Chapter 8

Pixel Graphics

8.1 Pixel Positions in the Execution Window

So far we have plotted graphics using characters. In this part we will plot graphics using dots or **pixels** in much the same way as a television picture is plotted. With characters there were 2000 positions in the window where a character could be placed (25 rows with 80 character positions in each row). With pixel graphics there are many more positions where dots can be plotted, over 150,000 in a standard output window (300 rows with 640 pixel positions in each row). Pixel graphics have a much **higher resolution**.

Pixel positions are given in terms of **coordinates**. The *x*-coordinate is the position number along a line starting from the left-hand side of the window. The first position has number 0; the last position depends on the type of graphics. For a standard output window it is 639. The *y*-coordinate is the row number starting at the bottom of the window at 0 and numbering upward. In a standard output window the last row number is 299. The *x*- and *y*-coordinates correspond to the usual way of plotting graphs mathematically. The **origin** of the coordinates is in the lower left-hand corner of the window.

Turing Execution windows can be one of two types: text windows or graphics windows. Text windows only support the **put** and **get** commands. You cannot use the locate or graphics commands in a text window. All output sent to a text Execution window can be viewed using the window's scroll bar. Saving the text Execution window produces a text file with all the output.

Graphics windows allow all available output commands but any output that scrolls off the top of the screen is lost. Saving a graphics Execution window creates a graphics file.

Here are the commands to set the window to text or graphics.

```
setscreen ("text")
setscreen ("graphics")
```

They must appear before any output.

8.2 Plotting Dots in the Execution Window

To place a dot in the window at a point whose coordinates are (*x, y*) you use the statement

drawdot (*x*, *y*, *c*)

This calls the predefined Turing procedure drawdot. If the values of *x* or *y* are outside the allowed range, the dot is not plotted.

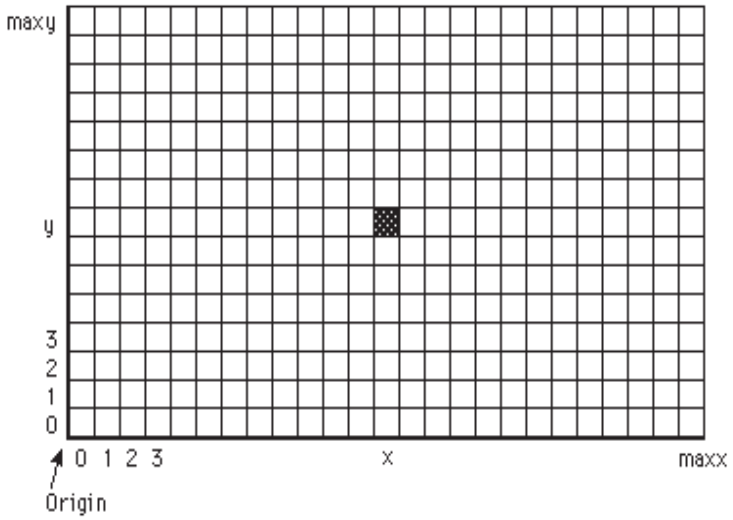

Figure 8.1 Pixel Locations

The *c* is an integer (or integer variable) that sets the color of the dot to be plotted. In graphics a zero value of *c* gives the same color as the background, normally white. Color values range from 0 up to 255, although it is preferable to use the predefined names for colors.

Because different computers may use differently sized output windows, rather than explicitly placing the size of the Execution window in a program, Turing allows you to use maxx and maxy.

The maxx predefined function represents the maximum *x*-coordinate available in the window and maxy represents the maximum *y*-coordinate available in the window. By using maxx and maxy, programs are resolution independent and will work regardless of Execution window size.

Here is a program like the *LeafFall* program in the section 7.4 on character graphics. Execution windows in Turing are normally in pixel graphics mode when the program begins. However, it is possible to set the Turing environment to start the Execution window in text mode. In text mode, output that scrolls off the end of the Execution window can be viewed using the scroll bars.

