Package: rgl (via r-universe)

October 28, 2026

Version 1.3.36

Title 3D Visualization Using OpenGL

Depends R (>= 3.6.0)

Suggests MASS, rmarkdown (>= 2.16), deldir (>= 1.0-4), orientlib, lattice, misc3d, magick, plotrix (>= 3.7-3), interp, alphashape3d, tcltk, js (>= 1.2), webshot2 (>= 0.1.0), downlit (>= 0.4.0), pkgdown (>= 2.0.0), extrafont, shiny, manipulateWidget (>= 0.9.0), testthat, crosstalk, V8, chromote, jpeg, png, markdown, xdvir, grid

Imports graphics, grDevices, stats, utils, htmlwidgets (>= 1.6.0), htmltools, knitr (>= 1.33), jsonlite (>= 0.9.20), magrittr, R6, base64enc, mime

Enhances waldo

Description Provides medium to high level functions for 3D interactive graphics, including functions modelled on base graphics (plot3d(), etc.) as well as functions for constructing representations of geometric objects (cube3d(), etc.). Output may be on screen using OpenGL, or to various standard 3D file formats including WebGL, PLY, OBJ, STL as well as 2D image formats, including PNG, Postscript, SVG, PGF.

License GPL

URL https://github.com/dmurdoch/rgl, https://dmurdoch.github.io/rgl/

SystemRequirements OpenGL and GLU Library (Required for display in R. See ``Installing OpenGL support'' in README.md. Not needed if only browser displays using rglwidget() are wanted.), zlib (optional), libpng (>=1.2.9, optional), FreeType (optional), pandoc (>=1.14, needed for vignettes), LaTeX needed for optional latex3d() examples.

BugReports https://github.com/dmurdoch/rgl/issues

VignetteBuilder knitr, rmarkdown

Biarch true

RoxygenNote 7.3.3

NeedsCompilation yes

Author Duncan Murdoch [aut, cre], Daniel Adler [aut], Oleg Nenadic [ctb], Simon Urbanek [ctb], Ming Chen [ctb], Albrecht Gebhardt [ctb], Ben Bolker [ctb], Gabor Csardi [ctb], Adam Strzelecki [ctb], Alexander Senger [ctb], The R Core Team [ctb, cph], Dirk Eddelbuettel [ctb], The authors of Shiny [cph], The authors of knitr [cph], Jeroen Ooms [ctb], Yohann Demont [ctb], Joshua Ulrich [ctb], Xavier Fernandez i Marin [ctb], George Helffrich [ctb], Ivan Krylov [ctb], Michael Sumner [ctb], Mike Stein [ctb], Jonathon Love [ctb], Mapbox team [ctb, cph]

Maintainer Duncan Murdoch <murdoch.duncan@gmail.com>

Config/pak/sysreqs libfreetype6-dev libglu1-mesa-dev texlive libpng-dev libgl1-mesa-dev zlib1g-dev

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

Date/Publication 2026-03-06 05:00:02 UTC

RemoteUrl https://github.com/cran/rgl

RemoteRef HEAD

RemoteSha e804083563b2d26960bebec923e42c96611746eb

Contents

rgl-package 5
.check3d 6
abclines3d 7
addNormals 8
ageControl 9
all.equal.mesh3d 10
arc3d 11
arrow3d 13
as.mesh3d 14
as.mesh3d.ashape3d 16
as.mesh3d.rglId 18
as.rglscene 20
as.tmesh3d 20
as.triangles3d 21
aspect3d 22
asRow 23
axes3d 25
bbox3d 28
bg3d 29
bgplot3d 31
Buffer 32
callbacks 37
checkDeldir 39
clipMesh3d 39
clipplaneControl 43

contourLines3d . . . 44
cube3d . . . 46
cylinder3d . . . 47
decorate3d . . . 49
drape3d . . . 50
ellipse3d . . . 52
expect_known_scene . . . 54
extrude3d . . . 55
facing3d . . . 56
figWidth . . . 57
getBoundary3d . . . 58
gltfTypes . . . 59
GramSchmidt . . . 59
grid3d . . . 60
hover3d . . . 62
identify3d . . . 64
import . . . 65
in_pkgdown_example . . . 65
light . . . 66
makeDependency . . . 67
material3d . . . 69
matrices . . . 74
merge.mesh3d . . . 76
mergeVertices . . . 77
mesh3d . . . 78
mfrow3d . . . 80
observer3d . . . 82
open3d . . . 83
par3d . . . 85
par3dinterp . . . 89
par3dinterpControl . . . 91
pch3d . . . 92
persp3d . . . 93
persp3d.deldir . . . 96
persp3d.function . . . 98
persp3d.triSht . . . 100
planes3d . . . 102
play3d . . . 103
playwidget . . . 105
plot3d . . . 108
plot3d.formula . . . 110
plot3d.lm . . . 111
plotmath3d . . . 113
polygon3d . . . 115
primitives . . . 116
propertyControl . . . 118
r3d . . . 120
readSTL . . . 121

