5

The Grid Graphics Model

Chapter preview

This chapter describes the fundamental tools that grid provides for drawing graphical scenes (including plots). There are basic features such as functions for drawing lines, rectangles, and text, together with more sophisticated and powerful concepts such as viewports, layouts, and units, which allow basic output to be located and sized in very flexible ways.

This chapter is useful for drawing a wide variety of pictures, including statistical plots from scratch, and for adding output to lattice plots.

The functions that make up the grid graphics system are provided in an add-on package called `grid`. The grid system is loaded into R as follows.

```
> library(grid)
```

In addition to the standard on-line documentation available via the `help()` function, grid provides both broader and more in-depth on-line documentation in a series of vignettes, which are available via the `vignette()` function.

The grid graphics system only provides low-level graphics functions. There are no high-level functions for producing complete plots. Section 5.1 briefly introduces the concepts underlying the grid system, but this only provides an indication of how to work with grid and some of the things that are possible. An effective direct use of grid functions requires a deeper understanding of the grid system (see later sections of this chapter and Chapter 6).

The lattice package described in Chapter 4 provides a good demonstration of the high-level results that can be achieved using grid. Other examples in this book are Figure 1.7 in Chapter 1 and Figures 7.1 and 7.18 in Chapter 7.

5.1 A brief overview of grid graphics

This chapter describes how to use grid to produce graphical output. There are functions to produce basic output, such as lines and rectangles and text, and there are functions to establish the context for drawing, such as specifying where output should be placed and what colors and fonts to use for drawing.

Like the traditional system, all grid output occurs on the current device,* and later output obscures any earlier output that it overlaps (i.e.,output follows the "painters model"). In this way, images can be constructed incrementally using grid by calling functions in sequence to add more and more output.

There are grid functions to draw primitive graphical output such as lines, text, and polygons, plus some slightly higher-level graphical components such as axes (see Section 5.2). Complex graphical output is produced by making a sequence of calls to these primitive functions.

The colors, line types, fonts, and other aspects that affect the appearance of graphical output are controlled via a set of graphical parameters (see Section 5.4).

Grid provides no predefined regions for graphical output, but there is a powerful facility for defining regions, based on the idea of a *viewport* (see Section 5.5). It is quite simple to produce a set of regions that are convenient for producing a single plot (see the example in the next section), but it is also possible to produce very complex sets of regions such as those used in the production of Trellis plots (see Chapter 4).

All viewports have a large set of coordinate systems associated with them so that it is possible to position and size output in physical terms (e.g., in centimeters) as well as relative to the scales on axes, and in a variety of other ways (see Section 5.3).

All grid output occurs relative to the current viewport (region) on a page. In order to start a new page of output, the user must call the `grid.newpage()`

*See Section 1.3.1 for information on devices and selecting a current device when more than one device is open.

function. The function `grid.prompt()` controls whether the user is prompted when moving to a new page.

As well as the side effect of producing graphical output, grid graphics functions produce objects representing output. These objects can be saved to produce a persistent record of a plot, and other grid functions exist to modify these graphical objects (for example, it is possible to interactively edit a plot). It is also possible to work entirely with graphical descriptions, without producing any output. Functions for working with graphical objects are described in detail in Chapter 6.

5.1.1 A simple example

The following example demonstrates the construction of a simple scatterplot using grid. This is more work than a single function call to produce the plot, but it shows some of the advantages that can be gained by producing the plot using grid.

This example uses the `pressure` data to produce a scatterplot much like that in Figure 1.1.

Firstly, some regions are created that will correspond to the "plot region" (the area within which the data symbols will be drawn) and the "margins" (the area used to draw axes and labels).

The following code creates two viewports. The first viewport is a rectangular region that leaves space for 5 lines of text at the bottom, 4 lines of text at the left side, 2 lines at the top, and 2 lines to the right. The second viewport is in the same location as the first, but it has x- and y-scales corresponding to the range of the pressure data to be plotted.

```
> pushViewport(plotViewport(c(5, 4, 2, 2)))
> pushViewport(dataViewport(pressure$temperature,
                            pressure$pressure,
                            name="plotRegion"))
```

The following code draws the scatterplot one piece at a time. Grid output occurs relative to the most recent viewport, which in this case is the viewport with the appropriate axis scales. The data symbols are drawn relative to the x- and y-scales, a rectangle is drawn around the entire plot region, and x- and y-axes are drawn to represent the scales.

```
> grid.points(pressure$temperature, pressure$pressure,
              name="dataSymbols")
> grid.rect()
> grid.xaxis()
> grid.yaxis()
```

Adding labels to the axes demonstrates the use of the different coordinate systems available. The label text is drawn outside the edges of the plot region and is positioned in terms of a number of lines of text (i.e.,the height that a line of text would occupy).

```
> grid.text("temperature", y=unit(-3, "lines"))
> grid.text("pressure", x=unit(-3, "lines"), rot=90)
```

The obvious result of running the above code is the graphical output (see the top-left image in Figure 5.1). Less obvious is the fact that several objects have been created. There are objects representing the viewport regions and there are objects representing the graphical output. The following code makes use of this fact to modify the plotting symbol from a circle to a triangle (see the top-right image in Figure 5.1). The object representing the data symbols was named `"dataSymbols"` (see the code above) and this name is used to find that object and modify it using the `grid.edit()` function.

```
> grid.edit("dataSymbols", pch=2)
```

The next piece of code makes use of the objects representing the viewports. The `upViewport()` and `downViewport()` functions are used to navigate between the different viewport regions to perform some extra annotations. First of all, a call to the `upViewport()` function is used to go back to working within the entire device so that a dashed rectangle can be drawn around the complete plot. Next, the `downViewport()` function is used to return to the plot region to add a text annotation that is positioned relative to the scale on the axes of the plot (see bottom-right image in Figure 5.1).

```
> upViewport(2)
> grid.rect(gp=gpar(lty="dashed"))
> downViewport("plotRegion")
> grid.text("Pressure (mm Hg)\nversus\nTemperature (Celsius)",
            x=unit(150, "native"), y=unit(600, "native"))
```

The final scatterplot is still quite simple in this example, but the techniques that were used to produce it are very general and powerful. It is possible to

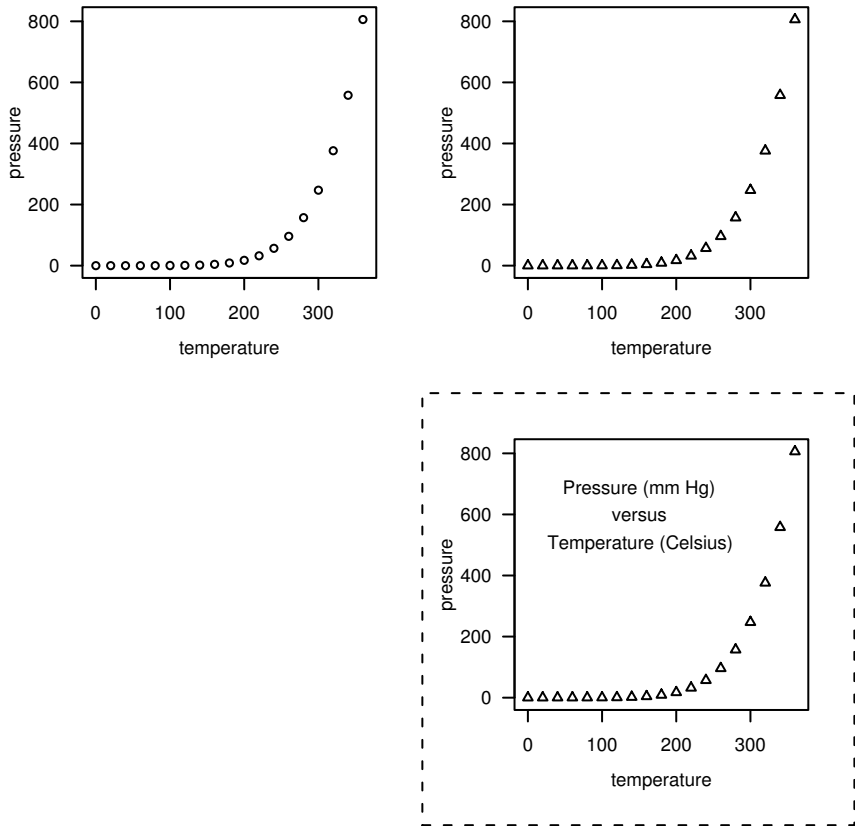

Figure 5.1
A simple scatterplot produced using grid. The top-left plot was constructed from a series of calls to primitive grid functions that produce graphical output. The top-right plot shows the result of calling the `grid.edit()` function to interactively modify the plotting symbol. The bottom-right plot was created by making calls to `upViewport()` and `downViewport()` to navigate between different drawing regions and adding further output (a dashed border and text within the plot).

produce a very complex plot, yet still have complete access to modify and add to any part of the plot.

In the remaining sections of this chapter, and in Chapter 6, the basic grid concepts of viewports and units are discussed in full detail. A complete understanding of the grid system will be useful in two ways: it will allow the user to produce very complex images from scratch (the issue of making them available to others is addressed in Chapter 7) and it will allow the user to work effectively with (e.g., modify and add to) complex grid output that is produced by other people's code (e.g. lattice plots).

5.2 Graphical primitives

The most simple grid functions to understand are those that draw something. There are a set of grid functions for producing basic graphical output such as lines, circles, and text.* Table 5.1 lists the full set of these functions.

The first arguments to most of these functions is a set of locations and dimensions for the graphical object to draw. For example, `grid.rect()` has arguments `x`, `y`, `width`, and `height` for specifying the locations and sizes of the rectangles to draw. An important exception is the `grid.text()` function, which requires the text to draw as its first argument.

In most cases, multiple locations and sizes can be specified and multiple primitives will be produced in response. For example, the following function call produces 100 circles because 100 locations and radii are specified (see Figure 5.2).

```
> grid.circle(x=seq(0.1, 0.9, length=100),
              y=0.5 + 0.4*sin(seq(0, 2*pi, length=100)),
              r=abs(0.1*cos(seq(0, 2*pi, length=100))))
```

The `grid.move.to()` and `grid.line.to()` functions are unusual in that they both only accept one location. These functions refer to and modify a "current location." The `grid.move.to()` function sets the current location and `grid.line.to()` draws from the current location to a new location, then sets

*All of these functions are of the form `grid.*()` and, for each one, there is a corresponding `*Grob()` function that creates an object containing a description of primitive graphical output, but does not draw anything. The `*Grob()` versions are addressed fully in Chapter 6.

Table 5.1
Graphical primitives in grid. This is the complete set of low-level functions that produce graphical output. For each function that produces graphical output (left-most column), there is a corresponding function that returns a graphical object containing a description of graphical output instead of producing graphical output (right-most column). The latter set of functions is described further in Chapter 6.

