Package: quickPlot (via r-universe)

September 7, 2024

Type Package

Title A System of Plotting Optimized for Speed and Modularity

Description A high-level plotting system, compatible with `ggplot2` objects, maps from `sf`, `terra`, `raster`, `sp`. It is built primarily on the 'grid' package. The objective of the package is to provide a plotting system that is built for speed and modularity. This is useful for quick visualizations when testing code and for plotting multiple figures to the same device from independent sources that may be independent of one another (i.e., different function or modules the create the visualizations). The suggested package 'fastshp' can be installed from the repository (<https://PredictiveEcology.r-universe.dev>).

URL https://quickplot.predictiveecology.org, https://github.com/PredictiveEcology/quickPlot

Version 1.0.2.9003

Date 2024-06-08

Depends R (>= 4.1)

Imports data.table (>= 1.10.4), fpCompare, grDevices, grid, methods, stats, terra, utils

Suggests covr, fastshp, ggplot2, knitr, raster, RColorBrewer, rmarkdown, sf, sp, testthat (>= 1.0.2), withr

Additional_repositories https://predictiveecology.r-universe.dev/

Encoding UTF-8

Language en-CA

License GPL-3

VignetteBuilder knitr, rmarkdown

BugReports https://github.com/PredictiveEcology/quickPlot/issues

ByteCompile yes

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.1

Collate 'environment.R' 'plotting-classes.R' 'plotting-colours.R' 'plotting-helpers.R' 'plotting-other.R' 'plotting.R' 'quickPlot-classes.R' 'quickPlot-package.R' 'testing-helpers.R' 'zzz.R'

Config/testthat/edition 3

Repository https://predictiveecology.r-universe.dev

RemoteUrl https://github.com/PredictiveEcology/quickPlot

RemoteRef development

RemoteSha 9bfd2801989e1de9da13a99649759edeed83189a

Contents

`quickPlot-package` *The* `quickPlot` *package*

Description

A high-level plotting system, built using 'grid' graphics, that is optimized for speed and modularity. This has great utility for quick visualizations when testing code, with the key benefit that visualizations are updated independently of one another.

Note

The suggested package **fastshp** can be installed with `install.packages("fastshp", repos = "https://rforge.net", type = "source")`.

Author(s)

Maintainer: Eliot J B McIntire `<eliot.mcintire@canada.ca>` (ORCID)

Authors:

- Alex M Chubaty `<achubaty@for-cast.ca>` (ORCID)

Other contributors:

- His Majesty the King in Right of Canada, as represented by the Minister of Natural Resources Canada [copyright holder]

See Also

Useful links:

- `https://quickplot.predictiveecology.org`
- `https://github.com/PredictiveEcology/quickPlot`
- Report bugs at `https://github.com/PredictiveEcology/quickPlot/issues`

`.hasBbox` *Test whether class has* `bbox` *method*

Description

For internal use only.

Usage

```
.hasBbox(z, objClass, objName, objEnv)
```

Arguments

`z`	Logical, whether this object is a `SpatialObject`
`objClass`	The class of the object
`objName`	The character string name of the object
`objEnv`	The environment where the object can be found

`.parseElems` *Parsing of elements*

Description

This is a generic definition that can be extended according to class. Intended only for development use.

Usage

```
.parseElems(tmp, elems, envir)

## S4 method for signature 'ANY'
.parseElems(tmp, elems, envir)
```

Arguments

`tmp`	A evaluated object
`elems`	A character string to be parsed
`envir`	An environment

Value

An object, parsed from a character string and and environment

Author(s)

Eliot McIntire

`clearPlot` *Clear plotting device*

Description

Under some conditions, a device and its metadata need to be cleared manually. This can be done with either the `new = TRUE` argument within the call to `Plot`. Sometimes, the metadata of a previous plot will prevent correct plotting of a new `Plot` call. Use `clearPlot` to clear the device and all the associated metadata manually.

Usage

```
clearPlot(
  dev = dev.cur(),
  removeData = TRUE,
  force = FALSE,
  verbose = getOption("quickPlot.verbose")
)

## S4 method for signature 'numeric,logical'
clearPlot(
  dev = dev.cur(),
  removeData = TRUE,
  force = FALSE,
  verbose = getOption("quickPlot.verbose")
)

## S4 method for signature 'numeric,missing'
clearPlot(dev, force)

## S4 method for signature 'missing,logical'
clearPlot(removeData, force)

## S4 method for signature 'missing,missing'
clearPlot(dev, removeData, force)
```

Arguments

`dev`	Numeric. Device number to clear.
`removeData`	Logical indicating whether any data that was stored in the `.quickPlotEnv` should also be removed; i.e., not just the plot window wiped.
`force`	Logical or "all". Sometimes the graphics state cannot be fixed by a simple `clearPlot()`. If TRUE, this will close the device and reopen the same device number. If "all", then all `quickPlot` related data from all devices will be cleared, in addition to device closing and reopening.
`verbose`	Numeric or logical. If `TRUE` or `>0`, then messages will be shown. If `FALSE` or `0`, most messages will be suppressed.

Author(s)

Eliot McIntire

See Also

`Plot()`.