rgl.attrib . . . 123
rgl.attrib.info . . . 124
rgl.bringtotop . . . 125
rgl.getAxisCallback . . . 126
rgl.incrementID . . . 127
rgl.init . . . 127
rgl.pixels . . . 128
rgl.postscript . . . 130
rgl.select . . . 131
rgl.Sweave . . . 132
rgl.useNULL . . . 133
rgl.user2window . . . 134
rglExtrafonts . . . 136
rglFonts . . . 137
rglIds . . . 138
rglMouse . . . 140
rglShared . . . 141
rglToLattice . . . 143
rglwidget . . . 144
safe.dev.off . . . 148
scene . . . 149
scene3d . . . 150
sceneChange . . . 153
select3d . . . 154
selectpoints3d . . . 156
setAxisCallbacks . . . 157
setGraphicsDelay . . . 159
setupKnitr . . . 160
setUserCallbacks . . . 162
setUserShaders . . . 166
shade3d . . . 169
shadow3d . . . 172
shapelist3d . . . 173
shiny . . . 174
shinyGetPar3d . . . 176
show2d . . . 178
snapshot3d . . . 180
spheres3d . . . 182
spin3d . . . 183
sprites . . . 185
subdivision3d . . . 187
subscene3d . . . 188
subsceneInfo . . . 191
surface3d . . . 192
tagged3d . . . 194
text3d . . . 195
textureSource . . . 197
thigmophobe3d . . . 198

tkpar3dsave 199
tkrgl 201
tkspin3d 201
tkspinControl 202
toggleWidget 203
triangulate 205
turn3d 206
vertexControl 207
viewpoint 209
writeASY 210
writeOBJ 212
writePLY 214

Index **217**

`rgl-package` *3D visualization device system*

Description

3D real-time rendering system.

Details

RGL is a 3D real-time rendering system for R. Multiple windows are managed at a time. Windows may be divided into "subscenes", where one has the current focus that receives instructions from the R command-line. The device design is oriented towards the R device metaphor. If you send scene management instructions, and there's no device open, it will be opened automatically. Opened devices automatically get the current device focus. The focus may be changed by using `set3d()` or `useSubscene3d()`.

RGL provides medium to high level functions for 3D interactive graphics, including functions modelled on base graphics (`plot3d()`, etc.) as well as functions for constructing geometric objects (`cube3d()`, etc.). Output may be on screen using OpenGL, or to various standard 3D file formats including WebGL, PLY, OBJ, STL as well as 2D image formats, including PNG, Postscript, SVG, PGF.

The `open3d()` function attempts to open a new RGL window, using default settings specified by the user.

See the first example below to display the ChangeLog.

See Also

`r3d` for a description of the `*3d` interface; `par3d` for a description of scene properties and the rendering pipeline; `rgl.useNULL` for a description of how to use RGL on a system with no graphics support.

Examples

```
if (!in_pkgdown_example())
  file.show(system.file("NEWS", package = "rgl"))
example(surface3d)
example(plot3d)
```

`.check3d` *Check for an open RGL window.*

Description

Mostly for internal use, this function returns the current device number if one exists, or opens a new device and returns that.

Usage

```
.check3d()
```

Value

The device number of an RGL device.

Author(s)

Duncan Murdoch

See Also

`open3d`

Examples

```
rgl.dev.list()
.check3d()
rgl.dev.list()
.check3d()
rgl.dev.list()
close3d()
```

`abclines3d` *Lines intersecting the bounding box*

Description

This adds mathematical lines to a scene. Their intersection with the current bounding box will be drawn.

Usage

```
abclines3d(x, y = NULL, z = NULL, a, b = NULL, c = NULL, ...)
```

Arguments

`x`, `y`, `z`	Coordinates of points through which each line passes.
`a`, `b`, `c`	Coordinates of the direction vectors for the lines.
`...`	Material properties.