Function to Produce Output	**Description**	**Function to Produce Object**
`grid.move.to()`	Set the current location	`moveToGrob()`
`grid.line.to()`	Draw a line from the current location to a new location and reset the current location.	`lineToGrob()`
`grid.lines()`	Draw a single line through multiple locations in sequence.	`linesGrob()`
`grid.segments()`	Draw multiple lines between pairs of locations.	`segmentsGrob()`
`grid.rect()`	Draw rectangles given locations and sizes.	`rectGrob()`
`grid.circle()`	Draw circles given locations and radii.	`circleGrob()`
`grid.polygon()`	Draw polygons given vertexes.	`polygonGrob()`
`grid.text()`	Draw text given strings, locations and rotations.	`textGrob()`
`grid.arrows()`	Draw arrows at either end of lines given locations or an object describing lines.	`arrowsGrob()`
`grid.points()`	Draw data symbols given locations.	`pointsGrob()`
`grid.xaxis()`	Draw x-axis.	`xaxisGrob()`
`grid.yaxis()`	Draw y-axis.	`yaxisGrob()`

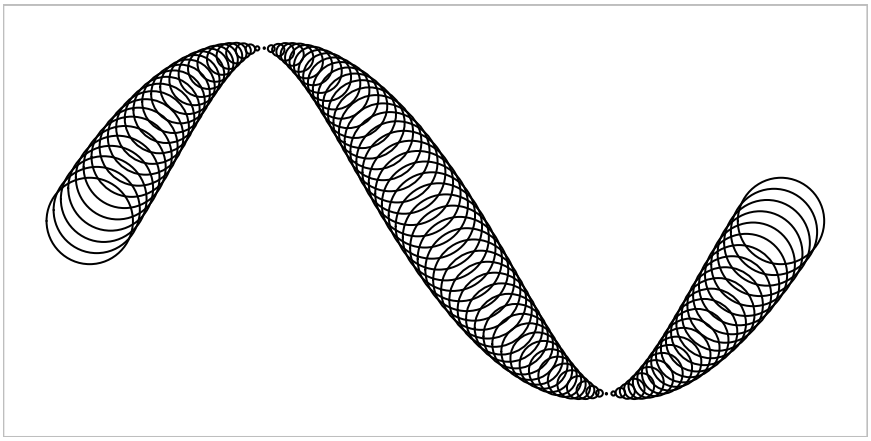

Figure 5.2
Primitive grid output. A demonstration of basic graphical output produced using a single call to the `grid.circle()` function. There are 100 circles of varying sizes, each at a different `(x, y)` location.

the current location to be the new location. The current location is not used by the other drawing functions*. In most cases, `grid.lines()` will be more convenient, but `grid.move.to()` and `grid.line.to()` are useful for drawing lines across multiple viewports (an example is given in Section 5.5.1).

The `grid.arrows()` function is used to add arrows to lines. A single line can be specified by `x` and `y` locations (through which a line will be drawn), *or* the `grob` argument can be used to specify an object that describes one or more lines (produced by `linesGrob()`, `segmentsGrob()`, or `lineToGrob()`). In the latter case, `grid.arrows()` will add arrows at the ends of the line(s). The following code demonstrates the different uses (see Figure 5.3). The first `grid.arrows()` call specifies locations via the `x` and `y` arguments to produce a single line, at the end of which an arrow is drawn. The second call specifies a `segments` graphical object via the `grob` argument, which describes three lines, and an arrow is added to the end of each of these lines.

```
> angle <- seq(0, 2*pi, length=50)
> grid.arrows(x=seq(0.1, 0.5, length=50),
              y=0.5 + 0.3*sin(angle))
> grid.arrows(grob=segmentsGrob(6:8/10, 0.2, 7:9/10, 0.8))
```

*There is one exception: the `grid.arrows()` function makes use of the current location when an arrow is added to a `line.to` graphical object produced by `lineToGrob()`.

Figure 5.3
Drawing arrows using the `grid.arrows()` function. Arrows can be added to: a single line through multiple points, as generated by `grid.lines()` (e.g., the sine curve in the left half of the figure); multiple straight line segments, as generated by `grid.segments()` (e.g., the three straight lines in the right half of the figure); the result of a line-to operation, as generated by `grid.line.to()` (example not shown here).

In simple usage, the `grid.polygon()` function draws a single polygon through the specified `x` and `y` locations (automatically joining the last location to the first to close the polygon). It is possible to produce multiple polygons from a single call (which is much faster than making multiple calls) if the `id` argument is specified. In this case, a polygon is drawn for each set of `x` and `y` locations corresponding to a different value of `id`. The following code demonstrates both usages (see Figure 5.4). The two `grid.polygon()` calls use the same `x` and `y` locations, but the second call splits the locations into three separate polygons using the `id` argument.

```
> angle <- seq(0, 2*pi, length=10)[-10]
> grid.polygon(x=0.25 + 0.15*cos(angle), y=0.5 + 0.3*sin(angle),
               gp=gpar(fill="grey"))
> grid.polygon(x=0.75 + 0.15*cos(angle), y=0.5 + 0.3*sin(angle),
               id=rep(1:3, each=3),
               gp=gpar(fill="grey"))
```

The `grid.xaxis()` and `grid.yaxis()` functions are not really graphical primitives as they produce relatively complex output consisting of both lines and text. They are included here because they complete the set of grid functions that produce graphical output. The main argument to these functions is the

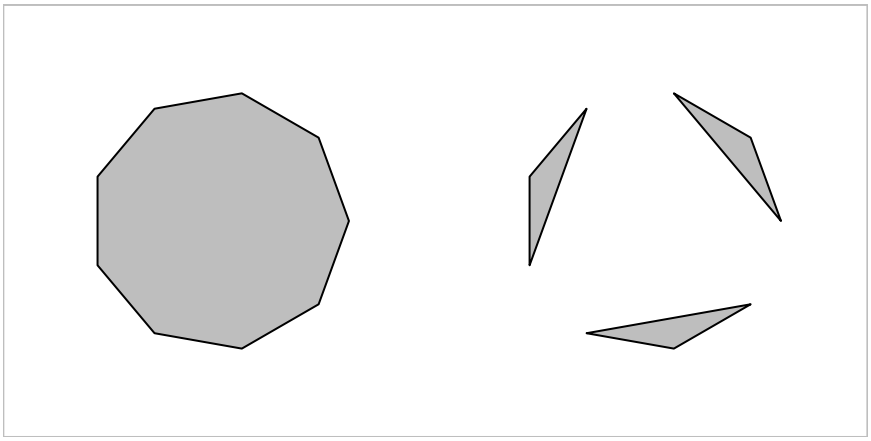

Figure 5.4
Drawing polygons using the `grid.polygon()` function. By default, a single polygon is produced from multiple `(x, y)` locations (the nonagon on the left), but it is possible to associate subsets of the locations with separate polygons using the `id` argument (the three triangles on the right).

`at` argument. This is used to specify where tick-marks should be placed. If the argument is not specified, sensible tick-marks are drawn based on the current scales in effect (see Section 5.5 for information about viewport scales). The values specified for the `at` argument are always relative to the current scales (see the concept of the `"native"` coordinate system in Section 5.3). These functions are much less flexible and general than the traditional `axis()` function. For example, they do not provide automatic support for generating labels from time- or date-based `at` locations.

Drawing curves

There is no native curve-drawing function in grid, but an approximation to a smooth curve consisting of many straight line segments is often sufficient. The example on the left of Figure 5.3 demonstrates how a series of line segments can appear very much like a smooth curve, if enough line segments are used.

5.2.1 Standard arguments

All primitive graphics functions accept a `gp` argument that allows control over aspects such as the color and line type of the relevant output. For example, the following code specifies that the boundary of the rectangle should be dashed

and colored red.

```
> grid.rect(gp=gpar(col="red", lty="dashed"))
```

Section 5.4 provides more information about setting graphical parameters.

All primitive graphics functions also accept a `vp` argument that can be used to specify a viewport in which to draw the relevant output. The following code shows a simple example of the syntax (the result is a rectangle drawn in the left half of the page); Section 5.5 describes viewports and the use of `vp` arguments in full detail.

```
> grid.rect(vp=viewport(x=0, width=0.5, just="left"))
```

Finally, all primitive graphics functions also accept a `name` argument. This can be used to identify the graphical object produced by the function. It is useful for interactively editing graphical output and when working with graphical objects (see Chapter 6). The following code demonstrates how to associate a name with a rectangle.

```
> grid.rect(name="myrect")
```

5.3 Coordinate systems

When drawing in grid, there are always a large number of coordinate systems available for specifying the locations and sizes of graphical output. For example, it is possible to specify an `x` location as a proportion of the width of the drawing region, or as a number of inches (or centimeters, or millimeters) from the left-hand edge of the drawing region, or relative to the current x-scale. The full set of coordinate systems available is shown in Table 5.2. The meaning of some of these will only become clear with an understanding of viewports (Section 5.5) and graphical objects (Chapter 6).*

With so many coordinate systems available, it is necessary to specify which coordinate system a location or size refers to. The `unit()` function is used

*Absolute units, such as inches, may not be rendered with full accuracy on screen devices (see the footnote on page 100).

Table 5.2
The full set of coordinate systems available in grid.

Coordinate System Name	Description
`"native"`	Locations and sizes are relative to the x- and y-scales for the current viewport.
`"npc"`	Normalized Parent Coordinates. Treats the bottom-left corner of the current viewport as the location $(0, 0)$ and the top-right corner as $(1, 1)$.
`"snpc"`	Square Normalized Parent Coordinates. Locations and sizes are expressed as a proportion of the *smaller* of the width and height of the current viewport.
`"inches"`	Locations and sizes are in terms of physical inches. For locations, $(0, 0)$ is at the bottom-left of the viewport.
`"cm"`	Same as `"inches"`, except in centimeters.
`"mm"`	Millimeters.
`"points"`	Points. There are 72.27 points per inch.
`"bigpts"`	Big points. There are 72 big points per inch.
`"picas"`	Picas. There are 12 points per pica.
`"dida"`	Dida. 1157 dida equals 1238 points.
`"cicero"`	Cicero. There are 12 dida per cicero.
`"scaledpts"`	Scaled points. There are 65536 scaled points per point.
`"char"`	Locations and sizes are specified in terms of multiples of the current nominal font size (dependent on the current `fontsize` and `cex`).
`"lines"`	Locations and sizes are specified in terms of multiples of the height of a line of text (dependent on the current `fontsize`, `cex`, and `lineheight`).
`"strwidth"` `"strheight"`	Locations and sizes are expressed as multiples of the width (or height) of a given string (dependent on the string and the current `fontsize`, `cex`, `fontfamily`, and `fontface`).
`"grobwidth"` `"grobheight"`	Locations and sizes are expressed as multiples of the width (or height) of a given graphical object (dependent on the type, location, and graphical settings of the graphical object).

to associate a numeric value with a coordinate system. This function creates an object of class `"unit"` (hereafter referred to simply as a *unit*), which acts very much like a normal `numeric` object — it is possible to perform basic operations such as sub-setting units, and adding and subtracting units.

Each value in a unit can be associated with a different coordinate system and each location and dimension of a graphical object is a separate unit, so for example, a rectangle can have its x-location, y-location, width, and height all specified relative to different coordinate systems.

The following pieces of code demonstrate some of the flexibility of grid units. The first code examples show some different uses of the `unit()` function: a single value is associated with a coordinate system, then several values are associated with a coordinate system (notice the recycling of the coordinate system value), then several values are associated with different coordinate systems.

```
> unit(1, "mm")
```

```
[1] 1mm
```

```
> unit(1:4, "mm")
```

```
[1] 1mm 2mm 3mm 4mm
```

```
> unit(1:4, c("npc", "mm", "native", "lines"))
```

```
[1] 1npc    2mm     3native 4lines
```

The next code examples show how units can be manipulated in many of the ways that normal numeric vectors can: firstly by sub-setting, then simple addition (again notice the recycling), then finally the use of a summary function (`max()` in this case).

```
> unit(1:4, "mm")[2:3]
```

```
[1] 2mm 3mm
```

```
> unit(1, "npc") - unit(1:4, "mm")
```

```
[1] 1npc-1mm 1npc-2mm 1npc-3mm 1npc-4mm
```

```
> max(unit(1:4, c("npc", "mm", "native", "lines")))
```

```
[1] max(1npc, 2mm, 3native, 4lines)
```

Some operations on units are not as straightforward as with numeric vectors, but require the use of functions written specifically for units. For example, the length of units must be obtained using the `unit.length()` function rather than `length()`, units must be concatenated (in the sense of the `c()` function) using `unit.c()`, and there are special functions for repeating units and for calculating parallel maxima and minima (`unit.rep()`, `unit.pmin()`, and `unit.pmax()`).

The following code provides an example of using units to locate and size a rectangle. The rectangle is at a location 40% of the way across the drawing region and 1 inch from the bottom of the drawing region. It is as wide as the text `"very snug"`, and it is one line of text high (see Figure 5.5).

```
> grid.rect(x=unit(0.4, "npc"), y=unit(1, "inches"),
            width=stringWidth("very snug"),
            height=unit(1, "lines"),
            just=c("left", "bottom"))
```

5.3.1 Conversion functions

As demonstrated in the previous section, a unit is not simply a numeric value. Units only reduce to a simple numeric value (a physical location on a graphics device) when drawing occurs. A consequence of this is that a unit can mean very different things, depending on when it gets drawn (this should become more apparent with an understanding of graphical parameters in Section 5.4 and viewports in Section 5.5).

