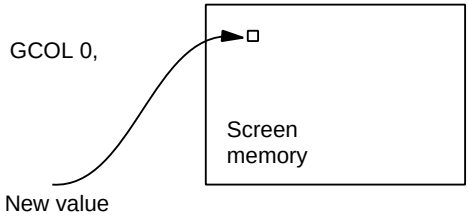
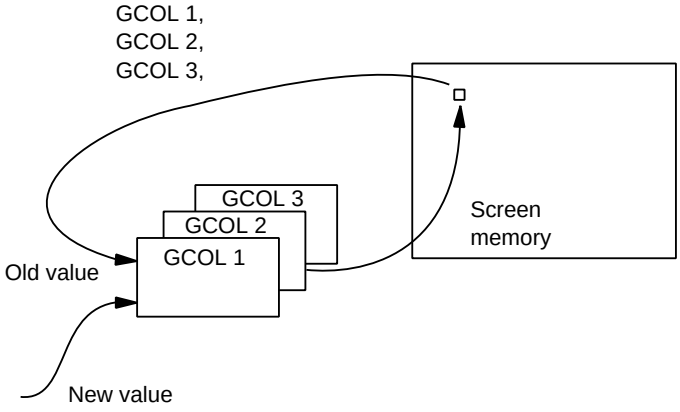
Chapter 2 Logical processing of colour and interactive graphics

In this chapter we examine in detail the uses of the logical processing facilities available in the GCOLstatement. In particular we look at how they can be used to create user defined mappings of the screen memory and to provide interactive graphics facilities.

The GCOL statement controls the way in which new values are loaded into the screen memory. Either a new value (or colour) specified for a particular pixel is loaded directly into the appropriate position in the screen memory:



or else a logical operation is performed between the new value and the current value in the screen memory.



The type of logical operation is specified by the first parameter in the GCOL statement. Tha GCOL statement provides

powerful logical and colour processing facilities that can be used in a wide variety of graphics applications. Some of these are developed in the remainder of this chapter and others are utilised in the chapter on animation. Without such logical facilities many advanced graphics applications would be impossible. The GCOL facilities are:

GCOL 0, colour	any subsequent plotting will be in the specified colour
GCOL 1, colour	the colour that results from subsequent plotting is produced by performing an operation between the specified colour and the existing screen colour at a pixel
GCOL 2, colour	as 1 but the logical operation is AND
GCOL 3, colour	as 1 but the logical operation is EOR (exclusive OR)
GC0L 4, colour	as 1 but the logical operation is NOT (i.e. the colour at any pixel visited is inverted)

The application of logical operations such as OR, AND, EOR and NOT is explained in Appendix 2. For details on the handling of foreground and background colour, consult our companion volume or the User Guide. GCOL 1 and 2 have applications in the dividing of an image into planes such as foreground, background and midground and GCOL 3 and 4 have applications in interactive graphics described later.

2.1 Image planes (GCOL 1 and GCOL 2)

We shall start by considering multi-plane images. A multi-plane image is an abstraction for the convenience of the programmer. He can build up images independently in planes that are (virtually) separate. This is useful in animation (later) and in being able to deal with a composite image, where different planes have a different priority, e.g. foreground, midground and background (below). We look first at the easier problem of constructing separate planes and switching between them, and then at the more general problem of building up a composite image from planes of different priority.

Virtual image planes: separate images

In a four-colour mode, two bits per pixel are used in the computer screen memory. However it is up to the programmer how he uses them. We could set up a scheme where we have two 'independent' planes or a scheme where we have a foreground and a midground plane of the same image. Consider the former scheme. We can set up the two bits at a pixel in this way:

```
0 = 00 = imagel and image2 background
1 = 01 = image1 foreground
2 = 10 = image2 foreground
3 = 11 = imagel and image2 foreground
```

The fourth code (3=11) is necessary to signify that for a particular pixel both imagel and image2 planes are 'on'. Starting with the easier consideration of switching between planes (assuming that both images are already built up) we would proceed as follows:

DISPLAY image1:

```
VDU 19, 0, backgroundcolimage1, 0,0,0
VDU 19, 1, foregroundcolimage1, 0,0,0
VDU 19, 2, backgroundcolimage1, 0,0,0
VDU 19, 3, foregroundcolimage1, 0,0,0
```

Image2 is thus set to the background colour selected for image1 and becomes invisible.

DISPLAY image2:

```
VDU 19, 0, backgroundcolimage2, 0,0,0
VDU 19, 1, backgroundcolimage2, 0,0,0
VDU 19, 2, foregroundcolimage2, 0,0,0
VDU 19, 3, foregroundcolimage2, 0,0,0
```

Now imagel is set to the background colour selected for image2 and becomes invisible.

To plot in the image1 plane, say, we have to proceed as follows for each pixel:

```
0 = 00 becomes 01
1 = 01 remains 01 (point already there)
2 = 10 becomes 11 (image2 point already there)
3 = 11 remains 11 (image2 and imagel point already there)
```

The third column is produced by ORing (inclusive) 01 with the second column and we simply precede any plotting statements with the appropriate GCOL statement:

PLOT image1: precede PLOT statements with GCOL 1, 1

Similarly to plot in the image2 plane:

```
0 = 00 becomes 10
1 = 01 becomes 11
2 = 10 remains 10
3 = 11 remains 11
```

and the appropriate GCOL is:

PLOT image2: precede PLOTs with GCOL 1, 2

The following program builds up a simple image in each image plane then repeatedly switches between them.

```
 10  MODE 5
 20  PROCplotimage1
 30  PROCplotimage2
 40  FOR screen = 1 TO 10
 50    PROCdisplayimage1
 60    PROCdelay
 70    PROCdisplayimage2
 80    PROCdelay
 90  NEXT screen
100  END

110  DEFPROCplotimage1
120    GCOL 1, 1
130    PROCdrawacircle(500, 500, 125)
140  ENDPROC

150  DEFPROCplotimage2
160    GCOL 1, 2
170    PROCdrawatriangle(327, 400, 346)
180  ENDPROC

190  DEFPROCdisp1ayimage1
200    VDU 19, 0, 2, 0,0,0
210    VDU 19, 1, 1, 0,0,0
220    VDU 19, 2, 2, 0,0,0
230    VDU 19, 3, 1, 0,0,0
240  ENDPROC
```

```
  250  DEFPROCdisplayimage2
  260    VDU 19, 0, 4, 0,0,0
  270    VDU 19, 1, 4, 0,0,0
  280    VDU 19, 2, 0, 0,0,0
  290    VDU 19, 3, 0, 0,0,0
  300  ENDPROC

  310  DEFPROCdrawacirc1e(xc, yc, r)
  320    MOVE xc + r, yc
  330    FOR theta = 10 TO 360 STEP 10
  340    x= r*COS(RAD(theta))
  350    y= r*SIN(RAD(theta))
  360    x= xc + x : y = yc + y
  370    MOVE xc, yc : PLOT 85, x, y
  380    NEXT theta
  390  ENDPROC

  390  DEFPROCdrawatriangle(xs, ys, s)
  400  MOVE xs, ys : DRAW xs+s, ys
  410  PLOT 85, xs+s/2, ys+s*0.866
  420  ENDPROC

  430  DEF PROCdelay
  440    TIME = 0
  450    REPEAT : UNTIL TIME>100
  460  ENDPROC
```

In the above program note that we are using completely different colours in each image. Switching between image planes that use the same colour is important in animation (Chapter 4).