Details

Draws the segment of a line that intersects the current bounding box of the scene using the parametrization $(x, y, z) + (a, b, c) * s$ where s is a real number.

Any reasonable way of defining the coordinates `x, y, z` and `a, b, c` is acceptable. See the function `xyz.coords` for details.

Value

A shape ID of the object is returned invisibly.

See Also

`planes3d` for mathematical planes.

`segments3d` draws sections of lines that do not adapt to the bounding box.

Examples

```
plot3d(rnorm(100), rnorm(100), rnorm(100))
abclines3d(0, 0, 0, a = diag(3), col = "gray")
```

addNormals *Add normal vectors to objects so they render more smoothly*

Description

This generic function adds normals at each of the vertices of a polyhedron by averaging the normals of each incident face. This has the effect of making the surface of the object appear smooth rather than faceted when rendered.

Usage

```
addNormals(x, ...)
## S3 method for class 'mesh3d'
addNormals(x, angleWeighted = TRUE, ...)
```

Arguments

x	An object to which to add normals.
...	Additional parameters which will be passed to the methods.
angleWeighted	See Details below.

Details

Currently methods are supplied for "mesh3d" and "shapelist3d" classes.

These methods work by averaging the normals on the faces incident at each vertex. By default these are weighted according to the angle in the polygon at that vertex. If angleWeighted = FALSE, a slightly faster but less accurate weighting by the triangle area is used.

Prior to **rgl** version 0.104.12 an incorrect weighting was used; it can be partially reproduced by using angleWeighted = NA, but not all the bugs in that scheme will be kept.

Value

A new object of the same class as x, with normals added.

Author(s)

Duncan Murdoch

Examples

```
open3d()
y <- subdivision3d(tetrahedron3d(col = "red"), depth = 3)
shade3d(y) # No normals
y <- addNormals(y)
shade3d(translate3d(y, x = 1, y = 0, z = 0)) # With normals
```

ageControl *Set attributes of vertices based on their age*

Description

This is a function to produce actions in response to a playwidget or Shiny input control. The mental model is that each of the vertices of some object has a certain birth time; a control sets the current time, so that vertices have ages depending on the control setting. Attributes of those vertices can then be changed.

Usage

```
ageControl(births, ages, objids = tagged3d(tags), tags, value = 0,
           colors = NULL, alpha = NULL, radii = NULL, vertices = NULL,
           normals = NULL, origins = NULL, texcoords = NULL,
           x = NULL, y = NULL, z = NULL,
           red = NULL, green = NULL, blue = NULL)
```

Arguments

births	Numeric birth times of vertices.
ages	Chosen ages at which the following attributes will apply.
objids	Object ids to which the changes apply.
tags	Alternate way to specify objids. Ignored if objids is given.
value	Initial value; typically overridden by input.
colors, alpha, radii, vertices, normals, origins, texcoords	Attributes of the vertices that can be changed. There should be one entry or row for each entry in ages.
x, y, z, red, green, blue	These one-dimensional components of vertices and colors are provided for convenience.

Details

All attributes must have the same number of entries (rows for the matrices) as the ages vector. The births vector must have the same number of entries as the number of vertices in the object.

Not all objects contain all attributes; if one is chosen that is not a property of the corresponding object, a Javascript alert() will be generated. (This restriction may be removed in the future by attempting to add the attribute when it makes sense.)

If a births entry is NA, no change will be made to that vertex.

Value

A list of class "rglControl" of cleaned up parameter values, to be used in an RGL widget.

Author(s)

Duncan Murdoch

See Also

The User Interaction in WebGL vignette gives more details.

Examples

```
saveopts <- options(rgl.useNULL = TRUE)

theta <- seq(0, 4*pi, length.out = 100)
xyz <- cbind(sin(theta), cos(theta), sin(theta/2))
lineid <- plot3d(xyz, type="l", alpha = 0, lwd = 5, col = "blue")["data"]

widget <- rglwidget() %>%
playwidget(ageControl(births = theta,
                      ages = c(-4*pi, -4*pi, 1-4*pi, 0, 0, 1),
                      objids = lineid,
                      alpha = c(0, 1, 0, 0, 1, 0)),
           start = 0, stop = 4*pi,
           step = 0.1, rate = 4)
if (interactive() || in_pkgdown_example())
  widget
options(saveopts)
```

---

all.equal.mesh3d *Compare mesh3d objects in a meaningful way.*

Description

These functions allow comparison of mesh3d objects, ignoring irrelevant differences.

