Mapyrus Project Version 0.807

Simon Chenery (simoc@users.sourceforge.net)

29-December-2008

Contents

1 Introduction **7**

2 Availability **11**

3 Feedback **12**

4 Tutorial and Cookbook **13**
4.1 First Step 13
4.2 Second Step 14
4.3 Using Variables 17
4.4 Building Procedures 19
4.5 Displaying Lines 21
4.6 Displaying Polygons 31
4.7 Displaying Labels 34
4.8 Displaying Data Stored In Text Files 41
4.9 Displaying Data Stored In Shape Files 44
4.10 Displaying OpenStreetMap Data 46
4.11 Displaying Data Stored In A Database 47
4.12 Displaying Geo-Referenced Images 51
4.13 Displaying Images From An OGC Web Mapping Service 52
4.14 Displaying Many Datasets or Geo-Referenced Images 53
4.15 Updating Existing Output Files 55
4.16 Display Performance 55
4.17 Displaying A Legend 56
4.18 Using Attributes To Control Display 62
4.19 Displaying A Scalebar 66
4.20 Displaying Piecharts And Histograms 68
4.21 Random Effects 70
4.22 Using Transparency 72
4.23 Shadow Effects 75
4.24 Displaying Tables 76
4.25 Wordwrapped Labels 78
4.26 Avoiding Overlapping Labels 79
4.27 Displaying Image Icons 82
4.28 Including Encapsulated PostScript Files 85
4.29 Mapyrus and Java Topology Suite Functions 86
4.30 Creating Landscape Output on Portrait Pages 89
4.31 Page Layout With Mapyrus 91
4.32 Creating Multiple Page Output 93
4.33 Using PostScript Fonts In PostScript Output 93
4.34 Using PostScript Fonts In PDF Output 93
4.35 Using TrueType Fonts In Output to Image Formats 94
4.36 Using Fonts In SVG Output 94
4.37 Running Mapyrus As An HTTP Server 94
4.38 Passing Variables To Mapyrus HTTP Server Through URLs . . . 96
4.39 Returning HTML Pages From Mapyrus HTTP Server 96
4.40 Using JavaScript with Mapyrus HTTP Server 97
4.41 Setting Expiry Dates, Cookies and Redirections from Mapyrus HTTP Server 98

4.42 Returning Additional Information From Mapyrus HTTP Server . 99
4.43 Running Mapyrus As A Java Servlet With Apache Tomcat . . . 99
4.44 Using Mapyrus In A Java Or Jython Application 100
4.45 Calling Java Functions From Mapyrus 101
4.46 Combining Mapyrus With Other Software 101
4.47 Creating Animations Using Mapyrus 102
4.48 Creating SVG Files With Event Handling 103
4.49 Building Mapyrus From Source 104
4.50 Sample Shapes And Patterns 105

5 Reference **109**
5.1 Software Requirements 109
5.2 Usage 109
5.2.1 Startup Configuration 109
5.3 Language 110
5.4 Internal Variables 117
5.5 Commands 119
5.5.1 addpath 120
5.5.2 arc 120
5.5.3 bezier 120
5.5.4 blend 120
5.5.5 box 121
5.5.6 box3d 121
5.5.7 chessboard 121
5.5.8 circle 121
5.5.9 clearpath 121
5.5.10 clip 122
5.5.11 closepath 122
5.5.12 color 122
5.5.13 cylinder 123
5.5.14 dataset 123
5.5.15 draw 126
5.5.16 ellipse 126
5.5.17 endpage 126
5.5.18 eps 126
5.5.19 eval 126
5.5.20 eventscript 127
5.5.21 fetch 127
5.5.22 fill 127
5.5.23 flowlabel 127
5.5.24 font 128
5.5.25 geoimage 128
5.5.26 gradientfill 129
5.5.27 guillotine 129
5.5.28 hexagon 130
5.5.29 httpresponse 130
5.5.30 icon 130
5.5.31 justify 131
5.5.32 key 131
5.5.33 label 131

5.5.34 legend . . . 131
5.5.35 let . . . 132
5.5.36 linestyle . . . 132
5.5.37 local . . . 132
5.5.38 mimetype . . . 133
5.5.39 move . . . 133
5.5.40 newpage . . . 133
5.5.41 parallelpath . . . 139
5.5.42 pdf . . . 139
5.5.43 pentagon . . . 139
5.5.44 print . . . 139
5.5.45 protect . . . 139
5.5.46 raindrop . . . 140
5.5.47 reversepath . . . 140
5.5.48 rotate . . . 140
5.5.49 rdraw . . . 140
5.5.50 roundedbox . . . 140
5.5.51 samplepath . . . 141
5.5.52 scale . . . 141
5.5.53 selectpath . . . 141
5.5.54 setoutput . . . 141
5.5.55 shiftpath . . . 141
5.5.56 sinewave . . . 142
5.5.57 sinkhole . . . 142
5.5.58 spiral . . . 142
5.5.59 star . . . 142
5.5.60 stripepath . . . 143
5.5.61 stroke . . . 143
5.5.62 svg . . . 143
5.5.63 svgcode . . . 143
5.5.64 table . . . 143
5.5.65 tree . . . 144
5.5.66 triangle . . . 144
5.5.67 unprotect . . . 145
5.5.68 wedge . . . 145
5.5.69 worlds . . . 145
5.6 Error Handling . . . 146
5.7 Mapyrus HTTP Server . . . 146
5.8 Mapyrus Servlet . . . 147

List of Figures

1 Shapes, Symbols And Patterns 7
2 Average Monthly Temperatures 8
3 Strip Map of Railways Lines in East Kent 9
4 Vegetation Classes 9
5 Inventory Levels at Warehouses 10
6 First Step 13
7 Second Step 14
8 Third Step 15
9 Fourth Step 17
10 Variables And Loops 18
11 Variables And Functions 19
12 Calling Procedures 20
13 Procedures And Move Points 21
14 Repeating Symbols Along A Line 22
15 Varying Repeated Symbols Along A Line 23
16 Combining Solid And Dashed Linestyles 24
17 Combining Several Linestyles 25
18 Linestyle Using Begin And End Points 26
19 Linestyle Using Sample Points 27
20 Parallel Linestyle 27
21 Selecting Parts Of Path 28
22 Selecting and Sampling Path 30
23 Sine Wave Curves 30
24 Hatching Polygons 31
25 Cross-Hatching Polygons 32
26 Displaying Points in Polygons 33
27 Displaying Polygon Borders 34
28 Displaying Polygons With Gradient Fill 35
29 Displaying Labels 36
30 Rotated Labels 36
31 Highlighted Labels 38
32 Labels Along a Line 39
33 Labelling Streets 40
34 Labelling Polygons 41
35 Text File tutorialdatasets1.txt 42
36 Displaying Contents Of A Text File 43
37 GIS Export File streets.EE 44
38 Displaying GIS Export Files 45
39 Displaying ESRI Shape Files 46
40 Displaying Geo-Referenced Images 52
41 Clipping Geo-Referenced Images 53
42 Updating An Existing Output File 56
43 Displaying A Legend 58
44 Displaying Legend Entries Individually 61
45 Displaying Frequency Count Of Legend Entries 62
46 Using Attributes 64
47 Using More Attributes 65
48 Displaying A Scalebar 67

49 Displaying Piecharts 69
50 Displaying Histograms 69
51 Random Color 71
52 Random Rotation 71
53 Random Position 72
54 Transparent Colors 73
55 Fading Lines 74
56 Shadow Effects 76
57 Displaying Tables In A Map 77
58 Displaying Map And Tables Separately 78
59 Wordwrapped Labels 79
60 Label Positioning 81
61 More Label Positioning 82
62 Icon Display 83
63 Icon Fill Pattern 84
64 Spray Paint Pattern 85
65 Displaying An Encapsulated PostScript File 86
66 Buffer Function 87
67 Contains Function 89
68 Difference Function 90
69 Rotated Landscape Page 90
70 Page Layout 106
71 Sample Shapes And Patterns 1 107
72 Sample Shapes And Patterns 2 108

1 Introduction

Mapyrus is software for creating plots of points, lines, polygons and labels to PostScript (high resolution, up to A0 paper size), Portable Document Format (PDF), Scalable Vector Graphics (SVG) format and web image output formats.

Mapyrus is open source software and is implemented entirely in Java enabling it to run on a wide range of operating systems.

The software combines the following four features.

1. A Logo or turtle graphics program.

   An imaginary pen is moved around a page, creating shapes that are drawn into an image file. Reusable routines are built up using a BASIC-like language. Branching and looping constructs enable complex shapes, symbols, patterns and graphs to be be defined.

   See Figure 1.

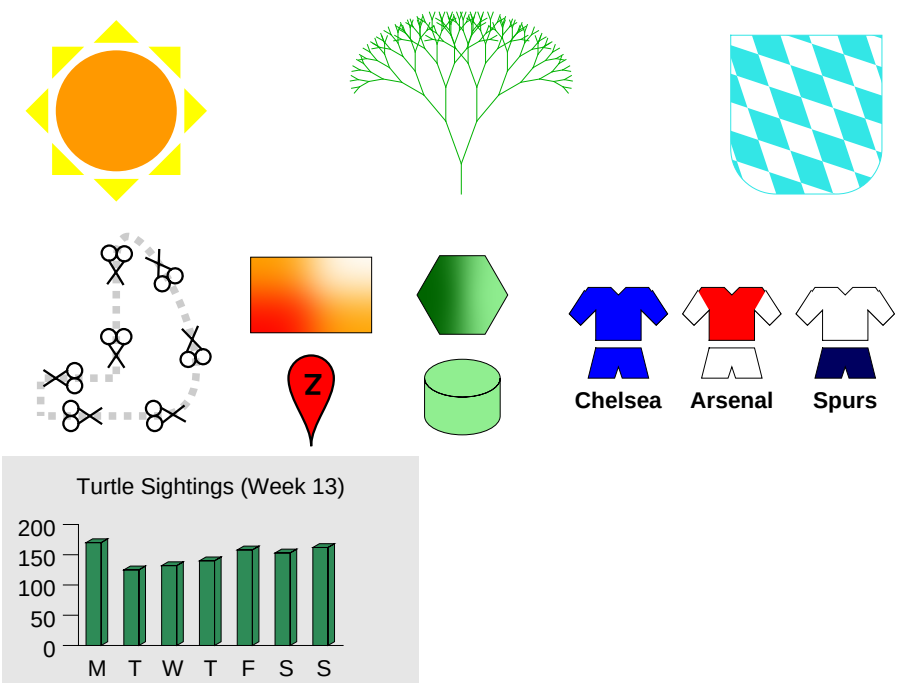

Figure 1: Shapes, Symbols And Patterns

2. Reading and displaying of geographic information system (GIS) datasets (including the OpenStreetMap project), text files, or tables held in a relational database (including spatially extended databases such as Oracle Spatial, PostGIS and MySQL).

   Drawing routines are applied to geographic data to produce annotated and symbolized maps and graphs. Attributes of the geographic data control the color, size, annotation and other characteristics of the appearance of the geographic data. Scalebars, legends, coordinate grids and north arrows are also available.

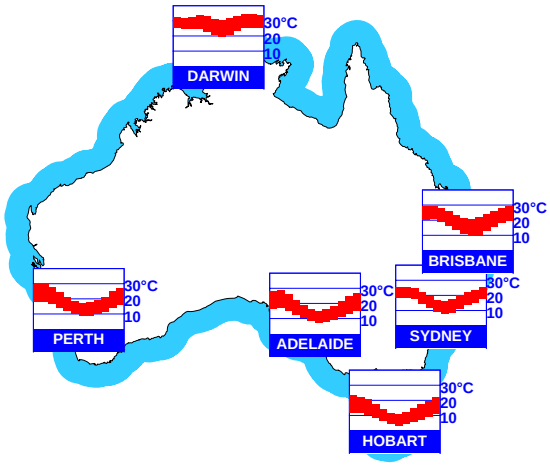

Figure 2: Average Monthly Temperatures of Australian Cities (degrees Celsius)

See Figures 2, 3, 4 and 5.

3. Integration with the freely-available Java Topology Suite [1]. This library provides geometric algorithms such as buffering, point-in-polygon test and polygon intersection.

4. Flexibility. Running in one of three ways.

   (a) As a stand-alone program for integration into scripts and batch tasks (suitable for generating a one-off map or a series of similar maps from a template showing different areas, or using different criteria for each map).

   (b) As a self-contained web server providing map and graph images to a web-based application via HTTP requests.

   (c) As a Java Servlet in a web server such as Apache Tomcat, generating map and graph images in response to HTTP requests.

Please note that:

- Mapyrus has no graphical user interface. Commands are read from a text file prepared by the user. Good user interfaces are hard to design and everyone wants something slightly different. Build a custom HTML or Java user interface on top of Mapyrus.

- Mapyrus cannot display three dimensional data and has nothing to do with 3D.

[1] Available from http://www.vividsolutions.com/JTS

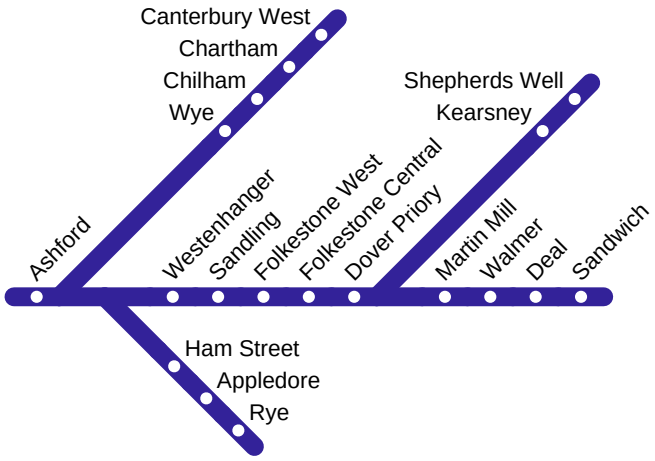

Figure 3: Strip Map of Railways Lines in East Kent

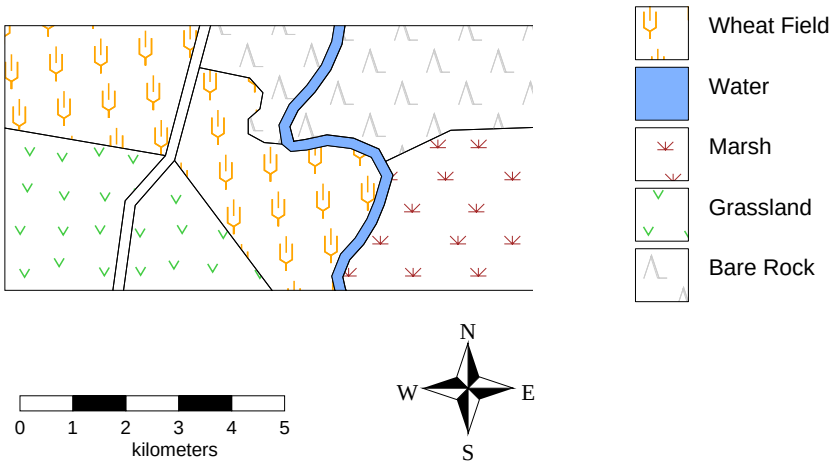

Figure 4: Vegetation Classes

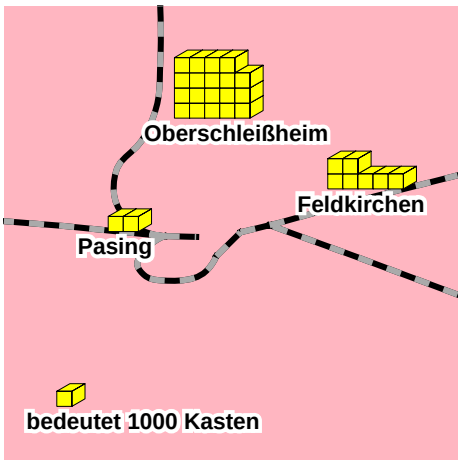

Figure 5: Inventory Levels at Warehouses

- Mapyrus includes a sample set of about 100 ready-to-use shapes and patterns for maps and charts (see Figure 4.50 on page 107). Icons found on the internet are another good solution for display in image files. For PostScript output, the resolution is higher and using characters from a font such as ZapfDingbats produces better output.
- Mapyrus cannot reproject coordinate values in geographic data from one projection to another. However, this functionality is available in spatial databases.

2 Availability

Mapyrus is released under the GNU Lesser General Public License. The software, documentation and source code is available for download from `http://mapyrus.sourceforge.net`.

For more information about licensing, see the file named `COPYING`.

3 Feedback

Send comments and bug reports by e-mail to

`simoc@users.sourceforge.net`

Good ideas for extensions to Mapyrus to make it more useful are also welcome!

4 Tutorial and Cookbook

The following examples demonstrate how to use Mapyrus. Each example is also included as a file in the `userdoc` subdirectory. All example files have file suffix `.mapyrus`. This suffix is not required but makes identification of files that are to be interpreted by Mapyrus easier.

If the Ghostscript program or a printer is not available, then change the `"eps"` arguments in `newpage` commands in examples to `"screen"` to display the output in a window on the screen instead.

4.1 First Step

Enter the following lines into a text file named `first.mapyrus`.

```
newpage "eps", "tutorialfirst1.eps", 30, 30
color "indigo"
linestyle 4, 'round', 'round'
move 5, 20
draw 20, 20, 5, 5, 20, 5
stroke
```

Each line in the file is a command for Mapyrus. First, output is set to a 30mm square Encapsulated PostScript format file named `tutorialfirst1.eps` with the `newpage` command, then a line is drawn with the given color and linestyle.

Open a terminal window (known as an "MS-DOS Prompt" in MicroSoft Windows) and run Mapyrus using following command to execute the commands in the file `first.mapyrus`.

```
java -classpath mapyrus-dir/mapyrus.jar org.mapyrus.Mapyrus first.mapyrus
```

When Mapyrus reaches the end of the file `first.mapyrus` the output file `tutorialfirst1.eps` is saved and Mapyrus exits. When the PostScript file is then printed, it appears as in Figure 6.

Figure 6: First Step

To perform these steps on a MicroSoft Windows machine where the the ZIP file containing Mapyrus has been unpacked in directory `C:\temp`, the following commands are used:

```
C:\>cd temp
```

```
C:\temp>dir /b *.jar        -- Check that Mapyrus is installed here
mapyrus.jar

C:\temp>notepad first.mapyrus        -- Create file with text editor

C:\temp>java -classpath mapyrus.jar org.mapyrus.Mapyrus first.mapyrus

C:\temp>gswin32 tutorialfirst1.eps        -- View output with GhostScript
```

4.2 Second Step

Change the file `first.mapyrus` to contain the following lines, then run Mapyrus again to create a new output file.

```
# Another simple example.
#
newpage "eps", "tutorialfirst2.eps", 30, 30
color "#bb0000"         # dark red
box 5, 5, 25, 25
fill
```

In this example, color is set as a hex value instead of a name, a rectangle is defined with the `box` command and the `fill` command is used to flood fill it. The characters on lines following a hash (`#`) character that is not enclosed in quotes are ignored by Mapyrus. Figure 7 shows the output of these commands.

Figure 7: Second Step

Change the file `first.mapyrus` to contain the following lines.

```
newpage "eps", "tutorialfirst3.eps", 70, 30
linestyle 0.1
color "rgb", 1, 1, 0
move 5, 15
arc 1, 15, 15, 25, 15
arc -1, 25, 5, 15, 5
rdraw -10, 0
closepath
fill
color "rgb", 0, 0.33, 0
stroke

# Clear path so we can begin a new shape.
```

```
#
clearpath
color "rgb", 0, 0, 0.7
move 30, 5
draw 30, 25, 60, 25, 60, 5, 30, 5
move 40, 8
draw 50, 8, 45, 22, 40, 8
fill
```

This example demonstrates drawing circular arcs, giving the direction (a positive value for clockwise and a negative value for anti-clockwise), center point and end point, drawing line segments relative to the last point in the path with the `rdraw` command, closing the path back to the starting point with the `closepath` command and yet another way of defining color. After being filled, the path remains and is used again to draw the outline of the shape. To clear the path before drawing another shape the `clearpath` command is used. The second shape contains an island (or hole). When a shape is self-intersecting or contains islands, the winding rule is used for determining which areas get filled. The output of these commands is shown in Figure 8.

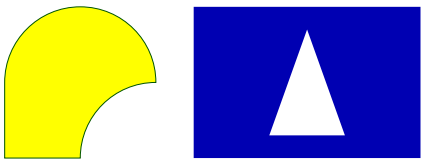

Figure 8: Third Step

Change the file `first.mapyrus` to contain the following lines.

```
# Set page size to 500x100 screen pixels and set scale
# so we can work with pixel coordinate values too.
#
newpage "eps", "tutorialfirst4.eps", \
  500 * Mapyrus.screen.resolution.mm, \
  100 * Mapyrus.screen.resolution.mm
scale Mapyrus.screen.resolution.mm

clearpath
color "Firebrick"
hexagon 25, 75, 25
fill

clearpath
color "orange"
triangle 25, 75, 15, 0
fill
```

```
clearpath
color "Forest Green"
circle 90, 75, 22
star 90, 75, 22, 5
stroke

clearpath
color "indigo"
wedge 150, 60, 30, 45, 90
fill

clearpath
color "black"
ellipse 200, 75, 10, 20
ellipse 200, 75, 20, 10
stroke

clearpath
color "blue"
linestyle 2
move 240, 60
bezier 280, 100, 280, 80, 260, 60
stroke

clearpath
color "SlateGray"
roundedbox 300, 60, 340, 90, 10
fill

clearpath
box3d 360, 60, 380, 80
color "yellow"
fill
color "black"
linestyle 0.01
stroke

clearpath
pentagon 430, 75, 20
color "orange"
fill

clearpath
color "Steel Blue"
spiral 25, 25, 20, 5, 0
stroke

clearpath
cylinder 90, 20, 25, 10
color "yellow"
```

```
fill
color "black"
stroke

clearpath
color "dodgerblue"
raindrop 150, 20, 10
fill

clearpath
color "crimson"
move 180, 20
sinewave 210, 20, 4, 15
stroke

clearpath
color "black"
move 240, 20
chessboard 240, 10, 270, 40, 5
fill
```

This example demonstrates setting the page size in screen pixels and setting a scale so that coordinates are also given as pixels. Various shapes are drawn using `circle`, `hexagon`, `triangle`, `star`, `wedge`, `ellipse`, `bezier`, `roundedbox`, `box3d`, `pentagon`, `spiral`, `cylinder`, `raindrop` `sinewave` and `chessboard` commands. The output of these commands is shown in Figure 9.

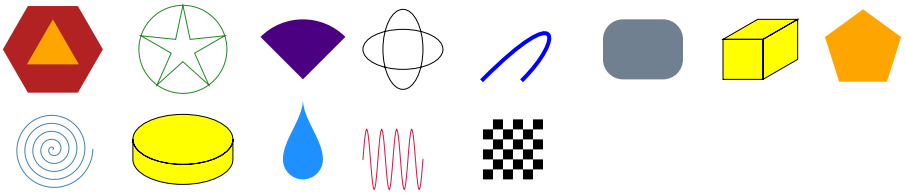

Figure 9: Fourth Step

4.3 Using Variables

Use variables and conditional tests to vary the appearance of the display. Variables named `c1` and `c2` are used in the following example. The variable `Mapyrus.page.width` is set by Mapyrus automatically. The output of this example is shown in Figure 10.

```
newpage "eps", "tutorialvar1.eps", 50, 50
color "green"
let c1 = 1
while c1 < Mapyrus.page.width
do
  let c2 = Mapyrus.page.width - c1
```

```
  clearpath
  move c1, 0
  draw 0, c2
  stroke
  let c1 = c1 + 4
done
```

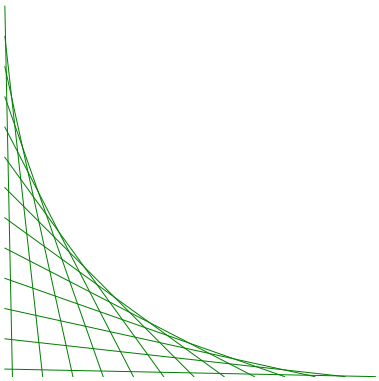

Figure 10: Variables And Loops

The second example demonstrates the use of the `length` and `substr` functions, the Java API function `java.lang.Integer.parseInt` and stepping through a string one element at a time. The output of this example is shown in Figure 11.

```
newpage "eps", "tutorialvar2.eps", 70, 20

# Set binary sequence
#
let seq = java.lang.Integer.toBinaryString(45678)
let seqlen = length(seq)

# Draw sequence, showing black bars for bits
# that have the value '1'.
#
let i = 1
while i <= seqlen
do
  clearpath
  box i * 4, 5, i * 4 + 3, 15
  if (substr(seq, i, 1) eq "1")
  then
    color "black"
  else
    color "yellow"
  endif
  fill
  let i = i + 1
```

```
done
```

Figure 11: Variables And Functions

4.4 Building Procedures

Store frequently used sequences of commands in a procedure. A procedure has a unique name, takes a fixed number of arguments and may be called from any place where a command is expected.

Private variables in a procedure are defined with the `local` command.

When a procedure is called, the graphics state is saved (see page 116), the commands for the procedure are executed and then the graphics state is restored before returning.

The following example shows a simple procedure named `trail`. The output of this example is shown in Figure 12.

```
begin trail grade
  # Draw a line marking a park trail.  Line color and
  # style depends on grade of trail.
  #
  if grade eq 'vehicle'
  then
    color "black"
    linestyle 0.5
  elif grade eq 'maintained'
  then
    color "gray"
    linestyle 0.5, 'butt', 'bevel', 0, 2, 2
  else
    # Unmaintained track is dotted line.
    #
    color "#A52A2A"
    linestyle 0.5, 'butt', 'bevel', 0, 1, 1
  endif
  stroke
end

newpage "eps", "tutorialprocedures1.eps", 60, 30

# Draw trails of differing grades.  In a real application this
# data would be read from a Geographic Information System (GIS).
#
clearpath
```

```
move 10, 10
draw 15, 16, 26, 24, 38, 29
trail 'vehicle'
clearpath
move 17, 16
draw 24, 15, 29, 12, 34, 5
trail 'maintained'
clearpath
move 24, 23
draw 12, 26, 6, 28
trail 'maintained'
clearpath
move 30, 11
draw 36, 9, 40, 10, 46, 9, 54, 8
trail 'unmaintained'
```

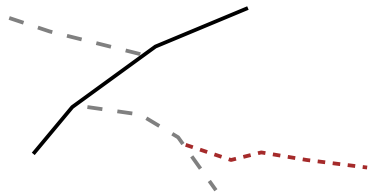

Figure 12: Calling Procedures

If the current path contains only points added with `move` commands when a procedure is called, then the procedure is called once for each point with the origin moved to that point. This permits several symbols to be drawn easily and is demonstrated in the following example. The output of this example is shown in Figure 13.

```
begin waypoint
  # Draw waypoint symbol.
  #
  color "red"
  box -2, -2, 2, 2
  stroke
  clearpath
  move -2, -2
  draw -2, 2, 2, -2
  fill
end

newpage "eps", "tutorialprocedures2.eps", 60, 30
clearpath
move 7, 5
move 16, 22
move 22, 11
```

```
move 34, 17
move 44, 9
waypoint
```

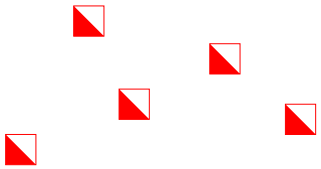

Figure 13: Procedures And Move Points

Put frequently used procedures in a separate file and use an `include` line to include it in other files. When this is done for the previous example, the procedure is put in a file.

```
# This file is tutorialprocedures3.mapyrus
begin waypoint
  # Draw waypoint symbol.
  #
  color "red"
  box -2, -2, 2, 2
  stroke
  clearpath
  move -2, -2
  draw -2, 2, 2, -2
  fill
end
```

and the file containing the commands to execute

```
# This file is tutorialprocedures4.mapyrus
include tutorialprocedures3.mapyrus

newpage "eps", "tutorialprocedures4.eps", 60, 30
clearpath
move 7, 5
move 16, 22
move 22, 11
move 34, 17
move 44, 9
waypoint
```

4.5 Displaying Lines

Simple solid line styles or dashed line styles are set with the `linestyle` command. More complex line styles are built using symbols repeated along a line or combinations of linestyles, plotted on top of each other.