Incidentally information common to both planes (such as text, say), need only be plotted once using GC0L 0, 3.

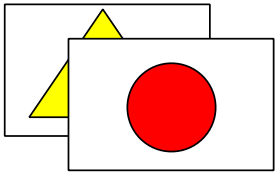
Composite image with priority (3 planes)

Again with a choice of four colours the above scheme can easily be adapted to set up a three-plane composite image (foreground, midground and background). Using GCOL the foreground and midground planes can be independently accessed and anything drawn in the midground plane that is shadowed by anything drawn in the foreground plane is automatically obscured in the composite image. Also we can delete from the foreground, delete from the midground, add to the foreground or add to the midground and the foreground/midground priority is automatically taken into account. Common operations that we might want to perform are:

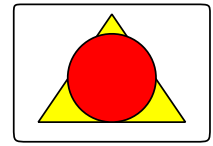
Logical image planes *Composite display image*

(The figures are meant to be solid or filled)

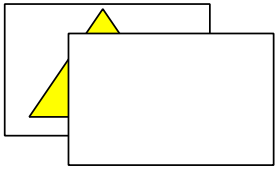
Initial image−a circle in the foreground against a triangle in the midground



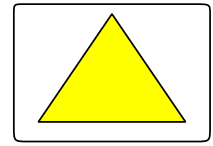
=



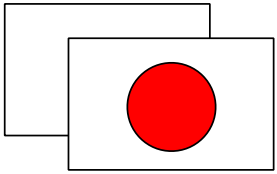
Delete foreground from initial image



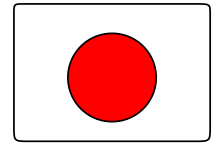
=



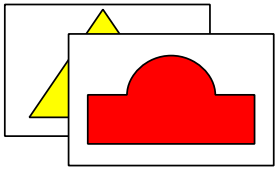
Delete midground from initial image



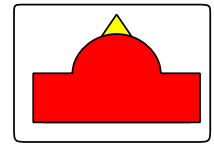
=



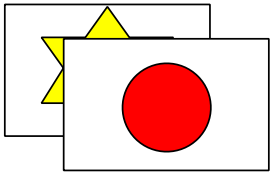
Add to foreground in initial image



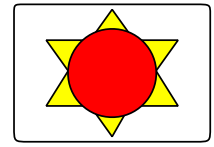
=



Add to midground in initial image

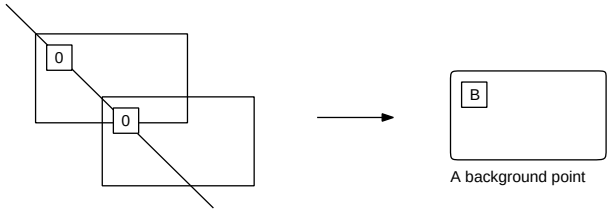


=

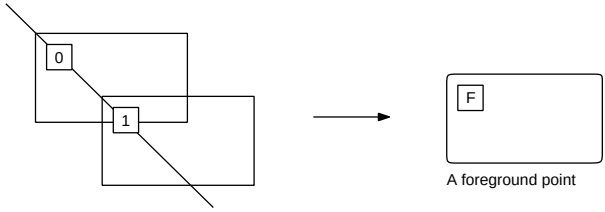


How this is accomplished is now explained. Suppose we are operating in a four colour mode (this allows two planes plus background). A four colour mode means that there are two bits per pixel, i.e. we can imagine the image memory as two one-bit planes.

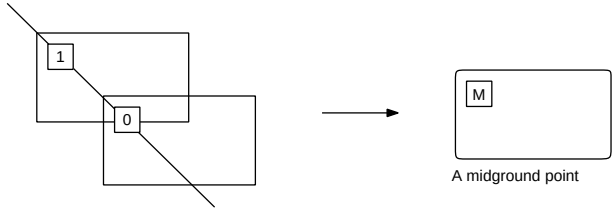
If the two planes have 0,0 in a pixel position, then the display image is a background point:



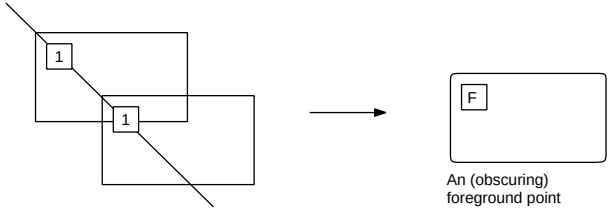
If the two planes have 0,1 in a pixel position then the display image is a foreground point:



1,0 Means a midground point:



Finally 1,1 means a foreground point but this time one that is obscuring amidground point:



Thus we have

```
0 = 00 =  background  point (. in illustrations)
1 = 01 =  foreground  point (F in illustrations)
2 = 10 =  midground   point (M in illustrations)
3 = 11 =  foreground  point (F in illustrations)
          (obscuring a midground)
```

Note that we use two logical colour codes to represent the foreground. This is because we can have 2 types of foreground - a foreground point obscuring a a background point only, and a foreground point obscuring a midground point. We can now give a few examples of 'plane' plotting and you can generalise from these examples.

To PLOT in the foreground

We precede any plot statements with GCOL 1,1 (inclusive OR):

```
GCOL 1,1
PLOT statements to plot figures in foreground plane
```

Now because

```
    00      OR       01      =       01
background       foreground      foreground
```

Background points are obscured by foreground points:

```
        ......................
        ...FFFFFFFFFFFF.....
        ...F.................
        ...F.................
        ...FFFFFFFFFF.......
        ...F.................
        ...F.................
        ...F.................
        ......................
```

To PLOT in the midground

We precede any plot statements with GCOL 1,2 (inclusive OR):

```
GCOL 1,2
PLOT statements to plot figures in midground plane
```

Now because

```
    00      OR       10      =       01
background       midground       midground
```

and

```
     01      OR       10      =       11
 foreground        midground       foreground
```

background points are obscured by midground points as you would expect, but points that are are already foreground remain in the foreground colour (but with code 11 indicating that they are obscuring a midground point). Thus to build up information in these two planes we use GCOL 1 (inclusive OR).

```
        ....................
        ...FFFFFFFFFFFF.....
        ...FM.........MM.....
        ...F.M.......M.M.....
        ...FFFFFFFFFF.M.....
        ...F...M...M...M.....
        ...F....M.M....M.....
        ...F.....M.....M.....
        ....................
```

You can perhaps see from this that after a composite set of planes has been built up any subsequent additions to the foreground or midground will be incorporated into the composite image according to their respective priority.