`compare_proxy.mesh3d` can function as a `compare_proxy` method for the **waldo** package, by stripping out `NULL` components and ordering other components alphabetically by name.

`all.equal.mesh3d` compares mesh3d objects by using `compare_proxy.mesh3d` to standardize them, then using the regular `all.equal` function to compare them.

Usage

```
## S3 method for class 'mesh3d'
all.equal(target, current, ...)
compare_proxy.mesh3d(x, path = "x")
```

Arguments

`target, current`	Two mesh3d objects to compare.
`x`	A single mesh3d object to standardize.
`path`	The string to use in a **waldo** display of this object.
`...`	Additional parameters to pass to `all.equal`.

Value

`all.equal.mesh3d` returns `TRUE`, or a character vector describing (some of) the differences.

`compare_proxy.mesh3d` returns a list containing two components:

object a copy of `x` with relevant components in alphabetical order.

path a modification of the path label for `x`

Note

waldo is not an installation requirement for **rgl** and **rgl** will never cause it to be loaded. The `compare_proxy.mesh3d` function will only be registered as a method for `waldo::compare_proxy` if you load **waldo** before **rgl**, as would normally happen during testing using **testthat**, or if you load it before calling `mesh3d`, as might happen if you are doing manual tests.

---

arc3d *Draw arcs*

Description

Given starting and ending points on a sphere and the center of the sphere, draw the great circle arc between the starting and ending points. If the starting and ending points have different radii, a segment of a logarithmic spiral will join them, unless they are in the same direction, in which case a straight line will join them.

Usage

```
arc3d(from, to, center, radius, n, circle = 50, base = 0,
plot = TRUE, ...)
```

Arguments

`from`	One or more points from which to start arcs.
`to`	One or more destination points.
`center`	One or more center points.
`radius`	If not missing, a vector of length `n` giving the radii at each point between `from` and `to`. If missing, the starting and ending points will be joined by a logarithmic spiral.
`n`	If not missing, how many segments to use between the first and last point. If missing, a value will be calculated based on the angle between starting and ending points as seen from the center.
`circle`	How many segments would be used if the arc went completely around a circle.
`base`	See Details below.
`plot`	Should the arcs be plotted, or returned as a matrix?
`...`	Additional parameters to pass to `points3d`.

Details

If any of `from`, `to` or `center` is an n by 3 matrix with n > 1, multiple arcs will be drawn by recycling each of these parameters to the number of rows of the longest one.

If the vector lengths of `from - center` and `to - center` differ, then instead of a spherical arc, the function will draw a segment of a logarithmic spiral joining the two points.

By default, the arc is drawn along the shortest great circle path from `from` to `to`, but the `base` parameter can be used to modify this. If `base = 1` is used, the longer arc will be followed. Larger positive integer values will result in `base - 1` loops in that direction completely around the sphere. Negative values will draw the curve in the same direction as the shortest arc, but with `abs(base)` full loops. It doesn't make much sense to ask for such loops unless the radii of `from` and `to` differ, because spherical arcs would overlap. Normally the `base` parameter is left at its default value of `0`.

When `base` is non-zero, the curve will be constructed in multiple pieces, between `from`, `to`, `-from` and `-to`, for as many steps as necessary. If `n` is specified, it will apply to each of these pieces.

Value

If `plot = TRUE`, called mainly for the side effect of drawing arcs. Invisibly returns the object ID of the collection of arcs.

If `plot = FALSE`, returns a 3 column matrix containing the points that would be drawn as the arcs.

Author(s)

Duncan Murdoch

Examples

```
normalize <- function(v) v/sqrt(sum(v^2))

# These vectors all have the same length

from <- t(apply(matrix(rnorm(9), ncol = 3), 1, normalize))
to <- normalize(rnorm(3))
center <- c(0, 0, 0)

open3d()
spheres3d(center, radius = 1, col = "white", alpha = 0.2)

arc3d(from, to, center, col = "red")
arc3d(from, 2*to, center, col = "blue")

text3d(rbind(from, to, center, 2*to),
       texts = c(paste0("from", 1:3), "to", "center", "2*to"),
       depth_mask = FALSE, depth_test = "always")
```

arrow3d *Draw an arrow*

Description

Draws various types of arrows in a scene.