To draw a line with repeated symbols, use the `samplepath` command to replace the path with evenly spaced sample points and then call a procedure to draw a symbol at each sample point. The following example demonstrates this, with the output shown in Figure 14.

```
begin arrow
  # Draws a filled '|>' symbol.
  #
  move -1, -1
  draw -1, 1, 2, 0
  fill
end

newpage "eps", "tutoriallines1.eps", 60, 30
color "orange"
clearpath
move 5, 5
draw 45, 5
arc -1, 45, 15, 45, 25
draw 35, 25, 35, 10, 5, 10
samplepath 5, 2.5
arrow
```

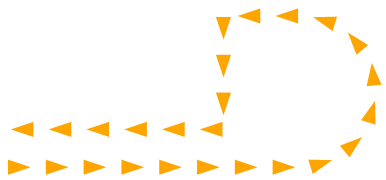

Figure 14: Repeating Symbols Along A Line

To vary the symbols that are drawn along the line, use a variable to count the number of symbols along the line and a conditional test to draw different symbols at different positions along the line. The following example demonstrates this, with the output shown in Figure 15.

```
begin spike
  move -1, 0
  let spikeCounter = spikeCounter + 1
  # Alternate between long and short spikes.
  #
  if spikeCounter % 2 == 0
  then
    draw 0, -3, 1, 0
  else
    draw 0, -6, 1, 0
  endif
```

```
  fill
end

newpage "eps", "tutoriallines2.eps", 60, 30
clearpath
move 5, 5
bezier 5, 30, 55, 5, 55, 25
samplepath 3, 1.5
let spikeCounter = 0
spike
```

Figure 15: Varying Repeated Symbols Along A Line

The next example demonstrates combining solid and dashed linestyles. For best results display all lines with a solid linestyle, then display all lines again with a dashed linestyle. The output of this example is shown in Figure 16.

```
newpage "eps", "tutoriallines3.eps", 60, 30

# Demonstrate combination of solid and dashed linestyles.
#
clearpath
move 5, 5
draw 45, 5
arc -1, 45, 15, 45, 25
draw 15, 25
arc -1, 15, 20, 15, 15
draw 35, 15
linestyle 2.5, 'butt', 'round'
color 'black'
stroke
linestyle 1.8, 'butt', 'round', 1, 2, 2
color 'yellow'
stroke
```

The following example demonstrates combining thick and thin linestyles and symbols at sampled points. For best results display all lines with a thick linestyle, then display all lines again with a thin linestyle. The output of this example is shown in Figure 17.

```
begin highway name
  # Force symbol to appear upright, regardless of
```

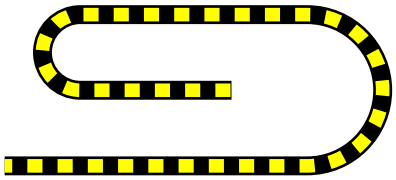

Figure 16: Combining Solid And Dashed Linestyles

```
  # orientation of line at point where symbol is drawn.
  #
  rotate -Mapyrus.rotation
  # Draw rectangle, then label inside it.
  #
  box -3, -2, 3, 2
  color "blue"
  fill
  clearpath
  color "white"
  font "Helvetica", 3
  justify "center"
  move 0, -1
  label name
end

newpage "eps", "tutoriallines4.eps", 90, 30
clearpath
move 5, 5
draw 45, 5
arc -1, 45, 15, 55, 15
arc 1, 65, 15, 65, 25
draw 85, 25
# Demonstrate overplotting of lines.
#
color 'red'
linestyle 1.32, 'round', 'round'
stroke
color 'yellow'
linestyle 0.1
stroke
# Draw path one more time as symbols showing name of highway.
#
samplepath 17, 9
highway 'A99'
```

The following two examples demonstrate creating sample points from the path to generate further line styles, with the output shown in Figures 18 and 19.

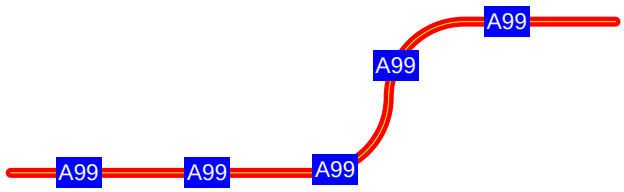

Figure 17: Combining Several Linestyles

```
begin arrowhead r
  # Draw arrowhead rotated to point to end of line.
  #
  rotate r
  move 5, 2
  draw 0, 0, 5, -2
  closepath
  fill
end
begin arrowstart
  # Split path into points with large spacing so there will be
  # only one sample point at start of line.  Draw arrowhead there.
  #
  samplepath 99999, 0
  arrowhead 0
end
begin arrowend
  # Draw arrowhead at end of line.
  #
  samplepath -99999, 0
  arrowhead 180
end
newpage "eps", "tutoriallines5.eps", 60, 30
clearpath
move 5, 5
draw 45, 5
arc -1, 45, 15, 55, 15
draw 55, 25
linestyle 0.1
color '#a020f0'
stroke
# Demonstrate drawing a line with arrows at both ends like <---->
#
arrowstart
arrowend

begin quarterLabel
  # Drawing tick and label at intermediate position along line.
  #
```

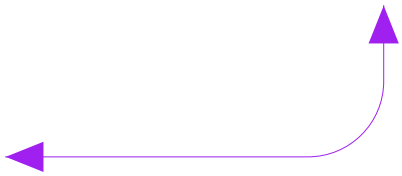

Figure 18: Linestyle Using Begin And End Points

```
  if percent < 0 or percent > 100
  then
    return
  endif

  # Draw triangle pointing to position on line.
  #
  triangle 0, -1, 1, 0
  fill

  # Label
  #
  clearpath
  move 0, 2
  justify "center"
  font "Helvetica", 3
  label percent . "%"
  let percent = percent + 25
end

begin labelQuarters pathLen
  # Divide path into quarters and drawing ticks and percentage
  # label at points 1/4, 2/4, 3/4 positions along line.
  #
  samplepath pathLen / 4, 0
  let percent = 0
  quarterLabel
end

newpage "eps", "tutoriallines6.eps", 60, 30
clearpath
move 5, 5
draw 45, 5
arc -1, 45, 15, 55, 15
draw 55, 25
linestyle 0.1
color '#006400'
stroke
labelQuarters Mapyrus.path.length
```

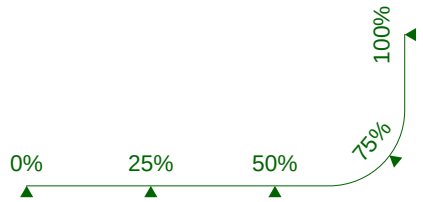

Figure 19: Linestyle Using Sample Points

The next example demonstrates using the `parallelpath` command to display parallel lines following the original path, with the output shown in Figure 20.

```
newpage "eps", "tutoriallines7.eps", 80, 40

move 10, 10
draw 17, 32, 18, 33, 27, 33, 27, 31, 24, 31, 25, 16, 36, 19, 40, 25
arc 1, 41, 19, 37, 8
hexagon 70, 15, 8
move 50, 35
draw 60, 35, 60.3, 30, 60.6, 30, 60.9, 35, 70, 35
color "blue"
stroke

# Show dashed lines to the left and right of original line.
#
parallelpath 1, -1
color "indigo"
linestyle 0.1, "round", "round", 0, 2, 1
stroke
```

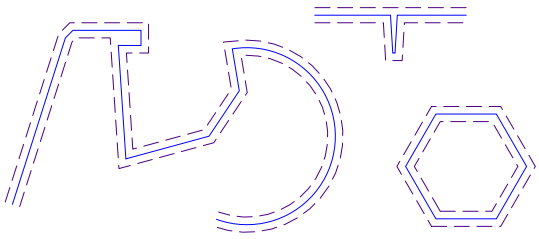

Figure 20: Parallel Linestyle

The next example demonstrates using the `selectpath` command to select short sections at the start and end of the path and displaying them in different colors to indicate the polarity of the connection between two points. The output of this example is shown in Figure 21.

```
begin positive_bar
  # Display black bar at start of line to show positive end of connection.
  selectpath 3, 5
  color "black"
  stroke
end

begin negative_bar
  # Display red bar at end of line to show negative end of connection.
  selectpath Mapyrus.path.length - 8, 5
  color "red"
  stroke
end

begin voltage_line
  linestyle 2, "square", "bevel"
  color "gray"
  stroke
  positive_bar
  negative_bar
end

newpage "eps", "tutoriallines8.eps", 60, 30
move 5, 25
draw 35, 5
voltage_line

clearpath
move 35, 5
draw 55, 10, 56, 16
voltage_line

clearpath
move 35, 5
bezier 35, 15, 40, 20, 45, 27
voltage_line
```

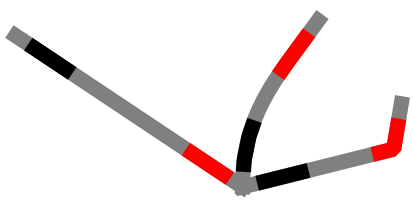

Figure 21: Selecting Parts Of Path

The `selectpath` command is also useful for displaying lines that stop short of the start and end points of the path, and for displaying lines that have thicker

and thinner sections in the middle, or at the ends of the lines.

The next example demonstrates combining the `selectpath` and `samplepath` commands to display a different linestyle in the middle of each line. The output of this example is shown in Figure 22.

```
begin zigzag_start
  selectpath 0, 5
  stroke
end
begin zigzag_end
  selectpath 5 + numzigs * ziglen, 999999
  stroke
end
begin zigzag_middle
  # Draw whole number of zigzag pattern in middle part of path.
  #
  selectpath 5, numzigs * ziglen
  samplepath ziglen, 2
  zig
end
begin zig
  # Draw zigzag symbol: /\/
  #
  move -ziglen / 2, 0
  draw -ziglen / 4, 2, ziglen / 4, -2, ziglen / 2, 0
  color "orange"
  stroke
end

begin zigzag_line
  linestyle 0.4, "round", "round"
  color "dodger blue"
  let ziglen = 4
  let numzigs = floor((Mapyrus.path.length - 5 - 5) / ziglen)
  zigzag_start
  zigzag_middle
  zigzag_end
end

newpage "eps", "tutoriallines9.eps", 60, 30
move 5, 25
draw 35, 25
zigzag_line

clearpath
move 7, 7
draw 47, 7, 47, 25
zigzag_line
```

Use the `sinewave` command to display a curved line between two points instead of a straight line.

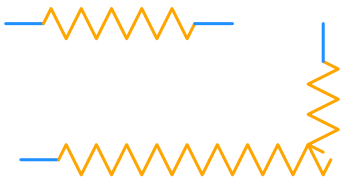

Figure 22: Selecting and Sampling Path

This is demonstrated in the following example, with the output shown in Figure 23.

```
newpage "eps", "tutoriallines10.eps", 60, 30

move 5, 5
draw 45, 25
stroke

clearpath
move 30, 5
draw 55, 25
stroke

# Draw lines a second time as sine wave curves.
#
color "red"
linestyle 2
clearpath
move 5, 5
sinewave 45, 25, 2, 3
stroke

color "green"
clearpath
move 30, 5
sinewave 55, 25, 0.5, -4
stroke
```

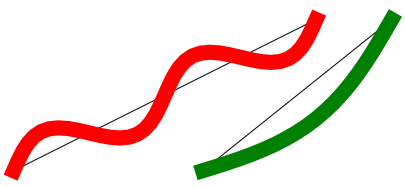

Figure 23: Sine Wave Curves

4.6 Displaying Polygons

Polygon outlines are drawn with the `stroke` command and filled with the `fill` command. To draw repeated stripes (also known as hatching) through the polygon, use the `clip` command to limit the area displayed to inside the polygon, then the `stripepath` command to replace the path with evenly spaced stripes and then draw each stripe. The following example demonstrates this, with the output shown in Figure 24.

```
newpage "eps", "tutorialpolygons1.eps", 60, 30
color "forestgreen"
clearpath
move 5, 5
draw 45, 5
arc -1, 45, 15, 45, 25
draw 35, 25, 35, 10, 5, 10, 5, 5
stroke  # Draw outline of polygon.
clip "inside"
stripepath 3, 60
stroke  # Draw hatch lines inside polygon.
```

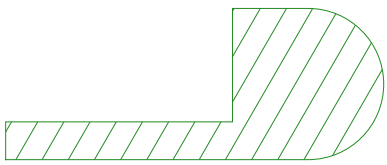

Figure 24: Hatching Polygons

To fill a polygon with cross-hatching use procedures to draw each set of hatch lines at different angles. Two procedures are used so that modifying the current path with the `stripepath` command is isolated in one procedure and the original path remains unmodified in the calling procedure. This is demonstrated in the following example, with the output shown in Figure 25.

```
begin hatch45
  clip "inside"
  stripepath 4, 45
  stroke
end
begin hatchMinus45
  clip "inside"
  stripepath 4, -45
  stroke
end
begin crosshatch
  linestyle 0.01
  color "red"
```

```
  stroke
  hatch45
  hatchMinus45
end

newpage "eps", "tutorialpolygons2.eps", 60, 30
clearpath
roundedbox 5, 3, 30, 25
crosshatch

clearpath
roundedbox 31, 5, 55, 27
crosshatch
```

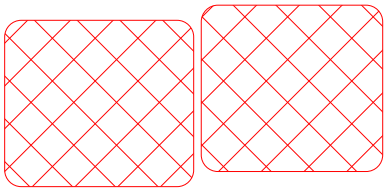

Figure 25: Cross-Hatching Polygons

Use the `stripepath` and `samplepath` commands in combination to generate a grid of points through the polygon, then call a procedure to draw a symbol at each point. This is demonstrated in the following example, with the output shown in Figure 26.

```
begin star6
  # Draws a 6 pointed star.
  #
  star 0, 0, 1.5, 6
  fill
end
begin starfill
  color "navy blue"
  stroke
  clip "inside"

  # Replace path with sample points covering the whole polygon.
  #
  stripepath 4, 0
  samplepath 4, 0
  color "goldenrod"
  star6
end

newpage "eps", "tutorialpolygons3.eps", 60, 30
```

```
clearpath
move 5, 5
draw 45, 5
arc -1, 45, 15, 45, 25
draw 35, 25, 35, 10, 25, 10, 30, 25, 10, 25, 15, 10
draw 5, 10, 5, 5
starfill
```

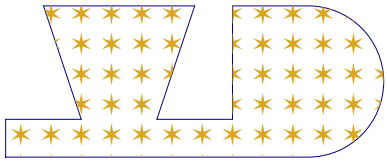

Figure 26: Displaying Points in Polygons

To display a border around the inside of the polygon use the `clip` command to limit the area displayed to inside the polygon and then draw the outline with a thick linestyle. The following example demonstrates this, with the output shown in Figure 27. Give the argument `outside` to the `clip` command to display a border around the outside of a polygon instead.

```
begin borderthick c
  clip "inside"
  color c
  linestyle 5
  stroke
end
begin border c
  # Draw border as thick line on inside of polygon then solid border.
  #
  borderthick c
  linestyle 0.1
  color "black"
  stroke
end

# Display adjacent polygons, each with a thick interior border.
#
newpage "eps", "tutorialpolygons4.eps", 65, 40
clearpath
move 5, 5
draw 7, 31, 42, 33, 40, 19, 20, 19, 19, 24, 15, 24, 13, 5, 5, 5
border "pink"
clearpath
move 13, 5
```

```
draw 15, 24, 19, 24, 20, 19, 40, 19, 42, 9, 13, 5
border "yellow"
clearpath
move 42, 9
draw 40, 19, 42, 33, 48, 33, 56, 7, 42, 9
border "green"
```

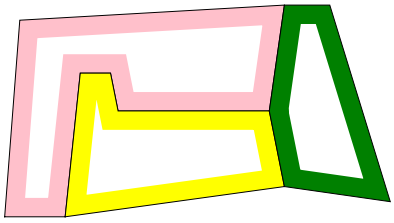

Figure 27: Displaying Polygon Borders

To display polygons with a gradient fill pattern use the `gradientfill` command. This command fills the current path with colors that fade across the polygon. The following example demonstrates fading from olive at the lower-left corner of each polygon to white in all other corners of the polygon. The output is shown in Figure 28.

```
begin oliveGradient
  gradientfill "olive", "white", "white", "white"
  color "black"
  stroke
end

newpage "eps", "tutorialgradient.eps", 65, 40
clearpath
move 5, 5
draw 7, 31, 42, 33, 40, 19, 20, 19, 19, 24, 15, 24, 13, 5, 5, 5
oliveGradient
clearpath
move 13, 5
draw 15, 24, 19, 24, 20, 19, 40, 19, 42, 9, 13, 5
oliveGradient
clearpath
move 42, 9
draw 40, 19, 42, 33, 48, 33, 56, 7, 42, 9
oliveGradient
```

4.7 Displaying Labels

Labels are displayed using the `move` command to define one or more points for labelling, followed by a `label` command. The font and justification for labels

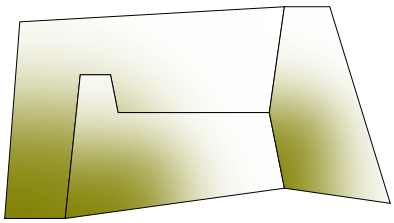

Figure 28: Displaying Polygons With Gradient Fill

are set with the `font` and `justify` commands. Setting fonts, justification and displaying multiple lines are demonstrated in the following example, with output shown in Figure 29.

```
newpage "eps", "tutoriallabels1.eps", 125, 30, \
  "isolatinfonts=Times-Roman"

# Draw a label containing o with umlaut and degree symbol.
# These octal character codes entered in ISO encoding so
# font is marked for ISO encoding in newpage command above.
#
color "black"
font 'Times-Roman', 10
clearpath
move 2, 5
label 'M\374nchen 17\260'

# Draw a three line centered label.
#
clearpath
move 67, 18
color "red"
font 'Helvetica', 4
justify 'center'
label 'For Free Advice Call\n1-800-HOTLINE\n24 hours a day'

# Draw the same label at several points.
#
clearpath
justify 'left'
color "Orange"
move 85, 5
move 90, 15
move 95, 25
label 'READY'

# Draw symbols from ZapfDingbats font.
```

```
#
color "black"
font 'ZapfDingbats', 12
clearpath
move 105, 10
label '0&'
```

Figure 29: Displaying Labels

Labels are displayed horizontally. Rotate the axes with the `rotate` command before setting the font to display labels at an angle. This is demonstrated in the following example, with output shown in Figure 30.

```
newpage "eps", "tutoriallabels2.eps", 40, 40
clearpath
move 20, 20
let r = 0
while r < 360
do
  color "hsb", r / 360, r / 360, 0.6
  rotate 20
  let r = r + 20
  font 'Times-Roman', 4 + r / 180
  label 'Spirally'
done
```

Figure 30: Rotated Labels

Several methods are available for changing the appearance of labels.

To highlight a label, draw the outline of each letter with a thick line. Then draw the label again at the same position in a different color.

The `stringwidth` function (see Table 3 on page 110) calculates the width of a label. This value is used to underline a label, draw a box surrounding a label, or to join several labels together on a single line.

These techniques are demonstrated in the following example, with output shown in Figure 31.

```
newpage "eps", "tutoriallabels3.eps", 100, 30

# Draw label, then underline it.
#
font "Helvetica-Narrow-Bold", 6
let h1 = "Abcdefg", h2 = "Hij"
move 5, 5
label h1
draw 5 + stringwidth(h1), 5
stroke

# Draw box, then label inside it.
#
clearpath
roundedbox 5, 20 - 2, 5 + stringwidth(h1), 26
fill
clearpath
move 5, 20
color "yellow"
label h1

# Draw one label, then another immediately following it
# in a different color and font.
#
font "Palatino-Roman", 6
clearpath
color "red"
move 40, 4
label h1
color "blue"
shiftpath stringwidth(h1), 0
font "Palatino-Italic", 8
label h2

# Draw yellow label with red outline to highlight it.
#
clearpath
move 35, 16
color "red"
font "Helvetica", 12, "outlinewidth=2"
label h1
color "yellow"
font "Helvetica", 12
label h1
```

```
# Draw only outline of letter.
#
clearpath
move 82, 6
color "black"
font "Helvetica", 24, "outlinewidth=0.2"
label "A"
```

Figure 31: Highlighted Labels

The `flowlabel` command is used to draw a label following along a line. This is useful for labelling streets or rivers.

Use of this command is demonstrated in the following example, with output shown in Figure 32.

```
newpage "eps", "tutorialflowlabel1.eps", 120, 60

# Draw label following J-shaped path.
#
font "Helvetica", 4
move 10, 5
draw 10, 40
arc 1, 20, 40, 30, 40
draw 30, 35
stroke

parallelpath -2
flowlabel 0, 15, "This was drawn with flowlabel command"

# Draw river, with label just above it.
#
clearpath
move 50, 40
draw 55, 42, 60, 41, 64, 43, 69, 43, 74, 45, 79, 46, 85, 50, 93, 51
color "blue"
font "Helvetica-Bold", 4
stroke
parallelpath -1
flowlabel 0.5, 2, "Parramatta River"
```

```
# Draw street with centered name.
#
clearpath
move 50, 20
draw 55, 22, 60, 21, 64, 23, 69, 23, 74, 25, 79, 26, 85, 30, 93, 31
linestyle 4, "round", "round"
color "red"
stroke
linestyle 3, "round", "round"
color "yellow"
stroke
color "black"
parallelpath 1.5
justify "center"
flowlabel 0.25, Mapyrus.path.length / 2, "Panorama Ave"
```

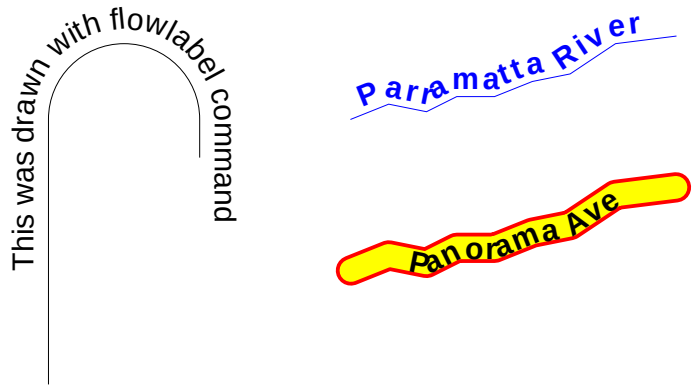

Figure 32: Labels Along a Line

When labelling streets using the `flowlabel` command, labels will appear above or below the street depending on whether the street was digitized left-to-right or right-to-left. To ensure that all streets are labelled on the same side, compare the `Mapyrus.path.start.x` and `Mapyrus.path.end.x` variables. Reverse the direction of the streets digitized in the wrong direction using the `reversepath` command. This is demonstrated in the following example, with two streets digitized in opposite directions. The output is shown in Figure 33.

```
newpage "eps", "tutorialflowlabel2.eps", 120, 30
font "Helvetica", 4
linestyle 1

move 10, 10
draw 50, 20
stroke
if Mapyrus.path.start.x > Mapyrus.path.end.x
```

```
then
  # Street was digitized right-to-left.  Change direction of street.
  reversepath
endif
parallelpath -1
flowlabel 0.1, 3, "R PASTEUR"

clearpath
move 110, 10
draw 70, 20
stroke
if Mapyrus.path.start.x > Mapyrus.path.end.x
then
  # Street was digitized right-to-left.  Change direction of street.
  reversepath
endif
parallelpath -1
flowlabel 0.1, 3, "R VICTOR HUGO"
```

Figure 33: Labelling Streets

The `sinkhole` command is used determine a position for a label in a polygon. This command replaces the current path defining a polygon with a single point in the middle of the polygon. This is demonstrated in the following example, with output shown in Figure 34.

```
# Draw outline of polygon, with label in the middle.
#
begin district name
  stroke
  sinkhole
  font "Helvetica-Narrow-Bold", 3
  justify "center, middle"
  label name
end

# Display labelled polygons.
#
newpage "eps", "tutorialsinkhole.eps", 65, 40
clearpath
move 5, 5
```

```
draw 7, 31, 42, 33, 40, 19, 20, 19, 19, 24, 15, 24, 13, 5, 5, 5
district "Packer"
clearpath
move 13, 5
draw 15, 24, 19, 24, 20, 19, 40, 19, 42, 9, 13, 5
district "Hamilton"
clearpath
move 42, 9
draw 40, 19, 42, 33, 48, 33, 56, 7, 42, 9
district "Ashley"
```

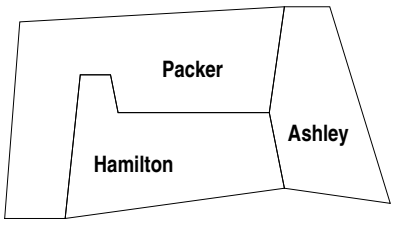

Figure 34: Labelling Polygons

If polygons are partially outside the page then use the `guillotine` command to chop the polygon at the edge of the page before using the `sinkhole` command.

To calculate two label positions for a polygon, use the `guillotine` command to cut the polygon into left and right (or top and bottom) halves and use the `sinkhole` command to calculate a position in each half.

4.8 Displaying Data Stored In Text Files

The simplest source of data is a text file, with one record per line. The filename and type of dataset to read is given in a `dataset` command. Then a `fetch` command is used in a loop to read each record and split it into fields. Fields are assigned to variables `$1`, `$2`, `$3`, ... and the whole record is assigned to variable `$0`.

Coordinates in GIS datasets are normally stored in a world coordinate system such as Universal Transverse Mercator. To transform these coordinates to millimeter coordinates on the page use the `worlds` command. This sets a transformation to map a range of world coordinates onto the whole page. All coordinates given for later `addpath`, `move`, `arc`, `box`, `box3d`, `draw`, `circle`, `ellipse`, `hexagon`, `pentagon`, `raindrop`, `spiral`, `star`, `triangle` and `wedge` commands are converted from world coordinate to page coordinates through this transformation.

The following example demonstrates setting a world coordinate system, reading geographic positions from text file shown in Figure 35, converting the positions to decimal values using the `parsegeo` function and displaying them. The output of this example is shown in Figure 36.

```
# @(#) $Id: tutorialdatasets1.txt,v 1.3 2005/07/16 19:06:57 schenery Exp $
# Latitude/Longitude positions of South Pacific capital cities.
33.83S 148.04E Canberra/Australia
41.08S 174.91E Wellington/New Zealand
22.13S 166.50E Noumea/New Caledonia
17.69S 168.41E Vila/Fiji
 9.48S 159.90E Honiara/Solomon Islands
 9.37S 147.29E Port Moresby/Papua New Guinea
```

Figure 35: Text File tutorialdatasets1.txt

```
begin capitalCity name
  # Draw a labelled dot marking a capital city.
  #
  color "red"
  box -1, -1, 1, 1
  fill

  clearpath
  move 2, 0
  font "Helvetica", 2.5
  color "black"
  label name
end

# Plot the geographic data in text file tutorialdatasets1.txt
# in an Encapsulated PostScript file.
#
newpage "eps", "tutorialdatasets1.eps", 85, 85

dataset "textfile", "tutorialdatasets1.txt", "comment=#"
worlds 150, -45, 180, -5
while Mapyrus.fetch.more
do
  # Draw label at each position read from text file.
  #
  fetch
  print "DEBUG:", $0
  clearpath
  move parsegeo($2), parsegeo($1)
  let cityAndCountryArray = split(substr($0, 16), "/")
  capitalCity cityAndCountryArray[1] . "\n(" . cityAndCountryArray[2] . ")"
done
```

Gazeteer files, GPS waypoint files and export files of GIS datasets are also read as text files.

For files containing lines and polygons, a loop and counter are used to read all coordinates for each line and polygon.

For files containing header lines, use the `fetch` command outside of a loop

Figure 36: Displaying Contents Of A Text File

to skip these lines.