Examples

```
if (interactive()) {
  Plot(1:10)
  clearPlot() # clears
}
```

---

coordinates — *Extract coordinates from a variety of spatial objects*

Description

This will extract using `terra::crds`, `sf::st_coordinates` and `raster::coordinates`. Other packages can create methods, as this is generic.

Usage

```
## S4 method for signature 'ANY'
coordinates(obj, ...)
```

Arguments

`obj`	An object from which to extract the coordinates (e.g., `sf`, `sp`)
`...`	Ignored.

Value

A 2 column matrix of coordinates (x and y)

Examples

```
library(terra)
caribou <- terra::vect(x = cbind(x = stats::runif(1e1, -50, 50),
                                 y = stats::runif(1e1, -50, 50)))
coordinates(caribou)
```

dev *Specify where to plot*

Description

Switch to an existing plot device, or if not already open, launch a new graphics device based on operating system used. On Windows and macOS, if `x` is not provided, this will open or switch to the first non-RStudio device, which is much faster than the ‘png’-based RStudio plot device. Currently, this will not open anything new.

Usage

```
dev(x, ..., verbose = getOption("quickPlot.verbose"))
```

Arguments

`x`	The number of a plot device. If missing, will open a new non-RStudio plotting device
`...`	Additional arguments passed to `newPlot()`.
`verbose`	Numeric or logical. If `TRUE` or `>0`, then messages will be shown. If `FALSE` or `0`, most messages will be suppressed.

Details

For example, `dev(6)` switches the active plot device to device 6. If it does not exist, it opens it. If devices 1-5 don’t exist they will be opened too.

Value

Opens a new plot device on the screen. Invisibly returns the device number selected.

Author(s)

Eliot McIntire and Alex Chubaty

Examples

```
## Not run:
  dev(4)

## End(Not run)
```

divergentColors *Divergent colour palette*

Description

Creates a palette for the current session for a divergent-colour graphic with a non-symmetric range. Based on ideas from Maureen Kennedy, Nick Povak, and Alina Cansler.

Usage

```
divergentColors(
  start.color,
  end.color,
  min.value,
  max.value,
  mid.value = 0,
  mid.color = "white"
)

## S4 method for signature 'character,character,numeric,numeric'
divergentColors(
  start.color,
  end.color,
  min.value,
  max.value,
  mid.value = 0,
  mid.color = "white"
)
```

Arguments

`start.color`	Start colour to be passed to `colorRampPalette`.
`end.color`	End colour to be passed to `colorRampPalette`.
`min.value`	Numeric minimum value corresponding to `start.colour`. If attempting to change the colour of a `RasterLayer`, this can be set to `minFn(RasterObject)`.
`max.value`	Numeric maximum value corresponding to `end.colour`. If attempting to change the colour of a `RasterLayer`, this can be set to `maxFn(RasterObject)`.
`mid.value`	Numeric middle value corresponding to `mid.colour`. Default is `0`.
`mid.color`	Middle colour to be passed to `colorRampPalette`. Defaults to `"white"`.

Value

A diverging colour palette.

Author(s)

Eliot McIntire and Alex Chubaty

See Also

`colorRampPalette()`

Examples

```
divergentColors("darkred", "darkblue", -10, 10, 0, "white")
```

`equalExtent` *Assess whether a list of extents are all equal*

Description

Assess whether a list of extents are all equal

Usage

```
equalExtent(extents)

## S4 method for signature 'list'
equalExtent(extents)
```

Arguments

`extents`	list of extents objects

Author(s)

Eliot McIntire

Examples

```
library(terra)

files <- system.file("maps", package = "quickPlot")
files <- dir(files, full.names = TRUE, pattern = "tif")
maps <- lapply(files, function(x) terra::rast(x))
names(maps) <- sapply(basename(files), function(x) {
  strsplit(x, split = "\\.")[[1]][1]
})
extnts <- lapply(maps, terra::ext)
equalExtent(extnts) ## TRUE
```

extent — *Get extent of a variety of spatial objects*

Description

This is a wrapper around `terra::ext`, `sf::st_bbox`, and `raster::extent`.

Usage

```
## S4 method for signature 'ANY'
extent(x, ...)
```

Arguments

`x`	The spatial object from which to extract the extent.
`...`	Not used.

Value

Returns a list of length 4 with elements `xmin`, `xmax`, `ymin`, and `ymax`, in that order.

getColors — *Get and set colours for plotting `Raster*` objects*

Description

Get and set colours for plotting `Raster*` objects

`setColors` works as a replacement method or a normal function call. This function can accept `RColorBrewer` colours by name. See examples.

Usage

```
getColors(object)

setColors(object, ..., n, verbose = getOption("quickPlot.verbose")) <- value

setColors(object, value, n, verbose = getOption("quickPlot.verbose"))
```

Arguments

object	A Raster* object.
...	Additional arguments to colorRampPalette.
n	An optional vector of values specifying the number of levels from which to interpolate the colour palette.
verbose	Numeric or logical. If TRUE or >0, then messages will be shown. If FALSE or 0, most messages will be suppressed.
value	Named list of hex colour codes (e.g., from RColorBrewer::brewer.pal), corresponding to the names of RasterLayers in x.

