Working Paper Series
ISSN 1170-487X

Drawing diagrams in TEX documents

by William J. Rogers & Geoffrey Holmes

Working Paper 92/1

January, 1992

Department of Computer Science
The University of Waikato
Private Bag 3105
Hamilton, New Zealand

Drawing Diagrams in TEX* Documents

W. J. Rogers and G. Holmes

Department of Computer Science

University of Waikato

Abstract

The typesetting language TEX [Knuth (1984)] is now available on a range of computers from mainframes to micros. It has an unequalled ability to typeset mathematical text, with its many formatting features and fonts. TEX has facilities for drawing diagrams using the packages *tpic* and PiCTEX. This paper describes a TEX preprocessor written in Pascal which allows a programmer to embed diagrams in TEX documents. These diagrams may involve straight or curved lines and labelling text. The package is provided for people who either do not have access to *tpic* or PiCTEX or who prefer to program in Pascal.

e-mail : w.rogers or g.holmes@waikato.ac.nz

* TEX is a trademark of the American Mathematical Society.

1. Introduction

It is frequently the case, when typesetting a document in TEX that a diagram is required. Such diagrams should be nested neatly in surrounding text. If the diagram, as well as the surrounding text is specified in TEX, then all the usual advantages of automatic typesetting are available: automatic reformatting after text (or diagram) changes; aesthetically pleasing page layout; etc.

In order to make the production of diagrams simple we provide a higher level drawing abstraction, designed to meet the needs of an illustrator. Our experiences with programming in TEX lead us to build a preprocessor for diagram construction in Pascal. The preprocessor provides a drawing abstraction as a series of procedures, and permits an illustrator to write Pascal programs built on this abstraction, to draw diagrams. The preprocessor translates the actions of the program into calls on a simple set of TEX macros. Its output takes the form of a file of TEX commands which can be "input" into any TEX document. It might have been possible to design a WYSIWYG drawing package which outputs TEX code, however the approach taken has the advantage of simplicity, and suited our application - the typesetting of a programming text which makes extensive use of graphics in example programs.

2. Drawing Abstraction

The drawing abstraction, Kiwi graphics - developed for teaching purposes [Hopper *et al* (1991)] - is loosely based on the LOGO Turtle Graphics system [Goodyear (1984)]. The drawing surface is an abstract piece of graph paper, with scales in millimetres. Diagrams are constructed over the first and fourth quadrant of the paper. For the purposes of TEX, the diagram is placed into an "hbox", stretching horizontally from the origin, to the diagram's right boundary, and vertically over whatever coordinate range is required. This range should not, of course, exceed the height set for the page. The 'Leonardo' diagram below indicates the quadrants used. The dotted lines indicate the extent of the "hbox".

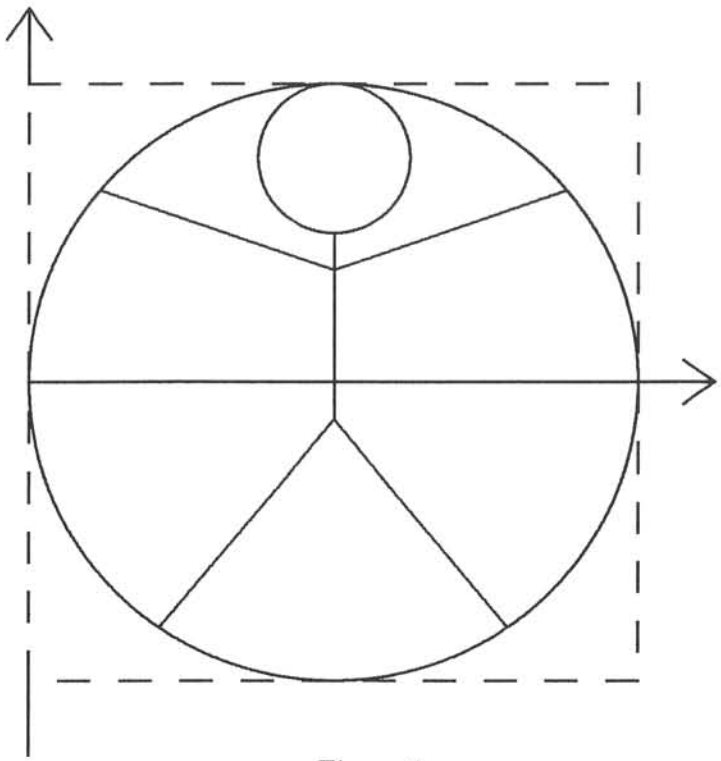

Figure 1

Drawing is achieved by moving an abstract pen, which may be raised above the surface using the routine `Up`, or lowered using the routine `Down` to leave a mark on the drawing surface as it moves.