In some cases, it can be useful to convert a unit to a simple numeric value. For example, it is sometimes necessary to know the current scale limits for numerical calculations. There are several functions that can assist with this problem: `convertUnit()`, `convertX()`, `convertY()`, `convertWidth()`, and `convertHeight()`. The following code demonstrates how to calculate the current scale limits for the x-dimension. First of all, a scale is defined on the x-axis with the range `c(-10, 50)` (see Section 5.5 for more about viewports).

```
> pushViewport(viewport(xscale=c(-10, 50)))
```

The next expression performs a query to obtain the current x-axis scale. The expression `unit(0:1, "npc")` represents the left and right boundaries of the

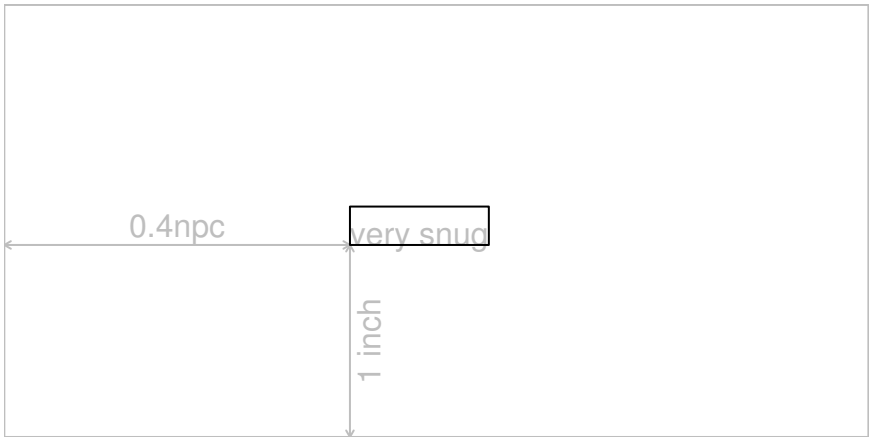

Figure 5.5
A demonstration of grid units. A diagram demonstrating how graphical output can be located and sized using grid units to associate numeric values with different coordinate systems. The grey border represents the current viewport. A black rectangle has been drawn with its bottom-left corner 40% of the way across the current viewport and 1 inch above the bottom of the current viewport. The rectangle is 1 line of text high and as wide as the text "very snug" (as it would be drawn in the current font).

current drawing region and `convertX()` is used to convert these locations into values in the `"native"` coordinate system, which is relative to the current scales.

```
> convertX(unit(0:1, "npc"), "native", valueOnly=TRUE)
```

```
[1] -10  50
```

WARNING: These conversion functions must be used with care. The output from these functions is only valid for the current device size. If, for example, a window on screen is resized, or output is copied from one device to another device with a different physical size, these calculations may no longer be correct. In other words, only rely on these functions when it is known that the size of the graphics device will not change. See Appendix B for more information on this topic and for a way to be able to use these functions on devices that may be resized. The discussion on the use of these functions in `drawDetails()` methods and the function `grid.record()` is also relevant (see "Calculations during drawing" in Section 7.3.10).

5.3.2 Complex units

There are two peculiarities of the "strwidth", "strheight", "grobwidth", and "grobheight" coordinate systems that require further explanation. In all of these cases, a value is interpreted as a multiple of the size of some other object. In the former two cases, the other object is just a text string (e.g., "a label"), but in the latter two cases, the other object can be any graphical object (see Chapter 6). It is necessary to specify the other object when generating a unit for these coordinate systems and this is achieved via the data argument. The following code shows some simple examples.

```
> unit(1, "strwidth", "some text")
```

```
[1] 1strwidth
```

```
> unit(1, "grobwidth", textGrob("some text"))
```

```
[1] 1grobwidth
```

A more convenient interface for generating units, when all values are relative to a single coordinate system, is also available via the stringWidth(), stringHeight(), grobWidth(), and grobHeight() functions. The following code is equivalent to the previous example.

```
> stringWidth("some text")
```

```
[1] 1strwidth
```

```
> grobWidth(textGrob("some text"))
```

```
[1] 1grobwidth
```

In this particular example, the "strwidth" and "grobwidth" units will be identical as they are based on identical pieces of text. The difference is that a graphical object can contain not only the text to draw, but other information that may affect the size of the text, such as the font family and size. In the following code, the two units are no longer identical because the text grob represents text drawn at font size of 18, whereas the simple string represents text at the default size of 10. The convertWidth() function is used to demonstrate the difference.

```
> convertWidth(stringWidth("some text"), "inches")
```

```
[1] 0.7175inches
```

```
> convertWidth(grobWidth(textGrob("some text",
                                  gp=gpar(fontsize=18))),
               "inches")
```

```
[1] 1.07625inches
```

For units that contain multiple values, there must be an object specified for every `"strwidth"`, `"strheight"`, `"grobwidth"`, and `"grobheight"` value. Where there is a mixture of coordinate systems within a unit, a value of `NULL` can be supplied for the coordinate systems that do not require data. The following code demonstrates this.

```
> unit(rep(1, 3), "strwidth", list("one", "two", "three"))
```

```
[1] 1strwidth 1strwidth 1strwidth
```

```
> unit(rep(1, 3),
       c("npc", "strwidth", "grobwidth"),
       list(NULL, "two", textGrob("three")))
```

```
[1] 1npc       1strwidth  1grobwidth
```

Again, there is a simpler interface for straightforward situations.

```
> stringWidth(c("one", "two", "three"))
```

```
[1] 1strwidth 1strwidth 1strwidth
```

For `"grobwidth"` and `"grobheight"` units, it is also possible to specify the name of a graphical object rather than the graphical object itself. This can be useful for establishing a reference to a graphical object, so that when the named graphical object is modified, the unit is updated for the change. The following code demonstrates this idea. First of all, a `text` grob is created with the name `"tgrob"`.

```
> grid.text("some text", name="tgrob")
```

Next, a unit is created that is based on the width of the grob called "tgrob".

```
> theUnit <- grobWidth("tgrob")
```

The convertWidth() function can be used to show the current value of the unit.

```
> convertWidth(theUnit, "inches")
```

```
[1] 0.7175inches
```

The following code modifies the grob named "tgrob" and convertWidth() is used to show that the value of the unit reflects the new width of the text grob.

```
> grid.edit("tgrob", gp=gpar(fontsize=18))
> convertWidth(theUnit, "inches")
```

```
[1] 1.07625inches
```

5.4 Controlling the appearance of output

All graphical primitives functions (and the viewport() function — see Section 5.5) — have a gp argument that can be used to provide a set of graphical parameters to control the appearance of the graphical output. There is a fixed set of graphical parameters (see Table 5.3), all of which can be specified for all types of graphical output.

The value supplied for the gp argument must be an object of class "gpar", and a gpar object can be produced using the gpar() function. For example, the following code produces a gpar object containing graphical parameter settings controlling color and line type.

```
> gpar(col="red", lty="dashed")
```

```
$col
[1] "red"

$lty
[1] "dashed"
```

Table 5.3
The full set of graphical parameters available in grid. The `lex` parameter has only been available since R version 2.1.0.

Parameter	Description
`col`	Color of lines, text, rectangle borders, ...
`fill`	Color for filling rectangles, circles, polygons, ...
`gamma`	Gamma correction for colors
`alpha`	Alpha blending coefficient for transparency
`lwd`	Line width
`lex`	Line width expansion multiplier applied to `lwd` to obtain final line width
`lty`	Line type
`lineend`	Line end style (round, butt, square)
`linejoin`	Line join style (round, mitre, bevel)
`linemitre`	Line mitre limit
`cex`	Character expansion multiplier applied to `fontsize` to obtain final font size
`fontsize`	Size of text (in points)
`fontface`	Font face (bold, italic, ...)
`fontfamily`	Font family
`lineheight`	Multiplier applied to final font size to obtain the height of a line

The function `get.gpar()` can be used to obtain current graphical parameter settings. The following code shows how to query the current line type and fill color. When called with no arguments, the function returns a complete list of current settings.

```
> get.gpar(c("lty", "fill"))

$lty
[1] "solid"

$fill
[1] "transparent"
```

A `gpar` object represents an *explicit graphical context* — settings for a small number of specific graphical parameters. The example above produces a graphical context that ensures that the color setting is `"red"` and the line-type setting is `"dashed"`. There is always an *implicit graphical context* consisting of default settings for all graphical parameters. The implicit graphical context is initialized automatically by grid for every graphics device and can be modified by viewports (see Section 5.5.5) or by gTrees (see Section 6.2.1).*

A graphical primitive will be drawn with graphical parameter settings taken from the implicit graphical context, except where there are explicit graphical parameter settings from the graphical primitive's `gp` argument. For graphical primitives, the explicit graphical context is only in effect for the duration of the drawing of the graphical primitive. The following code example demonstrates these rules.

The default initial implicit graphical context includes settings such as `lty="solid"` and `fill="transparent"`. The first (left-most) rectangle has an explicit setting `fill="black"` so it only uses the implicit setting `lty="solid"`. The second (right-most) rectangle uses all of the implicit graphical parameter settings. In particular, it is not at all affected by the explicit settings of the first rectangle (see Figure 5.6).

```
> grid.rect(x=0.33, height=0.7, width=0.2,
            gp=gpar(fill="black"))
> grid.rect(x=0.66, height=0.7, width=0.2)
```

*The ideas of implicit and explicit graphical contexts are similar to the specification of settings in Cascading Style Sheets[34] and the graphics state in PostScript[3].

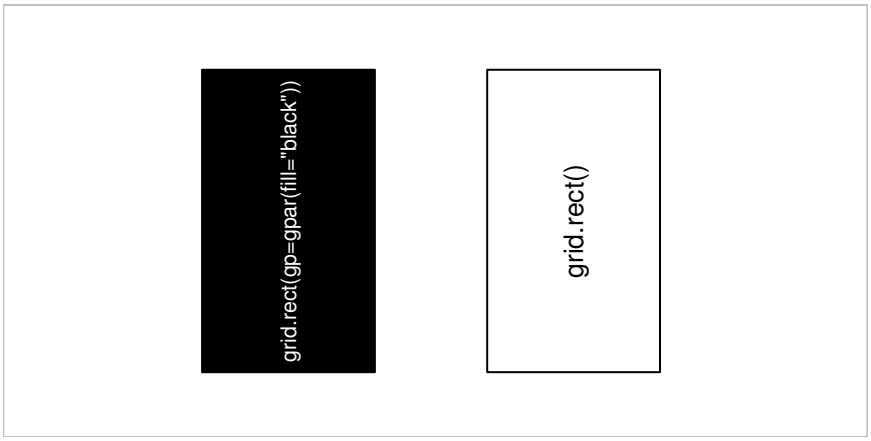

Figure 5.6
Graphical parameters for graphical primitives. The grey rectangle represents the current viewport. The right-hand rectangle has been drawn with no specific graphical parameters so it inherits the defaults for the current viewport (which in this case are a black border and no fill color). The left-hand rectangle has been drawn with a specific fill color of black (it is still drawn with the inherited black border). The graphical parameter settings for one rectangle have no effect on the other rectangle.

5.4.1 Specifying graphical parameter settings

The values that can be specified for colors, line types, line widths, line ends, line joins, and fonts are mostly the same as for the traditional graphics system. Sections 3.2.1, 3.2.2, and 3.2.3 contain descriptions of these specifications (for example, see the sub-section "Specifying colors"). In many cases, the graphical parameter in grid also has the same name as the traditional graphics state setting (e.g., `col`), though several of the grid parameters are slightly more verbose (e.g. `lineend` and `fontfamily`). Some other differences in the specification of graphical parameter values in the grid graphics system are described below.

In grid, the `fontface` value can be a string instead of an integer. Table 5.4 shows the possible string values.