The following example demonstrates reading and displaying the GenaMap GIS ZF19 format export file shown in Figure 37 containing line features. The output is shown in Figure 38.

```
# Draw the GenaMap export file streets.EE into a PNG file.
#
newpage "eps", "tutorialdatasets2.eps", 60, 60
dataset "textfile", "streets.EE", ""
worlds 324846, 1257086, 325300, 1257476
while Mapyrus.fetch.more
do
  # Header line for each feature contains coordinate count.
  #
  fetch
  let nCoords = substr($0, 51, 5)

  # Fetch first point of feature and add it to path.
  #
  clearpath
  fetch
  move $1, $2

  repeat (nCoords - 1)
  do
    # Add the rest of feature's coordinates to the path.
```

```
 1LINE      DOYALSON STREET                      2
325061.6800000000  1257380.2700000000         .0000000000
324930.4500000000  1257397.4400000000         .0000000000
 2LINE      WARD STREET                          5
324875.9000000000  1257182.7600000000         .0000000000
324892.4700000000  1257193.0000000000         .0000000000
324908.5000000000  1257203.0000000000         .0000000000
324956.0000000000  1257257.5000000000         .0000000000
324979.2900000000  1257284.4600000000         .0000000000
 3LINE      CAMBRIDGE AVENUE                     3
325061.6800000000  1257380.2700000000         .0000000000
325107.5000000000  1257433.5000000000         .0000000000
325130.9300000000  1257456.2700000000         .0000000000
 4LINE      ANDERSON ROAD                        3
325257.4600000000  1257100.0800000000         .0000000000
324887.0000000000  1257161.0000000000         .0000000000
324875.9000000000  1257182.2600000000         .0000000000
 5LINE      MILTON STREET                        3
325274.7500000000  1257206.6800000000         .0000000000
325029.0000000000  1257245.0000000000         .0000000000
324979.2100000000  1257284.4800000000         .0000000000
 6LINE      WARD STREET                          2
325061.6800000000  1257380.2700000000         .0000000000
324979.2200000000  1257284.4400000000         .0000000000
```

Figure 37: GIS Export File streets.EE

```
    #
    fetch
    draw $1, $2
  done

  # Draw each line in red.
  #
  color 'red'
  stroke
done
```

4.9 Displaying Data Stored In Shape Files

Reading data from an ESRI Shape file format is similar to reading a text file. A Shape file defines a bounding rectangle so the internal variables `Mapyrus.dataset.min.x`, `Mapyrus.dataset.min.y`, `Mapyrus.dataset.max.x`, and `Mapyrus.dataset.max.y` are available to set world coordinates to the bounding rectangle of the dataset.

An ESRI Shape file also defines field names. Fields fetched with the `fetch` command are assigned to variables with the same name as the field. The geometry for each record is assigned to a variable named `GEOMETRY` and is added to the current path with the `addpath` command.

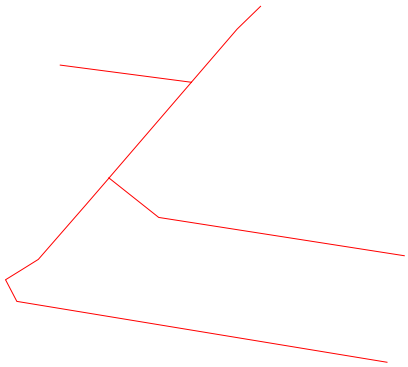

Figure 38: Displaying GIS Export Files

The following example demonstrates reading an ESRI Shape file containing points and attribute fields named `HOTELNAME` and `STARRATING`. The attribute fields are used as labels and to control the appearance of the symbols.

```
begin hotel name, nStars
  # Draw hotel name inside a box the same width as the name.
  #
  font 'Helvetica-Narrow-Bold', 3
  color '#A00000'
  clearpath
  move 2, 0
  box 2, 0, 2 + stringwidth(name), 4
  move 0, 2
  draw 2, 4, 2, 0, 0, 2
  fill
  clearpath
  move 2, 1
  color 'white'
  label name
  # Draw box below hotel name, then show number of stars inside it.
  #
  clearpath
  color '#A00000'
  let stars = '* ' x nStars
  box 2, -2, stringwidth(stars) + 2, 0
  fill
  clearpath
  color 'yellow'
  move 2, -2
  label stars
end
```

```
# Display the ESRI Shape file hotel.shp in a PostScript file.
#
newpage "eps", "tutorialdatasets3.eps", 65, 50
dataset "shapefile", "hotel.shp", ""
# Expand bounding box to the right so labels don't run off edge of page.
#
worlds Mapyrus.dataset.min.x, Mapyrus.dataset.min.y, \
  Mapyrus.dataset.max.x + Mapyrus.dataset.width / 5, Mapyrus.dataset.max.y
while Mapyrus.fetch.more
do
  fetch
  clearpath
  addpath GEOMETRY
  print "DEBUG:", GEOMETRY, HOTELNAME, STARRATING
  hotel HOTELNAME, STARRATING
done
```

Figure 39: Displaying ESRI Shape Files

4.10 Displaying OpenStreetMap Data

The OpenStreetMap Protocol enables street map data to be fetched using HTTP requests.

The following example demonstrates reading and displaying street data from the OpenStreetMap server.

```
newpage "eps", "tutorialdatasets4.eps", 210, 297
let x1 = 151.1, y1 = -34, x2 = 151.2, y2 = -33.9
let url = "http://api.openstreetmap.org/api/0.5/map?bbox=" . \
  x1 . "," . y1 . "," . x2 . "," . y2
dataset "osm", url, ""

worlds x1, y1, x2, y2

while Mapyrus.fetch.more
do
```

```
  fetch
  if TYPE eq "way"
  then
    if TAGS["highway"] eq "primary"
    then
      linestyle 0.7
      color "red"
    elif TAGS["highway"] eq "secondary"
    then
      linestyle 0.4
      color "orange"
    elif TAGS["natural"] eq "coastline"
    then
      linestyle 0.1
      color "lightgreen"
    else
      linestyle 0.1
      color "black"
    endif
    clearpath
    addpath GEOMETRY
    stroke
  endif
done
```

To avoid repeated requests to the OpenStreetMap server it is also possible to save the OpenStreetMap server output to a file and give the filename in the `dataset` command instead.

4.11 Displaying Data Stored In A Database

A simplistic method of reading from a relational database is to use a database front-end program to print selected records to a file and then to read this in Mapyrus.

Given the file `simplistic.mapyrus` containing the following commands,

```
newpage "png", "simplistic.png", 100, 100, "background=white"
worlds 1000, 1000, 2000, 2000
dataset "textfile", "-", "delimiter=|"
while Mapyrus.fetch.more
do
  fetch
  clearpath
  color $3
  star $1, $2, 3, 5
  fill
done
```

Then data read from an sqlite database is displayed as stars in a PNG image using commands like:

```
sqlite db77 "select X, Y, Color from SITES" > sites.txt
java -classpath mapyrus.jar org.mapyrus.Mapyrus simplistic.mapyrus < sites.txt
```

On UNIX operating systems, pipe the output directly from the database front-end program to Mapyrus. This is demonstrated in the following example, displaying roads fetched as OGC WKT strings from a PostGIS database.

```
psql -A -t -c 'select AsText(Geom) from ROADS' pogo | \
  java -classpath mapyrus.jar org.mapyrus.Mapyrus -e '
newpage "png", "simplistic.png", 100, 100, "background=white"
worlds 170000, 470000, 220000, 540000
dataset "textfile", "-", "delimiter=|"
while Mapyrus.fetch.more
do
  fetch
  clearpath
  addpath $1
  stroke
done'
```

A more efficient solution is to read from a relational database within Mapyrus using the Java JDBC interface and a JDBC driver provided as part of the database. The JAR file containing the JDBC driver must be included in the `-classpath` option when Mapyrus is run.

For example, when accessing a PostgreSQL database from a Linux machine, ensure that the `postgresql-jdbc` package is installed and use a command like:

```
java -classpath mapyrus.jar:/usr/share/pgsql/pg73b1jdbc1.jar \
  org.mapyrus.Mapyrus tutorialdataset5.mapyrus
```

The JDBC driver for PostgreSQL is available from `http://jdbc.postgresql.org`.

A JDBC driver for MySQL is available from `http://www.mysql.com/products/connector/j`.

A JDBC driver for Oracle is included in the `$ORACLE_HOME/jdbc` directory of an Oracle installation.

Each field in a database table has a name. Fields fetched with the `fetch` command are assigned to variables with the same name as the field. In the following example three fields are fetched from each row and assigned to variables named `longitude`, `latitude` and `assetcode`.

An SQL where clause is used to limit data read from the database to inside the area of interest.

```
begin asset
  # Draw filled square
  #
  box -1, -1, 1, 1
  fill
end

# Display the geographic data held in RDBMS table in a PNG file.
#
```

```
newpage "png", "tutorialdatasets5.png", 60, 60

let x1 = 151.03, y1 = -31.25, x2 = 151.04, y2 = -31.24

# Build SQL statement to fetch point data in area of interest.
#
let sql = "select Assetcode, Longitude, Latitude from SURVEY \
where Logdate > '1 Dec 2001' and LogDate < '15 Dec 2001' \
and Longitude >= " . x1 . " and Latitude >= " . y1 . " \
and Longitude <= " . x2 . " and Latitude <= " . y2

dataset "jdbc", sql, "driver=org.postgresql.Driver \
  url=jdbc:postgresql:nemo user=postgres password=postgres"

# Print names of fields being fetched from database for debugging,
# as some databases convert all field names to uppercase or lowercase.
#
let i = 1
while i <= length(Mapyrus.dataset.fieldnames)
do
  print "DEBUG: ", Mapyrus.dataset.fieldnames[i]
  let i = i + 1
done

# Fetch and draw each point, varying color depending on asset code.
#
worlds x1, y1, x2, y2
while Mapyrus.fetch.more
do
  clearpath
  fetch
  move longitude, latitude
  if assetcode eq "BN" or assetcode eq "BZ"
  then
    color "red"
  elif assetcode eq "CN"
  then
    color "blue"
  else
    color "green"
  endif

  asset
done
```

For databases supporting the *OpenGIS Simple Features Specification For SQL*, add fields containing OGC WKT geometry strings or WKB geometry values to the current path with an `addpath` command. Using WKT geometry strings is less efficient than WKB geometry strings because all geometry must be converted to a text string and then back to a geometry.

The following example demonstrates fetching a geometry field named `geom` from a PostGIS database as a WKT string and displaying it.

```
# Display the geographic data held in a PostGIS database
# in a PostScript file.
#
newpage "eps", "tutorialdatasets6.eps", 85, 85

let x1 = 191232, y1 = 243117, x2 = 191234, y2 = 243119

# Build spatially extended SQL statement to fetch
# road network data inside an area of interest.
#
let sql = "SELECT AsText(GEOM) AS GEOM FROM ROADS_GEOM \
WHERE GEOM && GeometryFromText('BOX3D(" . x1 . " " . y1 . \
"," . x2 . " " . y2 . ")'::box3d,-1)"

# Fetch each road as an OGC WKT geometry string and draw it.
#
dataset "jdbc", sql, "driver=org.postgresql.Driver \
url=jdbc:postgresql:pogo user=postgres password=postgres"
worlds x1, y1, x2, y2
while Mapyrus.fetch.more
do
  clearpath
  fetch
  addpath geom
  stroke
done
```

The next example demonstrates fetching geometry from an Oracle database as WKB geometry values and displaying it. The `Get_WKB` function is used to convert Oracle Spatial geometry to WKB geometry values.

```
# Display the spatial data held in an Oracle database
# in a PostScript file.
newpage "eps", "tutorialdatasets7.eps", 85, 85
worlds 0, 0, 20, 20

let sql = "select C.Shape.Get_WKB() AS WKB \
  from COLA_MARKETS C where C.Name<>'cola_d'"

dataset "jdbc", sql, "driver=oracle.jdbc.OracleDriver \
  url=jdbc:oracle:oci:@DEMO user=system password=manager"

while Mapyrus.fetch.more
do
  fetch
  clearpath
  addpath WKB
  stroke
```

```
done
```

Mapyrus is also able to read Oracle Spatial columns with type `MDSYS.SDO_GEOMETRY` directly without the `Get_WKB` function. When reading Oracle Spatial columns directly the `sdoapi.jar` JAR file from an Oracle database installation containing Oracle Spatial data types must also be included in the Java classpath.

4.12 Displaying Geo-Referenced Images

To display a geo-referenced image use the `geoimage` command. A "worlds" file with `.tfw` suffix is required, containing the world coordinate range covered by the image.

The following example demonstrates displaying the image `australia.png` as a background image, then displaying the data contained in an ESRI Shape file over the image. The output is shown in Figure 40.

```
newpage "eps", "tutorialgeoimage1.eps", 70, 70

dataset "shapefile", "coastline.shp", ""
worlds Mapyrus.dataset.min.x, Mapyrus.dataset.min.y, \
  Mapyrus.dataset.max.x, Mapyrus.dataset.max.y

geoimage "australia.png", "brightness=0.9"

color "red"
linestyle 1, "round", "round"
while Mapyrus.fetch.more
do
  clearpath
  fetch
  addpath GEOMETRY
  stroke
done
```

Some sets of geo-referenced images are provided as a grid of trapezoidal shaped images. To display several of these images it is necessary to clip each image as it is displayed to avoid overwriting neighbouring images. To define a clip polygon for a geo-referenced image, use the `clipfile` option to the `geoimage` command.

An alternative method of clipping an image is to use the `clip` command to set a clip path to polygons read from a dataset. The following example demonstrates this, clipping the image so that only parts of the image inside coastline polygons are displayed. The output is shown in Figure 41.

```
newpage "eps", "tutorialgeoimage2.eps", 70, 70

dataset "shapefile", "coastline.shp", ""
worlds Mapyrus.dataset.min.x, Mapyrus.dataset.min.y, \
  Mapyrus.dataset.max.x, Mapyrus.dataset.max.y

# Save all coastline polygons into path, then clip to path.
```

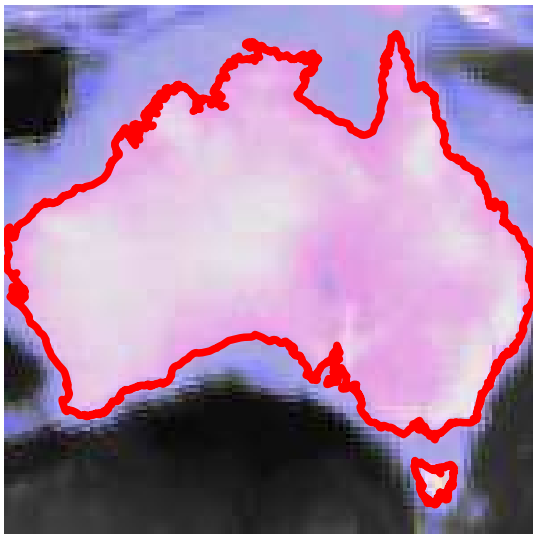

Figure 40: Displaying Geo-Referenced Images

```
#
while Mapyrus.fetch.more
do
  fetch
  addpath GEOMETRY
done

clip "inside"

# Draw image, clipped to coastline.
#
geoimage "australia.png"
```

4.13 Displaying Images From An OGC Web Mapping Service

An OGC Web Mapping Service (WMS) is an HTTP server that provides geo-referenced images. To fetch a geo-referenced image from a WMS and display it on the page use the `geoimage` command.

The following example demonstrates building a URL for a WMS request from a UMN Mapserver, defining the layers to be fetched and world coordinate range of the area of interest. The UMN Mapserver must be configured to return 24 bit PNG images as Mapyrus cannot read the default PNG format returned by Mapserver.

```
newpage "eps", "tutorialgeoimage3.eps", 85, 85

let x1 = 393381, y1 = 5207990, x2 = 495758, y2 = 5305370
worlds x1, y1, x2, y2
```

Figure 41: Clipping Geo-Referenced Images

```
# Build URL to fetch area covered by page from a Web Mapping Service.
#
let url = "http://tardis/cgi-bin/mapserv?map=demo.map&"
let url = url . "SERVICE=WMS&VERSION=1.1.1&REQUEST=GetMap&"
let url = url . "LAYERS=dlgstln2,cities&STYLES=&SRS=EPSG:26915&"
let url = url . "FORMAT=image/png&"
let url = url . "BBOX=" . x1 . "," . y1 . "," . x2 . "," . y2 . "&"
let url = url . "WIDTH=" . round((x2 - x1) / 100) . "&"
let url = url . "HEIGHT=" . round((y2 - y1) / 100)

# Fetch and display image.
#
geoimage url
```

4.14 Displaying Many Datasets or Geo-Referenced Images

To display data contained in many files, use the `listfiles` function to obtain a list of files in one or more directories matching a filename pattern. Then display each dataset in a loop. Displaying from many ESRI Shape files is demonstrated in the following example.

```
let allFiles = dir("H:\\data\\shape\\*\\*.shp")

newpage "eps", "tutorialdatasets7.eps", 180, 260
worlds 1600000, 610000, 1770000, 710000

for i in allFiles
do
```

```
  dataset "shapefile", allFiles[i], "dbffields="
  while Mapyrus.fetch.more
  do
    fetch
    clearpath
    addpath GEOMETRY
    stroke
  done
done
```

A common method of efficiently selecting the data that overlaps the page from hundreds of possible datasets and geo-referenced images is to create an index ESRI Shape file with a polygon containing the bounding rectangle of each dataset or image.

This ESRI Shape file index is then searched for overlapping polygons.

For ESRI Shape file datasets, the `tile4ms` program included in UMN Mapserver[2] is used to create an ESRI Shape file index.

For geo-referenced images, the `gdaltindex` program included in the gdal[3] library is used to create an ESRI Shape file index.

The following example demonstrates searching an ESRI Shape file index for images to display.

```
newpage "eps", "tutorialdatasets8.eps", "A4"
worlds 12000, 17000, 16000, 21000

# Find polygons (that represent images) in ESRI Shape file index
# that overlap world coordinates set for page.
#
let nTiles = 0
dataset "shapefile", "sydney-index.shp", \
    "xmin=" . Mapyrus.worlds.min.x . " ymin=" . Mapyrus.worlds.min.y . \
    " xmax=" . Mapyrus.worlds.max.x . " ymax=" . Mapyrus.worlds.max.y
while Mapyrus.fetch.more
do
    fetch
    let nTiles = nTiles + 1

    # Geo-referenced image filename stored in LOCATION attribute field.
    #
    let tiles[nTiles] = LOCATION
done

# Display each overlapping image.
#
let i = 1
while i <= nTiles
do
    geoimage tiles[i]
```

[2]Available from http://mapserver.gis.umn.edu
[3]Available from http://www.gdal.org

```
    let i = i + 1
done
```

4.15 Updating Existing Output Files

To edit or draw over an existing image file or PostScript file use the `update=true` option to the `newpage` command. This allows further data to be drawn over a base map, or a watermark or logo to be added to a map.

Updating works for both files created by Mapyrus and files created by other software.

This is demonstrated in the following example, drawing a grid over the output file `tutorialexisting1.eps` (a copy of the output file from Figure 36 from page 43). The output of this example is shown in figure Figure 42.

```
# Open existing file 'tutorialexisting1.eps' for update.
# New page size is ignored, the page size of existing file is used.
#
newpage "eps", "tutorialexisting1.eps", 1, 1, "update=true"
worlds 150, -45, 180, -5

# Draw vertical grid lines.
#
color "grey"
let x = floor(Mapyrus.worlds.min.x / 5) * 5
while x < Mapyrus.worlds.max.x
do
  clearpath
  move x, Mapyrus.worlds.min.y
  draw x, Mapyrus.worlds.max.y
  stroke
  let x = x + 5
done

# Draw horizontal grid lines.
#
let y = floor(Mapyrus.worlds.min.y / 5) * 5
while y < Mapyrus.worlds.max.y
do
  clearpath
  move Mapyrus.worlds.min.x, y
  draw Mapyrus.worlds.max.x, y
  stroke
  let y = y + 5
done
```

4.16 Display Performance

The following techniques help improve the display speed of large datasets.

Check the internal variable `Mapyrus.worlds.scale` before displaying a large, detailed dataset. If the scale is too high then the details will not be legible so

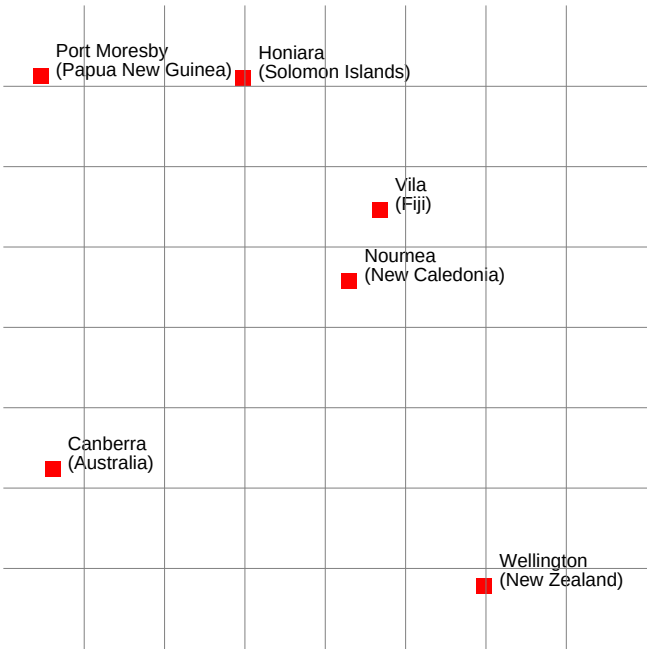

Figure 42: Updating An Existing Output File

skip display of the detailed part of the dataset. For example, display all streets when zoomed in, but only highways when zoomed out.

Store two copies of the same data. One at high resolution and one at low resolution. Select the dataset with the resolution closest to the current display scale.

For dense data, with many points very close to each other, skip every second piece of data in the dataset using two `fetch` commands together. When the dataset is in a database this is more efficiently done by ignoring odd numbered rows using an SQL statement like:

```
select X, Y, Freq from SCATTER where MOD(rowid, 2) = 0
```

Set an upper limit on the number of rows fetched or the time for display by checking the internal variables `Mapyrus.fetch.count` and `Mapyrus.timer` in a loop and ending when the limit is exceeded.

4.17 Displaying A Legend

A cartographic rule is that a map display must include a legend. The `key` command is used to save each entry to be include in a legend and the `legend` command displays a legend.

A `key` command in a procedure defines a legend entry. Mapyrus saves each legend entry encountered whilst executing commands. Each legend entry has a description label and a type, defining whether the entry appears as a point, line or box in the legend.

When display of all data is complete, points for legend entries are defined with `move` commands and the `legend` command is used to display each legend

entry. Each legend entry is automatically displayed by calling the procedure in which it was defined.

If insufficient `move` points are defined then some legend entries remain undisplayed.

Display the legend in a separate image file to prevent overwriting the map.

This is demonstrated in the following example. The two output files are shown in Figure 43. Note that the legend entry for the procedure `road` is not included in the legend because this procedure is not executed in the example.

```
begin river
  # Display current path as blue line, signifying a river.
  #
  key "line", "River"
  color "blue"
  linestyle 1
  stroke
end

begin road
  # Display current path as red line, signifying a road.
  #
  key "line", "Road"
  color "red"
  linestyle 0.1
  stroke
end

begin lake
  # Fill current path with light blue, signifying a lake.
  #
  key "box", "Lake"
  color "cyan"
  fill
end

begin church
  # Display a church symbol at current point.
  #
  key "point", "Church"
  color "black"
  linestyle 0.1
  move -1, -1
  arc 1, 0, -1, -1, -1
  fill
  clearpath
  move 0, 0
  draw 0, 2
  move -1, 1
  draw 1, 1
  stroke
```

```
end

newpage "eps", "tutoriallegend1.eps", 50, 30, "background=grey90"

clearpath
move 5, 5
draw 7, 22, 6, 29, 23, 27, 21, 9, 5, 5
lake

clearpath
move 27, 3
draw 34, 28
river
clearpath
move 29, 9
draw 40, 7, 47, 3
river

clearpath
move 40, 16
church

# Draw legend for map in a separate PostScript file, giving plenty
# of move points for the legend entries.
#
newpage "eps", "tutoriallegend1legend.eps", 40, 40
clearpath
color "black"
move 5, 5, 5, 12, 5, 19, 5, 26, 5, 33
justify "middle"
font "Helvetica", 2.5
legend 4
```

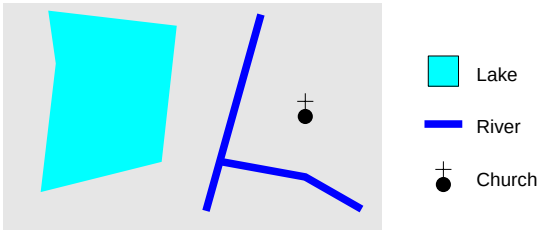

Figure 43: Displaying A Legend

To display each legend entry in a separate file use a loop, checking the internal variable `Mapyrus.key.count` to find how many legend entries remain to be displayed.

In each loop iteration, create a new output file, set a single `move` point and

then display a single legend entry with a `legend` command. Continue until all legend entries are displayed.

This is demonstrated in the following example, with each separate legend entry PostScript file inserted into a table. The output is shown in Figure 44. This approach is also useful for inserting legend entries into an HTML table.

```
begin monument
  # Display monument symbol at current point.
  #
  key "point", "Monument"
  color "black"
  move 0, 0
  draw -1, -2, 1, -2, 0, 0
  draw 0.5, 0.5, 0, 1, -0.5, 0.5, 0, 0
  fill
end
begin ruins
  # Display archaelogical ruins symbol at current point.
  #
  key "point", "Archaelogical Ruins"
  color "black"
  circle 0, 1.5, 0.5
  circle -1, 0, 0.5
  circle 1, 0, 0.5
  fill
end
begin church name, ruined
  # Display a church or church ruins symbol at current point on path.
  # Provide legend entry for each type of church.
  #
  key "point", "Church", "", 0
  key "point", "Church Ruins", "", 1
  # Draw name of church.
  #
  clearpath
  font "Helvetica", 2.5
  justify "middle"
  move 2, 0
  label name
  # Turn church symbol on its side if flagged as ruined.
  #
  if ruined == 1
  then
    rotate -30
  endif
  color "black"
  linestyle 0.1
  box -1, 0, 1, -2
  fill
  clearpath
```

```
  move 0, 0
  draw 0, 2
  move -1, 1
  draw 1, 1
  stroke
end

newpage "eps", "tutoriallegend2.eps", 50, 30, "background=grey90"
clearpath
move 5, 5
monument
clearpath
move 7, 22
monument
clearpath
move 24, 19
ruins
clearpath
move 32, 16
church "Ospringe", 1

# Draw each legend entry in a separate PostScript file, giving
# a single move point in each file to draw the next legend entry.
#
let counter = 0
while Mapyrus.key.count > 0
do
  let counter = counter + 1
  let filename = "tutoriallegend2legend" . counter . ".eps"
  newpage "eps", filename, 50, 8
  clearpath
  move 1, 1
  color "black"
  justify "middle"
  font "Helvetica", 2.5
  legend 6
done
```

To show the number of times a legend entry has been used on a map, include the special string `(#)` in legend entry descriptions. This string is replaced in the legend by the number of times the legend entry was encountered whilst executing commands. This feature is demonstrated in the following example, with output shown in Figure 45.

```
begin dormantcell
  key "box", "Dormant Cell (#) occurrences"
  color "skyblue"
  fill
  color "black"
  stroke
```

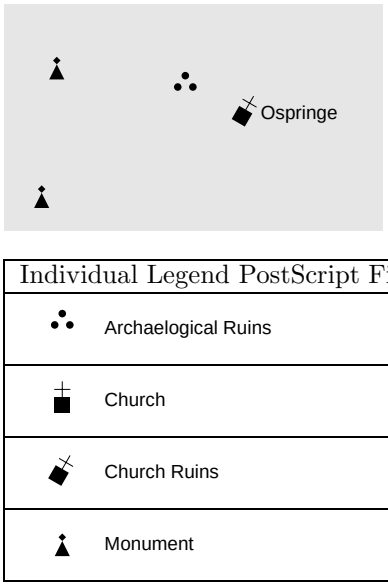

Figure 44: Displaying Legend Entries Individually

```
end
begin activecell
  key "box", "Active Cell (#) occurrences"
  color "red"
  fill
  color "black"
  stroke
end

newpage "eps", "tutoriallegend3.eps", 45, 30

clearpath
hexagon 10, 18, 8
dormantcell

clearpath
hexagon 22, 10, 8
activecell

clearpath
hexagon 34, 18, 8
activecell

# Make separate PostScript file containing legend.
#
newpage "eps", "tutoriallegend3legend.eps", 45, 40
clearpath
color "black"
```

```
move 5, 5, 5, 12, 5, 19, 5, 26, 5, 33
justify "middle"
font "Helvetica", 2.5
legend 4
```

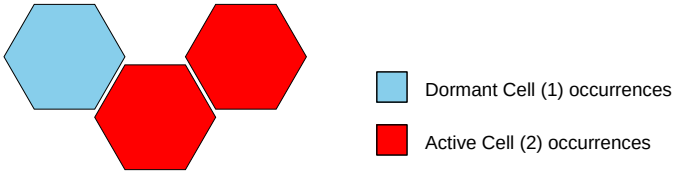

Figure 45: Displaying Frequency Count Of Legend Entries

4.18 Using Attributes To Control Display

Use attributes of geographic data to control color, size, shape, labelling, highlighting and alignment of symbols, lines and polygons. Use `if-then-endif` statements to display different classes of data differently. For example, display private, local, and major roads with different types of lines.