Value

Returns a named list of colours.

Returns a Raster with the colortable slot set to values.

Author(s)

Alex Chubaty

See Also

setColors<-(), brewer.pal(), RColorBrewer::ColorBrewer

brewer.pal(), RColorBrewer::ColorBrewer, colorRampPalette().

Examples

```
library(terra)

ras <- rast(matrix(c(0, 0, 1, 2), ncol = 2, nrow = 2))

getColors(ras) ## none

# Use replacement method
setColors(ras, n = 3) <- c("red", "blue", "green")
getColors(ras)

clearPlot()
Plot(ras)

# Use function method
ras <- setColors(ras, n = 3, c("red", "blue", "yellow"))
getColors(ras)

clearPlot()
Plot(ras)

# Using the wrong number of colors, e.g., here 2 provided,
# for a raster with 3 values... causes interpolation, which may be surprising
ras <- setColors(ras, c("red", "blue"))
```

```
clearPlot()
Plot(ras)

# Real number rasters - interpolation is used
ras <- rast(matrix(runif(9), ncol = 3, nrow = 3)) |>
  setColors(c("red", "yellow")) # interpolates when real numbers, gives warning

clearPlot()
Plot(ras)

# Factor rasters, can be contiguous (numerically) or not, in this case not:
ras <- rast(matrix(sample(c(1, 3, 6), size = 9, replace = TRUE), ncol = 3, nrow = 3))
levels(ras) <- data.frame(ID = c(1, 3, 6), Names = c("red", "purple", "yellow"))
ras <- setColors(ras, n = 3, c("red", "purple", "yellow"))
getColors(ras)

clearPlot()
Plot(ras)

# if a factor raster, and not enough labels are provided, then a warning
#   will be given, and colors will be interpolated
#   The level called purple is not purple, but interpolated betwen red and yellow
suppressWarnings({
  ras <- setColors(ras, c("red", "yellow"))
  clearPlot()
  Plot(ras)
})

# use RColorBrewer colors
setColors(ras) <- "Reds"
clearPlot()
Plot(ras)
```

`gpar` *Importing some grid functions*

Description

Currently only the `gpar` function is imported. This is a convenience so that users can change `Plot` arguments without having to load the entire grid package.

Usage

```
gpar(...)
```

Arguments

`...`	Any number of named arguments.

See Also

`grid::gpar()`

`isRstudioServer` *Determine if current session is RStudio Server*

Description

Determine if current session is RStudio Server

Usage

```
isRstudioServer()
```

Examples

```
isRstudioServer() # returns FALSE or TRUE
```

`layerNames` *Extract the layer names of Spatial Objects*

Description

There are already methods for `Raster*` objects. This adds methods for `SpatialPoints*`, `SpatialLines*`, and `SpatialPolygons*`, returning an empty character vector of length 1. This function was created to give consistent, meaningful results for all classes of objects plotted by `Plot`.

Usage

```
layerNames(object)

## S4 method for signature 'ANY'
layerNames(object)
```

Arguments

`object`	A `Raster*`, `SpatialPoints*`, `SpatialLines*`, or `SpatialPolygons*` object; or list of these.

Author(s)

Eliot McIntire

Examples

```
library(terra)

## RasterLayer objects
files <- system.file("maps", package = "quickPlot")
files <- dir(files, full.names = TRUE, pattern = "tif")
maps <- lapply(files, function(x) terra::rast(x))
names(maps) <- sapply(basename(files), function(x) {
  strsplit(x, split = "\\.")[[1]][1]
})
layerNames(maps)

## SpatVector objects
caribou <- terra::vect(cbind(x = stats::runif(1e2, -50, 50),
                             y = stats::runif(1e2, -50, 50)))
layerNames(caribou)
```

makeLines — *Make* `SpatialLines` *object from two* `SpatialPoints` *objects*

Description

The primary conceived usage of this is to draw arrows following the trajectories of agents.

Usage

```
makeLines(from, to)
```

Arguments

`from`	Starting spatial coordinates (`SpatialPointsDataFrame`).
`to`	Ending spatial coordinates (`SpatialPointsDataFrame`).

Value

A `SpatialLines` object. When this object is used within a `Plot` call and the `length` argument is specified, then arrow heads will be drawn. See examples.

Author(s)

Eliot McIntire

Examples

```
library(terra)
# Make 2 objects
caribou1 <- terra::vect(cbind(x = stats::runif(10, -50, 50),
                              y = stats::runif(10, -50, 50)))
caribou2 <- terra::vect(cbind(x = stats::runif(10, -50, 50),
                              y = stats::runif(10, -50, 50)))

caribouTraj <- makeLines(caribou1, caribou2)

if (interactive())
  Plot(caribouTraj, length = 0.1) # shows arrows

# or  to a previous Plot
files <- dir(system.file("maps", package = "quickPlot"), full.names = TRUE, pattern = "tif")
maps <- lapply(files, terra::rast)
names(maps) <- lapply(maps, names)

caribouTraj <- makeLines(caribou1, caribou2)

if (interactive()) {
  clearPlot()
  Plot(maps$DEM)
  Plot(caribouTraj, addTo = "maps$DEM", length = 0.1)
}
```