A complete program takes the following form.

```
program example(...);
        declarations of constants, variables, and routines
      which implement the drawing abstraction
        declarations of routines, etc., necessary for user program
begin
  Start;   {prompt for file name, open file, initialise pen etc}
   drawing commands
  Finish   {complete the TeX file, and close it}
end.
```

The pen has a current position and orientation. Its orientation may be set using the `Direction` routine. For example:

```
Direction(25 {degrees})
```

sets the current orientation to 25 degrees clockwise from the positive x axis. Relative changes in orientation may be made using the `Turn` routine. For example:

```
Turn(10 {degrees})
```

will add 10 degrees to the current orientation.

There are two styles of pen movement. The first uses the current orientation. The `Follow` command moves the pen through a specified distance, in a straight line with that orientation, (drawing if the pen is down) starting from the current position. For example:

```
Follow(20 {millimetres})
```

After such a call the current position is updated. If a curved line is required, the command `Curve` should be used. This also moves from the current position, starting in the current orientation, but following a curved path, being a portion of an ellipse with specified radii, for the specified change in orientation. For example, starting at the bottom left and heading east, the command:

```
Curve(140 {degrees}, 30 {semi-major axis}, 10 {semi-minor axis})
```

produces the following curve:

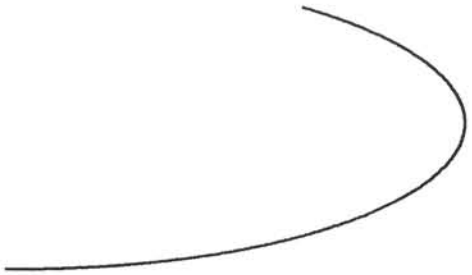

Figure 2

The large (20mm radius) circle in the first diagram above is achieved by

```
Curve(360 {degrees}, 20, 20 {millimetres})
```

The second style of pen movement is to specify rectilinear coordinates. The `Position` routine allows absolute positioning of the pen on the paper.

```
Position(100, 50 {millimetres})
```

moves in a straight line (drawing if the pen is down) from the current position to the point specified by the coordinate pair (100,50). The current position is updated by this action, but the current orientation remains unchanged. A relative move is also provided, again using rectilinear coordinates.

```
Offset(10, 20 {millimetres})
```

moves the pen 10mm horizontally and 20mm vertically from the current position, updating the current position when done.

3. Labels

Diagrams need to be labelled, and the appearance of a diagram is greatly enhanced by careful placement of labels. The human eye is very sensitive to misalignment of picture components. Accordingly a relatively complex labelling facility has been provided. The `Labelit` routine has

three parameters. The first is the label string. The second specifies horizontal alignment, and the third specifies vertical alignment.

For the label string, our implementation makes use of the DEC VAX Pascal conformant array facility. Calls with literal strings are then possible. For example:

```
Labelit( 'Hello World', centre, centre )
```

will centre the given string, both horizontally and vertically, about the current pen position. If there is a need to construct a label dynamically, then an array may be passed as the label parameter, either of the correct length, or with the required text terminated by a '~' (tilde) character.

The label string is not interpreted by the preprocessor, and so may contain any TEX commands in addition to text. Particularly useful is the option of setting the font, or font size for a label (see the **Shapes** example later). In fact the label command can be used to output TEX commands only, by including no displayable text in the string. On a Pascal implementation without conformant arrays, a fixed length array could be used, with the tilde character being used to delimit text.

The two alignment parameters of `Labelit` control positioning of text relative to the current pen position. The options for horizontal alignment are:

Figure 3

where the arrows show the current horizontal pen position. The options for vertical alignment are