To DELETE from foreground and midground

Now to delete images or parts of images from planes we use GCOL 2 (AND). To delete a foreground object, we redraw the object after using GCOL 2,2, where the second parameter happens to be the midground colour but is used here as a 'foreground delete code'. To delete a midground object, we use GCOL 2,1. For example to delete from the foreground:

```
   GCOL 2, 2
   PLOT statements to delete figure from foreground plane
```

and the PLOT statements will be exactly the same as the ones that were used to draw the object being deleted. Now we have

```
     00      AND       10      =       00
 background                        background
```

i.e. background points remain as background

```
     01      AND       10      =       00
 foreground                        background
```

'ordinary' foreground points revert to background

```
     10      AND       10      =       10
  midground                          midground
```

'ordinary' midground points are left unaltered

```
     11      AND       10      =       10
```

'obscured' midground points are now revealed

```
        ....................
        ...M...........M.....
        ...MM.........MM.....
        ...M.M.......M.M.....
        ...M..M.....M..M.....
        ...M...M...M...M.....
        ...M....M.M....M.....
        ...M.....M.....M.....
        ....................
```

Thus GCOL 2 (AND) can be used to delete and reveal. These operations are now demonstrated. The procedures are left undefined (see earlier sections) but should include colour fill. The following program draws a red circle in the foreground plane and a yellow triangle in the midground plane and any geometrical overlap is automatically taken care of. Note line 20; remember that we use 2 copies for the foreground and these of course should be the same colour.

```
10  MODE 5
20  VDU 19, 3, 1, 0,0,0
25  GCOL 1,1
30  PROCdrawacircle(500, 500, 125)
40  GCOL 1, 2
50  PROCdrawatriangle(327, 400, 346)
60  END
```

To delete the red circle and reveal any previously hidden parts of the yellow triangle we can add:

```
60  keypress = GET
70  GCOL 2, 2
80  PROCdrawacircle(500, 500, 125)
```

which 'undraws' the cirele.

A convenient alternative to the above uses of GCOL 1 and GCOL 2 is often useful. Provided that an object being plotted in a plane does not overlap any object that is already present in that plane, then GCOL 3 can be used for

both the drawing and deleting process. For example to draw an object in image plane 1:

```
GCOL 3, 1
PLOTs etc to draw the object
```

To delete the object we simply repeat exactly the same GCOL and PLOT statements.

Composite image with priority (5 planes)

On the Model B, in MODE 2, we have 4 bit colour codes (16 colours) and this gives us many more possibilities. The next program is designed to illustrate one such possibility.

In this program, we have set up four planes plus background :

```
foreground (white)
midground (yellow)
reareground ( red)
distant (blue)
background (black)
```

The different colour codes for a pixel together with their significance are:

code	binary	actual colour	interpretation
1	0001	white	fore
3	0011	white	fore obscuring mid
5	0101	white	fore obscuring rear
7	0111	white	fore obscuring rear, mid
9	1001	white	fore obscuring distant
11	1011	white	fore obscuring distant, mid
13	1101	white	fore obscuring distant, rear
15	1111	white	fore obscuring distant, rear, mid
2	0010	yellow	mid
6	0110	yellow	mid obscuring rear
10	1010	yellow	mid obscuring distant
14	1110	yellow	mid obscuring distant, rear
4	0100	red	rear
12	1100	red	rear obscuring distant
8	1000	blue	distant
0	0000	black	background

Each bit in a colour code represents one of the four planes.

The GCOL 1 colour code for drawing contains a one in the bit position for the plane involved and zeros in the other bit positions. The GCOL 2 colour code for erasing contains a

zero bit for the plane in which erasing is taking place an ones for the planes that are to to be unaffected. The GCOL statements needed for drawing or erasing in each plan without affecting the other planes are:

	draw	erase
foreground	GCOL 1,1	GCOL 2,14
midground	GCOL 1,2	GCOL 2,13
reareground	GCOL 1,4	GCOL 2,11
distant	GCOL 1,8	GCOL 2, 7

The program repeatedly draws or erases a colour-filled circle, in a plane specified by the user and with centre an radius specified by the user. A plane is specified using on of the keys F(oreground), M(idground), R(earground) D(istant) or Q(uit). You should experiment with the program and observe how the above priority system works when drawing and erasing overlapping circles in different planes.

```
 10  MODE 2
 20  VDU 28, 0,1, 19,0
 30  VDU 24, 0;0; 1279;963;
 40  sin5=SIN(RAD(5)) : cos5=COS(RAD(5))
 50  VDU 19,  1,7, 0,0,0 : VDU 19, 3,7, 0,0,0
 60  VDU 19,  5,7, 0,0,0
 70  VDU 19,  9,7, 0,0,0 : VDU 19, 11,7, 0,0,0
 80  VDU 19, 13,7, 0,0,0 : VDU 19, 15,7, 0,0,0
 90  VDU 19,  2,3, 0,0,0 : VDU 19,  6,3, 0,0,0
100  VDU 19, 10,3, 0,0,0 : VDU 19, 14,3, 0,0,0
110  VDU 19,  4,1, 0,0,0 : VDU 19, 12,1, 0,0,0
120  VDU 19,  8,4, 0,0,0
130  REPEAT : PROCcommand : UNTIL plane$="Q"
140  MODE 7 : END

150  DEF PROCcommand
160    plane$ = FNcommand( "Which plane" ,"FMRDQ")
170    IF plane$="Q" THEN ENDPROC
180    PROCcirclespec
190    IF plane$="F" THEN PROCdraworerase(1,14)
200    IF plane$="M" THEN PROCdraworerase(2,13)
210    IF plane$="R" THEN PROCdraworerase(4,l1)
220    IF plane$="D" THEN PROCdraworerase(8,7)
230  ENDPROC

240  DEF FNcommand(type$,coms$)
250  LOCAL c$
260    PRINT type$; "(";coms$;")?";
270    REPEAT : c$=GET$ : UNTIL INSTR(coms$,c$)>0
280    CLS
290  =c$
```

```
300  DEF PROCcirclespec
310  INPUT "Centre(x,y)",cx,cy
320  INPUT "Radius",r
330  CLS
340  ENDPROC

350  DEF PROCdraworerase(drawcode,erasecode)
360  LOCAL op$
370    op$ = FNcommand("Draw or Erase","DE")
380    IF op$="D" THEN GCOL 1,drawcode
        ELSE GCOL 2,erasecode
390  PROCcircle
400  ENDPROC

410  DEF PROCcircle
420  LOCAL oldx,oldy
430    MOVE cx+r ,cy
440    oldx=r : o1dy=0
450    FOR t=5 TO 360 STEP 5
460      x=  oldx*cos5 + oldy*sin5
470      y= -oldx*sin5 + oldy*cos5
480      MOVE cx ,cy
490      PLOT 81,x,y
500      oldx=x : oldy=y
510    NEXT t
520  ENDPROC
```

Another possibility would be to have three priority levels plus background with:

```
3 foreground colours   (spaceships?)
1 midground colour     (planets?)
1 reairground colour   (stars?)
1 background colour    (sky?)
```

This is discussed in Exercise 5 below.

Exercises

1 Undraw a rectangle by working from the centre as if it were a stage curtain being pulled to each wing. Underneath detail should be revealed in another colour: legendry that might be used in a caption sequence or anything else you fancy.