`newPlot` *Open a new plotting window*

Description

Open a new plotting window

Usage

```
newPlot(noRStudioGD = TRUE, ..., verbose = getOption("quickPlot.verbose"))

dev.useRSGD(useRSGD = FALSE)
```

Arguments

`noRStudioGD`	Logical Passed to `dev.new`. Default is TRUE to avoid using RStudio graphics device, which is slow.
`...`	Additional arguments.
`verbose`	Numeric or logical. If `TRUE` or `>0`, then messages will be shown. If `FALSE` or `0`, most messages will be suppressed.

useRSGD	Logical indicating whether the default device should be the RStudio graphic device, or the platform default (quartz on macOS; windows on Windows; x11 on others, e.g., Linux).

Note

dev.new() is supposed to be the correct way to open a new window in a platform-generic way; however, does not work in RStudio (SpaDES#116). Use dev.useRSGD(FALSE) to avoid RStudio for the remainder of this session, and dev.useRSGD(TRUE) to use the RStudio graphics device. (This sets the default device via the device option.)

Author(s)

Eliot McIntire and Alex Chubaty

See Also

dev().

Examples

```
## Not run:
  ## set option to avoid using Rstudio graphics device
  dev.useRSGD(FALSE)

  ## open new plotting window
  newPlot()

## End(Not run)
```

---

numLayers *Find the number of layers in an object*

Description

A unified function for raster::nlayers, terra::nlyrs, or lists of these. Default function returns 1L for all other classes.

Usage

```
numLayers(x)
```

Arguments

x	An object or list of objects.

Value

The number of layers in the object.

Author(s)

Eliot McIntire

Examples

```
library(terra)

files <- system.file("maps", package = "quickPlot")
files <- dir(files, full.names = TRUE, pattern = "tif")
maps <- lapply(files, function(x) rast(x))
names(maps) <- sapply(basename(files), function(x) {
  strsplit(x, split = "\\.")[[1]][1]
})
stck <- rast(maps)

numLayers(maps)
numLayers(stck)
```

`Plot` `Plot`*: Fast, optimally arranged, multi-panel plotting*

Description

This can take objects of type `Raster*`, `SpatialPoints*`, `SpatialPolygons*`, and any combination of those. These can be provided as individual objects, or a named list. If a named list, the names either represent a different original object in the calling environment and that will be used, or if the names don't exist in the calling environment, then they will be copied to `.quickPlotEnv` for reuse later. It can also handle `ggplot2` objects or `base::histogram` objects created via call to `exHist <- hist(1:10, plot = FALSE)`. It can also take arguments as if it were a call to `plot`. In this latter case, the user should be explicit about naming the plot area using `addTo`. Customization of the `ggplot2` elements can be done as a normal `ggplot2` plot, then added with `Plot(ggplotObject)`.

Usage

```
Plot(
  ...,
  new = FALSE,
  addTo = NULL,
  gp = gpar(),
  gpText = gpar(),
  gpAxis = gpar(),
  axes = FALSE,
  speedup = 1,
  size = 5,
  cols = NULL,
  col = NULL,
  zoomExtent = NULL,
  visualSqueeze = NULL,
  legend = TRUE,
  legendRange = NULL,
  legendText = NULL,
  pch = 19,
  title = NULL,
  na.color = "#FFFFFF00",
  zero.color = NULL,
  length = NULL,
  arr = NULL,
  plotFn = "plot",
  verbose = getOption("quickPlot.verbose")
)

## S4 method for signature 'ANY'
Plot(
  ...,
  new = FALSE,
  addTo = NULL,
  gp = gpar(),
  gpText = gpar(),
  gpAxis = gpar(),
  axes = FALSE,
  speedup = 1,
  size = 5,
  cols = NULL,
  col = NULL,
  zoomExtent = NULL,
  visualSqueeze = NULL,
  legend = TRUE,
  legendRange = NULL,
  legendText = NULL,
  pch = 19,
  title = NULL,
  na.color = "#FFFFFF00",
  zero.color = NULL,
  length = NULL,
  arr = NULL,
  plotFn = "plot",
  verbose = getOption("quickPlot.verbose")
)

rePlot(
  toDev = dev.cur(),
  fromDev = dev.cur(),
  clearFirst = TRUE,
  ...,
  verbose = getOption("quickPlot.verbose")
```

```
)
```