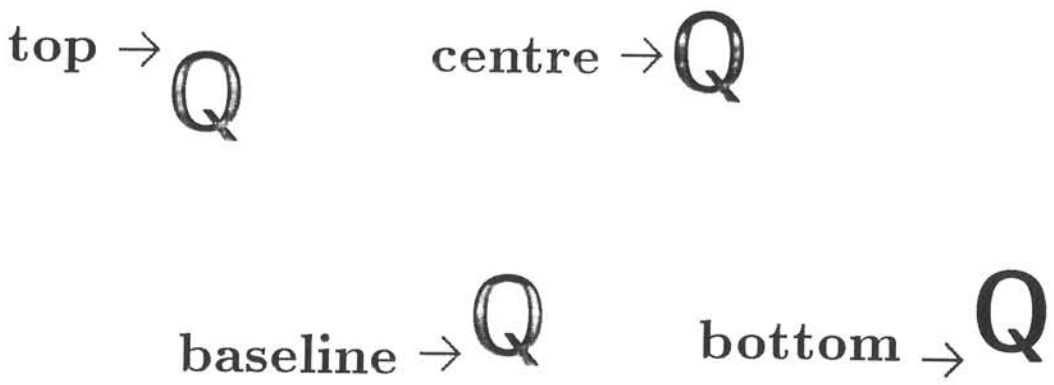

Figure 4

The position and orientation of the Pen are not altered by the `Labelit` routine.

4. Housekeeping

The procedures `Start` and `Finish` must be used as shown in the program outline given earlier. `Start` initialises the Pen position to (0,0), its orientation to 0 degrees (facing east), and leaves the pen raised off the surface of the paper.

Ideally, in software of this kind, there should be no magic parameters to adjust. Whilst provision of myriad controls may leave the author of a piece of software feeling that the maximum flexibility has been given to the user, in reality this is often asking the user to make decisions which would better have been made by the author of the software. We have provided only a single control - that of line thickness. The default is 1 point (1/72th of an inch). All the diagrams in this paper have used the default. However, the routine `Thickness` can be used to change the line thickness. The new value is used until another call to `Thickness` or a call to `Start`. For example, to halve the default line thickness:

```
Thickness(0.5 {point})
```

The line thickness control has no effect on labels. To alter the size or font used it is necessary to add TEX commands to the labels themselves (see examples). To further reduce the pain of choice, three constants are provided for use as parameters to `Thickness`:

```
thin, regular, thick
```

which should accomodate most needs. For example

```
Thickness( regular )
```

sets line thickness to the default value.

5. Implementation

The preprocessor code is (hopefully) self explanatory. The only tricks to explain are those involved in forcing TEX to draw. In TEX terms the diagram is mostly a sequence of zero width objects. Each zero width object is: a kern to the right, to position the item being drawn; the object itself; and a kern back to the origin to ensure that the width of the object is zero. TEX keeps track of the vertical size of the diagram by itself. The horizontal size that TEX is allowed to know of is established at the end with a single kern to the rightmost extent of the diagram. Objects are of four types: text for labels; horizontal lines; vertical lines; and blobs. The first three need no further explanation.

The production of diagonal lines and curves is achieved using the 'blob'. A blob is just an ink square of side equal to the current line thickness. The TEX macro `\drawline` builds diagonal lines out of overlapping blobs. The spacing of blobs is controlled by the TEX variable `\often`, which is roughly the number of tenths of a millimeter between blob centres.

Curves are built from blobs by the Pascal code. The preprocessor issues as many TEX calls to the blob macro as are necessary. The constant `Often` in the Pascal program is again roughly the number of tenths of a millimeter between blob centres.

The software has been built as simply as possible to ensure simplicity of use and clarity of presentation. Consequently it leads to TEX files which can take some time to process, and which sometimes exceed TEX's capacities. To those inclined to criticise the software for these foibles we ask only that they consider the effort required (which is considerable) to program TEX directly and efficiently (measured in machine terms) to draw diagrams, and then make that effort if they feel it worthwhile.

6. Examples

We present three examples to illustrate the use of the drawing abstraction. The first and third examples produce aesthetically pleasing diagrams from relatively short Pascal programs. We challenge anyone to produce these diagrams in TEX directly within the time taken to write the corresponding Pascal code! The second example is more typical of diagrams in documents, and illustrates the use of the `Labelit` feature using a variety of fonts.

Fractal Curves

Using recursion in graphics returns extremely impressive diagrams from very small programs. The fractal curve is a case in point [Rankin (1989)]. The code to produce the following diagram:

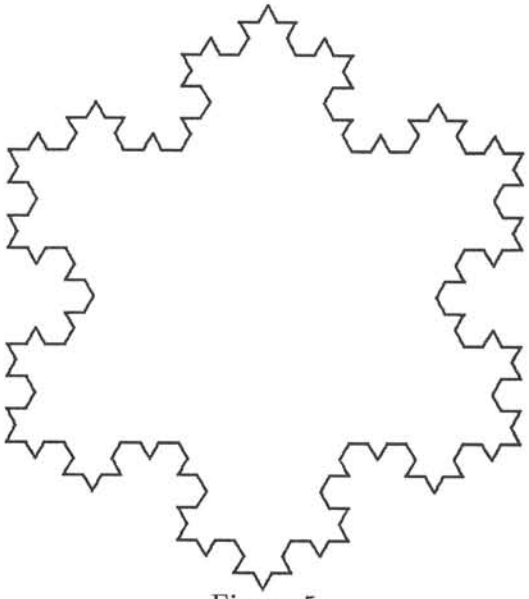

Figure 5

is the following:

```
program snowflake;
   procedure edge(side:  real;  level:  integer);
   begin
      if level <=1 then Follow(side) else begin
         edge(side/3.0,level-1);
         Turn(60);
         edge(side/3.0,level-1);
         Turn(-120);
         edge(side/3.0,level-1);
         Turn(60);
         edge(side/3.0,level-1);
         end
   end;
   procedure threeedges;
   begin
      edge(45,4);
      Turn(-120);
      edge(45,4);
      Turn(-120);
      edge(45,4)
   end;
begin
   Start;
   Down;
   threeedges;
   Finish
end.
```

Code Example 1

Spiral

The Pascal code for the second example is particularly pleasing in its brevity:

```
program spiral;
const
   turns = 3.25; radius = 20;
   incrangle = 10; {degrees}
   iterations = turns * 360 / incrangle;
   incrsize = radius / iterations;
```

```
var i :  integer;
begin
   Start;
   Position(radius, 0);
   Down;
   for i := 1 to round(iterations) do
      Curve(incrangle,i*incrsize,i*incrsize);
   Finish
end.
```

Code Example 2

It produces the following spiral curve:

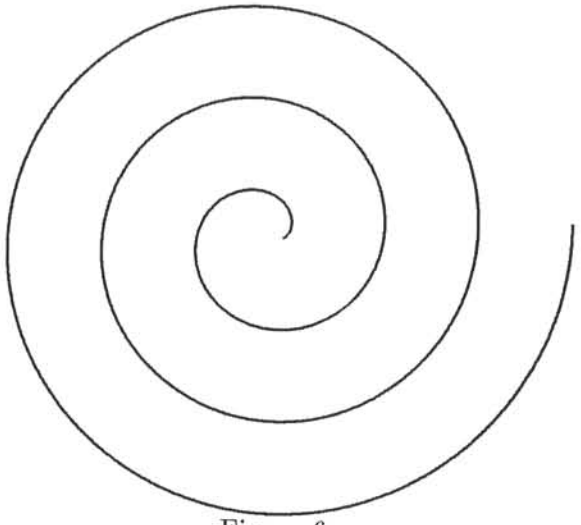

Figure 6

To draw this diagram by hand using a mouse would require a great deal of time and expertise. The time taken to write the Pascal code, however, is minimal.

Shapes and Labels

Many diagrams require labelled boxes. The following program produces a number of interconnected boxes with labels in different fonts.

```
program shapes;
const goleft = 90;
      side = 30;
  procedure rectangle;
  begin
     Position(5,45); Direction(0);
     Down;
     Follow(side); Turn(goleft); Follow(side); Turn(goleft);
     Follow(side); Turn(goleft); Follow(side);
     Up;
     Position(20,60); labelit('\rm Rectangle', centre, centre)
  end;
  procedure triangle;
  begin
     Position(65,45); Direction(0);
     Down;
     Follow(side); Turn(120); Follow(side); Turn(120); Follow(side);
     Up;
```

```
   Position(80,60); labelit('\it Triangle', centre, top)
end;

procedure circle;
const radius = 15;
begin
   Position(20,5); Direction(0);
   Down;
   curve(360, radius, radius);
   Up;
   Position(20, 5 + radius); labelit('\sl Circle', centre, centre)
end;

procedure ellipse;
const major = 20;
minor = 15;
begin
   Position(75,5); Direction(0);
   Down;
   curve(360, major, minor);
   Up;
   Position(75, 5 +  minor); labelit('\bf Ellipse', centre, centre)
end;

procedure arrowhead;
begin
   Turn(135); Follow(2.5);
   Up;
   Turn(180); Follow(2.5); Turn(-90);
   Down;
   Follow(2.5);
   Up;
   Turn(180); Follow(2.5); Turn(-45)
end;

procedure joinup;
begin
   Position(35,60); Direction(0); {rectangle to triangle}
   Down;
   Follow(38); arrowhead;
   Position(35,60); Direction(-45); {rectangle to ellipse}
   Down;
   Follow(40); arrowhead;
   Position(20,45); Direction(-90); {rectangle to circle}
   Down;
   Follow(10); arrowhead;
   Up;
   Direction(90); Follow(10); arrowhead {circle to rectangle}
end;

begin
   Start;
   rectangle; triangle; circle; ellipse; joinup;
   Finish
end.
```