To set the screen to the pixel graphics mode, we use the statement

```
setscreen ("graphics")
```

This instruction is not absolutely necessary since a call to any pixel graphics procedure, such as drawdot, will set the Execution window to pixel graphics mode even if it was in text mode.

```
% The "Confetti" program
% Color pixels in the window randomly
setscreen ("graphics")
var x, y, c : int
loop
    randint (x, 0, maxx)
    randint (y, 0, maxy)
    randint (c, 1, 15)
    drawdot (x, y, c)
end loop
```

8.3 Changing the Execution Window Size

By default, Turing creates an Execution window that 25 rows by 80 columns. It is possible to change the size of the Execution window. This is done with the instruction

```
setscreen ("graphics:<width>;<height>")
```

where <width> and <height> are the desired width and the height in pixels of the window. For example, to set the Execution window to be 300 pixels wide by 500 pixels tall, the statement

```
setscreen ("graphics:300;500")
```

would be used. Note that there is a colon after the ÒgraphicsÓ and a semicolon between the width and the height. The window can also be sized to be the largest possible window that will fit on the screen. This can be set with the instruction

```
setscreen ("graphics:max;max")
```

An Execution window's size can also be set in terms of the number of rows and columns using the instruction

```
setscreen ("screen:<rows>;<columns>")
```

With this, the Execution window is in graphics mode and <rows> and <columns> are the desired number of rows and columns in the window.

To determine the maximum color number available, a program can use the maxcolor predefined function, which is gives you the maximum allowable color number.

When you change the size of a window, the values returned by maxx and maxy will also change. After the statement

```
setscreen ("graphics:300;500")
```

maxx would give 299 and maxy would give 499.

For example, a modification of the *Confetti* program to produce output in a smaller window could be written this way.

```
% The "Wedding" program
% Gives randomly colored dots with greater variety of
% color and higher resolution than CGA graphics
setscreen ("graphics:200;200")
var x, y, c : int
loop
    randint (x, 0, maxx)
    randint (y, 0, maxy)
    randint (c, 0, maxcolor)
    drawdot (x, y, c)
end loop
```

8.4 Drawing Lines

A line can be drawn from a point in the window with coordinates (*x1, y1*) to a point with coordinates (*x2, y2*) in color *c*

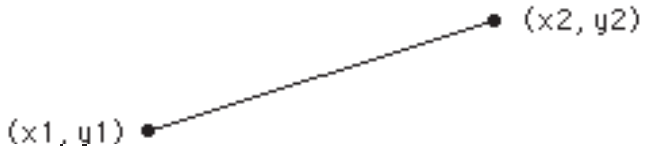

Figure 8.2 Drawing a Line with drawline

with the statement

drawline (*x1*, *y1*, *x2*, *y2*, *c*)

Here is a program that draws lines from the center of the window to points chosen randomly in random colors.

```
% The "StarLight" program
% Draws lines from center to random points
setscreen ("graphics")
var x, y, c : int
const centerx := maxx div 2
const centery := maxy div 2
```

```
loop
    randint (x, 0, maxx)
    randint (y, 0, maxy)
    randint (c, 0, maxcolor)
    drawline (centerx, centery, x, y, c)
end loop
```

Here is a program that draws a rectangle, or box, of width *w* and height *h* with its lower-left corner at (*x, y*).

The four drawline statements in the program could be replaced by the single statement

```
drawbox (x, y, x + w, y + h, c)
```

where (*x, y*) is the lower left corner and (*x + w, y + h*) the upper right corner of the box.

```
% The "DrawBox" program
% Draws a box of width, height,
% position, and color that you specify
var w, h, x, y, c : int
put "Enter width of box in pixels: " ..
get w
put "Enter height of box in pixels: " ..
get h
put "Enter x-coordinate of lower-left corner: " ..
get x
put "Enter y-coordinate of lower-left corner: " ..
get y
put "Enter color number: " ..
get c
setscreen ("graphics")
drawline (x, y, x + w, y, c)                % Base of box
drawline (x + w, y, x + w, y + h, c)    % Right side
drawline (x + w, y + h, x, y + h, c)     % Top
drawline (x, y + h, x, y, c)                % Left side
drawfill (x + 1, y + 1, c, c)               % Fill the box
```

The last statement of this program will fill the whole rectangle with the color number *c*. The form of this predefined procedure is

drawfill (*xinside*, *yinside*, *fillcolor*, *bordercolor*)

The area to be colored must contain the point (*xinside, yinside*) and be surrounded by the *bordercolor*. In this program the border color and the fill color are the same.

