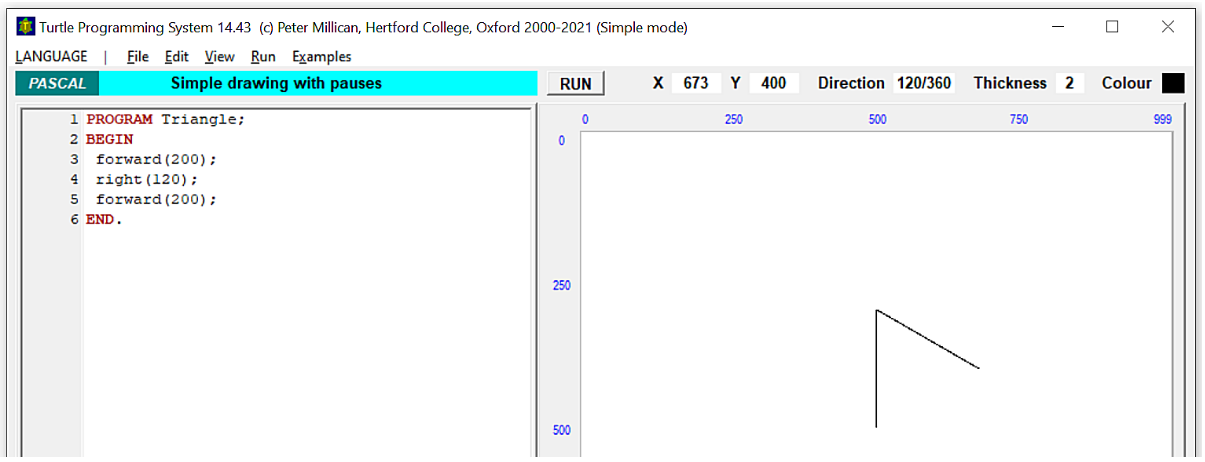
[2] If you want the circles to go on accumulating forever (or at least, until you HALT the program), then you could replace the FOR loop with an unending REPEAT loop. To do this, replace "`for count:=1 to 2000 do begin`" with "`repeat`"; and replace "end" with "`until 0=1`" (or some other condition that is *never* true).

5. Drawing a triangle, square, pentagon and hexagon

Now create another new empty program (click on the “File” menu, and then select “New program”), and copy the following code:

```
PROGRAM Triangle;
BEGIN
  forward(200);
  right(120);
  forward(200);
END.
```

Run the program to see what it does. On the Canvas on the right, you should see two sides of an equilateral triangle. The current position of the Turtle is shown by the **X** and **Y** boxes: **X** gives the distance from the left of the canvas, and **Y** gives the distance from the top. The Canvas is 1000 units square, so **X**=500 and **Y**=500 is in the centre. **Direction** gives the direction (in angles) that the Turtle is facing, with 0 meaning North, 90 meaning East, 180 meaning South, and 270 meaning West.


Turtle Programming System 14.43 (c) Peter Millican, Hertford College, Oxford 2000-2021 (Simple mode)
LANGUAGE | File Edit View Run Examples
PASCAL
Simple drawing with pauses
RUN
X 673 Y 400 Direction 120/360 Thickness 2 Colour
0 250 500 750 999
0 250 500

```
PROGRAM Triangle;
BEGIN
 forward(200);
 right(120);
 forward(200);
END.
```

Now try adding two commands to this program (before the “END.” line) so that it draws a complete equilateral triangle. Can you see how to do this? Can you also add a third command (making six in all) so that the *Turtle* ends up pointing in its original direction and in its original place?

Now start another new program, and write a similar program to draw a square, starting like this:

```
PROGRAM Square;
BEGIN
  forward(200);
  right(90);
  ...
```

Again make sure that it brings the *Turtle* back to its original position and direction. *(To enable our shape-drawing routines to be used within programs like the original spiral, without messing them up, we want all such routines to leave the Turtle’s position and direction unchanged.)*

Can you create a similar program to draw a regular pentagon (with 5 sides), and one to draw a regular hexagon (with 6 sides)? You might find it helpful to notice that for the regular triangle you had to turn 120° at each corner, and for the square 90°. So to draw a complete shape, the *Turtle* has to turn enough corners to get back to its original direction. So it has to turn through a total of 360° (a complete circle), which is equal to 3×120° or 4×90°.