Arguments

`...`	A combination of `spatialObjects` or non-spatial objects. For many object classes, there are specific `Plot` methods. Where there are no specific ones, the base plotting will be used internally. This means that for objects with no specific `Plot` methods, many arguments, such as `addTo`, will not work. See details.
`new`	Logical. If `TRUE`, then the previous named plot area is wiped and a new one made; if `FALSE`, then the `...` plots will be added to the current device, adding or rearranging the plot layout as necessary. Default is `FALSE`. This currently works best if there is only one object being plotted in a given `Plot` call. However, it is possible to pass a list of logicals to this, matching the length of the `...` objects. Use `clearPlot` to clear the whole plotting device. NOTE if `TRUE`: *Everything that was there, including the legend and the end points of the colour palette, will be removed and re-initiated.*
`addTo`	Character vector, with same length as `...`. This is for overplotting, when the overplot is not to occur on the plot with the same name, such as plotting a `SpatialPoints*` object on a `RasterLayer`.
`gp`	A `gpar` object, created by `gpar()`, to change plotting parameters (see **grid** package).
`gpText`	A `gpar` object for the title text. Default `gpar(col = "black")`.
`gpAxis`	A `gpar` object for the axes. Default `gpar(col = "black")`.
`axes`	Logical or `"L"`, representing the left and bottom axes, over all plots.
`speedup`	Numeric. The factor by which the number of pixels is divided by to plot rasters. See Details.
`size`	Numeric. The size, in points, for `SpatialPoints` symbols, if using a scalable symbol.
`cols`	(also `col`) Character vector or list of character vectors of colours. See details.
`col`	(also `cols`) Alternative to `cols` to be consistent with `plot`. `cols` takes precedence, if both are provided.
`zoomExtent`	An `Extent` object. Supplying a single extent that is smaller than the rasters will call a crop statement before plotting. Defaults to `NULL`. This occurs after any downsampling of rasters, so it may produce very pixelated maps.
`visualSqueeze`	Numeric. The proportion of the white space to be used for plots. Default is 0.75.
`legend`	Logical indicating whether a legend should be drawn. Default is `TRUE`.
`legendRange`	Numeric vector giving values that, representing the lower and upper bounds of a legend (i.e., `1:10` or `c(1,10)` will give same result) that will override the data bounds contained within the `grobToPlot`.
`legendText`	Character vector of legend value labels. Defaults to `NULL`, which results in a pretty numeric representation. If `Raster*` has a Raster Attribute Table (`rat`; see **raster** package), this will be used by default. Currently, only a single vector is accepted. The length of this must match the length of the legend, so this is mostly useful for discrete-valued rasters.

`pch`	see `?par`.
`title`	Logical or character string. If logical, it indicates whether to print the object name as the title above the plot. If a character string, it will print this above the plot. NOTE: the object name is used with `addTo`, not the title. Default NULL, which means print the object name as title, if no other already exists on the plot, in which case, keep the previous title.
`na.color`	Character string indicating the colour for `NA` values. Default transparent.
`zero.color`	Character string indicating the colour for zero values, when zero is the minimum value, otherwise, zero is treated as any other colour. Default transparent.
`length`	Numeric. Optional length, in inches, of the arrow head.
`arr`	A vector of length 2 indicating a desired arrangement of plot areas indicating number of rows, number of columns. Default NULL, meaning let Plot function do it automatically.
`plotFn`	An optional function name to do the plotting internally, e.g., "barplot" to get a barplot() call. Default "plot".
`verbose`	Numeric or logical. If `TRUE` or `>0`, then messages will be shown. If `FALSE` or `0`, most messages will be suppressed.
`toDev`	Numeric. Which device should the new replot be plotted to. Default is current device.
`fromDev`	Numeric. Which device should the replot information be taken from. Default is current device
`clearFirst`	Logical. Should `clearPlot` be run before replotting. Default TRUE.

Details

NOTE: Plot uses the **grid** package; therefore, it is NOT compatible with base R graphics. Also, because it does not by default wipe the plotting device before plotting, a call to `clearPlot()` is helpful to resolve many errors. Careful use of the other device tools, such as `dev.off()` and `dev.list()` might also clear problems that may arise.

If `new = TRUE`, a new plot will be generated, but only in the figure region that has the same name as the object being plotted. This is different than calling `clearPlot(); Plot(Object)`, i.e,. directly before creating a new Plot. `clearPlot()` will clear the entire plotting device. When `new = FALSE`, any plot that already exists will be overplotted, while plots that have not already been plotted will be added. This function rearranges the plotting device to maximize the size of all the plots, minimizing white space. If using the RStudio IDE, it is recommended to make and use a new device with `dev()`, because the built in device is not made for rapid redrawing. The function is based on the grid package.

Each panel in the multipanel plot must have a name. This name is used to overplot, rearrange the plots, or overlay using `addTo` when necessary. If the `...` are named `spatialObjects`, then `Plot` will use these names. However, this name will not persist when there is a future call to `Plot` that forces a rearrangement of the plots. A more stable way is to use the object names directly, and any layer names (in the case of `RasterLayer` or `RasterStack` objects). If plotting a `RasterLayer` and the layer name is "layer" or the same as the object name, then, for simplicity, only the object name will be used. In other words, only enough information is used to uniquely identify the plot.

For modularity, `Plot` must have access to the original objects that were plotted. These objects will be used if a subsequent Plot event forces a rearrangement of the plot device. Rather than saving all the plot information (including the data) at each `Plot` call (this is generally too much data to constantly make copies), the function saves a pointer to the original R object. If the plot needs to be rearranged because of a future addition, then `Plot` will search for that original object that created the first plots, and replot them. This has several consequences. First, that object must still exist and in the same environment. Second, if that object has changed between the first time it is plot and any subsequent time it is replotted (via a forced rearrangement), then it will take the object *as it exists*, not as it existed. Third, if passing a named list of objects, Plot will either create a link to objects with those names in the calling environment (e.g., `.GlobalEnv`) or, if they do not exist, then `Plot` will make a copy in the hidden `.quickPlotEnv` for later reuse.