Usage

```
arrow3d(p0 = c(1, 1, 1), p1 = c(0, 0, 0),
        barblen, s = 1/3, theta = pi/12,
        type = c("extrusion", "lines", "flat", "rotation"),
        n = 3, width = 1/3, thickness = 0.618 * width,
        spriteOrigin = NULL,
        plot = TRUE, ...)
```

Arguments

p0	The base of the arrow.
p1	The head of the arrow.
barblen	The length of the barbs (in display coordinates). Default given by s.
s	The length of the barbs as a fraction of line length. Ignored if barblen is present.
theta	Opening angle of barbs
type	Type of arrow to draw. Choose one from the list of defaults. Can be abbreviated. See below.
n	Number of barbs.
width	Width of shaft as fraction of barb width.
thickness	Thickness of shaft as fraction of barb width.
spriteOrigin	If arrow is to be replicated as sprites, the origins relative to which the sprites are drawn.
plot	If TRUE (the default), plot the object; otherwise return the computed data that would be used to plot it.
...	Material properties passed to polygon3d, shade3d or segments3d.

Details

Four types of arrows can be drawn. The shapes of all of them are affected by p0, p1, barblen, s, theta, material properties in ..., and spriteOrigin. Other parameters only affect some of the types, as shown.

"extrusion" (default) A 3-dimensional flat arrow, drawn with shade3d. Affected by width, thickness and smooth.

"lines" Drawn with lines, similar to arrows, drawn with segments3d. Affected by n.

"flat" A flat arrow, drawn with polygon3d. Affected by width and smooth.

"rotation" A solid of rotation, drawn with shade3d. Affected by n and width.

Normally this function draws just one arrow from p0 to p1, but if spriteOrigin is given (in any form that xyz.coords(spriteOrigin) can handle), arrows will be drawn for each point specified, with p0 and p1 interpreted relative to those origins. The arrows will be drawn as 3D sprites which will maintain their orientation as the scene is rotated, so this is a good way to indicate particular locations of interest in the scene.

Value

If plot = TRUE (the default), this is called mainly for the side effect of drawing the arrow; invisibly returns the id(s) of the objects drawn.

If plot = FALSE, the data that would be used in the plot (not including material properties) is returned.

Author(s)

Design based on heplots::arrow3d, which contains modifications by Michael Friendly to a function posted by Barry Rowlingson to R-help on 1/10/2010. Additions by Duncan Murdoch.

Examples

```
xyz <- matrix(rnorm(300), ncol = 3)
plot3d(xyz)
arrow3d(xyz[1,], xyz[2,], type = "extrusion", col = "red")
arrow3d(xyz[3,], xyz[4,], type = "flat",      col = "blue")
arrow3d(xyz[5,], xyz[6,], type = "rotation",  col = "green")
arrow3d(xyz[7,], xyz[8,], type = "lines",     col = "black")
arrow3d(spriteOrigin = xyz[9:12,],            col = "purple")
```

---

as.mesh3d *Convert object to mesh object*

Description

The as.mesh3d generic function converts various objects to mesh3d objects.

The default method takes takes a matrix of vertices as input and (optionally) merges repeated vertices, producing a mesh3d object as output. It will contain either triangles or quads or segments or points according to the type argument.

If the generic is called without any argument, it will pass all RGL ids from the current scene to the as.mesh3d.rglId method.

Usage

```
as.mesh3d(x, ...)
## Default S3 method:
as.mesh3d(x, y = NULL, z = NULL,
             type = c("triangles", "quads", "segments", "points"),
             smooth = FALSE,
             tolerance = sqrt(.Machine$double.eps),
             notEqual = NULL,
             merge = TRUE,
             ...,
             triangles)
```

Arguments

x, y, z	For the generic, x is the object to convert. For the default method, x, y and z are coordinates. Any reasonable way of defining the coordinates is acceptable. See the function xyz.coords for details.
type	What type of things should be in the mesh? Tries this list in order until it finds one that works.
smooth	If TRUE, addNormals will be called on the mesh object to make it render smoothly.
tolerance	The numerical tolerance to be used in all.equal to determine whether two vertices should be merged.
notEqual	If not NULL, an n by n matrix of logical values, where n is the number of vertices as input. TRUE entries indicate that the corresponding pair of vertices should not be merged even if they appear equal.
merge	Should apparently equal vertices be merged?
...	Material properties to pass to tmesh3d or qmesh3d.
triangles	Deprecated. If present, TRUE indicates type = "triangles" and FALSE indicates type = "quads".

Details

The motivation for this function is the following problem: I was asked whether RGL could render a surface made up of triangles or quadrilaterals to look smooth. It can do that, but needs normals at each vertex; they should be the average of the normals for each polygon sharing that vertex. Then OpenGL will interpolate the normals across the polygons and give the illusion of smoothness.