2 Plot two pictures using values from DATA statements and then repeatedly read a key. Depending on which key was pressed, display the first picture or display the second picture or display both pictures at once (without redrawing them).

3 Certain patterns are used for testing for various types

of colour blindness. These consist of a pattern of dots containing a large letter or number made up of dots in one colour, the rest of the dots being in another colour. Write a program that generates VDU 19 statements to switch through a sequence of colour combinations.

4 Write a program that uses data to draw four graphs (on the same axes) representing four year's sales and then uses VDU 19 commands to display one, two, three or four of the graphs in response to keys pressed by a user.

5 Modify the four-plane demonstration program so that there are three levels of priority, foreground, midground and reareground, with a choice of three foreground colours. You could use the following settings for the colour codes:

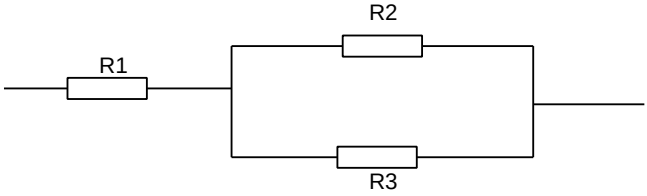
code	binary	actual colour	interpretation
1	0001	red	fore
2	0010	green	fore
3	0011	yellow	fore
5	0101	red	fore obscuring mid
6	0110	green	fore obscuring mid
7	0111	yellow	fore obscuring mid
9	1001	red	fore obscuring rear
10	1010	green	fore obscuring rear
11	1011	yellow	fore obscuring rear
13	1101	red	fore obscuring mid, rear
14	1110	green	fore obscuring mid, rear
15	1111	yellow	fore obscuring mid, rear
4	0100	blue	mid
12	1100	blue	mid obscuring rear
8	1000	magenta	rear
0	0000	black	background

You will find that to plot a foreground circle, you may first have to erase any existing foreground colour in the circle.

2.2 Basic interaction techniques (GCOL 3 and GCOL 4)

In this section two interaction techniques are implemented. Both of these use the keyboard, but clearly the principles are the same for either a keyboard or a more convenient device. Both interaction techniques can be used in picture

construction and this forms a part of most CAD (Computer Aided Design) systems. Such techniques enable designers to work in a two-dimensional or picture domain. This means for example that an electrical engineer can work with circuit diagrams and an architect with elevations or other projections of buildings, rather than just numbers. Now CAD techniques are an extensive topic by themselves and we shall only be concerned here with picture or line-drawing generation. It is not out of place to examine just briefly how such techniques 'fit in' to CAD programs. A CAD program that accepts a picture as input has to deduce certain information from it. An electrical engineer may draw a circuit diagram as input. A simple but somewhat unrealistic example serves to illustrate the point; say he inputs a series parallel resistor configuration:

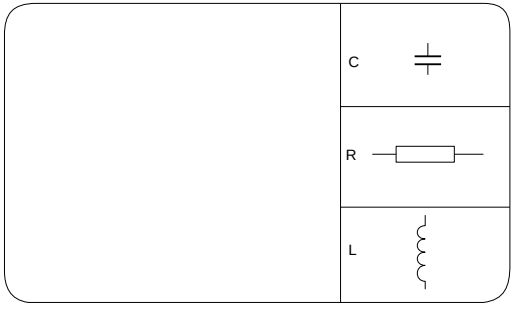


From this the CAD program will have to deduce that a resistor is connected in series to two resistors in parallel and that the total resistance is:

$$RT = R1 + R2*R3/(R2 + R3)$$

It can then evaluate numerical calculations and output required information graphically or otherwise back to the user. The CAD program will also be able to cope with alterations to the diagram - additions, deletions etc.

The circuit diagram could be built up using a technique known as 'picking and dragging'. A user is presented with a menu of objects and can pick a particular object and drag it to anywhere on the screen:



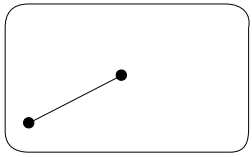
Other operations that might be available on objects are magnification and rotatation. Again in the case of an electrical circuit diagram, in parallel with the picture-drawing modules there will be procedures that keep track of the spatial relationship between components. The CAD program can then build up a formula reflecting some required attribute or behaviour of the circuit. This might be transfer characteristic, frequency response, etc. The computer program's view of the problem is numerical or formula based while the engineer's view remains pictorial. This is a tremendous advantage in most design problems.

In the same way an architect may sketch in the elevations of a house and ask for costing, insulation or sunlight calculations.

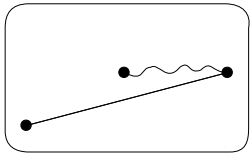
In the next two sections we look at the front end of such CAD programs firstly by looking at how we can sketch line drawings on the screen, and secondly how we can pick and drag predefined sub-pictures across the screen.

Rubberband line drawing

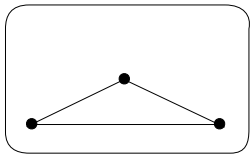
Using this technique we can build up a sketch or line drawing on the screen, using line segments whose length and direection are controlled from the keyboard. The program starts off by drawing an arbitrary line from (0,0) to (0,500). By using keys R, L, U, and D (Right, Left, Up and Down) as direction indicators we can move the end point of the line anywhere we want. Key F can be used to 'Fix' the end point of the line.



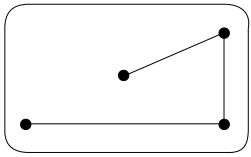
Start of program
Arbitrary line drawn from
(0,0) to (500,500)



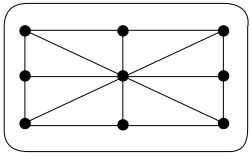
Line endpoint can be moved
anywhere from (500,500)



Key F depressed 2nd line arbitrarily drawn to (500,500) and 1st line permanently drawn



This line can be moved anywhere and key F depressed again



Thus any shape can be built up

Here is a program that illustrates a simple approach to 'rubberbanding' .

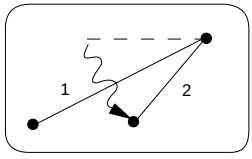
```
  10  MODE 4 : xstep =4 : ystep =4
  20  xs= 0 : ys= 0
  30  x= 640 : y = 512
  40  GCOL 3, 1
  50  PROCdrawordelete
  60  REPEAT
  70  command$ = GET$
  80  PROCprocesscommand
  90  UNTIL command$ = "Q"
 100  MODE 7 : END
```

```
110  DEF PROCprocesscommand
120  IF INSTR("FLRUD",command$)=0 THEN ENDPROC
130    PROCdrawordelete
140    IF command$ = "F" THEN PROCfix
150    IF command$ = "L" THEN x = x - xstep
160    IF command$ = "R" THEN x = x + xstep
170    IF command$ = "U" THEN y = y + ystep
180    IF command$ = "D" THEN y = y - ystep
190    PROCdrawordelete
200  ENDPROC

210  DEF PROCdrawordelete
220    MOVE xs, ys : DRAW x,y
230  ENDPROC

240  DEF PROCfix
250    REM Permanent draw
260    GCOL 0,1 : PROCdrawordelete
270    GCOL 3,1
280    xs = x : ys = y
290    x= 640 : y= 512
300  ENDPROC
```

'xs' and 'ys' always represent the start position of the line currently being drawn and 'x' and 'y' represent the position of the end of the line being moved. The program consists of a REPEAT loop that processes commands UNTIL the key Q (Quit) is typed.