`cols` is a vector of colours that can be understood directly, or by `colorRampPalette()`, such as `c("orange", "blue")`, will give a colour range from orange to blue, interpolated. If a list, it will be used, in order, for each item to be plotted. It will be recycled if it is shorter than the objects to be plotted. Note that when this approach to setting colours is used, any overplotting will revert to the `colortable` slot of the object, or the default for rasters, which is `terrain.color()`

`cols` can also accept `RColorBrewer` colours by keyword if it is character vector of length 1. i.e., this cannot be used to set many objects by keyword in the same Plot call. Default `terrain.color()`. See Details.

Some colouring will be automatic. If the object being plotted is a Raster, then this will take the `colorTable` slot (can be changed via `setColors()` or other ways). If this is a `SpatialPointsDataFrame`, this function will use a column called `colors` and apply these to the symbols.

For `SpatialPolygons`, `cols` can accept `RColorBrewer` colours by keyword as a character vector of length 1. For more control, pass a vector of colours to `cols` or to `gp = gpar(fill = vectorOfColours)`. In this second approach, the length of the `vectorOfColours` can be either less then or equal to the number of polygons in the `SpatialPolygons` object – each polygon within a `Polygons` object will share the same colour – or it can be greater than this number to give a different colour to each `Polygon` (of which there can be MANY more than `Polygons`. `Plot` will recycle these colours if there are not enough. The order provided will be the order assigned to each `Polygons` or `Polygon` object.

Silently, one hidden object is made, `.quickPlot` in the `.quickPlotEnv` environment, which is used for arranging plots in the device window, and identifying the objects to be replotted if rearranging is required, subsequent to a `new = FALSE` additional plot.

This function is optimized to allow modular Plotting. This means that several behaviours will appear unusual. For instance, if a first call to `Plot` is made, the legend will reflect the current colour scheme. If a second or subsequent call to `Plot` is made with the same object but with different colours (e.g., with `cols`), the legend will not update. This behaviour is made with the decision that the original layer takes precedence and all subsequent plots to that same frame are over-plots only.

`speedup` is not a precise number because it is faster to plot an non-resampled raster if the new resampling is close to the original number of pixels. At the moment, for rasters, this is set to 1/3 of the original pixels. In other words, `speedup` will not do anything if the factor for speeding up is not high enough (i.e., >3). If no sub-sampling is desired, use a speedup value less than 0.1.

These `gp*` parameters will specify plot parameters that are available with `gpar()`. `gp` will adjust plot parameters, `gpText` will adjust title and legend text, `gpAxis` will adjust the axes. `size` adjusts point size in a `SpatialPoints` object. These will persist with the original `Plot` call for each individual object. Multiple entries can be used, but they must be named list elements and they must match the ... items to plot. This is true for a `RasterStack` also, i.e., the list of named elements must be the same length as the number of layers being plotted. The naming convention used is: `RasterStackName$layerName`, i.e, `landscape$DEM`.

Value

Invisibly returns the `.quickPlot` class object. If this is assigned to an object, say `obj`, then this can be plotted again with `Plot(obj)`. This object is also stored in the locked `.quickPlotEnv`, so can simply be replotted with `rePlot()` or on a new device with `rePlot(n)`, where `n` is the new device number.

Author(s)

Eliot McIntire

See Also

`clearPlot()`, `rePlot()`, `gpar()`, `raster::raster()`, `par()`, `sp::SpatialPolygons()`, `grid.polyline()`, `ggplot2::ggplot()`, `dev()`, `terra::vect()`, `terra::rast()`