Code Example 3

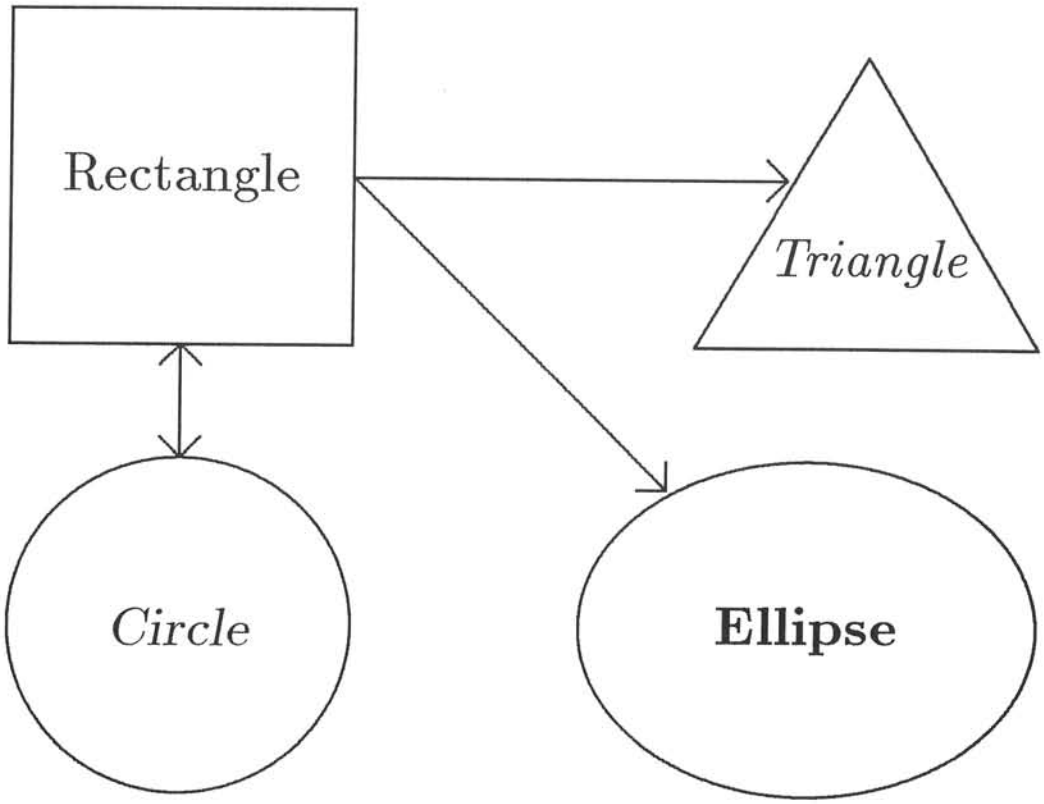

Figure 7

7. Conclusions

This paper presents a Pascal version of the preprocessor, in the expectation that the widespread availability of Pascal compilers will make this the most widely useful form of the program. We have also completed a version in Modula-2, a language much better suited than Pascal for the implementation of abstractions.

The manner of implementation of this software has left it with some faults. The worst of these is that the Pascal software cannot determine the size of label text. This means that diagrams cannot be sized about the text they contain. We have found that this is not a serious problem. Occasionally it necessitates an extra test run of a diagram to choose the best dimensions. In addition, text hanging out to the right of a diagram is not included in the diagram "hbox", thus leading to errors in positioning whole diagrams. The only cure for that is not to leave text hanging out to the right. Finally, the drawing pen has a square "nib". This means that diagonals are thicker than horizontal or vertical lines. It would be possible to correct this either by varying the blob size with line orientation, or by using a circular "nib".

It is interesting to speculate on the relative merits of a programming approach and a WYSIWYG approach. Clearly both have their place. However the example diagrams included here mostly seem easier to program than to draw with a mouse.

Copies of the Pascal preprocessor and TEX macros can be obtained directly from us or by electronic mail request.

References

Knuth, D.E. *The TEXbook*, Addison-Wesley, 1984.

Hopper, K., Rogers, W. J. & Holmes, G. *The Magic of Modula-2*, Addison-Wesley, 1991.

Goodyear, P. *LOGO - a guide to learning through programming*, Ellis Horwood, 1984.

Rankin, J. R. *Computer Graphics Software Construction*, Addison-Wesley, 1989.