To do this, it needs to know which polygons share each vertex. If the surface is described as a list of triangles or quadrilaterals, that means identifying vertices that are in multiple polygons, and converting the representation to a "mesh3d" object (which is a matrix of vertices and a matrix of vertex numbers making up triangles or quads). Then the addNormals function will add the normals.

Sometimes two polygons will share vertices (within numerical tolerance) without the user wanting them to be considered internal to the surface, or might want one sharp edge in an otherwise smooth surface. This means I needed a way to declare that two vertices from the original list of vertices in the triangles or quads are "not equal", even when they test numerically equal. That's what the notEqual matrix specifies.

Value

A "mesh3d" object with the same faces as in the input, but (if merge=TRUE) with vertices that test equal to within tolerance merged.

Author(s)

Duncan Murdoch

Examples

```
xyz <- matrix(c(-1, -1, -1,
                -1,  1, -1,
                 1,  1, -1,
                 1, -1, -1,
                -1,  1, -1,
                -1,  1,  1,
                 1,  1,  1,
                 1,  1, -1,
                 1, -1, -1,
                 1,  1, -1,
                 1,  1,  1,
                 1, -1,  1), byrow = TRUE, ncol = 3)
mesh <- as.mesh3d(xyz, type = "quads", col = "red")
mesh$vb
mesh$ib
open3d()
shade3d(mesh)

# Stop vertices 2 and 5 from being merged
notEQ <- matrix(FALSE, 12, 12)
notEQ[2, 5] <- TRUE
mesh <- as.mesh3d(xyz, type = "quads", notEqual = notEQ)
mesh$vb
mesh$ib
```

---

as.mesh3d.ashape3d *Convert alpha-shape surface of a cloud of points to RGL mesh object*

Description

The alphashape3d::ashape3d function computes the 3D α-shape of a cloud of points. This is an approximation to the visual outline of the cloud. It may include isolated points, line segments, and triangular faces: this function converts the triangular faces to an RGL tmesh3d object.

Usage

```
## S3 method for class 'ashape3d'
as.mesh3d(x,
                    alpha = x$alpha[1],
```

```
                    tri_to_keep = 2L,
                    col = "gray",
                    smooth = FALSE, normals = NULL,
                    texcoords = NULL, ...)
```

Arguments

x	An object of class "ashape3d".
alpha	Which alpha value stored in x should be converted?
tri_to_keep	Which triangles to keep. Expert use only: see triang entry in **Value** section of ashape3d for details.
col	The surface colour.
smooth	Whether to attempt to add normals to make the surface look smooth. See the Details below.
normals, texcoords	Normals and texture coordinates at each vertex can be specified.
...	Additional arguments to pass to use as material3d properties on the resulting mesh.

Details

Edelsbrunner and Mucke's (1994) α-shape algorithm is intended to compute a surface of a general cloud of points. Unlike the convex hull, the cloud may have voids, isolated points, and other oddities. This function is designed to work in the case where the surface is made up of simple polygons.

If smooth = TRUE, this method attempts to orient all of the triangles in the surface consistently and add normals at each vertex by averaging the triangle normals. However, for some point clouds, the α-shape will contain sheets of polygons with a few solid polyhedra embedded. This does not allow a consistent definition of "inside" and outside. If this is detected, a warning is issued and the resulting mesh will likely contain boundaries where the assumed orientation of triangles changes, resulting in ugly dark lines through the shape. Larger values of alpha in the call to alphashape3d::ashape3d may help.

Methods for plot3d and persp3d are also defined: they call the as.mesh3d method and then plot the result.

Value

A "mesh3d" object, suitable for plotting.

Author(s)

Duncan Murdoch

References

Edelsbrunner, H., Mucke, E. P. (1994). Three-Dimensional Alpha Shapes. ACM Transactions on Graphics, 13(1), pp.43-72.

Lafarge, T. and Pateiro-Lopez, B. (2017). alphashape3d: Implementation of the 3D Alpha-Shape for the Reconstruction of 3D Sets from a Point Cloud. R package version 1.3.