In grid, the `cex` value is cumulative. This means that it is multiplied by the previous `cex` value to obtain a current `cex` value. The following code shows a simple example. A viewport is pushed with `cex=0.5`. This means that text will be half size. Next, some text is drawn, also with `cex=0.5`. This text is drawn quarter size because `cex` was already `0.5` from the viewport (`0.5*0.5 = 0.25`).

Table 5.4
Possible font face specifications in grid.

Integer	String	Description
1	"plain"	Roman or upright face
2	"bold"	Bold face
3	"italic" or "oblique"	Slanted face
4	"bold.italic"	Bold and slanted face
For the HersheySerif font family		
5	"cyrillic"	Cyrillic font
6	"cyrillic.oblique"	Slanted Cyrillic font
7	"EUC"	Japanese characters

```
> pushViewport(viewport(gp=gpar(cex=0.5)))
> grid.text("How small do you think?", gp=gpar(cex=0.5))
```

The `alpha` graphical parameter setting is unique to grid. It is a value between 1 (fully opaque) and 0 (fully transparent). The `alpha` value is combined with the alpha channel of colors by multiplying the two and this setting is cumulative like the `cex` setting. The following code shows a simple example. A viewport is pushed with `alpha=0.5`, then a rectangle is drawn using a semitransparent red fill color (alpha channel set to `0.5`). The final alpha channel for the fill color is `0.25` (`0.5*0.5 = 0.25`).

```
> pushViewport(viewport(gp=gpar(alpha=0.5)))
> grid.rect(width=0.5, height=0.5,
            gp=gpar(fill=rgb(1, 0, 0, 0.5)))
```

Grid does not support fill patterns (see page 58).

5.4.2 Vectorized graphical parameter settings

All graphical parameter settings may be vector values. Many graphical primitive functions produce multiple primitives as output and graphical parameter settings will be recycled over those primitives. The following code produces 100 circles, cycling through 50 different shades of grey for the circles (see Figure 5.7).

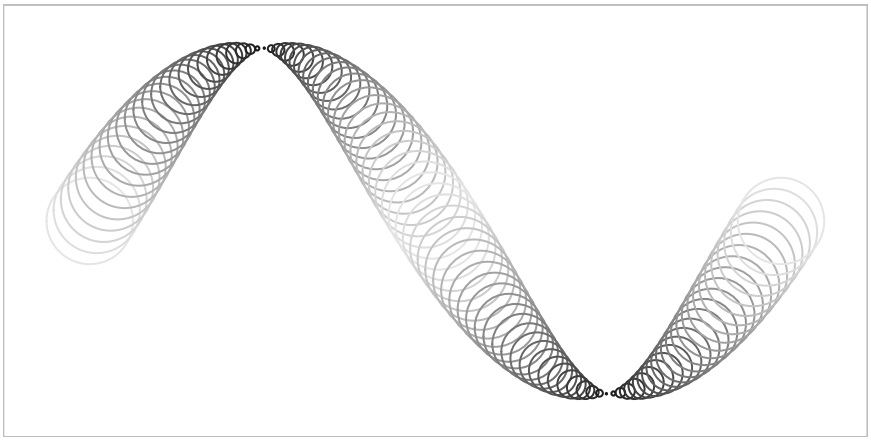

Figure 5.7
Recycling graphical parameters. The 100 circles are drawn by a single function call with 50 different greys specified for the border color (from a very light grey to a very dark grey and back to a very light grey). The 50 colors are recycled over the 100 circles so circle i gets the same color as circle $i + 50$.

```
> levels <- round(seq(90, 10, length=25))
> greys <- paste("grey", c(levels, rev(levels)), sep="")
> grid.circle(x=seq(0.1, 0.9, length=100),
              y=0.5 + 0.4*sin(seq(0, 2*pi, length=100)),
              r=abs(0.1*cos(seq(0, 2*pi, length=100))),
              gp=gpar(col=greys))
```

The `grid.polygon()` function is a slightly complex case. There are two ways in which this function will produce multiple polygons: when the `id` argument is specified *and* when there are `NA` values in the `x` or `y` locations (see Section 5.6). For `grid.polygon()`, a different graphical parameter will only be applied to each polygon identified by a different `id`. When a single polygon (as identified by a single `id` value) is split into multiple sub-polygons by `NA` values, all sub-polygons receive the same graphical parameter settings. The following code demonstrates these rules (see Figure 5.8). The first call to `grid.polygon()` draws two polygons as specified by the `id` argument. The `fill` graphical parameter setting contains two colors so the first polygon gets the first color (grey) and the second polygon gets the second color (white). In the second call, all that has changed is that an `NA` value has been introduced. This means that the first polygon as specified by the `id` argument is split into two separate polygons, but both of these polygons use the same `fill` setting because they both correspond to an `id` of `1`. Both of these polygons get the first color (grey).

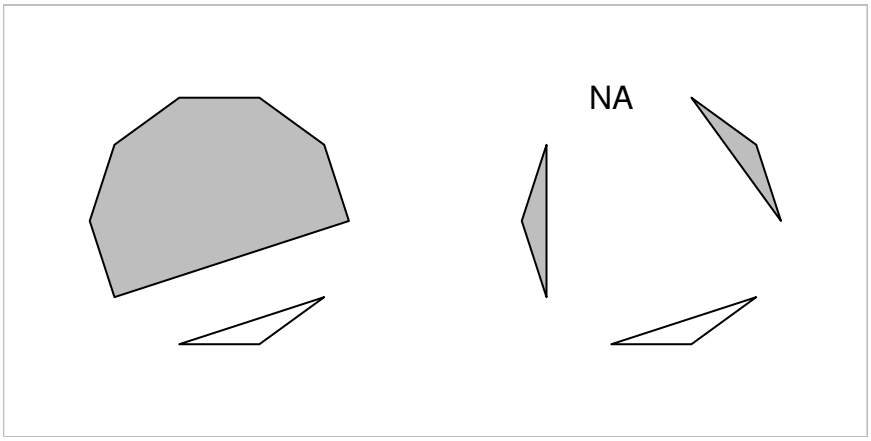

Figure 5.8
Recycling graphical parameters for polygons. On the left, a single function call produces two polygons with different fill colors by specifying an `id` argument and two fill colors. On the right, there are three polygons because an `NA value` has been introduced in the `(x, y)` locations for the polygon, but there are still only two colors specified. The colors are allocated to polygons using the `id` argument and ignoring any `NA` values.

```
> angle <- seq(0, 2*pi, length=11)[-11]
> grid.polygon(x=0.25 + 0.15*cos(angle), y=0.5 + 0.3*sin(angle),
               id=rep(1:2, c(7, 3)),
               gp=gpar(fill=c("grey", "white")))
> angle[4] <- NA
> grid.polygon(x=0.75 + 0.15*cos(angle), y=0.5 + 0.3*sin(angle),
               id=rep(1:2, c(7, 3)),
               gp=gpar(fill=c("grey", "white")))
```

All graphical primitives have a `gp` component, so it is possible to specify any graphical parameter setting for any graphical primitive. This may seem inefficient, and indeed in some cases the values are completely ignored (e.g., text drawing ignores the `lty` setting), but in many cases the values are potentially useful. For example, even when there is no text being drawn, the settings for `fontsize`, `cex`, and `lineheight` are always used to calculate the meaning of `"lines"` and `"char"` coordinates.

5.5 Viewports

A *viewport* is a rectangular region that provides a context for drawing.

A viewport provides a *drawing context* consisting of both a *geometric context* and a *graphical context*. A geometric context consists of a set of coordinate systems for locating and sizing output and all of the coordinate systems described in Section 5.3 are available within every viewport.* A graphical context consists of explicit graphical parameter settings for controlling the appearance of output. This is specified as a `gpar` object via the `gp` argument.

By default, grid creates a viewport that corresponds to the entire graphics device and, until another viewport is created, drawing occurs within the full extent of the device and using the default graphical parameter settings.

A new viewport is created using the `viewport()` function. A viewport has a location (given by `x` and `y`), a size (given by `width` and `height`), and it is justified relative to its location (according to the value of the `just` argument). The location and size of a viewport are specified in units, so a viewport can be positioned and sized within another viewport in a very flexible manner. The following code creates a viewport that is left-justified at an `x` location 0.4 of the way across the drawing region, and bottom-justified 1 centimeter from the bottom of the drawing region. It is as wide as the text `"very very snug indeed"`, and it is six lines of text high. Figure 5.9 shows a diagram representing this viewport.

```
> viewport(x=unit(0.4, "npc"), y=unit(1, "cm"),
           width=stringWidth("very very snug indeed"),
           height=unit(6, "lines"),
           just=c("left", "bottom"))
```

```
viewport[GRID.VP.33]
```

An important thing to notice in the above example is that the result of the `viewport()` function is an object of class `viewport`. No region has actually been created on a graphics device. In order to create regions on a graphics device, a `viewport` object must be *pushed* onto the device, as described in the next section.

*The idea of being able to define a geometric context is similar to the concept of the current transformation matrix (CTM) in PostScript[3] and the modeling transformation in OpenGL[55].

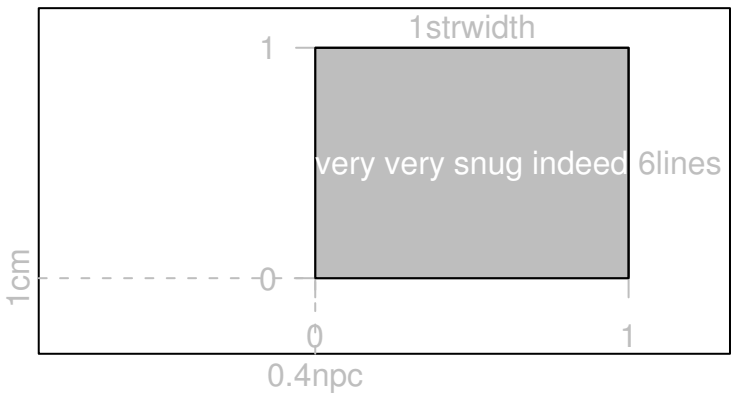

Figure 5.9
A diagram of a simple viewport. A viewport is a rectangular region specified by an `(x, y)` location, a `(width, height)` size, and a justification (and possibly a rotation). This diagram shows a viewport that is left-bottom justified 1 centimeter off the bottom of the page and 0.4 of the way across the page. It is 6 lines of text high and as wide as the text "very very snug indeed".

5.5.1 Pushing, popping, and navigating between viewports

The `pushViewport()` function takes a `viewport` object and uses it to create a region on the graphics device. This region becomes the drawing context for all subsequent graphical output, until the region is removed or another region is defined.

The following code demonstrates this idea (see Figure 5.10). To start with, the entire device, and the default graphical parameter settings, provide the drawing context. Within this context, the `grid.text()` call draws some text at the top-left corner of the device. A viewport is then pushed, which creates a region 80% as wide as the device, half the height of the device, and rotated at an angle of 10 degrees*. The viewport is given a name, `"vp1"`, which will help us to navigate back to this viewport from another viewport later.

Within the new drawing context defined by the viewport that has been pushed, *exactly the same* `grid.text()` call produces some text at the top-left corner of the viewport. A rectangle is also drawn to make the extent of the new viewport clear.

*It is not often very useful to rotate a viewport, but it helps in this case to dramatise the difference between the drawing regions.