To display data with a known range of attribute values in different colors based on the attribute value, define colors for minimum and maximum values and use interpolation to determine the color for each point, line or polygon. This is demonstrated in the following example by passing a temperature attribute value for each point that is displayed. The output is shown in Figure 46. A legend is also created to show how the coloring works.

```
begin temperature t
  # Define legend entries for various temperatures.
  #
  key "point", "temperature 20\260C", 20
  key "point", "temperature 25\260C", 25
  key "point", "temperature 30\260C", 30
  key "point", "temperature 35\260C", 35

  # Temperature is a value in range 20-35. Convert it to
  # a color in the range green-yellow-red.
  #
  color interpolate("20 green 25 yellow 35 red", t)
  box -2, -2, 2, 2
  fill
end

# Display map of Australia.   Then read temperature data from a file
# and display January temperatures for some cities, with different
```

```
# colors representing different temperatures.
#
newpage "eps", "tutorialattribute1.eps", 60, 40
dataset "shapefile", "coastline.shp", "dbffields="
worlds -2800000, 4800000, 2150000, 9190000
while Mapyrus.fetch.more
do
  fetch
  clearpath
  addpath GEOMETRY
  stroke
done
dataset "textfile", "aust_cities.dat", ""
while Mapyrus.fetch.more
do
  fetch
  let cityname = $1
  fetch
  let x = $1, y = $2
  clearpath
  move x, y
  fetch
  let januaryTemperature = $1
  temperature januaryTemperature
  fetch
done

# Draw legend in separate PostScript file.
#
newpage "eps", "tutorialattribute1legend.eps", 40, 50, \
  "isolatinfonts=Helvetica"
color "black"
move 5, 5, 5, 10, 5, 15, 5, 20
font "Helvetica", 2.5
legend 4
```

Another possibility is varying the size of a symbol depending on an attribute value. The next example demonstrates this, using the population value for each city to control the height of a cylinder drawn at each point. The output is shown in Figure 47.

```
begin populationCylinder name, pop
  # Define legend entries for various populations.
  #
  key "point", " 200000 people", "", 200000
  key "point", "1000000 people", "", 1000000
  key "point", "2000000 people", "", 2000000
  key "point", "5000000 people", "", 5000000

  # Population is a value in range 200000-5million. Convert
```

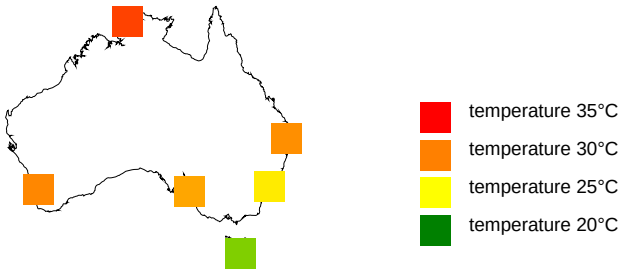

Figure 46: Using Attributes

```
  # it to a height for the cylinder in the range 2mm-8mm.
  #
  let height = interpolate("200000 2 5000000 8", pop)

  cylinder 0, 0, 4, height
  color "lightorange"
  fill
  color "black"
  stroke

  clearpath
  color "maroon"
  move 0, -6
  font "Helvetica-Bold", 3
  justify "center"
  label name
end

# Display map of Australia.   Then display populations of
# Australian cities, with different size cylinders representing
# different population levels.
#
newpage "eps", "tutorialattribute2.eps", 80, 70
dataset "shapefile", "coastline.shp", "dbffields="
worlds -2800000, 4800000, 2300000, 9190000
while Mapyrus.fetch.more
do
  fetch
  clearpath
  addpath GEOMETRY
  color "lightgray"
  stroke
done
```

```
dataset "textfile", "aust_cities.dat", ""
while Mapyrus.fetch.more
do
  fetch
  let city = $1, population = $2
  fetch
  let x = $1, y = $2
  clearpath
  move x, y
  fetch      # skip lines containing temperature data.
  fetch
  populationCylinder city, population
done

# Draw legend in separate PostScript file.
#
newpage "eps", "tutorialattribute2legend.eps", 40, 50
font "Helvetica-Bold", 3
color "black"
move 5, 5, 5, 15, 5, 25, 5, 35
legend 4
```

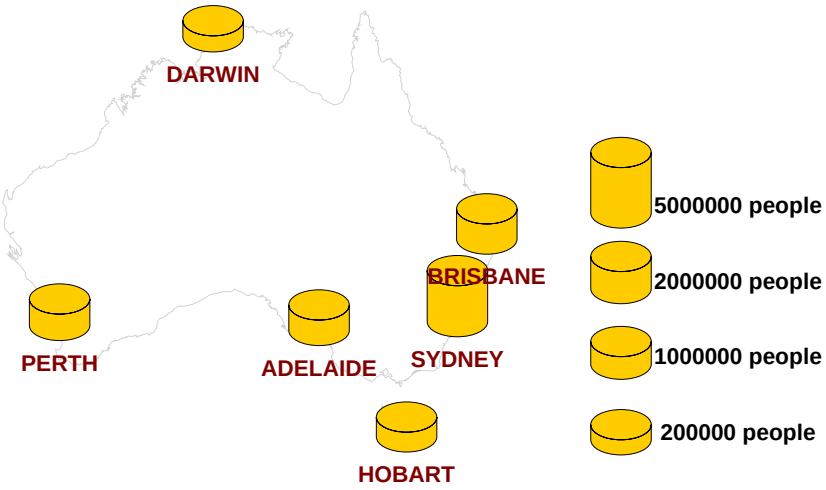

Figure 47: Using More Attributes

Further examples are varying line width depending on traffic between two points, varying labelling depending on the importance of data, or varying the symbols plotted inside polygons depending on soil or rock type.

Extend this technique to display two attribute values together to show relationships between the attributes (such as the correlation between fox populations and rabbit populations). Alternatively, display the same attribute measured at different times to show trends in the attribute data.

To display values of several attributes at point locations, draw a piechart or histogram (see page 68) at each point.

When reading from a relational database table, SQL aggregate functions are available for calculating statistics from the data.

The following SQL statement demonstrates calculating how many standard deviations each reading is from the average. Use the result of this calculation to control the display of the data, showing readings that are far from average differently to readings that are close to average.

```
select X, Y, (Reading - (select AVG(Reading) from Pollution)) /
(select STDDEV(Reading) from Pollution) as R from Pollution
```

4.19 Displaying A Scalebar

Another cartographic rule is that a map display must include a scalebar. The following example creates two PostScript files using the internal variable `Mapyrus.worlds.scale` to add a scale bar to the lower-left corner of each map display.

The included file `scalebar.mapyrus` (available in the `userdoc` subdirectory) defines a procedure named `scalebar`. This procedure performs the task of displaying the scalebar. Include this file and call the `scalebar` procedure whenever a scalebar is required for a map display.

Output from the example is shown in Figure 48.

An alternative approach for displaying scalebars is to save the scale value in a variable, use the `newpage` command to create a new output file and display the scalebar in a separate file from the map.

```
# Include file containing procedure to display scalebar.
#
include scalebar.mapyrus

newpage "eps", "tutorialscalebar1.eps", 60, 60, "background=pastelblue"

# Display entire Australian continent.
#
dataset "shapefile", "coastline.shp", ""
worlds -3000000, 5500000, 2600000, 8500000
while Mapyrus.fetch.more
do
  fetch
  clearpath
  addpath GEOMETRY
  color '#669900'
  fill
  color '#333333'
  linestyle 0.1
  stroke
done

# Add a scalebar.
#
```

```
clearpath
scalebar Mapyrus.worlds.scale, "m", 0, 0

# Begin a new page.
#
newpage "eps", "tutorialscalebar2.eps", 60, 60, "background=pastelblue"

# Display only northern part of Northern Territory.
#
dataset "shapefile", "coastline.shp", ""
worlds -1300000, 7700000, 300000, 8900000
while Mapyrus.fetch.more
do
  fetch
  clearpath
  addpath GEOMETRY
  color '#669900'
  fill
  color '#333333'
  linestyle 0.1
  stroke
done

# Add a scalebar to this map display too.
#
clearpath
scalebar Mapyrus.worlds.scale, "m", 0, 0
```

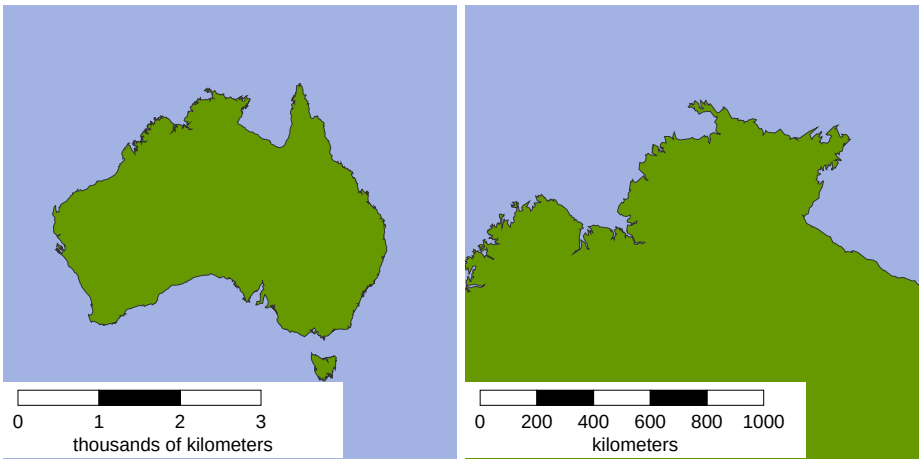

Figure 48: Displaying A Scalebar

4.20 Displaying Piecharts And Histograms

Piecharts and histograms are used to represent relative values of several attributes at different locations. For example, production levels or voting patterns. The following example demonstrate this using fixed data values. In a real application this information is read from a database or GIS dataset.

The included files `piechart.mapyrus` and `histogram.mapyrus` (available in the `userdoc` subdirectory) define procedures named `piechart` and `histogram`. These procedures perform the task of displaying piecharts and histograms. Include one of these files and call the `piechart` or `histogram` procedure whenever piecharts or histograms are to be displayed.

The output of this example is shown in Figures 49 and 50.

```
# Include file containing procedure to display piechart.
#
include piechart.mapyrus

newpage "eps", "tutorialpiechart1.eps", 90, 40

# Set fixed colors and labels for all pies.
#
let colors[1] = "green", colors[2] = "orange"
let colors[3] = "yellow", colors[4] = "#606c30"

let labels[1] = "Apples", labels[2] = "Oranges"
let labels[3] = "Bananas", labels[4] = "Olives"

# Set production levels for first site.  Then draw piechart.
#
let production[1] = 7000, production[2] = 2000
let production[3] = 1500, production[4] = 500
clearpath
move 22, 15
piechart 4, production, labels, colors

# Set production levels for second site.  Then draw piechart.
# Pass dummy variable for labels array to demonstrate omitting labels.
#
let production[1] = 5000, production[2] = 3000
let production[3] = 2500, production[4] = 500
clearpath
move 42, 28
piechart 4, production, dummy, colors

# Set production levels for third site.  Then draw piechart.
#
let production[1] = 8000, production[2] = 2000
let production[3] = 4500, production[4] = 3500
clearpath
move 70, 14
```

```
piechart 4, production, labels, colors

# Include file containing procedure to diplay histogram.
#
include histogram.mapyrus

newpage "eps", "tutorialhistogram1.eps", 80, 20

# Draw small histogram with four values without any labels.
#
move 20, 7
let maxProduction = 10000
histogram colors, dummy, production, maxProduction, 6

# Set different production levels and draw another small histogram.
#
let production[1] = 6000, production[2] = 4000
let production[3] = 3500, production[4] = 4500
clearpath
move 40, 14
histogram colors, dummy, production, maxProduction, 6

# Draw a larger histogram with a label for each bar.
#
clearpath
move 60, 5
histogram colors, labels, production, maxProduction, 12
```

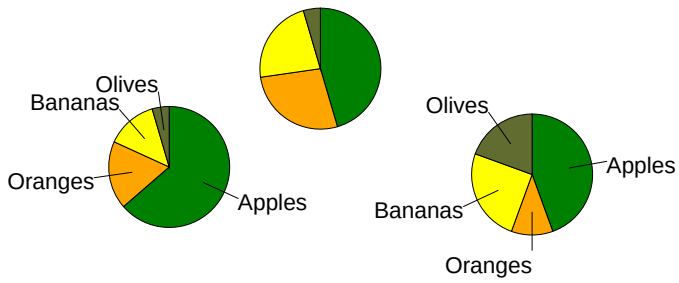

Figure 49: Displaying Piecharts

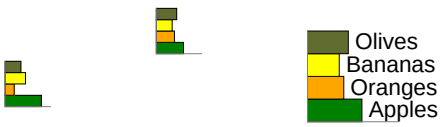

Figure 50: Displaying Histograms

4.21 Random Effects

Random effects are generated using the `random` function and help give a map a less mechanical appearance. Random effects are also useful for giving data that is not precise a blurred, inaccurate appearance. The following example demonstrates setting color randomly. The output is shown in Figure 51.

```
begin flower
  # Draw flower with 12 randomly colored petals.
  #
  repeat 12
  do
    clearpath
    ellipse 5, 0, 3, 1.2

    let r = random(3)
    if r < 1
    then
      color "pastelblue"
    elif r < 2
    then
      color "pastelpink"
    else
      color "pastelgreen"
    endif
    fill

    color "black"
    linestyle 0.1
    stroke

    rotate 30
  done
end

newpage "eps", "tutorialrand1.eps", 40, 40
move 9, 8
move 24, 10
move 8, 31
move 25, 28
flower
```

The next example demonstrates setting rotation randomly, with output shown in Figure 52.

```
begin cobblestone
  # Draw a square area of cobblestones, rotated
  # a random multiple of 90 degrees
  #
  let r = round(random(4))
  rotate r * 90
```

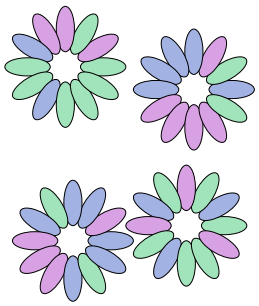

Figure 51: Random Color

```
  color "black"
  linestyle 0.2
  move -1.8, -1.8
  draw -1.6, -0.2, -0.3, -0.3, -0.4, -1.5, -1.8, -1.8
  move 0.3, -1.7
  draw 0.2, 1.8, 1.8, 1.7, 1.6, -1.8, 0.3, -1.7
  move -1.7, -0.3
  draw -1.8, 1.8, -0.2, 1.6, -0.3, 0.2, -1.7, -0.3
  stroke
end

newpage "eps", "tutorialrand2.eps", 40, 40
clearpath
move 4, 8
draw 36, 8, 36, 32, 12, 32, 12, 24, 4, 24
samplepath 4, 0
cobblestone
```

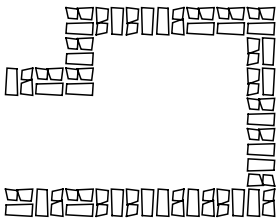

Figure 52: Random Rotation

The next example demonstrates setting position randomly for polygon fill, with output shown in Figure 53.

```
function randomLetter letters
  # Return a random letter from a list of letters
  #
```

```
  local n
  let n = substr(letters, random(length(letters)), 1)
  return n
end

begin placeLetter
  # Place random letter in random position.
  #
  move random(2), random(2)
  color "#0000A0"
  font "Helvetica", 3
  label randomLetter("ABCDEFGHIJKLMNOPQRSTUVWXYZ")
end
begin alphabetSoup
  # Draw polygon outline, then grid of letters inside polygon.
  #
  color "black"
  linestyle 0.1
  stroke
  clip "inside"
  stripepath 4, 0
  samplepath 4, 0
  placeLetter
end

newpage "eps", "tutorialrand3.eps", 40, 40
clearpath
move 5, 5
draw 7, 32, 31, 29, 27, 8, 5, 5
alphabetSoup
```

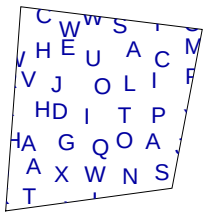

Figure 53: Random Position

4.22 Using Transparency

To set a partially transparent color in Mapyrus, give a transparency value (also known as an alpha value) for the color. The background will be partly visible behind shapes and labels drawn with transparent colors.

The following example demonstrates using transparent colors, with output shown in Figure 54.

Transparent colors cannot be set in PostScript files using `eps` or `ps` format output. Transparent colors are only available in a Encapsulated PostScript file containing an image created using `epsimage` format output.

```
newpage "epsimage", "tutorialtrans1.eps", 60, 50, "background=white"

color "lightgray"
clearpath
box 0, 25, 60, 50
fill

clearpath
color "red", 0.5   # half transparent
box 5, 15, 45, 40
fill

clearpath
color "green", 0.8   # mostly opaque (solid)
box 15, 5, 35, 35
fill

clearpath
color "blue", 0.2   # mostly transparent
box 25, 7, 50, 45
fill

clearpath
font 'Times-Roman-Bold', 9
color "black", 0.1  # nearly transparent
move 0, 22
label "Transparent"
```

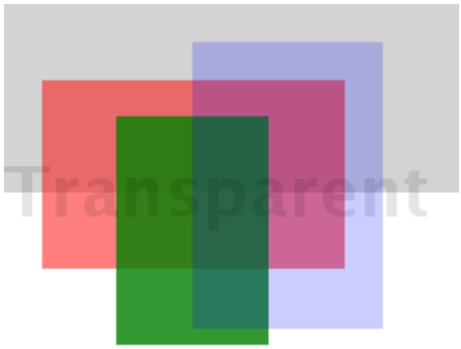

Figure 54: Transparent Colors

Transparency is also used to produce fading effects. This technique is demon-

strated in the following example, with lines fading from an opaque color to transparent. The output of this example is shown in Figure 55.

```
newpage "epsimage", "tutorialtrans2.eps", 80, 40, "background=white"

color "lightgray"
chessboard 0, 0, 80, 40, 5
fill

begin fadingDot startingColor
  color startingColor, pow(fadingCounter / fadingLength, 2)
  let fadingCounter = fadingCounter - 1
  circle 0, 0, 1
  fill
end

# Draw closely packed circles along path that fade
# from starting color to transparent at end of line.
#
begin fadingLine startingColor
  let fadingLength = Mapyrus.path.length
  let fadingCounter = fadingLength
  samplepath 1, 0
  fadingDot startingColor
end

clearpath
move 5, 30
draw 75, 35
fadingLine "blue"

clearpath
move 10, 18
draw 75, 8
fadingLine "red"
```

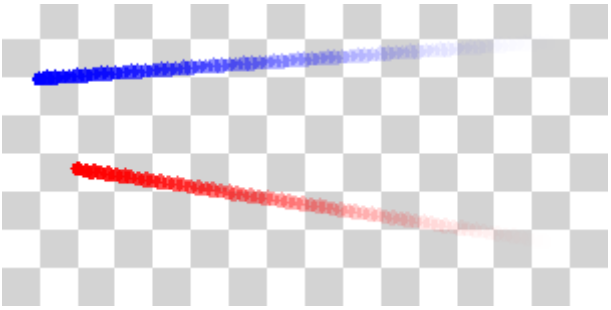

Figure 55: Fading Lines

When creating PDF output, the method of blending transparent colors with the background is modified using the `blend` command.

4.23 Shadow Effects

To highlight polygons with a shadow, draw the outline of the polygon, use a `clip "outside"` command to prevent the interior of the polygon being drawn, then repeatedly use the `shiftpath` command to move the polygon a small distance from its original position, drawing it each time. This is demonstrated in the following example, with output shown in Figure 56. Change the sign of the shift values in the `shiftpath` command to draw the highlight on a different side of the polygon.

```
begin shadowed2
  # Protect polygon interiors and draw shadow offset from polygon.
  #
  color '#cccc00'
  linestyle 0.1
  clip "outside"
  repeat 7
  do
    shiftpath 0.1, -0.1
    stroke
  done
end
begin shadowed
  # Display a polygon with shadow to the bottom-left.
  #
  shadowed2
  linestyle 0.1
  color 'black'
  stroke
end

# Fetch all polygons from coastline.shp into current path.
#
dataset "shapefile", "coastline.shp", "dbffields="
newpage "eps", "tutorialshadow1.eps", 60, 60
worlds -2800000, 4800000, 2150000, 9190000
clearpath
while Mapyrus.fetch.more
do
  fetch
  addpath GEOMETRY
done

# Draw all polygons with shadows together so that shadow from one
# polygon does not interfere with the interior of another polygon.
#
shadowed
```

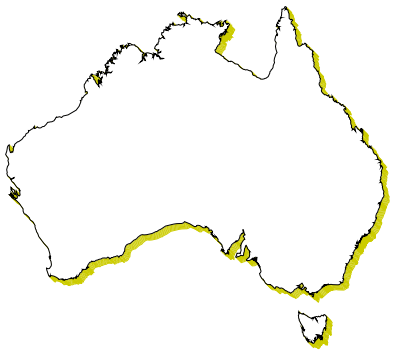

Figure 56: Shadow Effects

4.24 Displaying Tables

Attribute information read from a dataset is often best displayed in a table.

Tables are displayed in Mapyrus by creating arrays containing the row and column values and then using the `table` command.

The following example shows the positions of several vehicles, giving information about each vehicle as a table at the vehicle position. The output of this example is shown in Figure 57.

```
begin vehiclebox reg, cargo, driver, time
  local a, b
  let a[1] = reg, a[2] = cargo, b[1] = driver, b[2] = time

  # Draw vehicle info as two column table with arrow pointing left
  #
  font 'Helvetica-Narrow', 3
  color "Black"
  clearpath
  move 2, 4
  table "background=orange", a, b
  draw 2, -4, 0, 0, 2, 4
  fill
end

newpage "eps", "tutorialtable1.eps", 95, 30
move 22, 20
vehiclebox "MAC-259", "FROZEN", "Smith", "Bris 9:20"

clearpath
move 65, 14
vehiclebox "MTT-257", "FRESH", "Jones", "Syd 2:10"
```

Other uses of tables are to show a summary of the data displayed on the

Figure 57: Displaying Tables In A Map

page and to show a title block for a page.

The following example demonstrates both types of tables. The output of this example is shown in Figure 58.

```
begin hotelref counter
  box -1, -1, 1, 1
  stroke
  clearpath
  move 1.5, 0
  label counter
end

newpage "eps", "tutorialtable2.eps", 100, 50
color "black"
font 'Helvetica-Narrow-Bold', 3

# Display the ESRI Shape file hotel.shp on left side of page.
#
dataset "shapefile", "hotel.shp", ""
worlds Mapyrus.dataset.min.x, Mapyrus.dataset.min.y, \
  Mapyrus.dataset.max.x, Mapyrus.dataset.max.y, 0, 0, 60, 50
let n = 1
while Mapyrus.fetch.more
do
  fetch
  clearpath
  addpath GEOMETRY
  hotelref n

  # Save attribute information for each hotel in arrays for display in a table.
  #
  let ref[n] = n
  let name[n] = HOTELNAME
  let rating[n] = "*" x STARRATING
  let n = n + 1
done

# Display reference table for map and title block on right side of page.
#
```

```
worlds 0, 0, 100, 50
clearpath
move 60, 45
table "background=grey90,grey90,grey90,white,white,white", ref, name, rating
clearpath
move 60, 20
table "background=grey90", split("Title Block,Author,Date", ",")
```

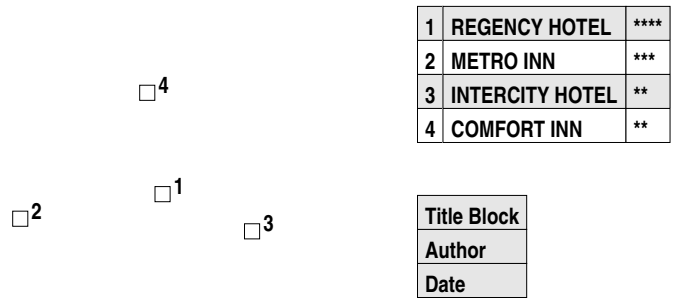

Figure 58: Displaying Map And Tables Separately

Give several background colors in the `table` command to make rows or columns in a table easier to read. For example, in a table with 2 columns, using background colors `white,white,yellow,yellow` results in rows having white and yellow backgrounds alternately.

4.25 Wordwrapped Labels

To split a long sentence into several lines for display as a label, use the `wordwrap` function. This function breaks lines at word boundaries like a word-processor and is useful for displaying speech bubbles, or tooltip type labels on the page. The following example shows sentences being word-wrapped and displayed inside a box using the `table` command. The output of this example is shown in Figure 59.

```
begin caption message, width
  local t

  # Word wrap message then display it as a table with only one entry.
  #
  font 'Helvetica-Narrow', 3
  let t[1] = wordwrap(message, width)
  color "Black"
  linestyle 0.1
  clearpath
  move 0, 0
  table "background=LightYellow", t
end
```

```
newpage "eps", "tutorialwordwrap1.eps", 95, 50
move 20, 24
caption "SYDNEY: A cloudy morning followed by a partly cloudy afternoon.
Mostly fine apart from the chance of a shower early.  Light to moderate
W/NW winds, turning moderate SW during the day.", 35

clearpath
move 30, 44
caption "BRISBANE: Rain easing to showers.  Mild to warm with moderate
to fresh gusty S/SW winds.", 35
```

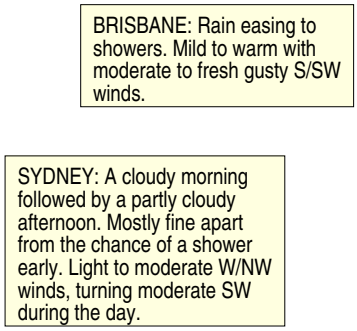

Figure 59: Wordwrapped Labels

4.26 Avoiding Overlapping Labels

Mapyrus stores a list of areas on the page that have been marked as protected by the `protect` command. To avoid overlapping labels, check that the area that will be covered by a label is not protected before displaying the label using the `stringwidth` and `protected` functions. After displaying a label use the `protect` command to mark the area covered by the label as protected.

This is demonstrated in the following example, with labels drawn to the right or left of a point, to avoid overlapping other labels. Output is shown in Figure 60.

```
begin populatedPlace name
  box -0.5, -0.5, 0.5, 0.5
  fill
  clearpath

  # Calculate size of place name label, then draw it to
  # the right or the left of the place, wherever there is
  # space that has not been used before.
  #
  let h = 4
  font "Helvetica", h
```

```
  let w = stringwidth(name)
  color "yellow"
  if not protected(1, 0, w + 1, h)
  then
    # Draw label inside a box to right of point.
    # Then mark labelled area as protected.
    #
    box 1, 0, w + 1, h
    fill
    clearpath
    move 1, 0
    justify "left"
    color "black"
    label name
    protect 1, 0, w + 1, h
  elif not protected(-1, 0, -(w + 1), h)
  then
    # Draw label inside a box to left of point.
    # Then mark labelled area as protected.
    #
    box -1, 0, -(w + 1), h
    fill
    clearpath
    move -1, 0
    justify "right"
    color "black"
    label name
    protect -1, 0, -(w + 1), h
  endif
end

# Draw coastline.
#
newpage "eps", "tutorialprotect1.eps", 75, 50
dataset "shapefile", "coastline.shp", ""
worlds 600000, 5100000, 1500000, 6100000
while Mapyrus.fetch.more
do
  fetch
  clearpath
  addpath GEOMETRY
  stroke
done

# Read positions and names of towns and cities from file.
#
dataset "textfile", "locations.txt", "delimiter=,"
while Mapyrus.fetch.more
do
  fetch
```

```
  clearpath
  move $1, $2
  populatedPlace $3
done
```

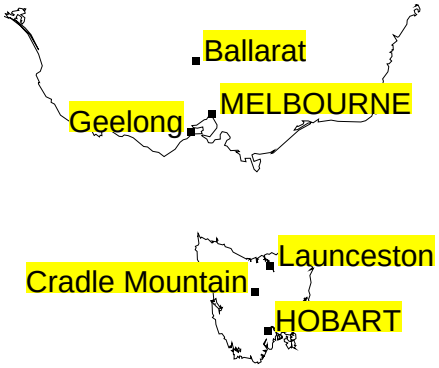

Figure 60: Label Positioning

For best results, display the most important labels first to maximise the chances of finding a position for a label that does not overlap previous labels. For example, label large towns before villages.

This method can be extended to draw labels left, right, above, below or offset from their true position, wherever they do not overlap other labels.

Areas of the page containing important symbols or icons can be protected too. However, in many cases it is simpler to display most important data last to ensure it is not overwritten.