Examples

```
if (requireNamespace("alphashape3d", quietly = TRUE)) {
  set.seed(123)
  n <- 400    # 1000 gives a nicer result, but takes longer
  xyz <- rbind(cbind(runif(n), runif(n), runif(n)),
               cbind(runif(n/8, 1, 1.5),
                     runif(n/8, 0.25, 0.75),
                     runif(n/8, 0.25, 0.75)))
  ash <- suppressMessages(alphashape3d::ashape3d(xyz, alpha = 0.2))
  m <- as.mesh3d(ash, smooth = TRUE)
  open3d()
  mfrow3d(1, 2, sharedMouse = TRUE)
  plot3d(xyz, size = 1)
  plot3d(m, col = "red", alpha = 0.5)
  points3d(xyz, size = 1)
}
```

---

as.mesh3d.rglId *Convert object in plot to RGL mesh object*

Description

This method attempts to read the attributes of objects in the rgl display and construct a mesh3d object to approximate them.

Usage

```
## S3 method for class 'rglId'
as.mesh3d(x, type = NA, subscene = NA, ...)
```

Arguments

x	A vector of RGL identifiers of objects in the specified subscene.
type	A vector of names of types of shapes to convert. Other shapes will be ignored.
subscene	Which subscene to look in; the default NA specifies the current subscene.
...	Ignored.

Details

This function attempts to construct a triangle mesh to approximate one or more objects from the current display. It can handle objects of types from `c("points", "lines", "linestrip", "triangles", "quads", "planes", "surface")`.

Since this method only produces meshes containing points, segments and triangles, they won't necessarily be an exact match to the original object.

If the generic `as.mesh3d` is called with no `x` argument, this method will be called with `x` set to the ids in the current scene.

Value

A mesh object.

Author(s)

Duncan Murdoch

See Also

`as.triangles3d.rglId` for extracting the triangles, `clipMesh3d` to apply complex clipping to a mesh object.

Examples

```
# volcano example taken from "persp"
#
data(volcano)

z <- 2 * volcano        # Exaggerate the relief

x <- 10 * (1:nrow(z))   # 10 meter spacing (S to N)
y <- 10 * (1:ncol(z))   # 10 meter spacing (E to W)

zlim <- range(y)
zlen <- zlim[2] - zlim[1] + 1

colorlut <- terrain.colors(zlen) # height color lookup table

col <- colorlut[ z - zlim[1] + 1 ] # assign colors to heights for each point

open3d(useNULL = TRUE)
surface3d(x, y, z, color = col)
m <- as.mesh3d()
close3d()

open3d()
shade3d(m)
```

as.rglscene *Convert an object to an* rglscene *object.*

Description

This is a placeholder generic function, to allow other packages to create "rglscene" objects compatible with the objects produced by scene3d.

No methods are currently defined in **rgl**.

Usage

```
as.rglscene(x, ...)
```

Arguments

x	Object to convert.
...	Other parameters to pass to methods.

as.tmesh3d *Convert object to a triangular mesh*

Description

Converts the quads in a mesh version of an object to triangles by splitting them up. Optionally drops any point or segment components.

Usage

```
as.tmesh3d(x, ...)
## Default S3 method:
as.tmesh3d(x, drop = FALSE, ...)
## S3 method for class 'mesh3d'
as.tmesh3d(x, drop = FALSE, keepTags = FALSE, ...)
```

Arguments

x	An object from which to create a triangular mesh object.
drop	If TRUE, drop any point or segment components.
keepTags	Whether to include the "tags" component in the output.
...	Ignored in the mesh3d method, passed to as.mesh3d in the default method.

Details

The default method simply calls as.mesh3d(x, ...) and passes the result to the "mesh3d" method.

Value

A "mesh3d" object containing no quads. If drop = TRUE, it will only contain triangles.

If keepTags = TRUE, a "tags" element will be added to the result. For details, see the clipMesh3d help page.

Note

Older versions of **rgl** had a "tmesh3d" class for meshes of triangles. That class is no longer used: as.tmesh3d and tmesh3d both produce "mesh3d" objects.

Author(s)

Duncan Murdoch

See Also

as.triangles3d to get just the coordinates.

Examples

```
x <- cuboctahedron3d()
x             # has quads and triangles
as.tmesh3d(x) # has only triangles
```

---

as.triangles3d *Convert an object to triangles*

Description

This generic and its methods extract or creates a matrix of coordinates of triangles from an object, suitable for passing to triangles3d.

Usage

```
as.triangles3d(obj, ...)
## S3 method for class 'rglId'
as.triangles3d(obj,
               attribute = c("vertices", "normals", "texcoords", "colors"),
               subscene = NA,
               ...)
```

Arguments

obj	The object to convert.
attribute	Which attribute of an RGL object to extract?
subscene	Which subscene is this object in?
...	Additional arguments used by the methods.