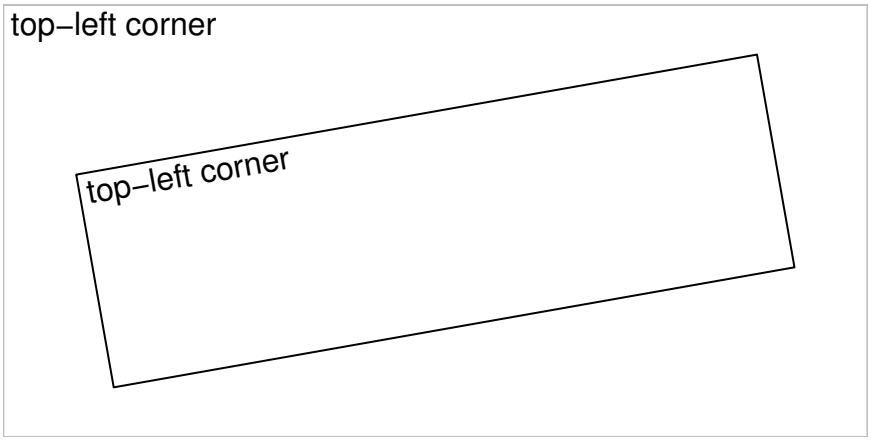

Figure 5.10
Pushing a viewport. Drawing occurs relative to the entire device until a viewport is pushed. For example, some text has been drawn in the top-left corner of the device. Once a viewport has been pushed, output is drawn relative to that viewport. The black rectangle represents a viewport that has been pushed and text has been drawn in the top-left corner of that viewport.

```
> grid.text("top-left corner", x=unit(1, "mm"),
            y=unit(1, "npc") - unit(1, "mm"),
            just=c("left", "top"))
> pushViewport(viewport(width=0.8, height=0.5, angle=10,
               name="vp1"))
> grid.rect()
> grid.text("top-left corner", x=unit(1, "mm"),
            y=unit(1, "npc") - unit(1, "mm"),
            just=c("left", "top"))
```

The pushing of viewports is entirely general. A viewport is pushed relative to the current drawing context. The following code slightly extends the previous example by pushing a further viewport, exactly like the first, and again drawing text at the top-left corner (see Figure 5.11). The location, size, and rotation of this second viewport are all relative to the context provided by the first viewport. Viewports can be nested like this to any depth.

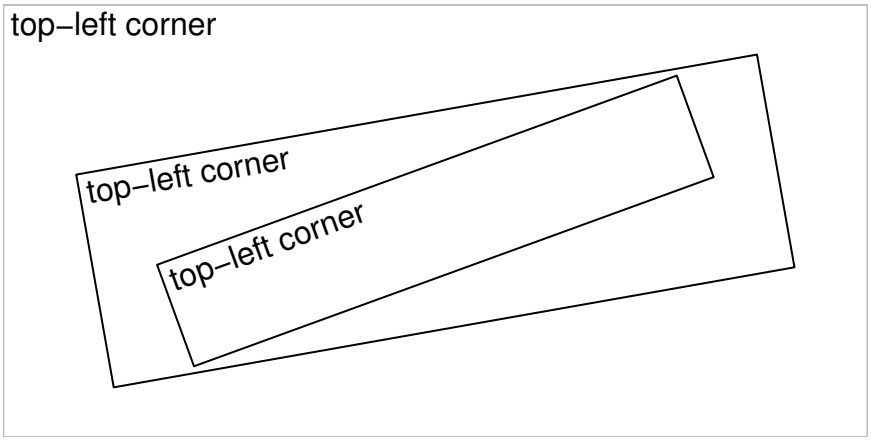

Figure 5.11
Pushing several viewports. Viewports are pushed relative to the current viewport. Here, a second viewport has been pushed relative to the viewport that was pushed in Figure 5.10. Again, text has been drawn in the top-left corner.

```
> pushViewport(viewport(width=0.8, height=0.5, angle=10,
               name="vp2"))
> grid.rect()
> grid.text("top-left corner", x=unit(1, "mm"),
            y=unit(1, "npc") - unit(1, "mm"),
            just=c("left", "top"))
```

In grid, drawing is always within the context of the current viewport. One way to change the current viewport is to push a viewport (as in the previous examples), but there are other ways too. For a start, it is possible to *pop* a viewport using the `popViewport()` function. This removes the current viewport and the drawing context reverts to whatever it was before the current viewport was pushed*. The following code demonstrates popping viewports (see Figure 5.12). The call to `popViewport()` removes the last viewport created on the device. Text is drawn at the bottom-right of the resulting drawing region (which has reverted back to being the first viewport that was pushed).

```
> popViewport()
> grid.text("bottom-right corner",
            x=unit(1, "npc") - unit(1, "mm"),
            y=unit(1, "mm"), just=c("right", "bottom"))
```

*It is illegal to pop the top-most viewport that represents the entire device region and the default graphical parameter settings. Trying to do so will result in an error.

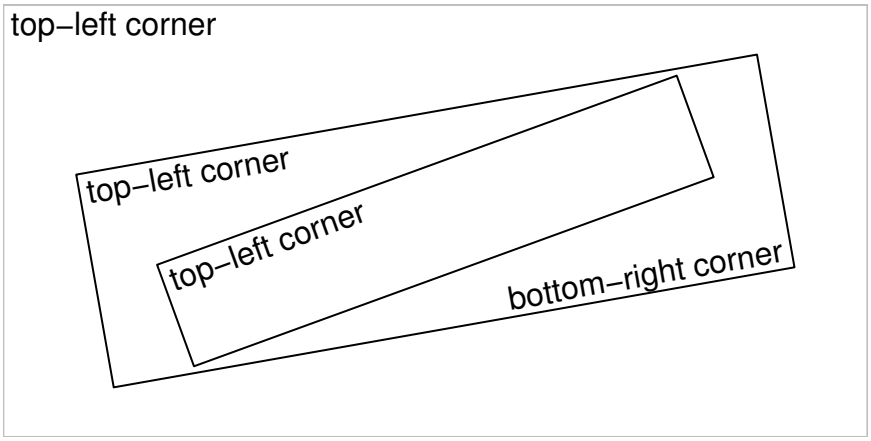

Figure 5.12
Popping a viewport. When a viewport is popped, the drawing context reverts to the parent viewport. In this figure, the second viewport (pushed in Figure 5.11) has been popped to go back to the first viewport (pushed in Figure 5.10). This time text has been drawn in the bottom-right corner.

The `popViewport()` function has an integer argument `n` that specifies how many viewports to pop. The default is `1`, but several viewports can be popped at once by specifying a larger value. The special value of `0` means that all viewports should be popped. In other words, the drawing context should revert to the entire device and the default graphical parameter settings.

Another way to change the current viewport is by using the `upViewport()` and `downViewport()` functions. The `upViewport()` function is similar to `popViewport()` in that the drawing context reverts to whatever it was prior to the current viewport being pushed. The difference is that `upViewport()` does not remove the current viewport from the device. This difference is significant because it means that that a viewport can be revisited without having to push it again. Revisiting a viewport is faster than pushing a viewport and it allows the creation of viewport regions to be separated from the production of output (see "viewport paths" in Section 5.5.3 and Chapter 7).

A viewport can be revisited using the `downViewport()` function. This function has an argument `name` that can be used to specify the name of an existing viewport. The result of `downViewport()` is to make the named viewport the current drawing context. The following code demonstrates the use of `upViewport()` and `downViewport()` (see Figure 5.13).

A call to `upViewport()` is made, which reverts the drawing context to the entire device (recall that prior to this navigation the current viewport was the first viewport that was pushed) and text is drawn in the bottom-right

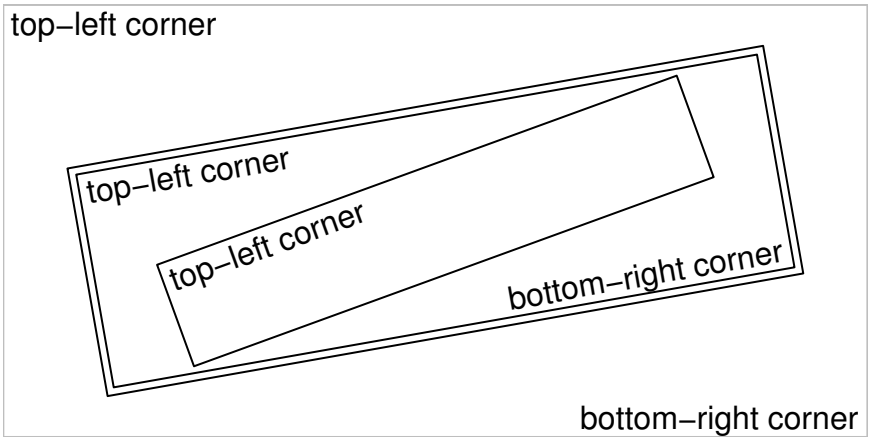

Figure 5.13
Navigating between viewports. Rather than popping a viewport, it is possible to navigate up from a viewport (and leave the viewport on the device). Here navigation has occurred from the first viewport to revert the drawing context to the entire device and text has been drawn in the bottom-right corner. Next, there has been a navigation down to the first viewport again and a second border has been drawn around the outside of the viewport.

corner. The `downViewport()` function is then used to navigate back down to the viewport that was first pushed and a second border is drawn around this viewport. The viewport to navigate down to is specified by its name, `"vp1"`.

```
> upViewport()
> grid.text("bottom-right corner",
            x=unit(1, "npc") - unit(1, "mm"),
            y=unit(1, "mm"), just=c("right", "bottom"))
> downViewport("vp1")
> grid.rect(width=unit(1, "npc") + unit(2, "mm"),
            height=unit(1, "npc") + unit(2, "mm"))
```

There is also a `seekViewport()` function that can be used to travel across the viewport tree. This can be convenient for interactive use, but the result is less predictable, so it is less suitable for use in writing grid functions for others to use. The call `seekViewport("avp")` is equivalent to `upViewport(0); downViewport("avp")`.

Drawing between viewports

Sometimes it is useful to be able to locate graphical output relative to more than one viewport. The only way to do this in grid is via the `grid.move.to()` and `grid.line.to()` functions. It is possible to call `grid.move.to()` within one viewport, change viewports, and call `grid.line.to()`. An example is provided in Section 5.8.2.

5.5.2 Clipping to viewports

Drawing can be restricted to only the interior of the current viewport (*clipped* to the viewport) by specifying the `clip` argument to the `viewport()` function. This argument has three values: `"on"` indicates that output should be clipped to the current viewport; `"off"` indicates that output should not be clipped at all; `"inherit"` means that the clipping region of the previous viewport should be used (this may not have been set by the previous viewport if that viewport's `clip` argument was also `"inherit"`). The following code provides a simple example (see Figure 5.14). A viewport is pushed with clipping on and a circle with a very thick black border is drawn relative to the viewport. A rectangle is also drawn to show the extent of the viewport. The circle partially extends beyond the limits of the viewport, so only those parts of the circle that lie within the viewport are drawn.

```
> pushViewport(viewport(w=.5, h=.5, clip="on"))
> grid.rect()
> grid.circle(r=.7, gp=gpar(lwd=20))
```

Next, another viewport is pushed and this viewport just inherits the clipping region from the first viewport. Another circle is drawn, this time with a grey and slightly thinner border and again the circle is clipped to the viewport.

```
> pushViewport(viewport(clip="inherit"))
> grid.circle(r=.7, gp=gpar(lwd=10, col="grey"))
```

Finally, a third viewport is pushed with clipping turned off. Now, when a third circle is drawn (with a thin, black border) all of the circle is drawn, even though parts of the circle extend beyond the viewport.

```
> pushViewport(viewport(clip="off"))
> grid.circle(r=.7)
> popViewport(3)
```

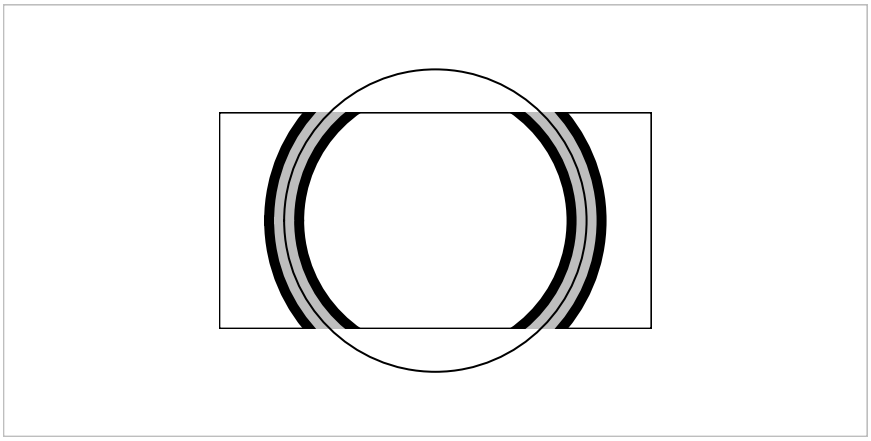

Figure 5.14
Clipping output in viewports. When a viewport is pushed, output can be clipped to that viewport, or the clipping region can be left in its current state, or clipping can be turned off entirely. In this figure, a viewport is pushed (the black rectangle) with clipping on. A circle is drawn with a very thick black border and it gets clipped. Next, another viewport is pushed (in the same location) with clipping left as it was. A second circle is drawn with a slightly thinner grey border and it is also clipped. Finally, a third viewport is pushed, which turns clipping off. A circle is drawn with a thin black border and this circle is not clipped.