The four drawline statements and the drawfill statement could be replaced by the single statement

drawfillbox (*x*, *y*, *x* + *w*, *y* + *h*, *c*)

where (*x, y*) is the lower-left corner and (*x* + *w, y* + *h*) the upper-right corner of the box. The instruction drawfillbox draws a box and then fills it in with color *c*.

8.5 Drawing Circles and Ellipses

To draw an oval whose center is at (*x, y*), whose half-width is *xRadius*, and whose half-height is *yRadius* in color *c*

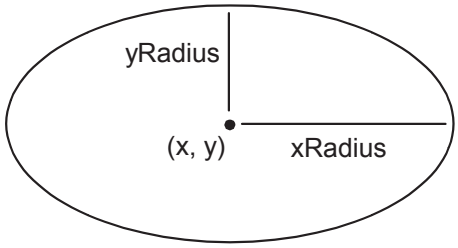

Figure 8.3 Drawing an Oval with drawoval

we use the statement

drawoval (*x*, *y*, *xRadius*, *yRadius*, *c*)

The resulting curve is an **ellipse**. To produce a circle we use equal values for *xRadius* and *yRadius.* Here is a program that draws a series of magenta circles radiating from the center of the window until the circles touch the boundary of the window.

```
% The "Pow" program
```

```
% Plots magenta circles of ever increasing radius centered
% in window until the edge of the window is reached
setscreen ("graphics")
const centerx := maxx div 2
const centery := maxy div 2
for radius : 1 .. centery
    drawoval (centerx, centery, radius, radius, magenta)
end for
```

In Turing, the predefined procedure drawfilloval draws a filled oval without the necessity of using drawfill. It has the form

drawfilloval (*x*, *y*, *xRadius*, *yRadius*, *c*)

and causes an oval to be drawn and filled with color *c*.

8.6 Animation

The effect of animation can be achieved with the *Pow* program by erasing each circle after a delay before plotting the next circle. To erase a circle you can re-plot it using the background color, number 0. The predefined procedure delay can be called to waste time between the plot of each circle in magenta and the plot of the same circle in black. It has the form

delay (*duration*)

where the duration is in milliseconds. The *Pow* program can thus be changed to the *Ripple* program in which a ripple starting at the center of the window appears to move out, by adding the two statements

```
delay (500)   % Delay half a second
drawoval (centerx, centery, radius, radius, 0)
```

after the drawoval that produces the magenta circle. This method of animation can be used for any simple form like a dot, line, box, or oval.

8.7 Drawing Arcs

A portion of an oval can be drawn using the predefined procedure drawarc in the form

drawarc (*x*, *y*, *xRadius*, *yRadius*, *initialAngle*, *finalAngle*, *c*)

where in addition to the parameters required to draw the oval you provide the initial and final angles in degrees measured counter clockwise that the lines from (*x, y*) to the end points of the arc make from the three o'clock position.

Figure 8.4 Drawing an Arc with drawarc

Here is a program that draws a series of circular arcs in green with their center at the center of the window and their radius changing by 1 pixel from 1 to 50. It is like the program *Pow* except that a portion of the circle is drawn rather than the whole circle. We will draw arcs of 60 degrees and a zero initial angle.

```
% The "CheeseSlice" program
% Draws a slice of green cheese
% that is one sixth of a round cheese
% by drawing circular arcs
setscreen ("graphics")
const maxRadius := 50
const xcenter := maxx div 2
const ycenter := maxy div 2
const theta := 60
for radius : 1 .. maxRadius
```

```
    drawarc (xcenter, ycenter, radius, radius, 0, theta, green)
end for
```

In Turing, you can also draw a filled in ÒsliceÓ by using the drawfillarc procedure. The call looks like

```
drawfillarc (x, y, xRadius, yRadius, initialAngle, finalAngle, c)
```

and draws a ÒsliceÓ filled with color *c*.