PROCprocesscommand first checks for a valid key. It then calls PROCdrawordelete to delete the line in its current position. If "F" has been pressed, then the line currently being operated on is fixed and the coordinates are set for a new line. One of the coordinates x, y is updated if one of the movement keys (L, R, U, D) has been pressed. The coordinate increments, 'xstep' and 'ystep', are set to the dimensions of a pixel in the mode being used. PROCprocesssccamnand terminates by drawing a line to the position now specified by the x-y coordinates.

The critical statement in the program is GCOL 3,1 (exclusive OR). This means that lines can be moved over existing lines without permanently wiping part of them out, as would be the case without this facility.



Normally to delete an object we would re-plot the object in the background colour but this would wipe out intersecting parts of existing lines. Using the above method, an existing line disappears only momentarily while the current moving line passes over it. Thus line segment 2 (above) can be swept over existing line segment 1 without rubbing it out. This can be explained by reference to the following table.

	1st. DRAW			2nd. DRAW	
old	plotting	new	old	plotting	new
0	1	1	1	1	0
1	1	0	0	1	1

You can see from the bottom row of the table that plotting a 1 on top of a 1 in the first DRAW results in a zero that is restored to a 1 by the 2nd DRAW. The top row of the table gives the effect of a normal draw and erase function. The second DRAW thus erases or undraws, at the same time restoring any holes in existing lines made by the 1st DRAW. We leave it as an exercise to work out why the behaviour is unaltered if GCOL 3 is replaced by GCOL 4.

If you try using this simple program, you will find that it suffers from a number of disadvantages. In order to make it more useful as a line-drawing program, we need to make a number of improvements and extensions.

First, we will look at an improved program structure that will speed up the rubberbanding process. In the above program, when the end-point of the rubberband line is moved several times in the same direction, the line is deleted and redrawn for each intermediate position of the end-point. Using this approach would make a realistic CAD (Computer Aided Design) program unacceptably slow. The improved program structure below permits the user to hold down one of the movement keys and the end-point of the rubberband line is moved in one step by an amount that depends on the length of time for which the key is pressed. The line is deleted and redrawn only once to effect the complete move.

```
 10  MODE 4 : xstep = 4 : ystep = 4
 20  xs = 0 : ys =0
 30  x= 640 : y = 512
 40  GCOL 3, 1
 50  PROCdrawordelete
 60  *FX 11,10
 70  *FX 12,1
 80  command$=GET$
 90  REPEAT
100  PROCprocesscommand
110  UNTIL command$ = "Q"
120  *FX 12, 0
130  MODE 7 : END
```

```
140  DEF PROCprocesscommand
150  PROCcountcoms
160  IF INSTR("FLRUD",comand$) = 0 THEN
       command$=GET$ : ENDPROC
170  PROCdrawordelete
180  IF command$ = "F" THEN PROCfix
190  IF command$ = "L" THEN x = x - xstep*coms
200  IF command$ = "R" THEN x = x + xstep*ccms
210  IF command$ = "U" THEN y = y + ystep*coms
220  IF command$ = "D" THEN y = y - ystep*coms
230  PROCdrawordelete
240  IF nextcom$="" THEN command$=GET$
      ELSE command$=nextcom$
250  ENDPROC

260  DEF PROCcountcoms
270    coms=0
280    REPEAT : coms=coms+1 : nextcom$=INKEY$(11)
290    UNTIL nextcom$<>command$
300  ENDPROC

310  DEF PROCdrawordelete
320    MOVE xs, ys : DRAW x, y
330  ENDPROC

340  DEF PROCfix
350    REM Permanent draw
360    GCOL 0,1 : PROCdrawordelete
370    GCOL 3,1
380    xs = x : ys = y
390    x = 640 : y = 512
400  ENDPROC
```

The *FX commands at lines 60 and 70 are used to increase the sensitivity of the keys. *FX 11 sets the delay before repeated copies of a character are sent to the computer by a continually depressed key. *FX 12 sets the delay between subsequent repeats of a character. (Each of these 'operating system commands' must appear on a separate numbered line.)

```
*FX 11,10
```

means that if a key is pressed for less than 10 hundredths of a second, then only one character is sent by that key.

```
*FX 12,1
```

now means that if a key is pressed for more than 10 hundredths of a second, then repeated copies of the character are sent by the key every 1 hundredth of a second.

The program enters PROCprocesscommand having read the next commmand character. PROCcountcoms is then used to count

any repeats of the command character in case the command key is being held down. If the command key is a movement key, then this count is used in changing x or y by an appropriate multiple of the basic increment.

PROCprocesscommand terminates (at line 160 or line 240) by ensuring that 'command$' has been set to the next command character, using GET$ if necessary, ready for the next execution of the main loop.

Now, even although we have speeded it up, the program is still slightly impractical - figures are constructed without the 'pen being lifted off the paper'. That is to say after a line is fixed it is assumed that another line is required. This may not be the case and the easiest way to incorporate a line on/off facility is to have another key controlling this option:

```
221  IF command$ = "0" THEN lineoff = NOT lineoff
```

This IF statement sets up a 'push on/push off' key - a mechanism that we shall use again. Whenever we introduce an extra command key, we must extend the string of permitted commands:

```
160  IF INSTR( "FLRUDO",command$)=0 THEN
       command$=GET$ : ENDPROC
```

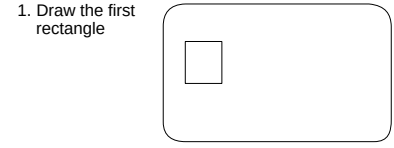
The variable 'lineoff' is originally set to FALSE:

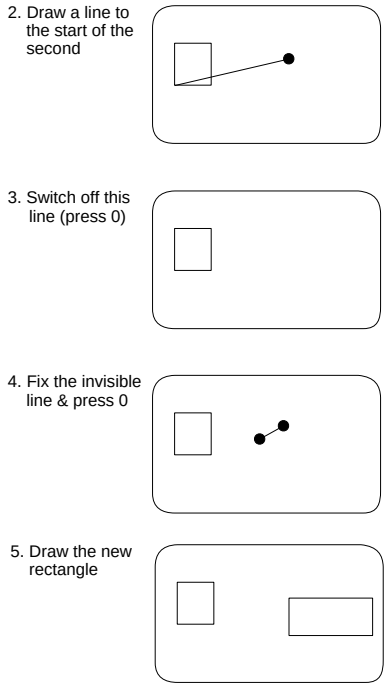
```
35 lineoff=FALSE
```

Pressing the appropriate key will change its value from FALSE to TRUE or vice versa. PROCdrawordelete can then be:

```
310  DEF PROCdrawordelete
315    IF lineoff THEN ENDPROC
320    MOVE xs, ys : DRAW x, y
330  ENDPROC
```

which prevents the drawing action if the line is switched off. Now, for example, to construct two isolated rectangles we would:





Rubberband drawing aids

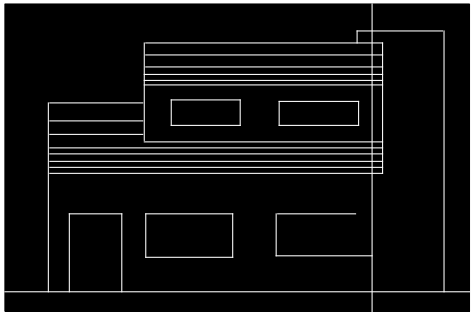
There are two useful elaborations that we can make to our rubberband line drawing program. Firstly we can include a horizontal and vertical cursor line to enable us to line up different parts of a drawing. This simply adds another two selections to PROCprocesscommand:

```
 36  hcursor = FALSE : vcursor = FALSE

160  IF INSTR("FLRUDOHV",command$)=0 THEN
       command$=GET$ : ENDPROC

225  IF command$ = "H" THEN hcursor = NOT hcursor
226  IF command$ = "V" THEN vcursor = NOT vcursor
```

This means that the H and V key functions are also pushon/ push off keys. PROCprocesscommand can now be further elaborated to check if cursors have to be drawn:

```
170  PROCdraworde1ete : PROCcheckcursors

230  PROCdrawordelete : PROCcheckcursors

410  DEF PROCcheckcursors
420    IF hcursor THEN MOVE 0,y:DRAW 1279,y
430    IF vcursor THEN MOVE x,0:DRAW x,1023
440  ENDPROC
```

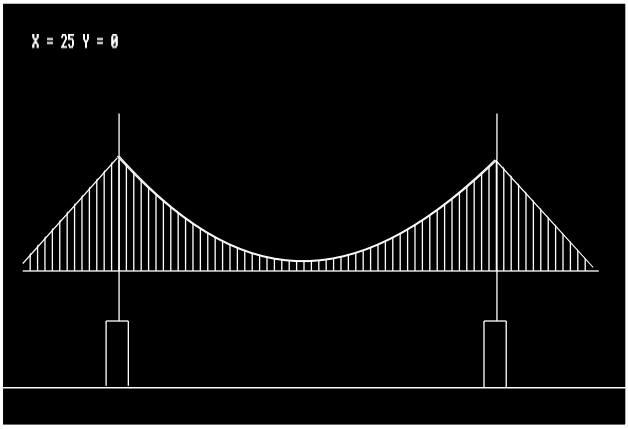
The next photograph shows the cursor being used in the course of a construction.



Another useful aid is a length measuring device that indicates the current length of a line. Consider for example measuring the current x projection of the line:

```
 37  printmeasure = FALSE

160  IF INSTR("FLRUDOHVM",command$ )=0 THEN
        command$=GET$ : ENDPROC

227  IF command$ = "M" THEN
        printmeasure = NOT printmeasure

235  PROCmeasure

450  DEF PROCmeasure
460  IF printmeasure THEN
        PRINT TAB(3,3); ABS(xs-x)
      ELSE PRINT TAB(3,3); SPC(4)
470  ENDPROC
```

If the measure option is switched on then PROCmeasure is obeyed and prints the current x projection of the line. In the next illlustration the hangers on the suspension bridge were accurately positioned using this facility.



Picking and dragging an object

We have already mentioned the use of this particular technique above so we'll jump straight in to doing it. In the next program we have set up a menu of objects in the right hand side of the screen. An object is selected by typing 1, 2 or 3. In practice, if we were using this technique frequently, an object would be selected from the menu by pointing a light pen at the appropriate position on the screen. When an object is selected it is dragged into transition and fixed as before. Instead of dragging a line we are now dragging a complete object.

```
  10  MODE 0 : xstep = 2 : ystep = 4
  20  PROCdrawmenu
  30  GCOL 3, 1
  40  PROCpick
  50  REPEAT
  60    x=100 : y=100
  70    PROCdrawordelete(selection$)
  80    *FX 11,10
  90    *FX 12,1
 100    command$=GET$
 110    fixed = FALSE
 120    REPEAT
 130      PROCprocesscommand
 140    UNTIL fixed
 150    *FX 12,0
 160    PROCpick
 170  UNTIL selection$ = "Q"
 180  MODE 7 : END
```

```
190  DEF PROCdrawmenu
200  MOVE 900,0 : DRAW 900,1000
210  PROCdrawresistor(1000,600)
220  PROCdrawcapacitor(1000,400)
230  PROCdrawdiode(1000,200)
240  PRINT TAB(60,12);"1"; TAB(60,18);"2";
     TAB(60,24);"3"
250  ENDPROC

260  DEF PROCpick
270    PRINT TAB(0,0);"Pick, (1/2/3/Q)";
280    REPEAT : selection$=GET$
290    UNTIL INSTR("123Q",selection$)>0
300    PRINTTAB(0,0);"                ";
310  ENDPROC

320  DEF PROCprocesscommand
330    PROCcountcoms
340    IF INSTR( "FLRUD",command$)=0 THEN
        command$=GET$ : ENDPROC
350    PROCdrawordelete(selection$)
360    IF command$="F" THEN
         PROCfix : fixed=TRUE : ENDPROC
370    IF command$="L" THEN x = x-xstep*coms
380    IF command$="R" THEN x = x+xstep*coms
390    IF command$="U" THEN y = y+ystep*coms
400    IF ccmmand$="D" THEN y = y-ystep*coms
410    PROCdrawordelete(selection$)
420    IF nextcom="" THEN command$=GET$
         ELSE command$=nextcom$
430  ENDPROC

440  DEF PROCcountcoms
450    coms=0
460    REPEAT : coms=coms+1 : nextcom$=INKEY$(11)
470    UNTIL command$<> nextcom$
480  ENDPROC

490  DEF PROCfix
500    GCOL 0,1:PROCdrawordelete(selection$)
510    GCOL 3,1
520  ENDPROC

530  DEF PROCdrawordelete(s$)
540    IF s$="1" THEN PROCdrawresistor(x,y)
550    IF s$="2" THEN PROCdrawcapacitor(x,y)
560    IF s$="3" THEN PROCdrawdiode(x,y)
570  ENDPROC
```

```
580   DEF PROCdrawresistor(x,y)
590     MOVE x,y : PLOT 1,30,0
600     PLOT 1,0,10 : PLOT 1,60,0
610     PLOT 1,0,-20 : PLOT 1,-60,0
620     PLOT 1,0,10 : PLOT 0,60,0
630     PLOT 1,30,0
640   ENDPROC

650   DEF PROCdrawcapacitor(x,y)
660     MOVE x,y : PLOT 1 ,30 ,0
670     PLOT 0, 0, -30 : PLOT 1 ,0 ,60
680     PLOT 0,20,0 : PLOT 1,0 ,-60
690     PLOT 0,0,30 : PLOT 1,30,0
700   ENDPROC

710   DEF PROCdrawdiode(x,y)
720     MOVE x,y : PLOT 1 ,30,0
730     PLOT 0,0, -25 : PLOT 1,0,50
740     PLOT 1,25,-25 : PLOT 1,-25,-25
750     PLOT 0,25,25 : PLOT 1,30,0
760   ENDPROC
```