Use the `unprotect` command to clear protected areas.

To avoid labels overlapping other important data on the page, set the area to be protected in the path and use the `protect` command with no arguments to protect that area.

This is demonstrated in the following example, with labels drawn in random positions but avoiding overlaps with other data already displayed on the page. Output is shown in Figure 61.

```
newpage "eps", "tutorialprotect2.eps", 75, 50

# Draw a shape then mark that area as protected.
#
move 10, 10
draw 60, 40, 55, 45, 5, 15, 10, 10
stroke
protect

font "Helvetica", 4
color "blue"
let w = stringwidth("ABCD")
let counter = 0
```

```
# Draw 25 randomly positioned labels, avoiding overlaps with anything
# we have already drawn.
#
while counter < 25
do
  let x = random(75)
  let y = random(50)
  if (not protected(x, y, x + w, y + 4))
  then
    clearpath
    move x, y
    label "ABCD"
    protect x, y, x + w, y + 4
    let counter = counter + 1
  endif
done
```

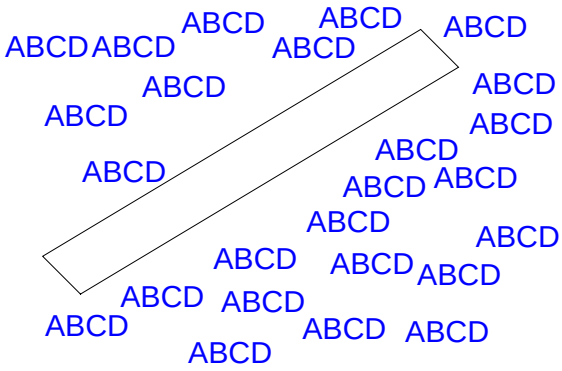

Figure 61: More Label Positioning

4.27 Displaying Image Icons

Display color image icons and clip-art using the `icon` command. Icons are scaled and rotated to any size and rotation angle. Displaying icons is demonstrated in the following example with output shown in Figure 62.

```
# Draw icons with different sizes and rotations.
#
newpage "eps", "tutorialicon1.eps", 40, 40
move 10, 15
icon "footballpitch.gif"
clearpath
move 25, 15
icon "footballpitch.gif", 15
clearpath
rotate 30
```

```
move 30, 20
icon "footballpitch.gif", 12
```

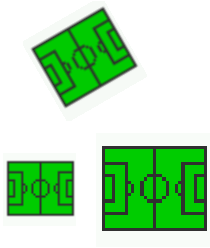

Figure 62: Icon Display

Combine the `icon` command with `stripepath` and `samplepath` commands to repeat the icon in a tile pattern inside a polygon. This is demonstrated in the following example with output shown in Figure 63.

```
begin singleIcon
  move 0, 0
  color "firebrick"
  icon "10000000
        11111110
        10000010
        10111010
        10100010
        10101110
        10100000
        10111111", 6
end

# Fill polygon with tiled icon image pattern.
#
begin iconPattern
  clip "inside"
  stripepath 6, 0
  samplepath 6, 0
  singleIcon
end

# Draw polygon filled with icons.
#
newpage "eps", "tutorialicon2.eps", 50, 50
clearpath
move 5, 5
draw 12, 29, 24, 43, 44, 37, 46, 29, 36, 26, 46, 25, 49, 7, 5, 5
iconPattern
linestyle 1.5, "round", "round"
```

```
stroke
```

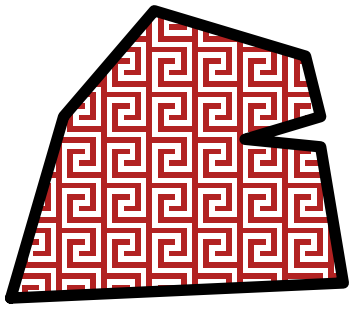

Figure 63: Icon Fill Pattern

Another use for icons is to achieve a spray paint effect, with scattered dots. This effect is demonstrated in the following example with output shown in Figure 64.

```
begin spray
  move 0, 0
  rotate random(360)
  color "limegreen"
  icon "00000000
        01000100
        00010010
        00101000
        01000010
        00010100
        01000001
        00101010", 3
end

# Fill polygon with spray paint pattern.
#
begin sprayPattern
  stroke
  clip "inside"
  stripepath 3, 0
  samplepath 3, 0
  spray
end

# Read map and plot it.
#
dataset "shapefile", "coastline.shp", "dbffields="
newpage "eps", "tutorialicon3.eps", 60, 60
worlds -2800000, 4800000, 2150000, 9190000
```

```
clearpath
while Mapyrus.fetch.more
do
  fetch
  clearpath
  addpath GEOMETRY
  sprayPattern
done
```

Figure 64: Spray Paint Pattern

4.28 Including Encapsulated PostScript Files

When creating PostScript or Encapsulated PostScript output it is possible to display the contents of another Encapsulated PostScript file on the page one or more times.

Both Encapsulated PostScript files Files generated by Mapyrus and other software may be included.

Using an Encapsulated PostScript file named `flag.eps` in a fill pattern is demonstrated in the following example with output shown in Figure 65.

```
# Fill polygon with Aboriginal flag pattern.
#
begin flagPattern
  clip "inside"
  stripepath 5, 0
  samplepath 6, 0
  eps "flag.eps", 5
end

# Read map and plot it.
#
dataset "shapefile", "coastline.shp", "dbffields="
```

```
newpage "eps", "tutorialeps1.eps", 50, 50
worlds Mapyrus.dataset.min.x, Mapyrus.dataset.min.y, \
  Mapyrus.dataset.max.x, Mapyrus.dataset.max.y
clearpath
while Mapyrus.fetch.more
do
  fetch
  clearpath
  addpath GEOMETRY
  flagPattern
done
```

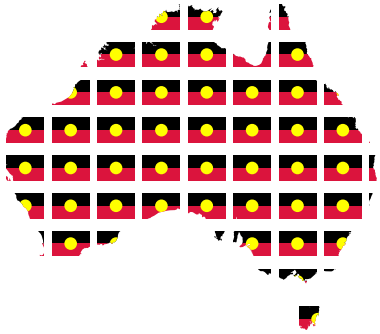

Figure 65: Displaying An Encapsulated PostScript File

4.29 Mapyrus and Java Topology Suite Functions

Advanced geometric functions in Mapyrus (see Figure 3 on page 110) are provided by the *Java Topology Suite*, available from (`http://www.vividsolutions.com/JTS`).

If any of these functions are used then the *Java Topology Suite* JAR file must be included in the classpath when running Mapyrus. The following command demonstrates this:

```
java -classpath install-dir/mapyrus.jar:jts-dir/jts-1.8.jar \
  org.mapyrus.Mapyrus tutorialjts1.mapyrus
```

The following example demonstrates using the `buffer` function to create a hatched buffer around a line with the output shown in Figure 66.

```
newpage "eps", "tutorialjts1.eps", 60, 60
worlds 5000, 1000, 6000, 2000

# Demonstrate drawing buffer around a line.  For simplicity, use
# only a single line.  In reality, geometries would be read
# from a dataset.
#
let wkt = 'LINESTRING (5200 1200, 5337 1664, 5798 1643, 5347 1120)'
```

```
addpath wkt
linestyle 3
color "orange"
stroke

clearpath
addpath buffer(wkt, 100, "round")
linestyle 0.1
color "red"
stroke
# Fill buffered area with a hatch pattern.
#
clip "in"
stripepath 3, -45
stroke
```

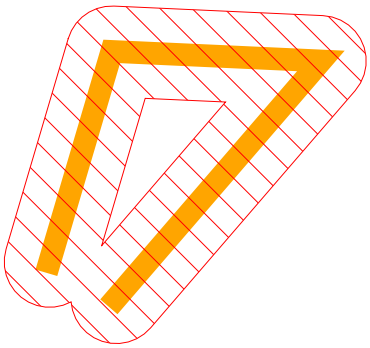

Figure 66: Buffer Function

The `contains` function is used to determine whether one geometry is contained inside another. This is useful for displaying sample points differently, depending on which polygon they are inside. A simple use of the `contains` function is demonstrated in the following example with the output shown in Figure 66.

```
begin locpt name
  box -1, -1, 1, 1
  fill
  clearpath
  move 2, 0
  font "Helvetica", 3.5
  label name
end

let polygonColorList = split("red blue green orange")
```

```
# Draw all polygons in different colors and also save them into a list.
#
newpage "eps", "tutorialjts2.eps", 85, 50
dataset "shapefile", "coastline.shp", ""
worlds 600000, 5100000, 1500000, 6100000
while Mapyrus.fetch.more
do
  fetch
  let polygonList[Mapyrus.fetch.count] = GEOMETRY
  clearpath
  addpath GEOMETRY
  color "lightgray"
  fill
  color polygonColorList[Mapyrus.fetch.count]
  stroke
done
let nPolygons = length(polygonList)

# Read list of points, finding which polygon contains each point,
# and then drawing the point in a different colour depending on
# which polygon contains the point.
#
dataset "textfile", "locations.txt", "delimiter=,"
while Mapyrus.fetch.more
do
  fetch
  let found = 0, i = 1
  while found == 0 and i <= nPolygons
  do
    if contains(polygonList[i], $1, $2)
    then
      # This polygon contains point, draw point in color of polygon.
      #
      color polygonColorList[i]
      clearpath
      move $1, $2
      locpt $3
      let found = 1
    endif
    let i = i + 1
  done
done
```

Combine the `buffer` and `contains` functions to display data differently depending on whether it falls within a certain distance of another geometry.

Use the `overlaps` function to test whether two polygons or two lines overlap.

Use the `difference`, `intersection` and `union` functions to perform operations on geometries. Use of the `difference` function is demonstrated in the following example to find the parts of a polygon that are not inside a second

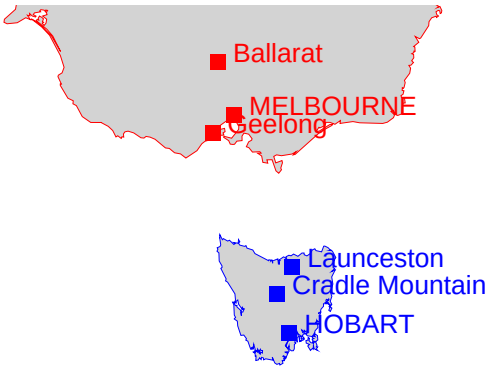

Figure 67: Contains Function

polygon. The output of this example is shown in Figure 68.

```
newpage "eps", "tutorialjts3.eps", 60, 60
worlds 5000, 1000, 6000, 2000

let wkt1 = 'POLYGON ((5200 1200, 5337 1664, 5798 1643, 5347 1120, 5200 1200))'
let wkt2 = 'POLYGON ((5100 1300, 5100 1900, 5500 1800, 5700 1350, 5100 1300))'

# Draw both polygons in different colors.
#
linestyle 3
clearpath
addpath wkt1
color "purple"
stroke
clearpath
addpath wkt2
color "forestgreen"
stroke

# Draw area of first polygon not covered by second polygon.
#
let wkt3 = difference(wkt1, wkt2)
clearpath
addpath wkt3
linestyle 1
color "yellow"
stroke
```

4.30 Creating Landscape Output on Portrait Pages

Many PostScript printers accept only portrait orientation pages. To print a landscape orientation page, give the `turnpage=true` option to the `newpage` com-

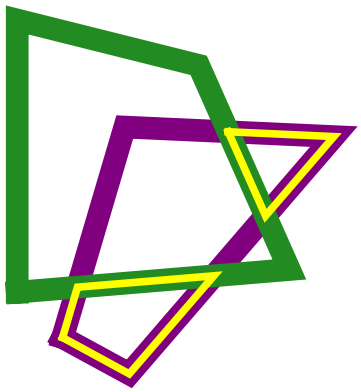

Figure 68: Difference Function

mand. This option turns a landscape page 90 degrees to fit on a portrait page and is demonstrated in the following example, with output shown in Figure 69.

```
# Create landscape page 50mm wide and 30mm high.
newpage "eps", "tutorialturnpage1.eps", 50, 30, "turnpage=true"
# Draw page border.
linestyle 4
box 0, 0, Mapyrus.page.width, Mapyrus.page.height
stroke
# Draw horizontal label on page.
clearpath
move 5, 10
font "Palatino", 8
label "Horizontal"
```

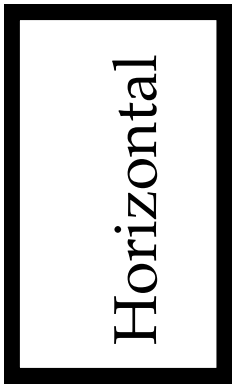

Figure 69: Rotated Landscape Page

4.31 Page Layout With Mapyrus

Place a map, a legend, a scalebar, a title, a logo and other information on a page to create a complete page layout.

Calculate a page position in millimeters at which to display each item. A map is displayed in a given region of the page using the `worlds` command with both world coordinates and a page position.

The following example demonstrates placing each item on an A6 size page with output shown in Figure 70 on page 106.

```
include scalebar.mapyrus

# Settings for page layout.  To use this a template, read these settings
# from files using commands like 'let title=spool("mytitle.txt")' instead.
#
let title = "My Title"
let map = "coastline.shp"

newpage "eps", "tutoriallayout1.eps", "A6", "background=lightgrey"

# Draw the title and logo.
#
move 10, 125
font "Palatino-Roman", 24
color "black"
label title
clearpath
move 80, 10
icon "mapyrus.png"

begin land
  key "box", "Land"
  color "khaki"
  fill
  color "black"
  stroke
end

begin sea
  key "box", "Sea"
  color "seablue"
  fill
end

begin drawMap
  # Set world coordinates in center of page, then draw map.
  #
  clearpath
  box 4, 20, 78, 115
  stroke
```

```
  dataset "shapefile", map, ""
  worlds -1300000, 7000000, 300000, 8900000, 10, 25, 75, 110

  clearpath
  box -1300000, 7000000, 300000, 8900000
  clip "inside"
  sea

  while Mapyrus.fetch.more
  do
    fetch
    clearpath
    addpath GEOMETRY
    land
  done
  let mapScale = Mapyrus.worlds.scale
end

# Draw map in procedure so world coordinates and clip
# rectangle are restored when procedure finishes.
#
clearpath
drawMap

# Now draw the legend to the right of the map.
#
clearpath
move 85, 100, 85, 90
font "Palatino-Roman", 4
legend 5

# Draw a table to the right of the map too.
#
font "Palatino-Roman", 3.5
clearpath
move 82, 80
table "", split("State ACT NSW NT QLD SA TAS VIC WA"), \
  split("Rating B A- C AA B C+ D B")

# Finally draw a scalebar below the map.
#
clearpath
scalebar mapScale, "m", 4, 5
```

Use a single page layout as a template for many map plots. Read the map title, map filename, colors and other settings from files at the start of each plot. An example of this is creating a daily weather map, with the map and title changing each day.

An alternative page layout method is to create three Encapsulated PostScript files for the map, legend and scalebar and to position these Encapsulated PostScript

files on a page using page layout software such as MicroSoft Word.

4.32 Creating Multiple Page Output

To create multi-page PDF files, create each page as a separate PostScript file in Mapyrus. Then use the following GhostScript command to merge the pages into a single PDF file:

```
gs -dNOPAUSE -sDEVICE=pdfwrite -sOutputFile=p.pdf p1.ps p2.ps -c quit
```

4.33 Using PostScript Fonts In PostScript Output

PostScript printers and the Ghostscript program understand a limited number of fonts, normally only 35 basic fonts including AvantGarde, Bookman, Courier, Helvetica, NewCenturySchlbk, Palatino, Symbol, Times-Roman, ZapfChancery, ZapfDingbats and the bold and oblique (italic) variations of these fonts.

When a PostScript printer prints a file containing an unknown font, a substitute font is used.

To make a font known to the printer, include the ASCII definition of the font in the PostScript file being printed.

An ASCII PostScript Type 1 font is defined in two files with suffixes `.pfa` and `.afm`. PostScript font definition files are included in a PostScript file created by Mapyrus using the *extras* `pfafiles` and `afmfiles` options to the `newpage` command. Include filenames of PostScript font definition files for all fonts to be used in the page that are not known by the printer.

If extended characters from any fonts are to be used then also include the list of font names in the `isolatinfonts` option to the `newpage` command.

The first line of an ASCII PostScript Type 1 font definition file contains the name of the font defined in the file.

To convert a True Type Font file to an ASCII PostScript Type 1 font use the ttf2pt1[4] program. The following command converts the font in the file `BEANTOWN.TTF` to a PostScript Type 1 font in files `BEANTOWN.pfa` and `BEANTOWN.afm`.

```
ttf2pt1 -e BEANTOWN.TTF BEANTOWN
```

4.34 Using PostScript Fonts In PDF Output

PDF format defines the following 14 standard fonts that are always available: Times-Roman, Times-Bold, Times-Italic, Times-BoldItalic, Helvetica, Helvetica-Bold, Helvetica-Oblique, Helvetica-BoldOblique, Courier, Courier-Bold, Courier-Oblique, Courier-BoldOblique, Symbol, ZapfDingbats.

For other fonts, the definition of the font must be included in the PDF file as a binary PostScript Type 1 font.

A binary PostScript Type 1 font is defined in two files with suffixes `.pfb` and `.afm`. A binary PostScript Type 1 font is included in a PDF file created by Mapyrus using the *extras* `pfbfiles` and `afmfiles` options to the `newpage` command. Include filenames of PostScript font definition files for all non-standard fonts to be used in the PDF file.

[4] Available from http://ttf2pt1.sourceforge.net

If extended characters from any fonts are to be used then also include the list of font names in the `isolatinfonts` option to the `newpage` command.

Binary PostScript Type 1 font definition files are binary files. However, many parts are plain text and the first plain text line in the file with `.pfb` suffix contains the name of the font defined in the file.

To convert a True Type Font file to a binary PostScript Type 1 font use the ttf2pt1[5] program. The following command converts the font in the file `BEANTOWN.TTF` to a binary PostScript Type 1 font in files `BEANTOWN.pfb` and `BEANTOWN.afm`.

```
ttf2pt1 -b BEANTOWN.TTF BEANTOWN
```

4.35 Using TrueType Fonts In Output to Image Formats

For output to image formats, TrueType format fonts are used.

MicroSoft Windows and Macintosh operating systems support TrueType fonts directly and all installed TrueType fonts are available for use in the `font` command. Additional TrueType fonts are installed using operating system commands.

A TrueType font is defined in a binary file with suffix `.ttf`. The name of a TrueType font normally does not match the filename exactly. MicroSoft Windows Explorer and font display programs show the name of a font contained in a TrueType font file.

Other operating systems do not support TrueType fonts. To use TrueType fonts on other operating systems using the *extras* `ttffiles` option to the `newpage` command. Include filenames of TrueType font definition files to be used in the page.

Some TrueType fonts do not include a complete set of characters. For example, letters with accents or the copyright symbol are often missing. To view the available characters in a font on MicroSoft Windows use the Character map program in the Program-Accessories menu. On UNIX operating systems use the font editor pfaedit[6].

4.36 Using Fonts In SVG Output

When creating Scalable Vector Graphics format output files, limit the use of fonts to commonly used fonts such as `Courier` and `Arial`, or fonts that are known to be available in the software to be used to display the SVG output files.

4.37 Running Mapyrus As An HTTP Server

Enter the following lines into a text file named `tutorialhttpserver1.mapyrus`.

```
httpresponse "HTTP/1.0 200 OK
Content-Type: image/png"

# Display the file coastline.shp in a PNG file that
```

[5] Available from http://ttf2pt1.sourceforge.net
[6] Available from http://pfaedit.sourceforge.net

```
# is sent to standard output.
#
newpage "png", "-", 100, 100
color "red"
dataset "shapefile", "coastline.shp", ""
worlds Mapyrus.dataset.min.x, Mapyrus.dataset.min.y, \
  Mapyrus.dataset.max.x, Mapyrus.dataset.max.y
while Mapyrus.fetch.more
do
  clearpath
  fetch
  addpath GEOMETRY
  stroke
done
```

Copy the file `coastline.shp` from the `userdoc` subdirectory in the Mapyrus installation into the same directory as the file `tutorialhttpserver1.mapyrus`. In a terminal window, change to the directory containing the two files and start Mapyrus as an HTTP server on port 8410 with the following command (where *install-dir* is the directory in which Mapyrus is installed).

`java -classpath` *`install-dir`*`/mapyrus.jar org.mapyrus.Mapyrus -s 8410`

Then enter the URL `http://localhost:8410/tutorialhttpserver1.mapyrus` in a web browser. Mapyrus receives the request, executes the commands in the file `tutorialhttpserver1.mapyrus` and returns the PNG image that is written to standard output to the web browser.

To return a PDF file instead of a PNG image, enter the following lines into a text file named `tutorialhttpserver2.mapyrus`.

```
httpresponse "HTTP/1.0 200 OK
Content-Type: application/pdf"

# Display the file coastline.shp in a PDF file.
#
newpage "pdf", "-", 100, 100
color "forestgreen"
dataset "shapefile", "coastline.shp", "dbffields="
worlds Mapyrus.dataset.min.x, Mapyrus.dataset.min.y, \
  Mapyrus.dataset.max.x, Mapyrus.dataset.max.y
while Mapyrus.fetch.more
do
  clearpath
  fetch
  addpath GEOMETRY
  stroke
done
```

Then enter the URL `http://localhost:8410/tutorialhttpserver2.mapyrus` in a web browser. Mapyrus generates a PDF file instead of a PNG image and returns it to the web browser.

To return a Scalable Vector Graphics (SVG) file instead of a PNG image, enter the following lines into a text file named `tutorialhttpserver3.mapyrus` and enter the URL `http://localhost:8410/tutorialhttpserver3.mapyrus` in a web browser.

```
httpresponse "HTTP/1.0 200 OK
Content-Type: image/svg+xml"

# Display the file coastline.shp in an uncompressed SVG
# file that is sent to standard output.
#
newpage "svg", "-", 100, 100
dataset "shapefile", "coastline.shp", ""
worlds Mapyrus.dataset.min.x, Mapyrus.dataset.min.y, \
  Mapyrus.dataset.max.x, Mapyrus.dataset.max.y
while Mapyrus.fetch.more
do
  clearpath
  fetch
  addpath GEOMETRY
  color "Sandy Brown"
  fill
  color "Black"
  stroke
done
```

4.38 Passing Variables To Mapyrus HTTP Server Through URLs

To vary the display that is created by the Mapyrus HTTP Server, include variables in the URL. For example,

```
http://localhost:8410/tutorialurl.mapyrus?x=11.13&y=48.24&pLabels=off
```

The variables `X`, `Y` and `PLABELS` are automatically set in Mapyrus with the values passed in the URL before interpreting the commands in the file `tutorialurl.mapyrus`. Note that variable names are always uppercase.

For applications that manage state and provide a graphical user interface (such as clients written in Java or PHP), use this method for returning data to the application.

4.39 Returning HTML Pages From Mapyrus HTTP Server

An application for web browsers is based on HTML pages. Generate HTML in Mapyrus and return it with `print` commands. Generate temporary images for the HTML page with unique filenames using the `tempname` function and return references to the images in the HTML.

The example file `tutorialhtmlpage1.mapyrus` demonstrates this, displaying ESRI Shape file `coastline.shp` and text file `aust_temperatures.dat` and using an HTML form to allow the user to change the map display. These three

files are stored in the `userdoc` subdirectory in a Mapyrus installation. In a terminal window change to this directory and start Mapyrus with the HTTP Server option, as described in section 4.37. Then enter the following URL in a web browser.

```
http://localhost:8410/tutorialhtmlpage1.mapyrus
```

To simplify editing of HTML pages for an application, setup template HTML pages with placeholders for the information that Mapyrus provides. This enables the HTML interface to be designed independently from Mapyrus.

In Mapyrus, return the HTML pages with the placeholders replaced by real values using the `replace` function. The Mapyrus commands in the file `tutorialhtmlpage2.mapyrus` and the template HTML file `tutorialhtmlpage2.txt` demonstrate this. Both files are found in the `userdoc` subdirectory.

4.40 Using JavaScript with Mapyrus HTTP Server

Images generated by Mapyrus are made interactive by defining hyperlinks and JavaScript functions to execute when the mouse is moved or clicked over the image. This is achieved by using an HTML imagemap for an image.

A file containing an HTML imagemap is created using the *extras* `imagemap` option to the `newpage` command.

A hyperlink or JavaScript function is defined for an area in the image using the `eventscript` command. The list of areas in the image and hyperlinks and JavaScript functions are written to the imagemap file.

The image and imagemap file are then combined in an HTML file and returned to the web browser.

The following example file `tutorialhtmlpage3.mapyrus` demonstrates this, displaying a tooltip when the mouse is moved over Australia.

```
httpresponse "HTTP/1.0 200 OK
Content-Type: text/html"

# Create image of Australia and imagemap to show a tooltip when
# mouse moved over Australia.
let mapFilename = tempname(".html"), pngFilename = tempname(".png")
newpage "png", pngFilename, 90, 90, "imagemap=" . mapFilename
color "forestgreen"

dataset "shapefile", "coastline.shp", "dbffields="
worlds -2800000, 4800000, 2150000, 9190000
while Mapyrus.fetch.more
do
  fetch
  clearpath
  addpath GEOMETRY
  fill

  # Define event for area covered by current path.
  eventscript "onMouseOver=\"return overlib('Australia');\" \
    onMouseOut=\"return nd();\" href=\"http://www.abc.net.au\""
```

```
done
endpage

# Return HTML page containing image and imagemap we created, using
# overlib.js JavaScript library from http://www.bosrup.com/web/overlib/
# to provide tooltips over the image.
print '<html><head>'
print '<script type="text/javascript" src="overlib.js"></script>'
print '</head><body>'
print '<div id="overDiv" style="position:absolute; visibility:hidden;'
print 'z-index:1000;"></div>'
print '<map name="m1">' . spool(mapFilename) . '</map>'
print '<img src="' . pngFilename . '" usemap="#m1" border="0">'
print '</body></html>'
```

The JavaScript library `overlib.js` [7] is used to display the tooltips and must be copied to the same directory as the file `tutorialhtmlpage3.mapyrus`.

Enter the following URL in a web browser to display the HTML page.

```
http://localhost:8410/tutorialhtmlpage3.mapyrus
```

Some knowledge of JavaScript is necessary to understand this example.

This method is also suitable for generating static HTML pages and imagemaps.

4.41 Setting Expiry Dates, Cookies and Redirections from Mapyrus HTTP Server

For images that change infrequently, setting an expiry date prevents a web browser from making the same request to the HTTP server again.

Change the `httpresponse` command in the example from section 4.37 to the following value to stop a web browser from repeating the same HTTP request within the next 60 seconds.

```
httpresponse "HTTP/1.0 200 OK
Content-Type: image/png
Expires: " . timestamp(60)
```

To track the number of times a web browser makes a request to the HTTP server, include a cookie in the response to an HTTP request. The web browser then returns this cookie in the HTTP header of the next request. The following `httpresponse` command demonstrates this.

```
let cookie = Mapyrus.http.header['Cookie']
let cookie = replace(cookie, "^count=", "")
httpresponse "HTTP/1.0 200 OK
Content-Type: image/png
Set-Cookie: count=" . (cookie + 1)
```

[7] Available from http://www.bosrup.com/web/overlib/

To redirect a web browser to a different location, return the following HTTP header using the `httpresponse` command.

```
httpresponse "HTTP/1.0 301 OK
Content-Type: image/png
Location: other-image.png"
```

4.42 Returning Additional Information From Mapyrus HTTP Server

Further ideas for building a complete HTML application are:

- For datasets covering a wide area generate two images for each HTML page. Create a detailed map display and an overview map display with a box showing the position of the detailed map in the dataset.
- Generate an image containing a scalebar and include this in the HTML page. See section 4.19 on page 66.
- Restrict datasets to a limited scale range by checking the internal variable `Mapyrus.worlds.scale` before displaying a dataset.
- Return a map image as an HTML imagemap. This enables the user to click in the map to re-center or zoom the display. When the user clicks in the map another HTTP request is generated. When Mapyrus receives an HTTP request from a mouse click in an image map the internal variables `Mapyrus.imagemap.x` and `Mapyrus.imagemap.y` are automatically set with the pixel position clicked in the image (with origin in top-left corner of the image). Use these variables to calculate the new area to display.

4.43 Running Mapyrus As A Java Servlet With Apache Tomcat

The file `mapyrus.war` provided with Mapyrus is a web application containing the Mapyrus software `mapyrus.jar` and web application configuration file `web.xml`.