In the chapter on advanced pixel graphics we will use a different technique for drawing a **pie chart** but that requires more mathematics.

8.8 Plotting a Mathematical Function

To keep things simple we will plot a mathematical function that can be drawn with the origin of its coordinates at the lower left of the window, the bottom of the window as the *x*-axis, and the left side as the *y*-axis. We will plot the curve for the parabola

```
y = x ** 2
```

for values of *x* going from 0 to 14. We will let each unit in the *x*-direction be represented by 20 pixels. One pixel is thus .05 units. This means that in the x-direction we will use from pixel 0 to pixel 280. Each unit in the *y*-direction will be represented by 1 pixel since *y* will vary from 0 to 196. What we have done is to choose a proper **scale** for *x* and *y* so that the graph nearly fills the window.

```
% The "Parabola" program
% Draws the graph of the function y = x ** 2
% for x going from 0 to 14 in steps of .05
% Draw axes
drawline (0, 0, maxx, 0, blue)
drawline (0, 0, 0, maxy, blue)
for pixel : 0 .. 280
    const x := .05 * pixel
    drawdot (pixel, round (x ** 2), cyan)
```

```
end for
```

In the chapter on advanced pixel graphics we will show an all purpose mathematical graph plotting program.

8.9 Using Text with Pixel Graphics

Frequently we want to add text to a pixel graphics plot. The range of character colors is the same as for dots. The color number of characters is set by the statement

color (*chosenColor*)

The position of characters to be output is set using the statement

locatexy (*x*, *y*)

The next characters output by a **put** statement will begin approximately at the point (*x, y*). Note that the output will not necessarily appear at exactly (*x*, *y*). This is because the characters still appear in the 25 by 80 grid of character rows and columns. Instead, the character appears at the location closest to the *x* and *y* in the locatexy statement.

In the *Parabola* program we could have labelled the graph in magenta by adding these statements just before the last two statements.

```
color (magenta)
% Label x-axis
locatexy (160, 10)
put "x-axis"
% Label y-axis
locatexy (10, 100)
put "y-axis"
% Label graph of parabola
locatexy (100, 100)
put "Graph of parabola y = x**2"
```

8.10 Background Color

The color of the background of the Execution window can be set by using the statement

drawfillbox (0, 0, maxx, maxy, colorNumber)

This erases anything else in the window that is currently displayed.

Here is a program to change the color of the window randomly.

```
% The "FlashWindow" program
% Randomly changes the window color
setscreen ("graphics")
var c : int
loop
    randint (c, 0, maxcolor)
    drawfillbox (0, 0, maxx, maxy, c)
    delay (500)
end loop
```

8.11 Sound with Graphics

A simple sound is often all that is required in a graphic display. It can be obtained using the statement

sound (*frequency*, *duration*)

where the frequency is in hertz or cycles per second and the duration is in milliseconds. The frequency should generally be between 200 and 2000. Frequencies outside that range are too low or high to be reproduced on a computer speaker.

Often, computers require a sound card to process the sound or music commands.

Here is a program that draws graphics and plays a sound that goes up and then down.

```
% The "MakeSound" program
% Plays a rising then falling frequency along with colorful graphics
for i : 0 .. maxy
    drawfillbox (0, 0, i * 2, maxy – i, i mod 16)
    sound (i * 2 + 200, 50)
end for
for decreasing i : maxy .. 0
    drawfilloval (0, 0, i * 2, maxy – i, i mod 16)
    sound (i * 2 + 200, 50)
end for
```

8.12 Current Values of Graphic Parameters

You can find out the current values of various parameters when you are in pixel graphics mode. A number of functions provide such values.

```
whatdotcolor (x, y)
```

gives the color of the pixel at (*x, y*); the value of

```
whatcolorback
```

is the current background color number. Similarly the whatcolor function returns the current text color.