6. Drawing shapes with a FOR loop

Your program to draw a triangle used the same pair of commands (`forward` and `right`) three times in a row, and your program to draw a hexagon used the same pair of commands six times in a row. Can you see how these programs could be made more efficient by using a FOR loop?

Open a new program again ("File", "New program"). This program is supposed to draw a triangle, as before, but this time using a separate "triangle" procedure that uses a FOR loop and also takes a "size" parameter. In outline, your program should look like this:

```
PROGRAM Shapes;

  Procedure triangle(size: integer);
  Var count: integer;
  Begin
    for count:=1 to 3 do
      begin
        commands to draw each side of the triangle ...
      end
  End;

BEGIN
  triangle(200)
END.
```

Now rename your "triangle" procedure "square". Can you edit the commands so that it draws a square instead? Think about how many sides a square has, and the angle needed at each point.

Now rename your "square" procedure "shape" instead, and give it also a "corners" parameter:

```
Procedure shape(corners,size: integer);
```

Edit the procedure so that it can draw a shape of any given number of corners (as well as any given size). Test it for "corners" between 3 and 6. Does it work as well when the "corners" value is 7?

If you want your shapes to be filled with colour (rather than just outlines), you can use the built-in `polygon(n)` command, which fills in the shape made by the last *n* points the *Turtle* has visited, using the current *Turtle* colour. You might like to experiment with this a bit.

7. Putting it all together

You have now learned how to draw filled circles and various shapes – all in different sizes – and how to arrange patterns in spirals or randomly across the Canvas. So now try using these techniques together, to create an interesting pattern. Note that if you want your program to make a random choice of shapes, you could do something like this, where "test" is set randomly between 0 and 6:

```
VAR test: integer;
...
test:=random(7);
if test=0 then
  filledcircle(...)   <――――――――   don't put a semicolon before an "else"
else
  shape(test, ...);
```

The Program

The program is a sequence of commands in the programming language *Turtle Pascal*, which you type into the Programming Area on the left of the screen, and which are executed when you click the **RUN** button. These commands determine what is drawn on the Canvas (the square area on the right of the screen). You might find it helpful to think of them as instructions being given to a small (and invisible) *Turtle* which moves around the Canvas, leaving a coloured trail and drawing shapes as it goes. The *Turtle* starts in the middle of the Canvas, pointing "north" (i.e. up towards the top of the screen).

The easiest way to see how all this works is to go to the "Examples" menu at the top of the screen, select "Examples 1 – drawing and counting loops" and click on the first of these, called "Simple drawing with pauses". This will load the following program into the Programming Area:

```
PROGRAM DrawPause;
BEGIN
  colour(green);
  blot(100);
  pause(1000);
  colour(red);
  forward(450);
  pause(1000);
  right(90);
  thickness(9);
  colour(blue);
  pause(1000);
  forward(300)
END.
```

If you now click on **RUN** you will see the effect of this program on the Canvas. Here is a brief explanation, which should be sufficient to enable you to write similar programs of your own:

Every program has to have a one-word name (here `DrawPause`), which is given after the word **PROGRAM** and followed by a semicolon; then the commands of the program itself are "bracketed" between **BEGIN** and **END**,[3] and separated from each other by semicolons (you don't need a semicolon before **END**, but you can put one if you like). The final **END** must be followed by a full-stop, which signifies the very end of the program. In this example the first command is `colour(green)`, which specifies the drawing colour as green; then `blot(100)` accordingly draws a green "blot" (i.e. a filled-in circle) of radius 100 around the initial position of the Turtle in the centre of the Canvas.

Next we have a `pause(1000)` command, which tells the Turtle to pause for 1000 milliseconds (i.e. 1 second). Then after another `colour` command we have `forward(450)`, which instructs the Turtle to move forward from its initial position by 450 units. This will leave a red trail, in accordance with the previous `colour(red)` command. After another `pause(1000)`, the Turtle is told to turn right by 90 degrees with `right(90)`, then `thickness(9)` tells it to thicken its pen from the standard 2 unit thickness to 9 units. This means that after `colour(blue)` and another `pause(1000)`, the final command `forward(300)` will draw a thick blue line of 300 units in what is now the direction of the Turtle (i.e. horizontal, having turned right by 90 degrees from its original vertical direction).

[3] Note that capitalization in *Turtle Pascal* is entirely optional – **PROGRAM**, **BEGIN**, and **END** are shown in uppercase only for emphasis, and it would make no difference if you changed them to lower-case and/or if you capitalised other words (or even if you used a mixture of upper- and lower-case).