To use Mapyrus with Apache Tomcat, the file `mapyrus.war` must be deployed to the Apache Tomcat server. This is most easily done using one of the following methods:

- Copying the file `mapyrus.war` to the `webapps` subdirectory of a Tomcat installation, then starting Tomcat.
- Using the Tomcat Manager web interface of a running Tomcat installation to add the file `mapyrus.war`.

After deploying `mapyrus.war`, HTTP requests are sent to Mapyrus using the URL

```
http://localhost:8080/mapyrus/
```

The servlet initialisation parameter `c1` in the web application configuration file `web.xml` defines the Mapyrus commands that are run when Mapyrus receives an HTTP request. Any parameters in URL are set as variables in Mapyrus before running the Mapyrus commands. The standard output of the Mapyrus commands is returned from the HTTP request.

To change the Mapyrus commands, use the following steps:

1. Unpack the file `web.xml` from `mapyrus.war` using the command

   ```
   jar xf mapyrus.war WEB-INF/web.xml
   ```

2. Edit file `WEB-INF/web.xml` with a text editor and change the Mapyrus commands.

3. Update the file `mapyrus.war` using command

   ```
   jar uf mapyrus.war WEB-INF/web.xml
   ```

4. Remove the old Mapyrus web application from Apache Tomcat, then add the new Mapyrus web application.

These steps are avoided if the Mapyrus commands in file `web.xml` are changed to a single line such as

```
include /path/myfile.mapyrus
```

Then the Mapyrus commands can be edited in a file outside of Apache Tomcat and tested immediately without updating `mapyrus.war` and without redeploying it.

If Java Topology Suite functions or a JDBC driver are used then the *Java Topology Suite* or JDBC JAR file must be made available to Tomcat by copying it to the Tomcat `common/lib` subdirectory before starting Tomcat.

To define more than one set of Mapyrus commands in file `web.xml`, define each set of commands in a separate `<init-param>` and give the parameter `ipn=`*`param-name`* in the URL to Mapyrus. For example, the following URL runs Mapyrus commands defined in servlet initialisation parameter `c2`.

```
http://localhost:8080/mapyrus/?ipn=c2&hello=world
```

4.44 Using Mapyrus In A Java Or Jython Application

Jython is a Java implementation of the Python programming language.

To use Mapyrus in a Jython application, include the file `mapyrus.jar` in the Java classpath and then create and use a Mapyrus object in Jython using the following code:

```
from java.lang import System
from java.io import IOException
from org.mapyrus import Mapyrus, MapyrusException

cmds1 = ["newpage 'png', 'j.png', 100, 100", "color 'yellow'"]
cmds2 = ["hexagon 50, 50, 40", "fill"]
```

```
m = Mapyrus()
try:
        m.interpret(cmds1, System.in, System.out)
        m.interpret(cmds2, System.in, System.out)
        m.close()
except MapyrusException, ex:
        print "Error:", ex
except IOException, ex:
        print "Error:", ex
```

To make Mapyrus draw into an image created in the application, add the following code before interpreting Mapyrus commands:

```
        b = BufferedImage(200, 200, BufferedImage.TYPE_4BYTE_ABGR)
        m.setPage(b, "")
```

The same constructors and method calls are used to embed Mapyrus in a Java application.

4.45 Calling Java Functions From Mapyrus

Java provides a large library of classes. Class methods that are declared `public` and `static` are available as functions in Mapyrus. The following examples demonstrate calling Java methods from Mapyrus.

```
print java.lang.Integer.toHexString(32767)
let b = java.lang.Integer.parseInt("101110", 2)
let dummy = java.lang.Thread.sleep(1000)

print com.vividsolutions.jts.io.WKTWriter.stringOfChar("x", 12)
```

4.46 Combining Mapyrus With Other Software

To combine Mapyrus with other software use the following techniques.

Pass parameters to Mapyrus as environment variables, or use the Java `-D` option.

To use Mapyrus in a pipeline of commands, give `-` as the output filename in a `newpage` command to write an image file to standard output.

Give `-` as the input filename in a `dataset` command to read standard input as a textfile.

Give Mapyrus commands with the `-e` option instead of reading commands from a file.

Use Mapyrus in combination with the freely available software Netpbm[8] and Ghostscript[9] to post-process image output and convert images and PostScript output to other formats.

The following three examples demonstrate these methods. The first example creates a PPM image file and then converts it to ASCII art.

[8] Available from http://netpbm.sourceforge.net
[9] Available from http://www.ghostscript.com

```
java -DDATADIR=$DATADIR -DNameidx=Q7B80L -classpath mapyrus.jar \
  org.mapyrus.Mapyrus mycommands.mapyrus | ppmtopgm | \
  pgmtopbm | pbmtoascii -2x4
```

The second example creates a PostScript file and then converts it to a fax format.

```
java -classpath mapyrus.jar org.mapyrus.Mapyrus \
  $HOME/common.mapyrus mycommands.mapyrus > myfile.ps
gs -sDEVICE=faxg3 -sOutputFile=myfile.fax \
  -TextAlphaBits=4 myfile.ps
```

The third example reads a file listing and displays it in an PNG image.

```
/bin/ls -l | java -classpath mapyrus.jar org.mapyrus.Mapyrus -e '
newpage "png", "-", 210, 120, "background=white"  # write to stdout
font "Courier", 4
let y = Mapyrus.page.height - 4
dataset "textfile", "-", ""  # read from stdin
while Mapyrus.fetch.more
do
  fetch
  clearpath
  move 1, y
  # Display files with suffix "txt" in red, others in black.
  color match($0, ".txt$") ? "red" : "black"
  label $0
  let y = y - 4
done' > ls.png
```

The fourth example creates a PNG image containing a banner, running in headless mode on a server with no graphics display available.

```
java -Djava.awt.headless=true -classpath mapyrus.jar org.mapyrus.Mapyrus -e '
newpage "png", "banner.png", 180, 30, "ttffiles=BEANTOWN.ttf"
font "BeanTown", 24
color "red"
move 5, 5
label "HERE IS MY BANNER"
'
```

4.47 Creating Animations Using Mapyrus

To create an animated GIF image, create each frame as a separate GIF image using Mapyrus and the Netpbm program `ppmtogif`. The following example demonstrates this, creating frames that slowly zoom in.

```
let size = 7000000, i = 0
let centerx = 1445863, centery = 6206249
repeat 60
do
```

```
  let f = "frame" . format("000", i) . ".gif"
  newpage "ppm", "| ppmtogif > " . f, 60, 60, "background=pastelblue"
  worlds centerx - size / 2, centery - size / 2, \
    centerx + size / 2, centery + size / 2

  dataset "shapefile", "coastline.shp", ""
  while Mapyrus.fetch.more
  do
    fetch
    clearpath
    addpath GEOMETRY
    color '#669900'
    fill
    color '#333333'
    linestyle 0.1
    stroke
  done

  # Reduce area shown in next frame of the animation.
  #
  let size = size * 0.9
  let i = i + 1
done
```

Then combine all frames using the freely available program Gifsicle[10]. The following command makes an animated GIF file from all frames.

```
gifsicle frame*.gif > anim.gif
```

4.48 Creating SVG Files With Event Handling

Scalable Vector Graphics files are made interactive by adding JavaScript functions and defining mouse events that call these functions.

Groups of shapes or layers are defined by adding extra XML `<g>` tags to an SVG file.

Using the following JavaScript contained in file `svg1.js`

```
<script type="text/ecmascript">
<![CDATA[
function highlight(evt, color)
{
alert("event at " + evt.clientX + ", " + evt.clientY);
evt.target.setAttribute("style", "fill:" + color);
}
]]></script>
```

the next example demonstrates creating an SVG file with mouse events that change the color of shapes when the mouse is moved over the shapes. XML `<g>` tags are also added to the SVG file using the `svgcode` command to identify different groups of shapes or layers.

[10] Available from http://www.lcdf.org/gifsicle

```
newpage "svg", "tutorialsvg1.svg", 30, 30, "scriptfile=svg1.js"

svgcode '<g id="layer1">'
color "indigo"
circle 20, 20, 10
fill "onmouseover='highlight(evt, \"yellow\")'"
svgcode '</g>'
clearpath

svgcode '<g id="layer2">'
box 0, 0, 10, 10
linestyle 2
color "red"

stroke "onmouseover='highlight(evt, \"green\")' onmouseout='highlight(evt, \"blue\")'"
svgcode '</g>'
```

4.49 Building Mapyrus From Source

To build Mapyrus from source code, the following development tools are required:

- CVS Version Control System.
- Ant build tool.
- Java 2 SDK 5, or higher.
- Java Topology Suite 1.8, or higher from `http://www.vividsolutions.com/JTS`.
- Oracle JDBC driver JAR file `ojdbc14.jar` and Oracle Spatial data types JAR file `sdoapi.jar` from an Oracle 10g database, or higher. This is only required for reading of Oracle Spatial column types from an Oracle database.
- Servlet API JAR file `servlet-api.jar` from Apache Tomcat 5 or later.
- LaTeX.

First, checkout the source code from the Mapyrus anonymous CVS server using the following command.

```
cvs -z3 -d :pserver:anonymous@mapyrus.cvs.sourceforge.net:/cvsroot/mapyrus co mapyrus
```

The build process build is defined in the file `build.xml` in the Mapyrus installation directory. To build the software, documentation and zip file for distribution, create a terminal window, change to the directory in which Mapyrus is installed and execute the following command.

```
ant -v
```

The build process also generates the PDF manual and manual examples in the `userdoc` subdirectory from source files using LaTeX.

All source code is in Java and contains javadoc style comments. To view and edit the source code, use a Java development environment such as Eclipse and import the source code into a new project.

4.50 Sample Shapes And Patterns

The file `symbols.mapyrus` included in the subdirectory `userdoc` contains many examples of shapes and patterns for lines and fills. A legend for all examples in this file is shown in Figure 4.50.

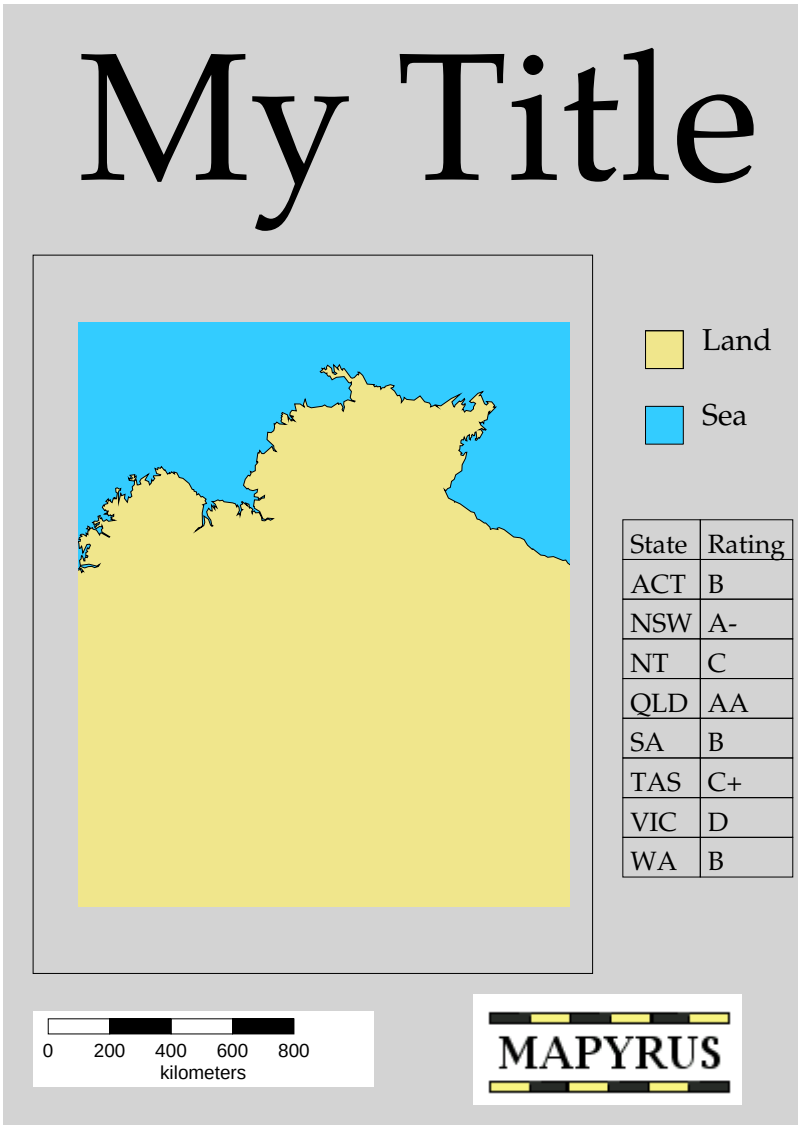

Figure 70: Page Layout

Figure 71: Sample Shapes And Patterns 1

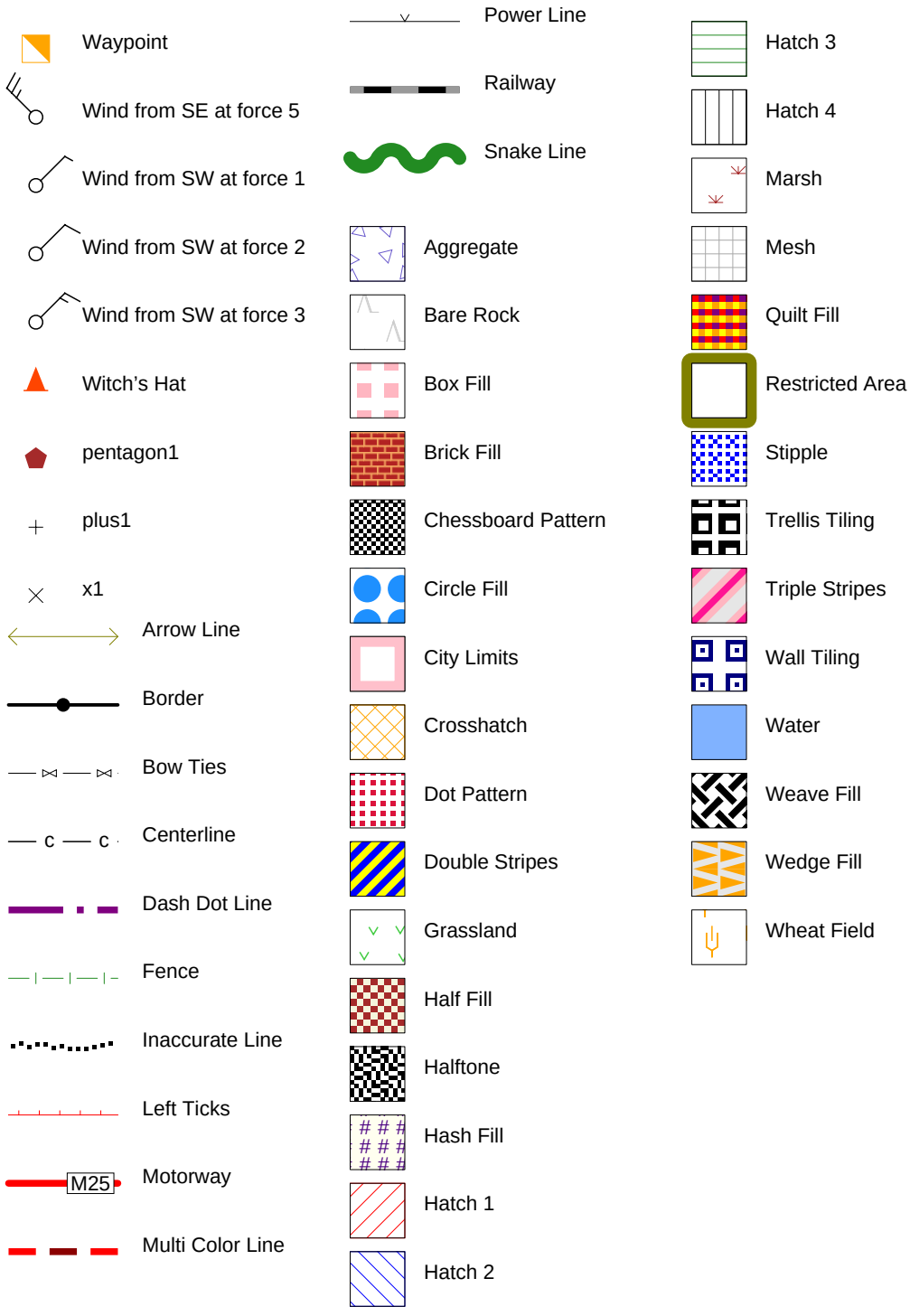

Figure 72: Sample Shapes And Patterns 2

5 Reference

5.1 Software Requirements

Mapyrus requires:

- Java 2 Runtime Environment, Standard Edition, (J2RE) 5 or higher, or Java 2 Software Developers Kit, Standard Edition (J2SDK) 5 or higher.
- The `$DISPLAY` environment variable set to an X-Windows display, if running on Linux or a UNIX operating system. If a real X-Windows display is not available, use the `-Djava.awt.headless=true` startup variable.
- The *Java Topology Suite* from `http://www.vividsolutions.com/JTS`, if geometric functions are required (see Table 3 on page 110).
- The OGR library from `http://openev.sourceforge.net` if reading of GIS data formats other than ESRI Shape format and relational database tables is required.

5.2 Usage

The Mapyrus software is contained in a single Java JAR file. Start Mapyrus in a Java interpreter with the following command.

```
java -classpath install-dir/mapyrus.jar org.mapyrus.Mapyrus filename
...
```

install-path is the directory in which Mapyrus is installed. *filename* is the name of a file or a URL for Mapyrus to read from. If *filename* is - then standard input is read. If several filenames and URLs are given then they are read in turn.

Environment variables and variables passed to Mapyrus using the Java `-D` option are available in the Mapyrus interpreter. The *Java Topology Suite* JAR file and other JAR files to be used in combination with Mapyrus are included in the `-classpath` option.

```
java -Dvariable=value ... \
  -classpath install-dir/mapyrus.jar:jts-dir/jts-1.8.jar:other-jarfile \
  org.mapyrus.Mapyrus filename
```

Mapyrus runs as an HTTP Server when started with the `-s` option.

```
java -classpath install-dir/mapyrus.jar:jarfile \
  org.mapyrus.Mapyrus -s port filename ...
```

Use the `-Xmx` Java option to make more memory available when running Mapyrus. To increase available memory to 256Mb, use the following command:

```
java -Xmx256m -classpath install-dir/mapyrus.jar org.mapyrus.Mapyrus filename ...
```

5.2.1 Startup Configuration

The variables available for configuring Mapyrus at startup are shown in Table 1.

Variable	Description
`Mapyrus.rgb.file=`*`filename`*	Defines an X Windows color names file containing additional color names for the `color` command. Default value is `/usr/lib/X11/rgb.txt`
`java.awt.headless=true`	Run in headless mode. Required when running on a server with no graphics display.
`jdbc.drivers=`*`class`*	Defines class containing JDBC 1.0 (or higher) driver to load at startup. A JDBC driver is required for connecting to a relational database and is provided as part of a relational database. See the Java JDBC DriverManager API documentation [11] for details. The JAR file containing the class must be included in the `-classpath` option when starting Mapyrus.

Table 1: Startup Variables

5.3 Language

Mapyrus interprets commands read from one or more plain text files. Each command begins on a separate line.

Any part of a line following a hash (`#`) character that is not part of a literal string is interpreted as a comment and is ignored. Leading and trailing spaces or tabs on a line are also ignored. A backslash (\) character at the end of a line is interpreted as a line continuation and the line and next line are joined into a single line.

A line beginning with the word `include`, followed by a filename or URL includes commands from another file.

Each command is passed zero or more arguments separated by commas. An argument is a number, a string literal in single quotes (´) or double quotes (¨), a variable name, an array, an array element or an expression.

The character sequence (***nnn**) in a string literal is interpreted as an octal character code (where ***nnn*** is one to three digits).

An expression contains arguments and operators and functions on those arguments, like in BASIC, C, or Python. Available operators are shown in Table 2. Pre-defined functions are shown in Table 3. Java methods are also available as functions by giving the class name and method name separated by a dot. Only Java methods declared as `public` and `static` are available as functions.

Function Name	Description
`abs(`*`n`*`)`	Returns the absolute value of *n*.

Table 3: Functions

[11] Available from `http://java.sun.com/j2se/1.5.0/docs/api/java/sql/DriverManager.html`

Function Name	Description
`axis(`*min*`, `*max*`, `*intervals*`)`	Generates a set of numbers that are suitable for an axis of a graph containing values in the range *min* to *max*. *intervals* sets the maximum number of values for the axis. An array is returned with each value for the axis.
`buffer(`*g*`, `*dist*`, `*cap*`)`	Returns a geometry containing a buffer calculated at a distance *dist* around the perimeter of geometry *g*. The value of *cap* defines the method of closing buffers at line endpoints, either `butt`, `round` or `square`. This function requires the *Java Topology Suite*.
`ceil(`*n*`)`	Returns the smallest integer value that is not less than *n*.
`contains(`*g1*`, `*x*`, `*y*`)` `contains(`*g1*`, `*g2*`)`	If point (*x*, *y*) or geometry *g2* is contained inside *g1* then 1 is returned. Otherwise 0 is returned. Geometries may be the same type or different types. This function requires the *Java Topology Suite*.
`convexhull(`*g*`)`	Returns a convex hull geometry that surrounds geometry *g*.
`cos(`*n*`)`	Returns the cosine of angle *n*, given in degrees.
`crosses(`*g1*`, `*g2*`)`	If geometry *g2* crosses *g1* then 1 is returned. Otherwise 0 is returned. Geometries must be of different types. To compare geometries of the same type, use `overlaps`. This function requires the *Java Topology Suite*.
`difference(`*g1*`, `*g2*`)`	Returns a geometry containing the difference between geometry *g1* and geometry *g2*. That is, parts of geometry *g1* that are not part of geometry *g2*. This function requires the *Java Topology Suite*.
`dir(`*p*`)`	Returns an array of all filenames matching the wildcard pattern *p* containing asterisk (`*`) characters.
`floor(`*n*`)`	Returns the largest integer value that is not larger than *n*.
`format(`*str*`, `*n*`)`	Returns the number *n* formatted using format string *str*. Format string is given using hash characters and zeroes for digits and an optional decimal point. For example, `00000` for a five digit number with leading zeroes, or `##.###` for a number rounded to three decimal places.

Table 3: Functions

Function Name	Description
`interpolate(`*str*`,` *n*`)`	Returns value calculated from n using linear interpolation. *str* contains list of numbers (given in increasing numeric order) and corresponding values: n_1 v_1 n_2 v_2 .... Result is found by finding range n_i to n_{i+1} containing `n` and using linear interpolation to calculate a value between v_i and v_{i+1}. Each value v_i is either a number or a named color.
`intersection(`*g1*`,` *g2*`)`	Returns a geometry containing the intersection of geometry *g1* and geometry *g2*. This function requires the *Java Topology Suite*.
`length(`*v*`)`	If v is an array, then the number of elements in the array is returned. Otherwise the string length of v is returned.
`log10(`*n*`)`	Returns the base 10 logarithm of n.
`lower(`*str*`)`	Returns *str* converted to lower case.
`lpad(`*str*`,` *len*`,` *pad*`)` `lpad(`*str*`,` *len*`)`	Returns string *str* left padded to length *len* using characters from string *pad*. Spaces are used for padding if *pad* is not given. String is truncated on the left if longer than length *len*.
`match(`*str*`,` *regex*`)`	Returns the index in the string *str*, where the regular expression *regex* is first matched. The index of the first character is 1. If the regular expression does not match *str*, then 0 is returned. The *Java API documentation* [12] describes the syntax of regular expressions.
`max(`*a*`,` *b*`)`	Returns the larger of values a and b.
`min(`*a*`,` *b*`)`	Returns the smaller of values a and b.
`overlaps(`*g1*`,` *g2*`)`	If geometry *g1* and geometry *g2* are the same type and overlap then 1 is returned. Otherwise 0 is returned. This function requires the *Java Topology Suite*.
`parsegeo(`*n*`)`	Parses string n containing a latitude or longitude position into a decimal value and returns it. Strings of many forms are accepted, including `42.196597N`, `42° 11´ 47.75¨`, `42d 11m 47.75s`, `N 42d 11m 47.75s` and `42deg 11min 47.75sec`.
`pow(`*a*`,` *b*`)`	Returns a to the power b.

Table 3: Functions

[12] Available from `http://java.sun.com/j2se/1.5.0/docs/api/java/util/regex/Pattern.html`

Function Name	Description
`protected(`*x1*`, `*y1*`, `*x2*`, `*y2*`)` `protected(`*g*`) protected()`	For points (*x1*, *y1*) and (*x2*, *y2*) defining any two opposite corners of a rectangle, returns 1 if any part of this rectangle has been protected using the `protect` command. For polygon geometry *g*, returns 1 if any part of the polygon has been protected. When no rectangle or geometry is given then 1 is returned if any part of the current path has been protected. Otherwise 0 is returned.
`random(`*n*`)`	Generates a random floating point number between 0 and *n*.
`readable(`*filename*`)`	Returns 1 if the file *filename* exists and is readable. Otherwise 0 is returned.
`replace(`*str*`, `*regex*`,` *replacement*`)`	Returns the string *str*, with all occurrences of the regular expression *regex* replaced by *replacement*.
`roman(`*n*`)`	Returns the number *n* converted to a Roman numerals string.
`round(`*n*`)`	Returns *n* rounded to nearest whole number.
`rpad(`*str*`, `*len*`, `*pad*`)` `rpad(`*str*`, `*len*`)`	Returns string *str* right padded to length *len* using characters from string *pad*. Spaces are used for padding if *pad* is not given. String is truncated if longer than length *len*.
`sin(`*n*`)`	Returns the sine of angle *n*, given in degrees.
`split(`*str*`, `*regex*`)` `split(`*str*`)`	Splits the string *str* into an array of strings, delimited by the regular expression *regex* or whitespace if no regular expression given. The array of split strings is returned, with the first string having array index 1, the second string having index 2, and so on.
`spool(`*filename*`)`	Returns string containing contents of text file *filename*. If *filename* has suffix `.gz` or `.zip` then it is automatically decompressed as it is read.
`sqrt(`*n*`)`	Returns square root of *n*.
`stringheight(`*str*`)`	Returns the height of the string *str* if it were displayed using the `label` command. For strings containing several lines, the total height of all lines is returned. The height is returned in world coordinate units if set with a `worlds` command, otherwise in page coordinates.

Table 3: Functions

Function Name	Description
`stringwidth(`*str*`)`	Returns the width of the string *str* if it were displayed using the `label` command. For strings containing several lines, the width of the longest line is returned. The width is returned in world coordinate units if set with a `worlds` command, otherwise in page coordinates.
`sum(`*a*`)`	Returns the sum of values in array *a*.
`substr(`*str*`, `*offset*`, `*n*`)` `substr(`*str*`, `*offset*`)`	Returns a substring of the string *str*, beginning at the character with index *offset* that is *n* characters long, or all characters from index *offset* if *n* is not given. The first character in *str* has an index of 1.
`tan(`*n*`)`	Returns the trigonometric tangent of angle *n*, given in degrees.
`tempname(`*suffix*`)`	Returns a unique temporary filename with given file suffix, for use when running as an HTTP server. Temporary files returned by this function are automatically deleted after 5 minutes.
`timestamp(`*n*`)`	Returns a time stamp containing the current GMT date and time plus *n* seconds, for use in setting expiry dates when running as an HTTP server.
`topage(`*x*`, `*y*`)` `topage(`*g*`)`	Transforms the point (*x*, *y*) or geometry *g* from the current world coordinate system to page coordinates.
`toworlds(`*x*`, `*y*`)` `toworlds(`*g*`)`	Transforms the point (*x*, *y*) or geometry *g* from page coordinates to the current world coordinate system. This is the inverse of the `topage` function.
`trim(`*str*`)`	Returns string *str* with whitespace trimmed from start and end.
`union(`*g1*`, `*g2*`)`	Returns a geometry containing the union of of geometry *g1* and geometry *g2*. This function requires the *Java Topology Suite*.
`upper(`*str*`)`	Returns *str* converted to upper case.

`wordwrap(`*str*`, `*width*`)`
`wordwrap(`*str*`, `*width*`,`
extras`)`
	Returns *str* broken into several lines for use in a `label` command. Each line will not be longer than *width* millimeters wide. If *extras* contains `hyphenation=`*str* then words containing the hyphenation string may also be split onto two lines at that point using a hyphen. If *extras* contains `adjustspacing=true` then additional spaces are added between words so that each line has the required width.