Examples

```
if (requireNamespace("RColorBrewer") && interactive()) {
  library(terra)

 files <- dir(system.file("maps", package = "quickPlot"), full.names = TRUE, pattern = "tif")
  maps <- lapply(files, rast)
  names(maps) <- lapply(maps, names)

  # put layers into a single stack for convenience
  landscape <- rast(maps)

  # can change color palette
  setColors(landscape, n = 50) <-
    list(DEM = topo.colors(50),
         forestCover = RColorBrewer::brewer.pal(9, "Set1"),
         forestAge = RColorBrewer::brewer.pal("Blues", n = 8),
         habitatQuality = RColorBrewer::brewer.pal(9, "Spectral"),
         percentPine = RColorBrewer::brewer.pal("GnBu", n = 8))

  # Make a new raster derived from a previous one; must give it a unique name
  habitatQuality2 <- landscape$habitatQuality ^ 0.3
  names(habitatQuality2) <- "habitatQuality2"

  # make a SpatialPoints object
  caribou <- terra::vect(cbind(x = stats::runif(1e2, -50, 50),
                               y = stats::runif(1e2, -50, 50)))

  # use factor raster to give legends as character strings
  ras <- rast(ext(0, 3, 0, 4), vals = sample(1:4, size = 12, replace = TRUE), res = 1)

  # needs to have a data.frame with ID as first column - see ?raster::ratify
```

```
levels(ras) <- data.frame(ID = 1:4, Name = paste0("Level", 1:4))
Plot(ras, new = TRUE)

# Arbitrary values for factors, including zero and not all levels represented in raster
levs <- c(0:5, 7:12)
ras <- rast(ext(0, 3, 0, 2), vals = c(1, 1, 3, 5, 8, 9), res = 1)
levels(ras) <- data.frame(ID = levs, Name = LETTERS[c(1:3, 8:16)])
Plot(ras, new = TRUE)

# Arbitrary values for factors, including zero and not all levels represented in raster
levs <- c(0:5, 7:23)
ras <- rast(ext(0, 3, 0, 2), vals = c(1, 1, 3, 5, 8, 9), res = 1)
levels(ras) <- data.frame(ID = levs, Name = LETTERS[1:23])
Plot(ras, new = TRUE)

# SpatialPolygons
sr1 <- cbind(object = 1, cbind(c(2, 4, 4, 1, 2), c(2, 3, 5, 4, 2)) * 20 - 50)
sr2 <- cbind(object = 2, cbind(c(5, 4, 2, 5), c(2, 3, 2, 2)) * 20 - 50)
spP <- vect(rbind(sr1, sr2))

clearPlot()
Plot(ras)

clearPlot()
Plot(landscape)

# Can overplot, using addTo
Plot(caribou, addTo = "landscape$forestAge", size = 4, axes = FALSE)

# can add a plot to the plotting window
Plot(caribou, new = FALSE)

# Can add two maps with same name, if one is in a stack; they are given
#  unique names based on object name
Plot(landscape, caribou, maps$DEM)

# can mix SpatRaster, SpatVector, RasterStack, RasterLayer, Spatial*
Plot(landscape, habitatQuality2, caribou)

# can mix stacks, rasters, SpatialPoint*, and SpatialPolygons*
Plot(landscape, caribou)
Plot(habitatQuality2, new = FALSE)
Plot(spP)
Plot(spP, addTo = "landscape$forestCover", gp = gpar(lwd = 2))

# provide manual arrangement, NumRow, NumCol
Plot(landscape, spP, arr = c(2, 5), new = TRUE)

# example base plot
clearPlot()
Plot(1:10, 1:10, addTo = "test", new = TRUE) # if there is no "test" then it will make it
Plot(4, 5, pch = 22, col = "blue", addTo = "test")
obj1 <- rnorm(1e2)
```

```
  Plot(obj1, axes = "L")

  # Can plot named lists of objects (but not base objects yet)
  ras1 <- ras2 <- ras
  a <- list()
  for (i in 1:2) {
    a[[paste0("ras", i)]] <- get(paste0("ras", i))
  }
  a$spP <- spP
  clearPlot()
  Plot(a)

  # Now all together
  Plot(obj1, title = "scatterplot")
  Plot(landscape)

  # do with sf --> these will add to previous plots
  if (requireNamespace("sf", quietly = TRUE)) {
    caribouSF <- sf::st_as_sf(caribou)
    Plot(caribouSF, axes = "L")
    Plot(caribouSF, addTo = "landscape$percentPine") # overlay on a specific plot
  }

  # clean up
  clearPlot()

}
```

quickPlotClasses   `quickPlot` *classes*

Description

`quickPlot` uses S4 classes. "Dot" classes are not exported and are therefore intended for internal use only.

Plotting classes - used within `Plot`

New classes

`.arrangement()`	The layout or "arrangement" of plot objects
`.quickPlot()`	Main class for `Plot` - contains `.quickGrob` and `.arrangement` objects
`.quickPlotGrob()`	GRaphical OBject used by `quickPlot` - smallest unit

Unions of existing classes:

`.quickPlottables`	The union of all object classes Plot can accept
`.quickPlotObjects`	The union of `spatialObjects` and several others

Author(s)

Eliot McIntire and Alex Chubaty

See Also

`Plot()`

`sample-maps` *Dummy maps included with* `quickPlot`

Description

All maps included here are randomly generated maps created using `SpaDES.tools::gaussMap()`. These are located within the `maps` folder of the package, and are used in the vignettes. Use `system.file("maps", package = "quickPlot")` to locate the '`maps/`' directory on your system.

Format

raster

Details

- `DEM.tif`: converted to a a small number of discrete levels (in 100m hypothetical units).
- `habitatQuality.tif`: made to look like a continuous habitat surface, rescaled to 0 to 1.
- `forestAge.tif`: rescaled to possible forest ages in a boreal forest setting.
- `forestCover.tif`: rescaled to possible forest cover in a boreal forest setting.
- `percentPine.tif`: rescaled to percentages.

sp2sl — *Convert pairs of coordinates to* `SpatialLines`

Description

This will convert 2 objects whose coordinates can be extracted with `coordinates` (e.g., `sp::SpatialPoints*`) to a single `SpatialLines` object. The first object is treated as the "to" (destination), and the second object the "from" (source). This can be used to represent directional `SpatialLines`, especially with with arrow heads, as in `Plot(sl, length = 0.1)`.