5.5.3 Viewport lists, stacks, and trees

It can be convenient to work with several viewports at once and there are several facilities for doing this in grid. The `pushViewport()` function will accept multiple arguments and will push the specified viewports one after another. For example, the fourth expression below is a shorter equivalent version of the first three expressions.

```
> pushViewport(vp1)
> pushViewport(vp2)
> pushViewport(vp3)

> pushViewport(vp1, vp2, vp3)
```

The `pushViewport()` function will also accept objects that contain several viewports: viewport lists, viewport stacks, and viewport trees. The function `vpList()` creates a list of viewports and these are pushed "in parallel." The first viewport in the list is pushed, then grid navigates back up before the next viewport in the list is pushed. The `vpStack()` function creates a stack of viewports and these are pushed "in series." Pushing a stack of viewports is exactly the same as specifying the viewports as multiple arguments to `pushViewport()`. The `vpTree()` function creates a tree of viewports that consists of a parent viewport and any number of child viewports. The parent viewport is pushed first, then the child viewports are pushed in parallel within the parent.

The current set of viewports that have been pushed on the current device constitute a viewport tree and the `current.vpTree()` function prints out a representation of the current viewport tree. The following code demonstrates the output from `current.vpTree()` and the difference between lists, stacks, and trees of viewports. First of all, some (trivial) viewports are created to work with.

```
> vp1 <- viewport(name="A")
> vp2 <- viewport(name="B")
> vp3 <- viewport(name="C")
```

The next piece of code shows these three viewports pushed as a list. The output of `current.vpTree()` shows the root viewport (which represents the entire device) and then all three viewports as children of the root viewport.

```
> pushViewport(vpList(vp1, vp2, vp3))
> current.vpTree()
```

```
viewport[ROOT]->(viewport[A], viewport[B], viewport[C])
```

This next code pushes the three viewports as a stack. The viewport `vp1` is now the only child of the root viewport with `vp2` a child of `vp1`, and `vp3` a child of `vp2`.

```
> grid.newpage()
> pushViewport(vpStack(vp1, vp2, vp3))
> current.vpTree()
```

```
viewport[ROOT]->(viewport[A]->(viewport[B]->(viewport[C])))
```

Finally, the three viewports are pushed as a tree, with `vp1` as the parent and `vp2` and `vp3` as its children.

```
> grid.newpage()
> pushViewport(vpTree(vp1, vpList(vp2, vp3)))
> current.vpTree()
```

```
viewport[ROOT]->(viewport[A]->(viewport[B], viewport[C]))
```

As with single viewports, viewport lists, stacks, and trees can be provided as the `vp` argument for graphical functions (see Section 5.5.4).

Viewport paths

The `downViewport()` function, by default, searches down the current viewport tree as far as is necessary to find a given viewport name. This is convenient for interactive use, but can be ambiguous if there is more than one viewport with the same name in the viewport tree.

Grid provides the concept of a *viewport path* to resolve such ambiguity. A viewport path is an ordered list of viewport names, which specify a series of parent-child relations. A viewport path is created using the `vpPath()` function. For example, the following code produces a viewport path that specifies a viewport called `"C"` with a parent called `"B"`, which in turn has a parent called `"A"`.

```
> vpPath("A", "B", "C")
```

```
A::B::C
```

For convenience in interactive use, a viewport path may be specified directly as a string. For example, the previous viewport path could be specified simply as `"A::B::C"`. The `vpPath()` function should be used when writing graphics functions for others to use.

The `name` argument to the `downViewport()` function will accept a viewport path, in which case it searches for a viewport that matches the entire path. The `strict` argument to `downViewport()` ensures that a viewport will only be found if the full viewport path is found, *starting from the current location in the viewport tree.*

5.5.4 Viewports as arguments to graphical primitives

As mentioned in Section 5.2.1, a viewport may be specified as an argument to functions that produce graphical output (via an argument called `vp`). When a viewport is specified in this way, the viewport gets pushed before the graphical output is produced and popped afterwards. To make this completely clear, the following two code segments are identical. First of all, a simple viewport is defined.

```
> vp1 <- viewport(width=0.5, height=0.5, name="vp1")
```

The next code explicitly pushes the viewport, draws some text, then pops the viewport.

```
> pushViewport(vp1)
> grid.text("Text drawn in a viewport")
> popViewport()
```

This next piece of code does the same thing in a single call.

```
> grid.text("Text drawn in a viewport", vp=vp1)
```

It is also possible to specify the name of a viewport (or a viewport path) for a `vp` argument. In this case, the name (or path) is used to navigate down to the viewport (via a call to `downViewport()`) and then back up again afterwards (via a call to `upViewport()`). This promotes the practice of pushing viewports once, then specifying where to draw different output by simply naming the appropriate viewport. The following code does the same thing as the previous example, but leaves the viewport intact (so that it can be used for further drawing).

```
> pushViewport(vp1)
> upViewport()
> grid.text("Text drawn in a viewport", vp="vp1")
```

This feature is also very useful when annotating a plot produced by a high-level graphics function. As long as the graphics function names the viewports that it creates and does not pop them, it is possible to revisit the viewports to add further output. Examples of this are given in Section 5.8 and this approach to writing high-level grid functions is discussed further in Chapter 7.

5.5.5 Graphical parameter settings in viewports

A viewport can have graphical parameter settings associated with it via the `gp` argument to `viewport()`. When a viewport has graphical parameter settings, those settings affect all graphical objects drawn within the viewport, and all other viewports pushed within the viewport, unless the graphical objects or the other viewports specify their own graphical parameter setting. In other words, the graphical parameter settings for a viewport modify the implicit graphical context (see page 168).

The following code demonstrates this rule. A viewport is pushed that has a `fill="grey"` setting. A rectangle with no graphical parameter settings is drawn within that viewport and this rectangle "inherits" the `fill="grey"` setting. Another rectangle is drawn with its own `fill` setting so it does not inherit the viewport setting (see Figure 5.15).

```
> pushViewport(viewport(gp=gpar(fill="grey")))
> grid.rect(x=0.33, height=0.7, width=0.2)
> grid.rect(x=0.66, height=0.7, width=0.2,
            gp=gpar(fill="black"))
> popViewport()
```

The graphical parameter settings in a viewport only affect other viewports and graphical output within that viewport. The settings do not affect the viewport itself. For example, parameters controlling the size of text (`fontsize`, `cex`, etc.) do not affect the meaning of `"lines"` units when determining the location and size of the viewport (but they will affect the location and size of other viewports or graphical output within the viewport). A layout (see Section 5.5.6) counts as being within the viewport (i.e., it is affected by the graphical parameter settings of the viewport).

If there are multiple values for a graphical parameter setting, only the first is used when determining the location and size of a viewport.

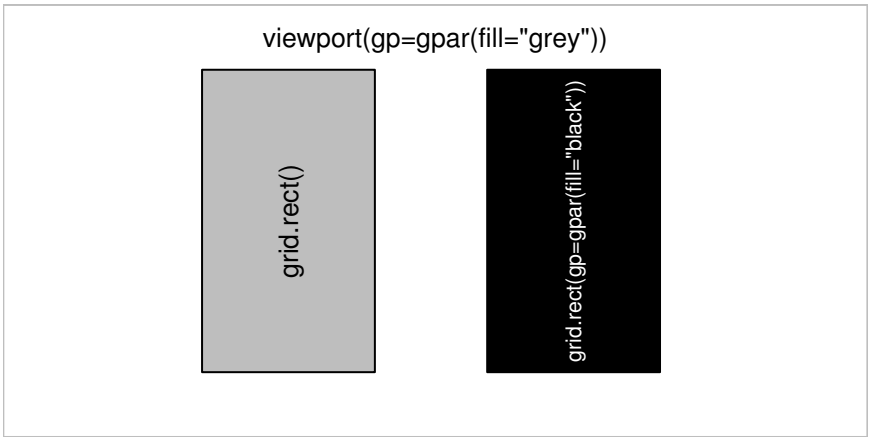

Figure 5.15
The inheritance of viewport graphical parameters. A diagram demonstrating how viewport graphical parameter settings are inherited by graphical output within the viewport. The viewport sets the default fill color to grey. The left-hand rectangle specifies no fill color itself so it is filled with grey. The right-hand rectangle specifies a black fill color that overrides the viewport setting.

5.5.6 Layouts

A viewport can have a *layout* specified via the `layout` argument. A layout in grid is similar to the same concept in traditional graphics (see Section 3.3.2). It divides the viewport region into several columns and rows, where each column can have a different width and each row can have a different height. For several reasons, however, layouts are much more flexible in grid: there are many more coordinate systems for specifying the widths of columns and the heights of rows (see Section 5.3); viewports can occupy overlapping areas within the layout; and each viewport within the viewport tree can have a layout (layouts can be nested). There is also a `just` argument to justify the layout within a viewport when the layout does not occupy the entire viewport region.

Layouts provide a convenient way to position viewports using the standard set of coordinate systems, and provide an extra coordinate system, `"null"`, which is specific to layouts.

The basic idea is that a viewport can be created with a layout and then subsequent viewports can be positioned relative to that layout. In simple cases, this can be just a convenient way to position viewports in a regular grid, but in more complex cases, layouts are the only way to apportion regions. There are very many ways that layouts can be used in grid; the following

sections attempt to provide a glimpse of the possibilities by demonstrating a series of example uses.

A grid layout is created using the function `grid.layout()` (*not* the traditional function `layout()`).

A simple layout

The following code produces a simple layout with three columns and three rows, where the central cell (row two, column two) is forced to always be square (using the `respect` argument).

```
> vplay <- grid.layout(3, 3,
                       respect=rbind(c(0, 0, 0),
                                     c(0, 1, 0),
                                     c(0, 0, 0)))
```

The next piece of code uses this layout in a viewport. Any subsequent viewports may make use of the layout, or they can ignore it completely.

```
> pushViewport(viewport(layout=vplay))
```

In the next piece of code, two further viewports are pushed within the viewport with the layout. The `layout.pos.col` and `layout.pos.row` arguments are used to specify which cells within the layout each viewport should occupy. The first viewport occupies all of column two and the second viewport occupies all of row 2. This demonstrates that viewports can occupy overlapping regions within a layout. A rectangle has been drawn within each viewport to show the region that the viewport occupies (see Figure 5.16).

```
> pushViewport(viewport(layout.pos.col=2, name="col2"))
> upViewport()
> pushViewport(viewport(layout.pos.row=2, name="row2"))
```

A layout with units

This section describes a layout that makes use of grid units. In the context of specifying the widths of columns and the heights of rows for a layout, there is an additional unit available, the `"null"` unit. All other units (`"cm"`, `"npc"`, etc.) are allocated first within a layout, then the `"null"` units are used to divide the remaining space proportionally (see Section 3.3.2). The following

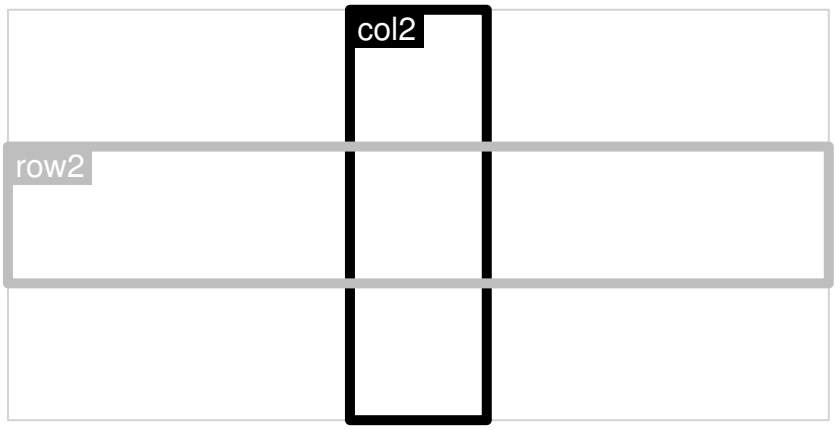

Figure 5.16
Layouts and viewports. Two viewports occupying overlapping regions within a layout. Each viewport is represented by a rectangle with the viewport name at the top-left corner. The layout has three columns and three rows with one viewport occupying all of row 2 and the other viewport occupying all of column 2.

code creates a layout with three columns and three rows. The left column is one inch wide and the top row is three lines of text high. The remainder of the current region is divided into two rows of equal height and two columns with the right column twice as wide as the left column (see Figure 5.17).

```
> unitlay <-
    grid.layout(3, 3,
                widths=unit(c(1, 1, 2),
                            c("inches", "null", "null")),
                heights=unit(c(3, 1, 1),
                             c("lines", "null", "null")))
```

With the use of `"strwidth"` and `"grobwidth"` units it is possible to produce columns that are just wide enough to fit graphical output that will be drawn in the column (and similarly for row heights — see Section 6.4).