Table 3: Functions

Operator	Description
`(`, `)`	parentheses
`++`, `--`	increments or decrements variable
`*`, `/`, `%`, `x`	numeric multiplication, numeric division, modulo (works with non-integer values too), string repetition
`+`, `-`, `.`	numeric addition, numeric subtraction, string concatenation
`<=`, `<`, `==`, `!=`, `>`, `>=`, `lt`, `le`, `eq`, `ne`, `gt`, `ge`	numeric comparisons and string comparisons
`?:`	ternary conditional operator
`and`, `or`, `not`	Logical and, or, not

Table 2: Operators

An argument or expression is assigned to a named variable using the `let` command and an equals sign (`=`). A variable name begins with a letter or dollar sign (`$`) and contains only letters, numbers, dots (`.`), underbars (`_`) and colons (`:`). Variable names are case-sensitive. Variables and array elements that are accessed without having been defined have a value of zero, or an empty string.

Variables accepted as parameters to a function or procedure, or declared in the function or procedure with the local command are local to that function or procedure and not visible outside the function or procedure. All other variables are global.

Array elements are accessed by giving the index between square brackets (`[` and `]`). The index is any string or number value. Multi-dimension arrays are available by using indexes named *index1*`."`*c*`".`*index2*, where *c* is any character that never appears in an index.

The `if`, `repeat`, `while`, `for` and `function` flow control structures found in other languages are available:

```
if condition then
  then-commands ...
else
  else-commands ...
endif
```

Executes *then-commands* if *condition* evaluates to a non-zero value, otherwise *else-commands* are executed. The `else` part is optional. Compound tests are built using the `elif` keyword:

```
if condition then
  commands ...
elif condition then
  commands ...
endif
```

The `repeat` keyword defines a loop in which *commands* will be executed *count* times:

```
repeat count do
```

```
  commands ...
done
```

The `while` keyword defines a loop in which *commands* will be executed for as long as *condition* continues to evaluate to a non-zero value:

```
while condition do
  commands ...
done
```

The `for ... in` keywords define a loop in which each element of *array* is assigned to variable *var* and *commands* are executed. Elements in *array* are accessed in numerical index order if indexes are numeric, otherwise in alphabetical index order:

```
for var in array do
  commands ...
done
```

Functions are used to repeat commonly used calculations and to return a value:

```
function name [arg1, ...]
  commands ...
  return arg
end
```

Procedures group frequently used commands together, save graphics state when they begin and restore it when they end, isolating the calling procedure from any changes:

```
begin name [arg1, ...]
  commands ...
end
```

A procedure is defined to take a fixed number of arguments. All procedure names are global and following the same naming rules as variables. Procedure definitions within a procedure are not allowed. The `return` keyword returns a from a procedure to the calling procedure.

A procedure is called from any place where a command is accepted and the number of arguments passed must match the number in the procedure definition.

Before commands in the procedure are executed the graphics state is saved. The graphics state is restored when the procedure finishes. The graphics state contains:

1. The path defined with `move`, `draw`, `box`, `arc`, `bezier`, `circle`, `cylinder`, `ellipse`, `hexagon`, `pentagon`, `sinewave`, `spiral`, `star`, `triangle`, `wedge` and `addpath` commands.
2. The clip path defined with `clip` commands.
3. Current drawing settings set by `color`, `linestyle`, `font` and `justify` commands.

4. Transformations set by `rotate` and `scale` commands.

5. The output file set by a `setoutput` command.

Any new page created in a procedure with a `newpage` command is completed when the procedure finishes and output returns to the page being created before the procedure was called.

Any dataset being read is global and protected regions are global. They are not saved and not restored.

Any world coordinate system set with the `worlds` command is cleared before the commands in the procedure are executed. This enables the calling procedure to work in a world coordinate system and the called procedure to draw at these world coordinate positions using measurements in millimeters.

If the current path contains only `move` points and no straight line or arc segments when a procedure is called then the procedure is called repeatedly, one time for each move point, with the origin (0, 0) translated to the `move` point each time and the path reset to empty. Therefore, coordinates in the called procedure are relative to the move point. This enables drawing commands in the called procedure to be given in millimeters, relative to each `move` point.

5.4 Internal Variables

All environment variables and Java standard system properties (`os.arch`, `user.dir`, etc.) are defined as variables in Mapyrus.

Mapyrus maintains the internal variables shown in Table 4.

Variable Name	Description
`Mapyrus.dataset.fieldnames`	An array containing the names of fields being read from the current dataset. The first fieldname has array index 1.
`Mapyrus.dataset.projection`	A description of the projection (coordinate system) in which coordinates of the current dataset are stored. Projection descriptions are not standardised between dataset formats. Different dataset formats will return different descriptions for the same projection.
`Mapyrus.dataset.min.x`, `Mapyrus.dataset.min.y`, `Mapyrus.dataset.max.x`, `Mapyrus.dataset.max.y`, `Mapyrus.dataset.center.x`, `Mapyrus.dataset.center.y`	The bounding rectangle of all data in the current dataset.
`Mapyrus.fetch.count`	The number of records already fetched from the current dataset.
`Mapyrus.fetch.more`	Flag value set to 1 if another record is available for `fetch` command, or 0 if no more records available.
`Mapyrus.filename`	The name of the file or URL being interpreted.

Table 4: Internal Variables

Variable Name	Description
`Mapyrus.freeMemory`	The amount of free memory that Java has available, in bytes.
`Mapyrus.http.header`	An array containing header information passed in the HTTP request when running as an HTTP server. Useful values are `Mapyrus.http.header['Referer']` giving the name of the referring HTML page, `Mapyrus.http.header['Cookie']` giving the contents of a cookie set by a previous HTTP request and `Mapyrus.http.header['User-Agent']` giving the name of the web browser making the HTTP request.
`Mapyrus.imagemap.x`, `Mapyrus.imagemap.y`	The pixel position of the point clicked in an HTML imagemap and passed to Mapyrus, for use when running as an HTTP server. Both values are set to -1 if no imagemap point passed in current URL.
`Mapyrus.key.count`	The number of legend entries defined with `key` commands that have not yet been displayed with a `legend` command.
`Mapyrus.key.next`	The name of the of the next procedure to be displayed by the `legend` command.
`Mapyrus.page.format`, `Mapyrus.page.height`, `Mapyrus.page.width`, `Mapyrus.page.resolution.dpi`, `Mapyrus.page.resolution.mm`	The file format, page height, page width and resolution that were passed to the `newpage` command. File format is in lowercase. Height and width are in millimeters. Resolution is available as either a dots-per-inch value, or a distance in millimeters between dots.
`Mapyrus.path`	The current path as an OGC WKT geometry with coordinates measured in millimetres.
`Mapyrus.path.length`, `Mapyrus.path.area`, `Mapyrus.path.start.angle`, `Mapyrus.path.start.x`, `Mapyrus.path.start.y`, `Mapyrus.path.end.angle`, `Mapyrus.path.end.x`, `Mapyrus.path.end.y`, `Mapyrus.path.min.x`, `Mapyrus.path.min.y`, `Mapyrus.path.max.x`, `Mapyrus.path.max.y`, `Mapyrus.path.center.x`, `Mapyrus.path.center.y`, `Mapyrus.path.width`, `Mapyrus.path.height`	The length of the current path on the page measured in millimeters, the area of the current path measured in square millimeters, the coordinates and angles at the start and end of the path in degrees measured counterclockwise, and the bounding rectangle of the current path.

Table 4: Internal Variables

Variable Name	Description
`Mapyrus.rotation`	The current rotation angle in degrees set by `rotate` command. Returned value is normalised to fall in the range -180 to +180 degrees.
`Mapyrus.scale`	The current scale factor set by `scale` command.
`Mapyrus.screen.height`, `Mapyrus.screen.width`, `Mapyrus.screen.resolution.dpi`, `Mapyrus.screen.resolution.mm`	The height, width and resolution of the screen in which Mapyrus is running. Height and width are in millimeters. Resolution is available as either a dots-per-inch value, or a distance in millimeters between dots.
`Mapyrus.time.day`, `Mapyrus.time.month`, `Mapyrus.time.year`, `Mapyrus.time.hour`, `Mapyrus.time.minute`, `Mapyrus.time.second`, `Mapyrus.time.day.of.week`, `Mapyrus.time.day.name`, `Mapyrus.time.month.name`, `Mapyrus.time.week.of.year`, `Mapyrus.time.stamp`	Components of the current date and time. Day of week has value 1 for Monday through to 7 for Sunday.
`Mapyrus.timer`	The elapsed processing time, measured in seconds.
`Mapyrus.totalMemory`	The total amount of memory available to Java, in bytes.
`Mapyrus.version`	The version of the software.
`Mapyrus.worlds.min.x`, `Mapyrus.worlds.min.y`, `Mapyrus.worlds.max.x`, `Mapyrus.worlds.max.y`, `Mapyrus.worlds.center.x`, `Mapyrus.worlds.center.y`, `Mapyrus.worlds.width`, `Mapyrus.worlds.height`	The bounding rectangle of world coordinates set with the `worlds` command.
`Mapyrus.worlds.scale`	The real-world scale factor, determined by dividing of the X axis world coordinate range by the page width.

Table 4: Internal Variables

5.5 Commands

Commands are listed alphabetically. The arguments required for each command are given. Some commands accept arguments in several ways. For these commands, each combination of arguments is given.

5.5.1 addpath

`addpath` *`geometry-field`* `[,` *`geometry-field`* `...]`

Adds geometry in each *geometry-field* to current path. A *geometry-field* is geometry fetched from a dataset with a `fetch` command or a string containing an OGC WKT geometry.

Coordinates are transformed through any transformation set with a `worlds` command, then scaled and rotated by `scale` and `rotate` values.

5.5.2 arc

`arc` *`direction`*`,` *`xCenter`*`,` *`yCenter`*`,` *`xEnd`*`,` *`yEnd`*

Adds a circular arc to the current path. The arc begins at the last point added to the path and ends at (*xEnd*, *yEnd*) with center at (*xCenter*, *yCenter*). If *direction* is a positive number, the arc travels clockwise, otherwise the arc travels in an anti-clockwise direction. If the begin and end points are the same then the arc is a complete circle. A straight line segment is first added to the path if the distance from the beginning point to the center is different to the distance from the center to the end point.

Points are transformed through any transformation set with a *worlds* command, then scaled and rotated by *scale* and *rotate* values.

5.5.3 bezier

`bezier` *`xControl1`*`,` *`yControl1`*`,` *`xControl2`*`,` *`yControl2`*`,` *`xEnd`*`,` *`yEnd`*

Adds a Bezier curve (a spline curve) to the current path. The curve begins at the last point added to the path and ends at (*xEnd*, *yEnd*) with control points (*xControl1*, *yControl1*) and (*xControl2*, *yControl2*).

The control points define the direction of the line at the start and end points of the Bezier curve. At the start of the Bezier curve, the direction of the curve is towards the first control point. At the end of the Bezier curve, the direction of the curve is from the second control point.

Points are transformed through any transformation set with a *worlds* command, then scaled and rotated by *scale* and *rotate* values.

5.5.4 blend

`blend` *`mode`*

Sets the blend mode for transparent colors. Transparent colors are mixed differently with background colors depending on the blend mode.

Blend mode is one of `Normal`, `Multiply`, `Screen`, `Overlay`, `Darken`, `Lighten`, `ColorDodge`, `ColorBurn`, `HardLight`, `SoftLight`, `Difference` or `Exclusion`.

The effect of each blend mode is described in the PDF Reference Manual, available from `http://www.adobe.com`. Only the first six blend modes are available for SVG format output.

5.5.5 box

`box` *x1* `,` *y1* `,` *x2* `,` *y2*

Adds a rectangle to the current path. The points (*x1*, *y1*) and (*x2*, *y2*) define any two opposite corners of the rectangle.

The two corner points of the box are first transformed through any world coordinate transformation set with a `worlds` command, then scaled and rotated by `scale` and `rotate` values.

5.5.6 box3d

`box3d` *x1* `,` *y1* `,` *x2* `,` *y2* `[,` *depth*`]`

Adds a rectangle to the current path in the same way as the `box` command. The right side and top sides of the box are also added to the current path to give a 3 dimensional effect.

The depth of the right and top sides is optional. If not given then the the smaller of box height and box width is used.

The two corner points and depth of the box are first transformed through any world coordinate transformation set with a `worlds` command, then scaled and rotated by `scale` and `rotate` values.

5.5.7 chessboard

`chessboard` *x1* `,` *y1* `,` *x2* `,` *y2* `,` *size*

Adds squares in a chessboard pattern to the current path. The points (*x1*, *y1*) and (*x2*, *y2*) define any two opposite corners of a rectangular area for the pattern, with *size* defining the size of each square.

The two corner points and size of squares are first transformed through any world coordinate transformation set with a `worlds` command, then scaled and rotated by `scale` and `rotate` values.

5.5.8 circle

`circle` *xCenter* `,` *yCenter* `,` *radius*

Adds a circle to the current path, with center point (*xCenter*, *yCenter*) and radius *radius*.

The center point and radius are transformed through any transformation set with a *worlds* command, then scaled and rotated by *scale* and *rotate* values.

5.5.9 clearpath

`clearpath`

Removes all points from the current path.

5.5.10 clip

`clip` *side*

Sets a clip path to the area covered by the current path, or excluding the area covered by the current path, depending on the value *side*.

If *side* has value `inside` then later drawing commands are limited to draw only inside the area covered by current path. If *side* has value `outside` then later drawing commands are limited to draw only outside the area covered by current path.

If the path is clipped in a procedure, then the area remains clipped until the procedure is complete. Otherwise, the area remains permanently clipped for the page. When more than one path is clipped, drawing is limited to areas that satisfy all clip paths.

The current path is not modified by this command.

5.5.11 closepath

`closepath`

Closes the current path by adding a straight line segment back to the last point added with a `move` command.

5.5.12 color

```
color name [, alpha]
color "contrast" [, alpha]
color "brighter" [, alpha]
color "darker" [, alpha]
color "softer" [, alpha]
color "current" [, alpha]
color "#hexdigits" [, alpha]
color "0xhexdigits" [, alpha]
color "rgb", red, green, blue [, alpha]
color "hsb", hue, saturation, brightness [, alpha]
```

Sets color for drawing. Around 500 commonly used color names are defined, additional color names are defined in a file given as a startup variable (see Table 1 on page 110). Color names are case-insensitive.

If color name is `contrast` then color is set to either black or white, whichever constrasts more with the current color.

If color name is `brighter`, `darker` or `softer` then color is set to a brighter, darker or softer version of the current color.

If color name is `current` then the current color is set again.

hexdigits is a 6 digit hexadecimal value defining RGB values, as used in HTML pages.

red, *green* and *blue* values for RGB colors and *hue*, *saturation* and *brightness* values for Hue-saturation-brightness (HSB) colors are given as intensities in the range 0-1.

The alpha value is optional and defines transparency as a value in the range 0-1. An alpha value of 1 is completely opaque and the color overwrites underlying colors. An alpha value of 0 is completely transparent and the color is not

visible. Intermediate values are partially transparent and the color is blended with colors of underlying shapes on the page. The `blend` command controls how transparent colors are mixed with background colors.

Colors are opaque if an alpha value is not given.

Transparent colors are only available for BMP, JPEG, PNG, PPM, SVG, PDF and Encapsulated PostScript format image (`epsimage`) output.

The PostScript language does not contain any functions for setting transparency. All colors in PostScript and Encapsulated PostScript files will be opaque.

The spelling `colour` is also accepted for this command.

5.5.13 cylinder

`cylinder` *xCenter*`,` *yCenter*`,` *radius*`,` *height*

Adds a cylindrical shape to the current path, with center point (*xCenter*, *yCenter*) and given radius and height.

The center point, radius and height are transformed through any transformation set with a *worlds* command, then scaled and rotated by *scale* and *rotate* values.

5.5.14 dataset

`dataset` *format*`,` *name*`,` *extras*

Defines a dataset to read from. A dataset contains geographic data, geometry, attributes, a lookup table, data to write to standard output, or a combination of these.

dataset is the filename of the dataset to read. *format* is the format of the dataset and *extras* defines further options for accessing the dataset, given as *variable=value* values, separated by whitespace. Data formats and options are shown in Table 5.

Dataset Format	Description and Extras
`grass`	Reads from GRASS site list file with filename *name*. A GRASS site list is stored in a `site_lists` directory in a GRASS mapset. The geometry for each site point is assigned to a variable named `GEOMETRY`, attribute field values are assigned to variables `$1`, `$2`, `$3`, ... If the site file contains three dimensional data then the third dimension is assigned to a variable named `Z`.

Table 5: Dataset Formats

Dataset Format	Description and Extras
`jdbc`	Accesses data held in a relational database with an SQL `select` statement via JDBC. *name* contains the SQL query to execute. For each fetched record, field values are assigned to variables with the name of the fields. Field values that are NULL are converted to either an empty string, or numeric zero, depending on their type. Binary and blob fields are interpreted as OGC WKB geometry values. Some databases convert all field names to upper case, or to lowercase. Use a field name alias for fields that are the result of an expression. Extras: `driver=`*`string`* The name of the Java class containing a JDBC 1.0 (or higher) driver for connecting to the database. This class name is required if not given in the startup variable `jdbc.drivers` (see Table 1 on page 110). The JAR file containing the class must be included in the `-classpath` option when starting Mapyrus. `url=`*`string`* URL containing the database name, host and other information for identifying the database to connect to. The format of this string is database dependent. The database remains connected after use and the connection is reused in later `dataset` commands with the same `url` value. When Mapyrus is run using the HTTP server option, a pool of database connections are used for each `url` value to avoid continually reconnecting to the database. Mapyrus automatically closes bad and idle connections and Mapyrus will reconnect if the database is restarted. `user=`*`string`* Username for connecting to the database. `password=`*`string`* Password for connecting to the database. Other values are set as properties for the JDBC driver.

Table 5: Dataset Formats

Dataset Format	Description and Extras
`ogrinfo`	Reads the output of the `ogrinfo` program, part of the freely-available OGR library. The filename *name* contains either, 1. An `ogrinfo` program command followed by a pipe (`|`) character. Mapyrus runs this command and reads the output. Use `ogrinfo` command options to limit the data to an area of interest, or to data matching an SQL-like where clause. 2. (`-`) to read standard input to Mapyrus. 3. The name of a file containing `ogrinfo` output. The geometry for each fetched record is assigned to a variable named `GEOMETRY`, attribute field values are assigned to variables with attribute field names.
`osm`	Reads from OpenStreetMap URL or file *name*. Each node or way is fetched as a separate record. For each node or way, the variable `TYPE` is set to either `node` or `way` to indicate the type of data, `ID` is set to the ID of the node or way, `GEOMETRY` is set to the geometry of the node or way, and `TAGS` is created as an array containing the tag information for the node or way.
`shapefile`	Reads from ESRI Shape format file with filename *name*. The geometry for each fetched record is assigned to a variable named `GEOMETRY`, attribute field values are assigned to variables with attribute field names. Extras: `dbffields=`*`field1,field2,...`* Comma-separated list of attribute fields to read from the DBF database file accompanying the Shape file. By default, all fields are read. Reading fewer attribute fields improves performance. `xmin=`*`x1`*, `ymin=`*`y1`*, `xmax=`*`x2`*, `ymax=`*`y2`* Bounding rectangle of data to fetch. Data outside this rectangle is not fetched. Setting bounding rectangle to same values as world coordinate values in `worlds` command improves performance.
`textfile`	Reads from delimited text file *name*, with one record per line. Fields in fetched record are assigned to variables `$1`, `$2`, `$3`, ... and the whole record is assigned to variable `$0`. If *name* is `-` then standard input is read. If *name* has suffix `.gz` or `.zip` then it is automatically decompressed as it is read. Extras: `comment=`*`string`* Character string at start of a line marking a comment line that is to be ignored. Default value is a hash character (`#`). `delimiter=`*`character`* Character separating fields in the text file. Default value is all whitespace characters.

Table 5: Dataset Formats

5.5.15 draw

`draw` *x*`,` *y*`, ...`

Adds one or more straight line segments to the current path. A straight line segment is added from the previously defined point to (x, y) and then to each further point given. Points are first transformed through any world coordinate transformation set with a `worlds` command then scaled and rotated by `scale` and `rotate` values.

5.5.16 ellipse

`ellipse` *xCenter*`,` *yCenter*`,` *xRadius*`,` *yRadius*

Adds an ellipse to the current path, with center point (*xCenter*, *yCenter*). The radius of the ellipse in the horizontal direction is *xRadius* and in the vertical direction *yRadius*.

The center point and radius values are transformed through any transformation set with a *worlds* command, then scaled and rotated by *scale* and *rotate* values.

5.5.17 endpage

`endpage`

Closes output file created with `newpage` command.

5.5.18 eps

`eps` *filename*`, [,` *size*`]`

Displays an Encapsulated PostScript file at each `move` point in the current path. The file is centered at each point.

Encapsulated PostScript files can only be displayed when creating PostScript or Encapsulated PostScript output. For other formats, a grey box is drawn showing where the Encapsulated PostScript file would be drawn.

filename is the name of an Encapsulated PostScript file.

size is the optional size for the Encapsulated PostScript file in millimeters. If no size is given or size is zero then the file is displayed at its natural size, as defined in the Encapsulated PostScript file. The file is scaled and rotated according to the current `scale` and `rotate` settings.

5.5.19 eval

`eval` *command*

Evaluates any variables in *command* and then runs the result as a new command. This command is identical to the `eval` command found in UNIX scripting and Perl and enables commands to be built and executed while Mapyrus runs.

5.5.20 eventscript

`eventscript` *`tags ...`*

This command is used in combination with the `imagemap` option of the `newpage` command to create an HTML imagemap.

This command creates an entry in the imagemap with the given HTML tags for the area covered by the current path. Useful HTML tags include hyperlinks and callbacks for mouse events. Example HTML tags are:

```
href="australia.html"
```

```
onMouseClicked="return alert('Message!');"
```

To create an imagemap entry for a single point, first use the `box` command to define a box a few pixels in size around the point.

To create an imagemap entry for a line, first create a polygon around the line using the `buffer` function.

See section 4.40 for an example displaying tooltips as the mouse is moved over an image.

5.5.21 fetch

`fetch`

Fetches next record from current dataset. For each field in the record, a variable is defined with the name of the field and the value of the field for the next record. Before fetching a record, check the variable `Mapyrus.fetch.more` to ensure that another record is available from the dataset.

5.5.22 fill

`fill [`*`xml-attributes`*`]`

Flood fills the current path with the current color. The winding rule is used for determining the inside and outside regions of polygons containing islands. The current path is not modified by this command.

For SVG output, any XML attributes given in *xml-attributes* are included in the `<path>` XML element for the filled path.

5.5.23 flowlabel

`flowlabel` *`spacing`*`,` *`offset`*`,` *`string`* `[,` *`string ...`*`]`

Draws a label following the current path, using the font set with the `font` command. *string* values are separated by spaces. *offset* is the distance along the path at which to begin the label, given in millimeters. *spacing* is the spacing distance between each letter, given in millimeters.

If the label would appear upside down on the page, then it is rotated 180 degrees so it appears the right way up and is readable.

5.5.24 font

`font` *name* `,` *size* `[,` *extras* `...]`

Sets font for labelling with the `label` command. Font *name* and *size* are the name and size in millimeters of the font to use.

If a scale factor was set with the `scale` command then font size is scaled by this factor.

If a rotation was set with the `rotate` command then labels follow current rotation angle. If no rotation is set then labels are displayed horizontally.

Font *name* depends on the output format set with the `newpage` command. For PostScript output, *name* is the name of a PostScript Type 1 font. For output to an image format, *name* is one of the Java Logical font names (`Serif`, `SansSerif`, `Monospaced`, `Dialog`, or `DialogInput`) or a TrueType font name.

Tutorial Sections 4.33, 4.35 and 4.36 describe different font formats.

extras defines further options for the font, given as *variable=value* values, separated by whitespace. See Table 6 for available options.

Extra	Description
`outlinewidth=`*width*	Sets line width to use for drawing outline of each letter in label. Only the outline of each letter is drawn, no part of the letter is filled.
`linespacing=`*spacing*	Sets spacing between lines for labels with multiple lines. Line spacing is given as a multiple of the font size. The default line spacing is 1.

Table 6: Font Extras

5.5.25 geoimage

`geoimage` *filename* `[,` *extras* `]`
`geoimage` *url* `[,` *extras* `]`
`geoimage` *WebMapServiceUrl* `[,` *extras* `]`

Displays a geo-referenced image.

filename or *url* is the name of a BMP, GIF, JPEG, PNG, PPM or XBM format image file. An associated "worlds" file with suffix `.tfw` must exist, defining the world coordinate range covered by the image. The `extras` option `readerclass` enables additional image formats to be read using external Java classes.

webMapServiceUrl is a URL request to an OGC Web Mapping Service (WMS) for an image. The request type must be `GetMap`. The world coordinate range for the image is parsed from the `BBOX` parameter in the URL. See the Web Map Service Implementation specification at `http://www.opengis.org` for details of all parameters that must be included in the URL.

extras defines further options for the image, given as *variable=value* values, separated by whitespace. See Table 7 for available options.

Images cannot be displayed when creating SVG format output.

Extra	Description
`clipfile=`*`filename`*	Gives name of a text file containing a clip polygon for the image. Each line of the text file defines one (X, Y) world coordinate of the clip polygon. Only image data inside the clip polygon is displayed. Using a clip polygon prevents display of a non-rectangular image from overwriting a neighbouring image.
`hue=`*`factor`* `saturation=`*`factor`* `brightness=`*`factor`*	Defines a hue, saturation or brightness multiplication factor for the image.
`readerclass=`*`classname`*	Gives the name of a Java class to read the image and the world coordinate range covered by the image. The Java class must be included in the Java classpath and must contain the following methods: • `constructor(String filename, String extras)` • `java.awt.image.BufferedImage read()` • `java.awt.geom.Rectangle2D getBounds()` The methods may throw any type of Java exception on error. This option enables Mapyrus to be extended to read additional image formats.

Table 7: Geo-referenced Image Extras

5.5.26 gradientfill

`gradientfill` *`color1`*`,` *`color2`*`,` *`color3`*`,` *`color4`* `[,` *`color5`* `...]`

Fills the current path with a gradient fill pattern. Color names *color1*, *color2*, *color3* and *color4* define the color for the lower-left corner, lower-right corner, upper-left corner and upper-right corner of the polygon.

If *color5* is given then it defines an additional color at the the center of the polygon.

Colors in the interior of the polygon fade from the color defined in each corner to the colors in the other corners.

The winding rule is used for determining the inside and outside regions of polygons containing islands. The current path and current color are not modified by this command.

For SVG output only gradient fill patterns that fade in one dimension (vertically or horizontally) are possible. This is a limitation of the SVG format.

5.5.27 guillotine

`guillotine` *`x1`*`,` *`y1`*`,` *`x2`*`,` *`y2`*

Cuts path against a rectangle. Any part of the path inside or on the boundary of the rectangle remains. Any part of the path outside the rectangle is removed. The points (*x1*, *y1*) and (*x2*, *y2*) define any two opposite corners of the rectangle to cut against.

The four corner points of the rectangle are first transformed through any world coordinate transformation set with a `worlds` command, then scaled and rotated by `scale` and `rotate` values.

The path is always cut against a rectangle aligned with the X and Y axes of the page, regardless of any rotation angle.

5.5.28 hexagon

`hexagon` *`xCenter, yCenter, radius`*

Adds a hexagon shape to the current path, with center point (*xCenter*, *yCenter*) and distance *radius* from the center point to each vertex.

The center point and radius are transformed through any transformation set with a *worlds* command, then scaled and rotated by *scale* and *rotate* values.

5.5.29 httpresponse

`httpresponse` *`header`*

Sets the complete header to return from an HTTP request when Mapyrus is running as an HTTP server. The header must include at least two lines containing an HTTP server response code and a MIME content type. See section 4.41 for example HTTP headers.

5.5.30 icon

`icon` *`filename`*`, [, `*`size`*`]`
`icon "`*`binarydigits`*`" [, `*`size`*`]`
`icon "0x`*`hexdigits`*`" [, `*`size`*`]`
`icon "resource:`*`resourcename`*`" [, `*`size`*`]`

Displays an image icon at each `move` point in the current path. The icon is centered at each point.

filename is a the name of a file, URL or Java resource containing the icon. The icon must be either `BMP`, `GIF`, `JPEG`, `PAT`[13], `PNG`, `PPM` or `XBM` image format.

binarydigits are 16, 64 or 256 binary digits (all 0's and 1's) defining a square single color bitmap of size 4x4, 8x8 or 16x16 pixels. Any other characters in the string are ignored.

hexdigits are 4, 16 or 64 hexadecimal digits defining a square single color bitmap image of size 4x4, 8x8 or 16x16 pixels. Any non-hexadecimal characters in the string are ignored.

resourcename is the name of the Java resource containing the image, in the form `au/com/company/filename.png`. This option enables images from a Java JAR file included in the in the `-classpath` startup option to be displayed.

size is the optional size for the icon in millimeters. If no size is given or size is zero then the icon is displayed at its natural size, as it would appear in an

[13]The Gimp pattern file format

image viewer with one image pixel per display pixel. The image is scaled and rotated according to the current `scale` and `rotate` settings.