8.13 Exercises

1. Write a program to plot a horizontal band 5 pixels wide centered on any *y*-value you specify and with any color.

2. Change the program so that you can continue to add such horizontal lines as long as you want.
3. Write a program to draw a striped pattern of lines of random color at an angle of 45¡ to the bottom of the window. Arrange to stop execution when the window is completely covered.
4. Draw four circular arcs of width 5 pixels and radius 50 pixels centered on the four corners of the window using the drawarc procedure. What happens if you try to draw four complete circles using these centers and the drawoval procedure?
5. Draw a slice of watermelon with red flesh and green rind. Make it one quarter of a complete circular slice.
6. Write a program like the *BrownianMotion* program but instead of having an asterisk, use a small magenta ball. Can you arrange to display this on a window that is not black?
7. Write a program like the *Bounce* program only using a small circle instead of an asterisk to represent a puck bouncing off the boards in a hockey game. Make a sound as you bounce. Use delay rather than a time-wasting loop.
8. Write a program to draw a cartoon of a smiling face in the Execution window. Change the face to frown; now alternate between these two faces.
9. Draw the face of a clown centered on the window. Use maxx **div** 2, and maxy **div** 2 as the center of the face. Make the face in any shape and size oval that you wish. Add a nose in the center of the face. Add eyes so that they are symmetrically placed on either side of the face (they should be equidistant from the center of the face). Add a mouth, an arc. It's center coordinate should have maxx **div** 2 as its x. Add an ear on each side of the head. Make them so that their center has maxy **div** 2 for its y. Now that your clown is drawn, use animation to close the eye and open it again. The eye should close by reducing the size of the oval until it hits a size of 0. It should open by drawing an eye which grows from size 0 to the previous eye size. Use a delay to control the speed of the closing and opening of the eye. Which eye you animate is your choice.

10. Modify the clown above so that the eye closes in a different manner. The manner is of your choice. You may have it close from top to bottom, like a eyelid coming down, from left side and right side simultaneously until they hit the middle, or any method you choose. Again make the eye open. Use delays to control the speed of the closing and the opening.
11. Modify either clown#1 or clown#2. Have the eye close as well as the mouth close. The actions should appear to be simultaneous. Then have the eyes and mouth reopen, again simultaneously.
12. Draw an arrow on the bottom left of the window using drawlines. The point of the arrow should face right. Have the arrow move across the window from left to right using delay to control the speed. Once the arrow is off the window, make it reappear on the bottom- right corner with the point facing left.
13. Draw an oval at the bottom left of the window. Have the oval move from the bottom to the top of the window. Once it hits the top, make it change direction and come to the bottom. Once it hits the bottom, make it go diagonally to the top-right corner of the window and then drop vertically to the bottom of the window again (bottom- right).
14. Draw a ball in motion starting at the top of the window so that its center moves to the position it would move to under gravity in each step of the animation. The equation for the *y*-coordinate of the center is

    ```
    y = ymax – (1/2) * a * t ** 2
    ```

    where *a* is the acceleration of gravity, namely 9.8 m sec^{-2}, and *t* is the time in seconds. If each pixel in the *y*-direction represents 1 meter choose an appropriate time scale to watch the motion: in real time, in slow motion, and in time lapse motion.
15. Draw an animated graph to represent a ripple being reflected from the top of the window when the wave of the program *ripple* in this chapter strikes it. What happens when you let the original ripple go on out beyond the top of the window?

16. Write a program to plot either a cosine or a sine curve for values of the angle going from 0 to 720 degrees. Draw the *x*- and *y*-axes, and label the graph appropriately. Choose a scale for plotting that has the graph more or less fill the window. Remember that the values of cosine and sine vary between –1 and +1.

8.14 Technical Terms

pixel
dot
plot
graphics mode
resolution
coordinate
axis
origin of coordinates
drawdot
aspect ratio
palette
setscreen
maxx
maxy
maxcolor
drawline
drawfill
drawbox
drawoval
getch
delay
drawarc
pie chart
scale
parabola
sound
whatdotcolor
whatcolorback
whatcolor