Usage

```
sp2sl(sp1, from)
```

Arguments

`sp1`	a `SpatialPoints*` object
`from`	a `SpatialPoints*` object. Optional. If not provided, then the function will attempt to find the "previous" coordinates as columns (`prevX`, `prevY`) in the `sp1` object.

Examples

```
caribou <- terra::vect(x = cbind(x = stats::runif(1e1, -50, 50),
                                        y = stats::runif(1e1, -50, 50)))
caribouFrom <- terra::vect(x = cbind(x = stats::runif(1e1, -50, 50),
                                        y = stats::runif(1e1, -50, 50)))
caribouLines <- sp2sl(caribou, caribouFrom)
if (interactive()) {
  clearPlot()
  Plot(caribouLines, length = 0.1)
}
```

thin — *Thin a polygon using* `fastshp::thin`

Description

For visualizing, it is sometimes useful to remove points in `Spatial*` objects. This will change the geometry, so it is not recommended for computation. This is similar to `sf::st_simplify`, but faster (see examples) for large shapefiles, particularly if `returnDataFrame` is `TRUE`. `thin` *will not attempt to preserve topology.* It is strictly for making smaller polygons for the (likely) purpose of visualizing more quickly.

Usage

```
thin(
  x,
  tolerance,
  returnDataFrame,
  minCoordsToThin,
  ...,
  verbose = getOption("quickPlot.verbose")
)

thnSpatialPolygons(
  x,
  tolerance = NULL,
  returnDataFrame = FALSE,
  minCoordsToThin = 1e+05,
  maxNumPolygons = getOption("quickPlot.maxNumPolygons", 3000),
  ...,
  verbose = getOption("quickPlot.verbose")
)

## Default S3 method:
thin(
  x,
  tolerance,
  returnDataFrame,
  minCoordsToThin,
  maxNumPolygons,
  ...,
  verbose = getOption("quickPlot.verbose")
)
```

Arguments

`x`	A `Spatial*` object
`tolerance`	Maximum allowable distance for a point to be removed.
`returnDataFrame`	If `TRUE`, this will return a list of 3 elements, `xyOrd`, `hole`, and `idLength`. If `FALSE` (default), it will return a `SpatialPolygons` object.
`minCoordsToThin`	If the number of coordinates is smaller than this number, then thin will just pass through, though it will take the time required to calculate how many points there are (which is not `NROW(coordinates(x))` for a `SpatialPolygon`)
`...`	Passed to methods (e.g., `maxNumPolygons`)
`verbose`	Numeric or logical. If `TRUE` or `>0`, then messages will be shown. If `FALSE` or `0`, most messages will be suppressed.
`maxNumPolygons`	For speed, `thin` can also simply remove some of the polygons. This is likely only a reasonable thing to do if there are a lot of polygons being plotted in a small

space. Current default is taken from options('quickPlot.maxNumPolygons'), with a message.

whereInStack *Find the environment in the call stack that contains an object by name*

Description

This is similar to pryr::where, except instead of working up the search() path of packages, it searches up the call stack for an object. Ostensibly similar to base::dynGet, but it will only return the environment, not the object itself and it will try to extract just the object name from name, even if supplied with a more complicated name (e.g., if obj$firstElement@slot1$size is supplied, the function will only search for obj). The function is fairly fast. This function is an important component to the Plot function.

Usage

```
whereInStack(name, whFrame = -1)
```

Arguments

name	An object name to find in the call stack
whFrame	A numeric indicating which sys.frame (by negative number) to start searching in.

Details

The difference between this and what get and exists do, is that these other functions search up the enclosing environments, i.e., it matters where the functions were defined. whereInStack looks up the call stack environments. See the example for the difference.

Value

The environment that is in the call stack where the object exists, that is closest to the frame in which this function is called.

Examples

```
b <- 1
inner <- function(y) {
  objEnv <- whereInStack("b")
  get("b", envir = objEnv)
}
findB <- function(x) {
  b <- 2
  inner()
}
findB() # Finds 2 because it is looking up the call stack, i.e., the user's perspective
```

```
# defined outside of findB2, so its enclosing environment is the same as findB2
innerGet <- function(y) {
   get("b")
}
findB2 <- function(x) {
  b <- 2
  innerGet()
}
findB2() # Finds 1 because b has a value of 1 in the enclosing environment of innerGet
b <- 3
findB2() # Finds 3 because b has a value of 3 in the enclosing environment of innerGet,
         #  i.e., the environment in which innerGet was defined
findB() # Still finds 2 because the call stack hasn't changed

# compare base::dynGet
findB3 <- function(x) {
  b <- 2
  dynGet("b")
}
findB3() # same as findB(), but marginally faster, because it omits the stripping on
         #   sub elements that may be part of the name argument


b <- list()
findB3 <- function(x) {
  b$a <- 2
  dynGet("b$a")
}
testthat::expect_error(findB3()) # fails because not an object name

findB <- function(x) {
  b$a <- 2
  env <- whereInStack("b$a")
  env
}
findB() # finds it
```

Index