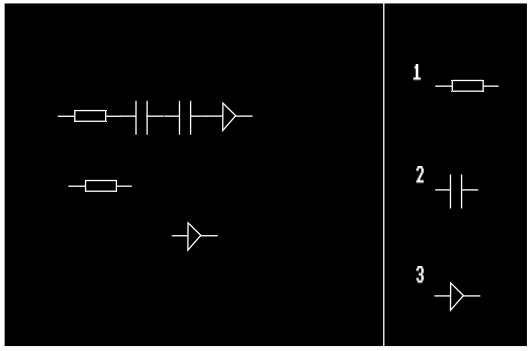
The program to drag an object is identical to the rubberband proyram with

```
PROCCdrawordelete
```

replaced by:

```
PROCdrawordelete(selection$)
```

This procedure selects one out of the three drawing procedures and the selected object is drawn at a position under control of the directional keys. The next illustration shows the screen during execution of the above program.



Scaling and rotating a dragged object

Other common facilities in picking and dragging programs are magnification and rotation. For example in the above dragging program, another key option could be "M" for 'Magnify' and T (turn) for rotation. The structural alterations now required in the program are significant. In particular we have to change the way in which we store shape information. Currently this information is embedded in the drawing procedures as parameters of the PLOT 1 statement. The most convenient scheme is to store the current displacement coordinate values for an object in an array. These displacements will of course change as a function of the angle of rotation. Initially we could set up an array for a square, for example, as:

```
   squarex(1)     100        squarey(1)        0
          (2)       0                (2)      100
          (3)    -100                (3)        0
```

To draw the square in any (dragged) position (x,y) we need:

```
  570  DEF PROCdrawsquare(x, y)
  580    MOVE x, y
  590    FOR i = 1 TO 3
  600    PLOT 1, squarex(i), squarey(i)
  610    NEXT i
  620    DRAW x, y
  630  ENDPROC
```

This is the same scheme as we have in the component drawing procedures (above) except that we are now storing the displacements in an array. Now to rotate an object we would press T (turn) and make the object rotate by a predetermined angular increment of, say, 10 degrees by altering the relative displacements. To do this we simply use a standard two-dimensional rotation transform (see Chapter 3):

```
  800  DEF PROCrotate
  810  LOCAL x,y
  820    sintheta = SIN(RAD(10))
  830    costheta = COS(RAD(10))
  840    FOR i = 1 TO 3
  850      x = x(i) : y = y(i)
  860      x(i) = x*costheta + y*sintheta
  870      y(i) = -x*sintheta+ y*costheta
  880    NEXT i
  890  ENDPROC
```

Each time the key is depressed new displacements are

calculated {previous. Note that the figure is stationary while it is being rotated; it cannot be rotated and dragged at the same time.

Saving a line drawing

An image that has been created by rubberbanding can be saved as a list of coordinates and subsequently regenerated by a simple program reading the coordinates from a file and using DRAW. The coordinates can be saved initially in two parallel arrays and when the drawing is complete, the array contents dumped into a file. The coordinate saving should clearly be part of the 'fixing' process:

```
340  DEF PROCfix
350    REM Permanent draw
360    GCOL 0 ,1 : PROCdraworde1ete
370    GCOL 3, 1
375    line = line + 1
376    xcoord(line) = x
377    ycoord(line) = y
380    xs = x : ys = y
390    x = 640 : y = 512
400  ENDPROC
```

Similarly an image that has been created by picking and dragging an object can be saved, most economically, using three parallel arrays. The program would store, for each object, a pair of coordinates followed by a code indicating the class of object drawn at that position. The program would terminate by outputting the contents of the three arrays to a file. The regenerating program would contain the object-generating procedures again called from a shape selection procedure, the appropriate procedure for each shape being selected according to the stored code.

Exercises

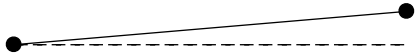
1 Improve the rubberband program so that the start coordinate is input from the keyboard.

2 Introduce colour so that the fixed lines are displayed in one colour, but the moving line appears in a contrasting colour.

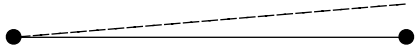
3 Consult the User Guide and change the rubberband program so that the cursor arrow keys are used for controlling the movement of the endpoints of the line.

4 Write a rubberband program where the permanent lines are to be constrained to the horizontal or vertical direction. For example an imperfectly drawn horizontal

line:

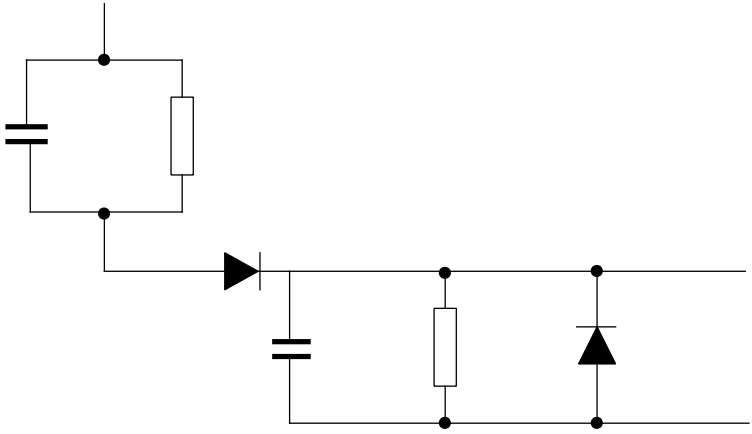


is to be corrected to a perfect horizontal line:



5 Write a picking and dragging program that picks either a hexagon or an equilateral triangle and drags it to a required position, fixes it there and colours it in a colour that is selected by another key. The hexagon and triangle should each have the same length of side so that they can be fitted together.

6 Write a picking and dragging program that will allow such diagrams as the following to be constructed: Note that this will have to contain both object dragging and rotation (0 or 90 degrees only) as well as rubberband line drawing.



7 Incorporate the picture-filing suggestions in your programs.

2.3 Co1our-fi11 - general algorithms

The triangular fill facility (PLOT 80 to 87) is generally inconvenient in interactive graphics. In particular figures containing interior holes or concavities are difficult to fill using this method. Also if we are drawing a region

outline using a light pen or graphics tablet it is inconvenient to store the coordinates of pixels on the outline. We require a general algorithm that will fill any region already delineated on the screen. Algorithms that fill the interior of any closed figure sometimes called 'flood-fill' algorithms and they work assuming that the region to be filled is delineated by a boundary of pixels in a non-background colour and that the interior of the region is '4-connected'. This means that all pixels within the region can be reached one from the other by a sequence of any of the movements up, down, left and right.