A nested layout

This section demonstrates the nesting of layouts. The following code defines a function that includes a trivial use of a layout consisting of two equal-width columns to produce grid output.

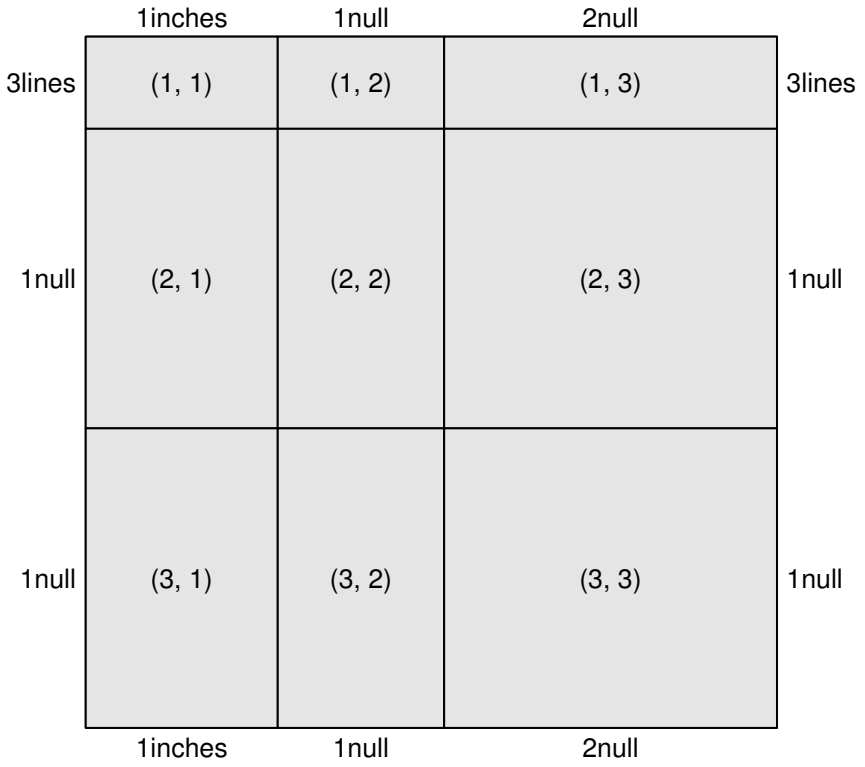

Figure 5.17
Layouts and units. A grid layout using a variety of coordinate systems to specify the widths of columns and the heights of rows.

```
> gridfun <- function() {
    pushViewport(viewport(layout=grid.layout(1, 2)))
    pushViewport(viewport(layout.pos.col=1))
    grid.rect()
    grid.text("black")
    grid.text("&", x=1)
    popViewport()
    pushViewport(viewport(layout.pos.col=2, clip="on"))
    grid.rect(gp=gpar(fill="black"))
    grid.text("white", gp=gpar(col="white"))
    grid.text("&", x=0, gp=gpar(col="white"))
    popViewport(2)
  }
```

The next piece of code creates a viewport with a layout and places the output from the above function within a particular cell of that layout (see Figure 5.18).

```
> pushViewport(
    viewport(
      layout=grid.layout(5, 5,
                         widths=unit(c(5, 1, 5, 2, 5),
                                     c("mm", "null", "mm",
                                       "null", "mm")),
                         heights=unit(c(5, 1, 5, 2, 5),
                                      c("mm", "null", "mm",
                                        "null", "mm")))))
> pushViewport(viewport(layout.pos.col=2, layout.pos.row=2))
> gridfun()
> popViewport()
> pushViewport(viewport(layout.pos.col=4, layout.pos.row=4))
> gridfun()
> popViewport(2)
```

Although the result of this particular example could be achieved using a single layout, what this shows is that it is possible to take grid code that makes use of a layout (and may have been written by someone else) and embed it within a layout of your own. A more sophisticated example of this involving lattice plots is given in Section 5.8.2.

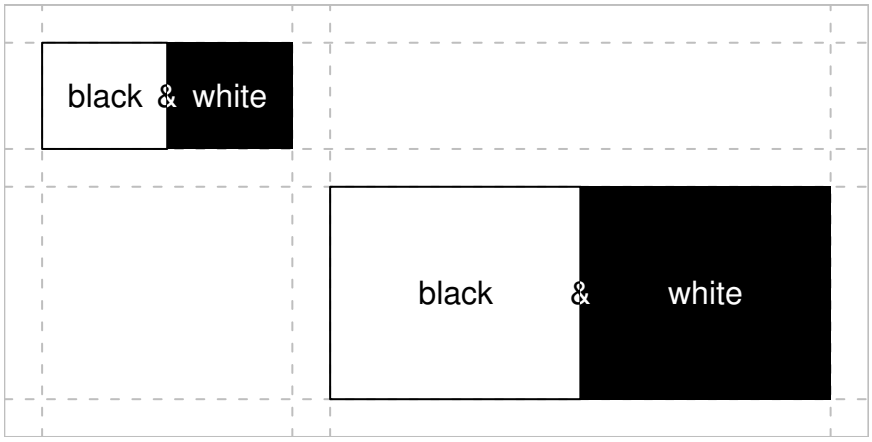

Figure 5.18
Nested layouts. An example of a layout nested within a layout. The black and white squares are drawn within a layout that has two equal-width columns. One instance of the black and white squares has been embedded within cell $(2, 2)$ of a layout consisting of five columns and five rows of varying widths and heights (as indicated by the dashed lines). Another instance has been embedded within cell $(4, 4)$.

5.6 Missing values and non-finite values

Non-finite values are not permitted in the location, size, or scales of a viewport. Viewport scales are checked when a viewport is created, but it is impossible to be certain that locations and sizes are not non-finite when the viewport is created, so this is only checked when the viewport is pushed. Non-finite values result in error messages.

The locations and sizes of graphical objects can be specified as missing values (`NA`, `"NA"`) or non-finite values (`NaN`, `Inf`, `-Inf`). For most graphical primitives, non-finite values for locations or sizes result in the corresponding primitive not being drawn. For the `grid.line.to()` function, a line segment is only drawn if the previous location and the new location are both not non-finite. For `grid.polygon()`, a non-finite value breaks the polygon into two separate polygons. This break happens within the current polygon as specified by the `id` argument. All polygons with the same `id` receive the same `gp` settings. For `grid.arrows()`, an arrow head is only drawn if the first or last line segment is drawn.

Figure 5.19 shows the behavior of these primitives where x- and y-locations

are seven equally-spaced locations around the perimeter of a circle. In the top-left figure, all locations are not non-finite. In each of the other figures, two locations have been made non-finite (indicated in each case by grey text).

5.7 Interactive graphics

The strength of the grid system is in the production of static graphics. There is only very basic support for user interaction, consisting of the `grid.locator()` function. This function returns the location of a single mouse click relative to the current viewport. The result is a list containing an `x` and a `y` unit. The `unit` argument can be used to specify the coordinate system to be used for the result.

From R version 2.1.0, the `getGraphicsEvent()` function provides additional capability (on Windows) to respond to mouse movements, mouse ups, and key strokes. However, with this function, mouse activity is only reported relative to the native coordinate system of the device.

5.8 Customizing lattice plots

This section provides some demonstrations of the basic grid functions within the context of a complete lattice plot.

The lattice package described in Chapter 4 produces complete and very sophisticated plots using grid. It makes use of a sometimes large number of viewports to arrange the graphical output. A page of lattice output contains a top-level viewport with a quite complex layout that provides space for all of the panels and strips and margins used in the plot. Viewports are created for each panel and for each strip (among other things), and the plot is constructed from a large number of rectangles, lines, text, and data points.

In many cases, it is possible to use lattice without having to know anything about grid. However, a knowledge of grid provides a number of more advanced ways to work with lattice output (see Section 6.7). A simple example is provided by the `panel.width` and `panel.height` arguments to the `print.trellis()` method. These provide an alternative to the `aspect` argument for controlling the size of panels within a lattice plot using grid units.

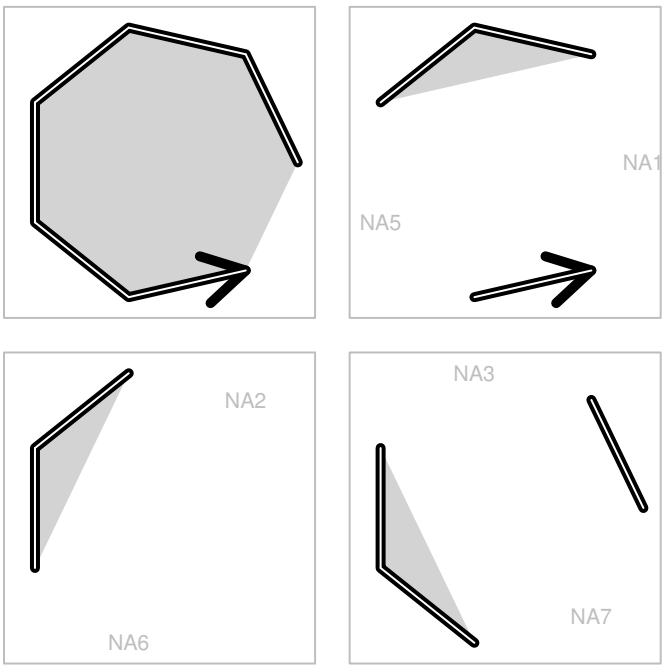

Figure 5.19
Non-finite values for line-tos, polygons, and arrows. The effect of non-finite values for `grid.line.to()`, `grid.polygon()`, and `grid.arrows`. In each panel, a single grey polygon, a single arrow (at the end of a thick black line), and a series of thin white line-tos are drawn through the same set of seven points. In some cases, certain locations have been set to `NA` (indicated by grey text), which causes the polygon to become cropped, creates gaps in the lines, and can cause the arrow head to disappear. In the bottom-left panel, the seventh location is not `NA`, but it produces no output.

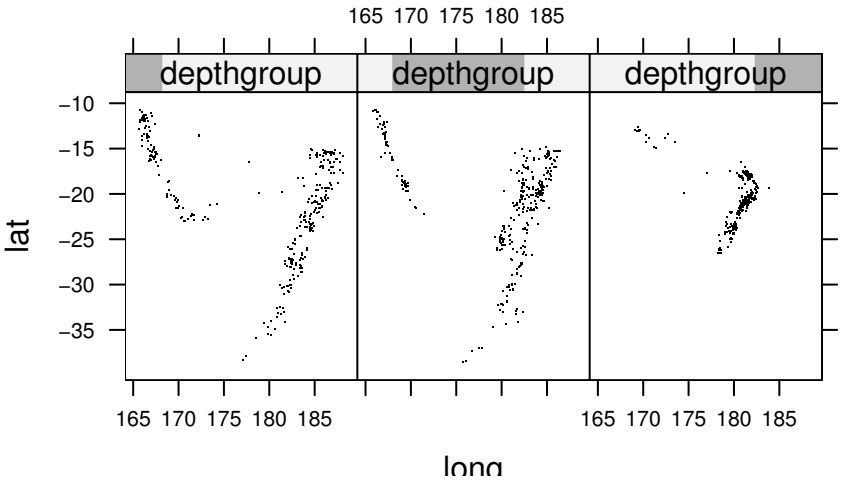

Figure 5.20
Controlling the size of lattice panels using grid units. Each panel is exactly 1.21 inches wide and 1.5 inches high.