In PostScript and PDF files, icons with more than one color are displayed with an opaque white background. This is a limitation of PostScript and PDF output.

Icon images are loaded into memory. Loading very large images will use a large amount of memory. Page 109 describes how to make more memory available for Mapyrus.

Icons cannot be displayed when creating SVG format output.

5.5.31 justify

`justify` *`justification`*

Sets justification for labelling with the `label` command. *justification* is a string containing either `left`, `right`, `center` for justifying labels horizontally and/or `top`, `middle`, `bottom` for justifying labels vertically.

5.5.32 key

`key` *`type`*`,` *`description`*`, [`*`arg1`*`,` *`arg2`* `...]`

Defines an entry for a legend. The procedure containing this command will be called with arguments *arg1*, *arg2* ... to display a sample of the symbol when a legend is generated with a `legend` command. This command is ignored if used outside of a procedure.

If *description* contains the string `(#)` then it will be replaced in a legend by the number of times that the legend entry is defined.

type is either `point` to display the legend entry as a single point, `line` to display the legend entry as a horizontal line, `zigzag` to display the legend entry as a zig-zag line, or `box` to display the legend as a box. *description* is the label for the legend entry.

If a procedure displays more than one type of type of symbol depending on the arguments passed to it then use a separate `key` command for each, with different descriptions and different arguments.

5.5.33 label

`label` *`string`* `[,` *`string`* `...]`

Draws a label at each point in the path set with the `move` command, using the font, justification and rotation set with the `font`, `justify` and `rotate` commands. *string* values are separated by spaces. If *string* contains newline characters (`\n`) then labels are displayed as multiple lines, one below the other.

5.5.34 legend

`legend` *`size`*

Displays legend entries defined with `key` commands at points defined with `move` commands.

Each legend entry corresponds to a procedure and a set of arguments. The first legend entry is displayed at the first `move` point by calling the procedure in which the first legend entry was defined. Then the second legend entry is displayed at the second `move` point. This continues until either all legend entries are displayed or all move points are used.

The variable `Mapyrus.key.count` contains the number of legend entries that remain to be displayed.

If there are more legend entries than `move` points then some legend entries remain undisplayed and will be displayed in the next legend.

Legend entries are displayed in the order in which they are encountered in called procedures.

Legend entries in procedures that were never called are not included in the legend. Therefore, the legend only shows entries that were actually displayed.

size defines the size of each legend entry, in millimeters.

The description label is displayed to the right of each legend entry, using the current `color`, `font` and `justify` settings.

5.5.35 let

`let` *`var`* `=` *`expression`*`, ...`

Assigns result of evaluating *expression* to a variable with name *var*. The variable is globally accessible unless defined as local to the current procedure with a `local` command.

Variable *var* is either a simple variable name, an array, or an array element of the form *var*[*index*].

Several variables are assigned by separating each *var* and *expression* pair by a comma.

5.5.36 linestyle

`linestyle` *`width`*
`linestyle` *`width`*`,` *`cap`*`,` *`join`*
`linestyle` *`width`*`,` *`cap`*`,` *`join`*`,` *`phase`*`,` *`dash length`*`, ...`

Sets style line drawing by the `stroke` command. Line *width* given in millimeters. *cap* is the style to use at the ends of lines, either `butt`, `round` or `square`. *join* is the style to use where lines join, either `bevel`, `miter` or `round`. One or more *dash length* values are given, alternating between the length of one dash and the length of space between dashes in a dash pattern. Each *dash length* is given in millimeters. *phase* is the offset in millimeters into the dash pattern at which to begin.

5.5.37 local

`local` *`name`*`, [`*`name`* `...]`

Declares the listed variable names as local to a procedure. The variables are not visible outside the enclosing procedure and their values are lost when the procedure ends.

5.5.38 mimetype

`mimetype` *`type`*

Sets MIME type for content being returned from HTTP request when Mapyrus is running as an HTTP server. A more general solution is to use the `httpresponse` command to set the complete header returned from the HTTP request.

5.5.39 move

`move` *`x`*`,` *`y`*

Adds the point (x, y) to the current path. The point is first transformed through any world coordinate transformation set with a `worlds` command, then scaled and rotated by `scale` and `rotate` values.

5.5.40 newpage

`newpage` *`format`*`,` *`filename`*`,` *`width`*`,` *`height`*`,` *`extras`*
`newpage` *`format`*`,` *`filename`*`,` *`paper`*`,` *`extras`*

Begins output of a new page to a file. Any previous output is closed. The path, clipping path and world coordinates are cleared. The origin of the new page is in the lower-left corner. *format* is the file format to use for output, one of:

- `eps` for Encapsulated PostScript output, with shapes and labels defined geometrically.
- `ps`, `postscript` or `application/postscript` for PostScript output, with shapes and labels defined geometrically.
- `pdf` or `application/pdf` for Portable Document Format output.
- `screen` to display output in a window on the screen.
- `bmp` or `image/bmp` for BMP image output.
- `jpeg` or `image/jpeg` for JPEG image output.
- `png` or `image/png` for PNG image output.
- `ppm` or `image/x-portable-pixmap` for Netpbm PPM image output.
- `epsimage` for Encapsulated PostScript image output. Output contains a single image in which all shapes and labels have been drawn.
- `svg` or `image/svg+xml` for Scalable Vector Graphics (SVG) output.

paper is a paper size name for the page. Alternatively, *width* and *height* are the dimensions of the page in millimeters. To set dimensions in pixels for an image file, use the calculation *`pixelSize`* `* Mapyrus.screen.dpi.mm` for width and height.

filename is the name of the file to write the page to. If *filename* is a dash (`-`) then the page is written to standard output. If *filename* begins with a pipe

(|) then the rest of *filename* is interpreted as an operating system command. The operating system command is executed and Mapyrus writes the page to the standard input of the executing operating system command.

extras defines further options for the new page, given as *variable=value* values, separated by whitespace. See Table 8 for options available for each type of output.

File Format	Extras

PostScript, Encapsulated PostScript	`background=`*color*
	Background color for page, as a named color or as hex digits.
	
	`afmfiles=`*filename*`,`*filename2*`,...`
	Comma-separated list of PostScript Type 1 font metrics filenames to include in this PostScript file. PostScript Type 1 font metrics are defined in a file with suffix `.afm`. Include files for all PostScript fonts to be used in this page that are not known by the printer.
	See section 4.33 for information on converting TrueType fonts to PostScript Type 1 format.
	
	`pfafiles=`*filename*`,`*filename2*`,...`
	Comma-separated list of ASCII PostScript Type 1 font definition filenames to include in this PostScript file. An ASCII PostScript Type 1 font definition is defined in a file with suffix `.pfa`. Include files for all PostScript fonts to be used in this page that are not known by the printer.
	See section 4.33 for information on converting TrueType fonts to ASCII PostScript Type 1 format.
	
	`isolatinfonts=`*fontname*`,`*fontname2*`,...`
	Comma-separated list of PostScript Type 1 font names for which ISO Latin1 character encoding (also known as ISO-8859-1 encoding) is required. Use ISO Latin1 encoding when extended characters such as accented characters or a copyright symbol are to be displayed from the font.
	
	`minimumlinewidth=`*value*
	Sets a minimum line width. Thinner lines will be changed to this width. This avoids very thin lines which appear differently in different output formats.
	
	`resolution=`*value*
	Resolution for page, given as a dots-per-inch value. Replaces default value of 300.
	
	`turnpage=`*flag*
	If *flag* is `true` then turns a landscape orientation page 90 degrees so that it appears as a portrait page.
	
	`update=`*flag*
	If *flag* is `true` then the file with name *filename* is an existing PostScript file that is opened for editing. The existing file must be an Encapsulated PostScript file or a PostScript file containing only a single page. Drawing commands will draw over the top of the existing page. Page size is set to the size of the existing page, *width* and *height* of the new page are ignored.

Table 8: Output Formats

File Format	Extras
PDF	`afmfiles=`*`filename,filename2,...`* Comma-separated list of PostScript Type 1 font metrics filenames to include in this PDF file. PostScript Type 1 font metrics are defined in a file with suffix `.afm`. Include files for all PostScript fonts to be used in this page that are not one of the 14 standard PDF fonts. See section 4.34 for information on converting TrueType fonts to binary PostScript Type 1 format. `pfbfiles=`*`filename,filename2,...`* Comma-separated list of binary PostScript Type 1 font definition filenames to include in this PostScript file. A binary PostScript Type 1 font definition is defined in a file with suffix `.pfb`. Include files for all PostScript fonts to be used in this page that are not one of the 14 standard PDF fonts. See section 4.34 for information on converting TrueType fonts to binary PostScript Type 1 format. `isolatinfonts=`*`fontname,fontname2,...`* Comma-separated list of PostScript Type 1 font names for which ISO Latin1 character encoding (also known as ISO-8859-1 encoding) is required. Use ISO Latin1 encoding when extended characters such as accented characters or a copyright symbol are to be displayed from the font. `background=`*`color`* Background color for page, as a named color or as hex digits. `minimumlinewidth=`*`value`* Sets a minimum line width. Thinner lines will be changed to this width. This avoids very thin lines which appear differently in different output formats. `resolution=`*`value`* Resolution for page, given as a dots-per-inch value. Replaces default value of 72. `turnpage=`*`flag`* If *flag* is `true` then turns a landscape orientation page 90 degrees so that it appears as a portrait page.

Table 8: Output Formats

File Format	Extras
Scalable Vector Graphics (SVG)	`background=`*`color`* Background color for page, as a named color or as hex digits. `compress=`*`flag`* If *flag* is `true` then output is compressed with GZIP compression. `minimumlinewidth=`*`value`* Sets a minimum line width. Thinner lines will be changed to this width. This avoids very thin lines which appear differently in different output formats. `resolution=`*`value`* Resolution for page, given as a dots-per-inch value. Replaces default value of screen resolution. `scriptfile=`*`filename`* Name of file containing an XML `<script> ...</script>` element to add to SVG file.

Table 8: Output Formats

File Format	Extras
BMP, JPEG, PNG, PPM images, output to a window on screen and Encapsulated PostScript images	`background=`*`color`* Background color for image, as a named color or as hex digits. `fractionalfontmetrics=`*`flag`* If *flag* is `true` then slower, more accurate calculations are made for positioning letters in labels. Fractional font metrics are not used by default. `imagemap=`*`filename`* Creates a file containing an HTML imagemap for the image. An HTML imagemap contains hyperlinks to jump to and JavaScript functions to execute when the mouse is moved or clicked over the image. Entries in the imagemap are defined using the `eventscript` command. A completed imagemap file is surrounded by an HTML `<map>` tag and included in an HTML file. `labelantialiasing=`*`flag`* If *flag* is `true` then labels are drawn with anti-aliasing, improving readability. Labels are drawn with anti-aliasing by default. `lineantialiasing=`*`flag`* If *flag* is `true` then lines are drawn with anti-aliasing. Lines are not drawn with anti-aliasing by default. `minimumlinewidth=`*`value`* Sets a minimum line width. Thinner lines will be changed to this width. This avoids very thin lines which appear differently in different output formats. `resolution=`*`value`* Resolution for page, given as a dots-per-inch value. Replaces default value of screen resolution. `ttffiles=`*`filename`*`,`*`filename2`*`,...` Comma-separated list of TrueType font filenames to load for this page. A TrueType font is defined in a file with suffix `.ttf`. Do not use this option on operating systems that support TrueType fonts (Windows, Mac). All TrueType fonts are already available from the operating system. On operating systems that do not support TrueType fonts (Linux, UNIX) include filenames of all TrueType fonts to be used on this page. These fonts are loaded by Mapyrus. `update=`*`flag`* If *flag* is `true` then the file with name *filename* is an existing file that is opened for editing. Drawing commands will draw over the top of the existing image in the file. Page size is set to the size of the existing image, *width* and *height* of the new page are ignored.

Table 8: Output Formats

5.5.41 parallelpath

`parallelpath` *`distance`* `[,` *`distance`* `...]`

Replaces current path with new paths parallel to current path. For each given distance, a new path is created at *distance* millimeters to the right of current path. If a *distance* is negative then path is created to the left of the current path.

When used on complex paths with sharp angles, this command creates paths that self-intersect.

5.5.42 pdf

`pdf` *`filename`*`,` *`page`* `[,` *`size`*`]`

Displays a Portable Document Format (PDF) file at each `move` point in the current path.

filename is the name of a PDF file. *page* is the page number from the PDF file to display.

size is the optional size for the PDF file in millimeters. If no size is given or size is zero then the file is displayed at its natural size, as defined in the PDF file. The file is scaled and rotated according to the current `scale` and `rotate` settings.

PDF files can only be displayed when creating PDF output. For other formats, a grey box is drawn showing where the PDF file would be drawn.

5.5.43 pentagon

`pentagon` *`xCenter`*`,` *`yCenter`*`,` *`radius`*

Adds a pentagon shape to the current path, with center point (*xCenter*, *yCenter*) and distance *radius* from the center point to each vertex.

The center point and radius are transformed through any transformation set with a *worlds* command, then scaled and rotated by *scale* and *rotate* values.

5.5.44 print

`print` *`string`* `[,` *`string`* `...]`

Prints each *string* to standard output, separated by spaces. A newline is added after the final *string*.

Standard output is redirected to a different file or destination using the `setoutput` command.

5.5.45 protect

`protect` *`x1`*`,` *`y1`*`,` *`x2`*`,` *`y2`*
`protect` *`geometry`*
`protect`

Marks a region of the page as protected. The function `protected` will then return 1 for any point in this region.

The points (*x1*, *y1*) and (*x2*, *y2*) define any two opposite corners of the rectangle to mark as protected.

If *geometry* containing a polygon is given, then the region covered by that polygon is protected.

If no arguments are given then the region inside the current path is protected.

The rectangle or geometry is first transformed through any world coordinate transformation set with a `worlds` command, then scaled and rotated by `scale` and `rotate` values.

5.5.46 raindrop

`raindrop` *`xCenter , yCenter , radius`*

Adds a raindrop shape to the current path, with center point (*xCenter*, *yCenter*) and radius *radius*.

The center point and radius are transformed through any transformation set with a *worlds* command, then scaled and rotated by *scale* and *rotate* values.

5.5.47 reversepath

`reversepath`

Reverses the direction of the current path.

5.5.48 rotate

`rotate` *`angle`*

Rotates the coordinate system, adding to any existing rotation. *angle* is given in degrees, measured counter-clockwise. All later coordinates given in `move`, `draw`, `arc` and `addpath` commands are rotated by this angle.

5.5.49 rdraw

`rdraw` *`dx , dy , ...`*

Adds one or more straight line segments to the current path using relative distances. A straight line segment is added from the previously defined point a relative distance (*dx*, *dy*). Each further point adds a line segment relative to the point before. Points are first transformed through any world coordinate transformation set with a `worlds` command then scaled and rotated by `scale` and `rotate` values.

5.5.50 roundedbox

`roundedbox` *`x1 , y1 , x2 , y2`*
`roundedbox` *`x1 , y1 , x2 , y2 , radius`*

Adds a rectangle with rounded corners to the current path. The points (*x1*, *y1*) and (*x2*, *y2*) define any two opposite corners of the rectangle.

The radius of circular arcs at the rounded corners is *radius*, or 10% of the size of the rectangle if not given.

The points and radius are transformed through any transformation set with a *worlds* command, then scaled and rotated by *scale* and *rotate* values.

5.5.51 samplepath

`samplepath` *spacing*`,` *offset*

Replaces current path with equally spaced points along the path. *offset* is the distance along the path at which to place first point, given in millimeters. *spacing* is the distance between points, given in millimeters. The sign of *spacing* controls the direction in which the path is travelled. If *spacing* is a positive value, the path is travelled from the beginning towards the end. If *spacing* is a negative value, then the absolute value of *spacing* is used and the path is travelled from the end towards the beginning. Using a very large positive or negative value for *spacing* results in current path being replaced by a single point at the beginning or end of the path.

5.5.52 scale

`scale` *factor*

Scales the coordinate system, adding to any existing scaling. *factor* is scale factor for X and Y axes. All later coordinates given in `move`, `draw`, `arc` and `addpath` commands are scaled by this factor.

5.5.53 selectpath

`selectpath` *offset*`,` *length* `[,` *offset*`,` *length* `...]`

Selects one or more parts of the current path.

Each *offset* is a distance along the path at which to begin selecting the path, measured in millimeters. *length* is the length of path to select at that offset, measured in millimeters.

Offsets and lengths are scaled by `scale` values but are independent of any world coordinate transformation.

5.5.54 setoutput

`setoutput` *filename*

Sets file that all `print` commands will be written to. *filename* is the name of a file to write to, overwriting any existing file with this name.

5.5.55 shiftpath

`shiftpath` *x*`,` *y*

Shifts all points in the current path x millimeters along the X axis and y millimeters along the Y axis. Shift values are scaled and rotated by `scale` and `rotate` values but are independent of any world coordinate transformation.

Use this command repeatedly following a `clip "outside"` command to produce a shadow effect for polygons, as shown in Section 4.23 on page 75.

5.5.56 sinewave

`sinewave` *`xEnd , yEnd , repeats , height`*

Adds a a sine wave curve to the current path. The curve begins at the last point added to the path and ends at (*xEnd*, *yEnd*)

The sine wave is repeated *repeats* number of times.

height defines the height of the sine wave. If *height* is a negative number then a mirror image of the sine wave is produced.

The end point and height are transformed through any transformation set with a *worlds* command, then scaled and rotated by *scale* and *rotate* values.

5.5.57 sinkhole

`sinkhole`

Replaces the current path containing a polygon with a single point in the middle of the polygon, farthest from the polygon perimeter.

5.5.58 spiral

`spiral` *`xCenter , yCenter , radius , revolutions , startAngle`*

Adds a spiral to the current path, with center point (*xCenter*, *yCenter*) and given radius.

revolutions defines the number of loops of the spiral. If *revolutions* is a positive number then the spiral is drawn in a clockwise direction. If *revolutions* is a negative number then the spiral is drawn in an anti-clockwise direction.

startAngle defines the angle at which the outer revolution of the spiral starts.

The center point, radius and start angle are transformed through any transformation set with a *worlds* command, then scaled and rotated by *scale* and *rotate* values.

5.5.59 star

`star` *`xCenter , yCenter , radius , points`*

Adds a star shape to the current path, with center point (*xCenter*, *yCenter*), distance *radius* from the center to each point of the star. *points* is the number of points for the star.

The center point and radius are transformed through any transformation set with a *worlds* command, then scaled and rotated by *scale* and *rotate* values.

5.5.60 stripepath

`stripepath` *spacing*`,` *angle*

Replaces current path with equally spaced parallel lines that completely cover the path. *spacing* is the distance between lines, measured in millimeters. *angle* is angle of each line, measured counter-clockwise in degrees, with zero being horizontal. Follow this command with a `clip` command to produce a hatched fill pattern.

5.5.61 stroke

`stroke [`*xml-attributes*`]`

Draws the current path using the current color and linestyle. The current path is not modified by this command.

For SVG output, any XML attributes given in *xml-attributes* are included in the `<path>` XML element for the drawn path.

5.5.62 svg

`svg` *filename*`, [,` *size*`]`

Displays a Scalable Vector Graphics (SVG) file at each `move` point in the current path.

filename is the name of an SVG file, with either an `.svg` or `.svgz` suffix.

size is the optional size for the SVG file in millimeters. If no size is given or size is zero then the file is displayed at its natural size, as defined in the SVG file. The file is scaled and rotated according to the current `scale` and `rotate` settings.

SVG files can only be displayed when creating SVG output. For other formats, a grey box is drawn showing where the SVG file would be drawn.

5.5.63 svgcode

`svgcode` *xml*

Adds XML code to the output file.

XML code can only be added to Scalable Vector Graphics (SVG) files.

This enables the default Mapyrus settings in SVG files to be overridden and for graphics to be grouped together as layers.

5.5.64 table

`table` *extras*`,` *column1*`,` *column2* `...`

Draws a table at each point in the path set with the `move` command, One or more arrays are given defining values for each column of the table. Values in each column array are displayed as one column in the table.

Labels in the table are drawn using the current color and font settings.

extras defines further options for the table, given as *variable=value* values, separated by whitespace. See Table 9 for available options.

Extra	Description
`background=`*`colors`*	Comma-separated list of colors to use as background for entries in the table, as named colors or as hex digits. The colors are used in turn for each column in each row. When the end of the list is reached, the list is repeated. By default the background is not displayed.
`borders=`*`flag`*	If *flag* is `true` then a border is drawn around each entry in the table using the current linestyle and color. By default borders are drawn.
`justify=`*`justifications`*	A comma-separated list of horizontal justification values for each column in the table. Each justification is one of `left`, `right` or `center`.
`sortcolumn=`*`index`*	Index of column to sort on, with first column having index 1. Values in given column are sorted and all columns are displayed in the order of the sorted column. By default values are not sorted.
`sortorder=`*`order`*	Ordering for sort column. Either `asc` for ascending order, or `desc` for descending order. Default is ascending order.

Table 9: Table Extras

5.5.65 tree

`tree` *`extras`*`,` *`entries`*

Draws a tree of labels at each point in the path set with the `move` command. An array is given defining tree entries. Each entry is split using the delimiter. An entry starting with the same values as a previous entry is indented to the right with an arrow linking it to the previous entry.

Labels are drawn using the current color and font settings.

extras defines further options for the tree, given as *variable=value* values, separated by whitespace. See Table 10 for available options.

Extra	Description
`delimiter=`*`string`*	Delimiter used to determine indentation of labels. By default whitespace is used as the delimiter.

Table 10: Tree Extras

5.5.66 triangle

`triangle` *`xCenter`*`,` *`yCenter`*`,` *`radius`*`,` *`rotation`*

Adds an equilateral triangle to the current path, with center point (*xCenter*, *yCenter*) and distance *radius* from the center point to each vertex.

The triangle is rotated clockwise *rotation* degrees.

The center point, radius and rotation are transformed through any transformation set with a *worlds* command, then scaled and rotated by *scale* and *rotate* values.

5.5.67 unprotect

`unprotect` *x1* `,` *y1* `,` *x2* `,` *y2*
`unprotect` *geometry*
`unprotect`

Clears all protected regions from an area on the page.

The points (*x1*, *y1*) and (*x2*, *y2*) define any two opposite corners of the rectangle to clear.

If *geometry* containing a polygon is given then the region inside that polygon is cleared.

If no arguments are given then the region inside the current path is cleared.

The rectangle or geometry is first transformed through any world coordinate transformation set with a `worlds` command, then scaled and rotated by `scale` and `rotate` values.

5.5.68 wedge

`wedge` *xCenter* `,` *yCenter* `,` *radius* `,` *angle* `,` *sweep* `[,` *height* `]`

Adds a wedge (pie slice) shape to the current path, with center point (*xCenter*, *yCenter*) and radius *radius*. The wedge begins at angle *angle* measured counter-clockwise in degrees, with zero being horizontal. The wedge is open *sweep* degrees in a counter-clockwise direction. If *sweep* is negative then the wedge opens in a clockwise direction.

If *height* is given then the wedge is extended downwards by this value to produce 3 dimensional effect.

The center point, radius and height are transformed through any transformation set with a *worlds* command, then scaled and rotated by *scale* and *rotate* values.

5.5.69 worlds

`worlds` *wx1* `,` *wy1* `,` *wx2* `,` *wy2* `[,` *extras* `]`
`worlds` *wx1* `,` *wy1* `,` *wx2* `,` *wy2* `,` *px1* `,` *py1* `,` *px2* `,` *py2* `[,` *extras* `]`

Defines a world coordinate system for the page.

The coordinates (*wx1*, *wy1*) and (*wx2*, *wy2*) define the lower-left and upper-right world coordinate values.

The coordinates (*px1*, *py1*) and (*px2*, *py2*) define the lower-left and upper-right positions on the page in millimetres. The world coordinates are mapped into this area of the page. If page coordinates are not given then the world coordinates are mapped to the whole page.

The new world coordinates replace any world coordinates set with a previous `worlds` command.

extras defines further options, given as *variable=value* values, separated by whitespace. See Table 11 for available options.

Extra	Description
`units=`*`units`*	Defines the units of the world coordinates, either `metres`, `meters` or `feet`. If not given, units are assumed to be meters.
`distortion=`*`flag`*	If `true` then non-uniform scaling in X and Y axes is allowed. If `false` then the world coordinate range is expanded, if necessary, to maintain uniform scaling. If not given, then scaling is uniform.

Table 11: Worlds Extras

5.6 Error Handling

If Mapyrus encounters an error when interpreting commands, an error message is printed including the filename and line number at which the error occurred and Mapyrus exits immediately. The Java interpreter exits with a non-zero status.

If an error occurs when using the HTTP Server option, an HTTP failure status and the error message are returned to the HTTP client. The HTTP server continues, handling the next request.

5.7 Mapyrus HTTP Server

Mapyrus runs as an HTTP server when started with the `-s` command line option.

The HTTP server accepts and replies to requests from HTTP clients on the given port number. If port number is 0 then any free port number is used. The port number used is written to the log file or to standard output.

The HTTP server is multi-threaded to enable several requests to be handled simultaneously. If the HTTP server receives an HTTP request for a file with a suffix matching a well-known MIME type (such as `html`, `txt`, `ps`, `pdf`, `svg`, `zip` or a web image format), then the contents of that file are returned by the HTTP server to the HTTP client. Requests for files with no suffix, or with unknown file suffix such as `.mapyrus` are interpreted by Mapyrus using the following steps.

1. Set any parameters passed in the URL (following the `?` character in the URL or passed in an HTML form) as variables in Mapyrus. Variable names are converted to uppercase.

2. Read and execute the commands from the filename given in the URL.

3. Capture the standard output of these commands and return it to the HTTP client. An image file is returned if the `newpage` command is used with output file set to standard output. Otherwise the output of any `print` commands is returned. The HTTP header information set in a `mimetype` or `httpresponse` command is returned to the HTTP client.

   An HTTP error state is returned if the request fails.

Requests using either GET or POST methods are accepted by Mapyrus.

The HTTP server runs forever and is stateless. Each HTTP request is independent and variables, graphics state and legend entries are not shared between requests. Any files or URLs given on the command line when Mapyrus is started are interpreted before accepting HTTP requests. Procedures defined in these files are available when interpreting HTTP requests. This enables common procedures to be loaded only once at startup and not with every HTTP request.

Files are not cached and are read for each HTTP request.

For security, the HTTP server only replies to requests from the directory in which Mapyrus was started. Requests for files from other directories return an error to the HTTP client. If communication between HTTP client and Mapyrus is blocked for longer than 5 minutes then the HTTP request is cancelled. When all threads in the HTTP server are busy handling requests, further requests are queued.

Logging of HTTP requests is controlled by the `-l` command line option.

5.8 Mapyrus Servlet

The file `mapyrus.war` contains a web application. After being deployed in a web server, the Mapyrus servlet handles HTTP requests to the following URL.

```
http://localhost:8080/mapyrus/
```

The commands run by Mapyrus servlet when an HTTP request is received are defined in a servlet initialisation parameter. Servlet initialisation parameters are set in the servlet configuration file `WEB-INF/web.xml` contained in the web application file `mapyrus.war`.

By default, Mapyrus commands are defined in the servlet initialisation parameter `c1`. If the URL parameter `ipn` is set to a servlet initialisation parameter name then Mapyrus commands are read from that initialisation parameter instead.

Any parameters passed in the URL (following the `?` character in the URL or passed in an HTML form) are set as variables in Mapyrus. Variable names are converted to uppercase.

The standard output of the Mapyrus commands is returned from the HTTP request. A `mimetype` or `httpresponse` command must be used to define the type of output being returned.

Mapyrus commands are not cached. If a single `include` statement is given for the Mapyrus commands in the servlet initialisation parameter then it is possible to define the Mapyrus commands run by the servlet in a file outside of the web server. It is then not necessary to redeploy the servlet after modifying the file.

The Mapyrus servlet throws a Java ServletException if there is an error in the Mapyrus commands, or an error running the Mapyrus commands.