There are two approaches that we can make to this problem: one is recursive and is described in Chapter 7; the other is non-recursive and is now described. The algorithm below is extremely slow, but it provides a good introduction to the ideas involved. This algorithm uses a FIFO (first in, first out) buffer or queue. A program that fills the area enclosed by two concentric circles is now given.

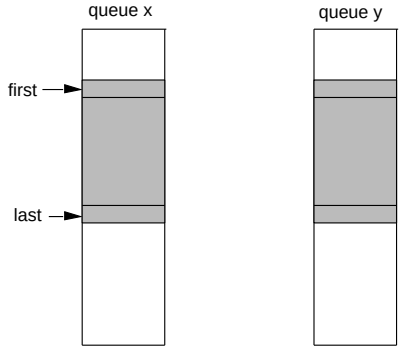
```
 10  INPUT "RADII",r1,r2
 20  MODE 1
 30  GCOL 0,1
 40  PROCcircle(r1,640,512)
 50  PROCcircle(r2,640,512)
 60  PROCfillfrom(640+(r1+r2)/2,512)
 70  END

 90  DEF PROCcircle(r,xc,yc)
100  LOCAL t
110    MOVE xc+r,yc
120    FOR t=10 TO 360 STEP 10
130      DRAW xc+r*COS(RAD(t)),yc+r*SIN(RAD(t))
140    NEXT t
150  ENDPROC

200  DEF PROCfillfrom(startx,starty)
210  DIM queuex(500), queuey(500)
220    first=1 : last=0
230    PROCfill(startx,starty)
240    REPEAT
250      PROCunqueue
260      PROCfi1l(x,y+4)
270      PROCfill(x,y-4)
280      PROCfill(x+4,y)
290      PROCfill(x-4,y)
300    UNTIL first=(last+1) MOD 500
310  ENDPROC
```

```
  330  DEF PROCfill(x,y)
  340    IF POINT(x,y)>0 THEN ENDPROC
  350    PLOT 69,x,y
  360    PROCqueue(x,y)
  370  ENDPROC

  390  DEF PROCqueue(x,y)
  400    last=(last+1)MOD500
  410    queuex(last)=x
  420    queuey(last)=y
  430  ENDPROC

  450  DEF PROCunqueue
  460    x=queuex(first)
  470    y=queuey(first)
  480    first=(first+1)MOD500
  490  ENDPROC
```

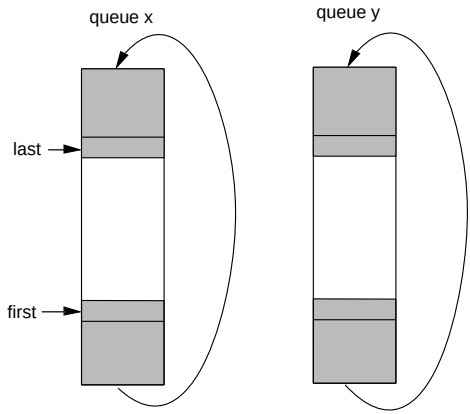
PROCfillfrom is initiated from a start point and that start point is coloured and added to a queue (by calling PROCfilll. PROCfillfrom then repeatedly takes the first point from the queue and examines each of the neighbouring N, S, E and W points (by calling PROCfill for each of these points in turn). Each time PROCfil1 is called, it colours the point it is given (if it is not already coloured) and adds that point to the end of the queue. Adding a point to the queue in this way ensures that it will subsequently be removed from the queue and its neighbours examined.

The reason the queue is made a FIFO is to prevent it becoming too large. If for example we made the queue an ordinary stack (LIFO or last in first out), as you may see suggested in computer graphics textbooks, it would gradually fill up and would run out of memory.

For the queue, we use two arrays, one for x-coordinates and one for y-coordinates. Two variables indicate the positions of the 'first' and 'last' items in the queue.



The arrays are treated as circular so that when the end of the queue reaches reaches the end of the arrays, the queue is 'wrapped around' and continues into the space that is now free at the start of the arrays.

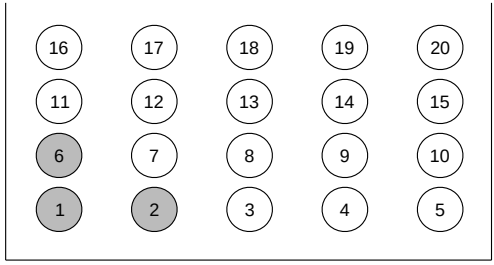


PROCfillfrom repeatedly takes the next point from the queue until the queue is empty. The photograph shows the algorithm in the course of filling. Note that the 'wavefronts' are diagonal. This is a consequence of using a FIFO queue in this particular context.

An illustrative sequence of how the algorithm works in detail is now given for a simple rectangular region. The start point is the bottom left hand corner.

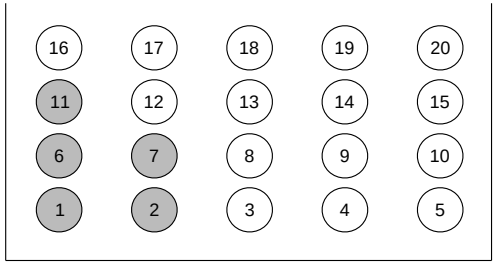
pixel 1 is filled and added to the queue

1st cycle of REPEAT loop in PROCfillfrom
pixel 1 is removed from queue and neighbouring points examined



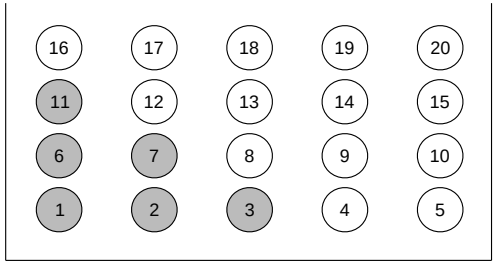
queue is now 6, 2
pixels 6 and 2 are filled

2nd cycle, pixel 6 removed and neighbours examined.



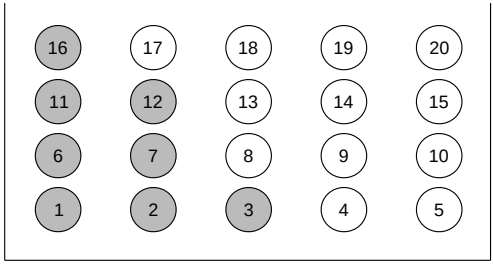
queue is now 2, 11, 7
pixels 11 and 7 are also filled.

3rd cycle, pixel 2 removed and neighbours examined.



queue is now 11, 7, 3
pixel 3 is filled

4th cycle, pixel 11 removed and neighbours examined.



queue is now 7, 3, 16, 12
pixels 16 and 12 are filled

This sequence continues until the queue is empty.

Later versions of the Operating System provide a PLOT command for horizontal filling of a row of pixels up to a boundary. It is convenient to postpone discussion of this facility until Chapter 7 (recursion).

Exercises

1 Draw a checker board or games board pattern using colour-fill.

2 As an aid to understanding the queue fill algorithm, build up on paper a sequence showing a shape being filled from a central point.

3 Write a rubberbanding program that includes a paint option for colouring the region containing the current point.