The following code produces a multipanel lattice plot of the `quakes` data set (see page 126) where the size of each panel is fixed at 1.21 inches wide and 1.5 inches high (see Figure 5.20).*

```
> temp <- xyplot(lat ~ long | depthgroup,
                 data=quakes, pch=".",
                 layout=c(3, 1))
> print(temp,
        panel.width=list(1.21, "inches"),
        panel.height=list(1.5, "inches"))
```

5.8.1 Adding grid output to lattice output

The functions that lattice provides for adding output to panels (`ltext()`, `lpoints()`, etc) are designed to make it easier to port code between R and S-PLUS. However, they are restricted because they only allow output to be located and sized relative to the `"native"` coordinate system. Grid graphical primitives cannot be ported to S-PLUS, but they provide much more control

*These specific sizes were chosen for this particular data set so that one unit of longitude corresponds to the same physical size on the page as one unit of latitude.

over the location and size of additional panel output. Furthermore, it is possible to create and push extra viewports within a panel if desired (although it is very important that they are popped again or lattice will get very confused).

In a similar vein, the facilities provided by the `upViewport()` and `downViewport()` functions in grid allow for more flexible navigation of a lattice plot compared to the `trellis.focus()` function.

The following code provides an example of using low-level grid functions to add output within a lattice panel function. This produces a variation on Figure 4.4 with a dot and a text label added to indicate the location of Auckland, New Zealand relative to the earthquakes (see Figure 5.21).*

```
> xyplot(lat ~ long | depthgroup, data=quakes, pch=".",
         panel=function(...) {
           grid.points(174.75, -36.87, pch=16,
                       size=unit(2, "mm"),
                       default.units="native")
           grid.text("Auckland",
                     unit(174.75, "native") - unit(2, "mm"),
                     unit(-36.87, "native"),
                     just="right")
           panel.xyplot(...)
         })
```

5.8.2 Adding lattice output to grid output

As well as the advantages of using grid functions to add further output to lattice plots, an understanding that lattice output is really grid output makes it possible to embed lattice output within grid output. The following code provides a simple example (see Figure 5.22).

First of all, two viewports are defined. The viewport `tvp` occupies the rightmost 1 inch of the device and will be used to draw a label. The viewport `lvp` occupies the rest of the device and will be used to draw a lattice plot.

```
> lvp <- viewport(x=0,
                  width=unit(1, "npc") - unit(1, "inches"),
                  just="left", name="lvp")
> tvp <- viewport(x=1, width=unit(1, "inches"),
                  just="right", name="tvp")
```

*The data are from the `quakes` data set (see page 126).

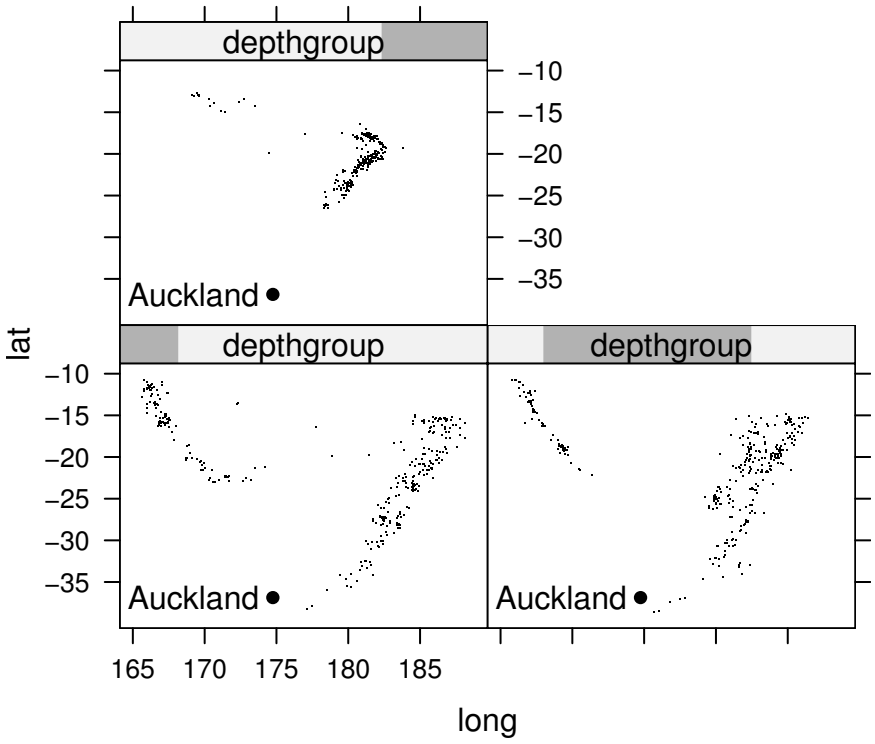

Figure 5.21
Adding grid output to a lattice plot (the lattice plot in Figure 4.4). The grid functions `grid.text()` and `grid.points()` are used within a lattice panel function to highlight the location of Auckland, New Zealand within each panel.

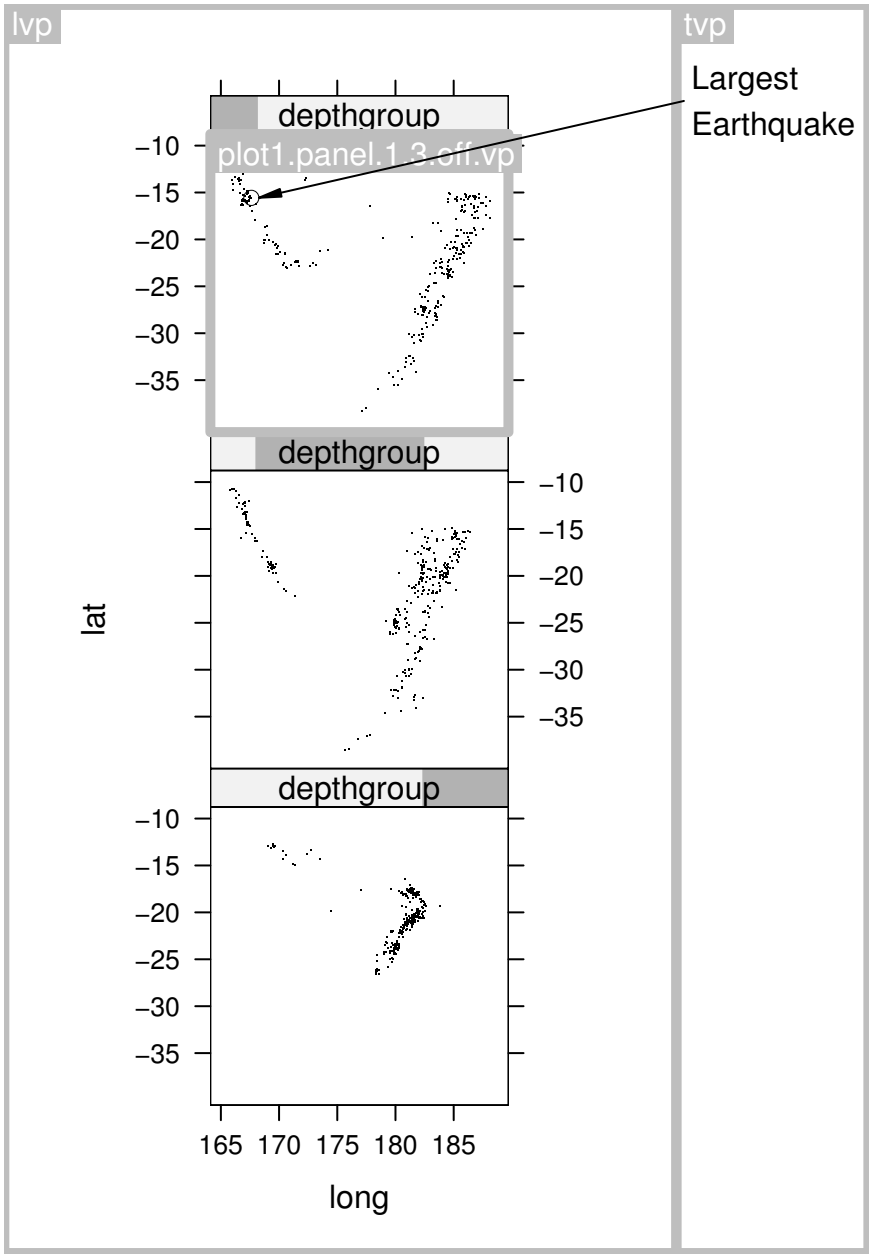

Figure 5.22
Embedding a lattice plot within grid output. The lattice plot is drawn within the viewport `"lvp"` and the text label is drawn within the viewport `"tvp"` (the viewports are indicated by grey rectangles with their names at the top-left corner). An arrow is drawn from viewport `"tvp"` where the text was drawn into viewport `"panel.1.3.off.vp"` — the top panel of the lattice plot.

The next piece of code produces (but does not draw) an object representing a multipanel scatterplot using the `quakes` data (see page 126).

```
> lplot <- xyplot(lat ~ long | depthgroup,
                  data=quakes, pch=".",
                  layout=c(1, 3), aspect=1,
                  index.cond=list(3:1))
```

The following pieces of code do all the drawing. First of all, the `lvp` viewport is pushed and the lattice plot is drawn inside that. The `upViewport()` function is used to navigate back up so that all of the lattice viewports are left intact.

```
> pushViewport(lvp)
> print(lplot, newpage=FALSE, prefix="plot1")
> upViewport()
```

Next, the `tvp` viewport is pushed and a text label is drawn in that.

```
> pushViewport(tvp)
> grid.text("Largest\nEarthquake", x=unit(2, "mm"),
            y=unit(1, "npc") - unit(0.5, "inches"),
            just="left")
```

The last step is to draw an arrow from the label to a data point within the lattice plot. While still in the `tvp` viewport, the `grid.move.to()` function is used to set the current location to a point just to the left of the text label. Next, `seekViewport()` is used to navigate to the top panel within the lattice plot.* Finally, `grid.arrows()` and `lineToGrob()` are used to draw a line from the text to an `(x ,y)` location within the top panel. A circle is also drawn to help identify the location being labelled.

*The name of the viewport representing the top panel in the lattice plot can be obtained using the `trellis.vpname()` function or by just visual inspection of the output of `current.vpTree()` and possibly some trial-and-error.

```
> grid.move.to(unit(1, "mm"),
               unit(1, "npc") - unit(0.5, "inches"))
> seekViewport("plot1.panel.1.3.off.vp")
> grid.arrows(grob=lineToGrob(unit(167.62, "native") +
                              unit(1, "mm"),
                              unit(-15.56, "native")),
              length=unit(3, "mm"), type="closed",
              angle=10, gp=gpar(fill="black"))
> grid.circle(unit(167.62, "native"),
              unit(-15.56, "native"),
              r=unit(1, "mm"),
              gp=gpar(lwd=0.1))
```

The final output is shown in Figure 5.22.

Chapter summary

Grid provides a number of functions for producing basic graphical output such as lines, polygons, rectangles, and text, plus some functions for producing slightly more complex output such as data symbols, arrows, and axes. Graphical output can be located and sized relative to a large number of coordinate systems and there are a number of graphical parameter settings for controlling the appearance of output, such as colors, fonts, and line types.

Viewports can be created to provide contexts for drawing. A viewport defines a rectangular region on the device and all coordinate systems are available within all viewports. Viewports can be arranged using layouts and nested within one another to produce sophisticated arrangements of graphical output.

Because lattice output is grid output, grid functions can be used to add further output to a lattice plot. Grid functions can also be used to control the size and placement of lattice plots.