Details

The method for "rglId" objects can extract several different attributes, organizing them as it would organize the vertices for the triangles.

Value

An n x 3 matrix containing the vertices of triangles making up the object. Each successive 3 rows of the matrix corresponds to a triangle.

If the attribute doesn't exist, NULL will be returned.

Author(s)

Duncan Murdoch

See Also

as.mesh3d to also capture material properties.

Examples

```
open3d()
x <- surface3d(x = 1:10, y = 1:10, z = rnorm(100), col = "red")
tri <- as.triangles3d(x)
open3d()
triangles3d(tri, col = "blue")
```

---

aspect3d *Set the aspect ratios of the current plot*

Description

This function sets the apparent ratios of the x, y, and z axes of the current bounding box.

Usage

```
aspect3d(x, y = NULL, z = NULL)
```

Arguments

x	The ratio for the x axis, or all three ratios, or "iso"
y	The ratio for the y axis
z	The ratio for the z axis

Details

If the ratios are all 1, the bounding box will be displayed as a cube approximately filling the display. Values may be set larger or smaller as desired. Aspect `"iso"` signifies that the coordinates should all be displayed at the same scale, i.e. the bounding box should not be rescaled. (This corresponds to the default display before `aspect3d` has been called.) Partial matches to `"iso"` are allowed.

`aspect3d` works by modifying `par3d("scale")`.

Value

The previous value of the scale is returned invisibly.

Author(s)

Duncan Murdoch

See Also

`plot3d`, `par3d`

Examples

```
x <- rnorm(100)
y <- rnorm(100)*2
z <- rnorm(100)*3

open3d()
plot3d(x, y, z)
aspect3d(1, 1, 0.5)
highlevel()  # To trigger display
open3d()
plot3d(x, y, z)
aspect3d("iso")
highlevel()
```

---

asRow *Convenience functions for RGL HTML layouts*

Description

The `asRow` function arranges objects in a row in the display; the `getWidgetId` function extracts the HTML element ID from an HTML widget.

Usage

```
asRow(..., last = NA, height = NULL, colsize = 1)
getWidgetId(widget)
```

Arguments

...	Either a single "combineWidgets" object produced by asRow or a %>% pipe of RGL objects, or several objects intended for rearrangement.
last	If not NA, the number of objects from ... that are to be arranged in a row. Earlier ones will remain in a column.
height	An optional height for the resulting row. This is normally specified in pixels, but will be rescaled as necessary to fit the display.
colsize	A vector of relative widths for the columns in the row.
widget	A single HTML widget from which to extract the HTML element ID.

Details

Using asRow requires that the **manipulateWidget** package is installed.

asRow produces a "combineWidgets" object which is a single column whose last element is another "combineWidgets" object which is a single row.

If n objects are given as input and last is given a value less than n, the first n - last objects will be displayed in a column above the row containing the last objects.

Value

asRow returns a single "combineWidgets" object suitable for display or nesting within a more complicated display.

getWidgetId returns a character string containing the HTML element ID of the widget.

Author(s)

Duncan Murdoch

See Also

pipe for the %>% operator. The User Interaction in WebGL vignette gives more details.

Examples

```
if (requireNamespace("manipulateWidget", quietly = TRUE) &&
    require("crosstalk", quietly = TRUE)) {
  sd <- SharedData$new(mtcars)
  ids <- plot3d(sd$origData(), col = mtcars$cyl, type = "s")
  # Copy the key and group from existing shared data
  rglsd <- rglShared(ids["data"], key = sd$key(), group = sd$groupName())
  w <- rglwidget(shared = rglsd) %>%
       asRow("Mouse mode: ", rglMouse(getWidgetId(.)),
             "Subset: ", filter_checkbox("cylinderselector",
                 "Cylinders", sd, ~ cyl, inline = TRUE),
             last = 4, colsize = c(1,2,1,2), height = 60)
  if (interactive() || in_pkgdown_example())
    w
}
```

axes3d *Draw boxes, axes and other text outside the data*

Description

These functions draw axes, boxes and text outside the range of the data. `axes3d`, `box3d` and `title3d` are the higher level functions; normally the others need not be called directly by users.

Usage

```
axes3d(edges = "bbox", labels = TRUE, tick = TRUE, nticks = 5,
 box = FALSE, expand = 1.03, ...)
box3d(...)
title3d(main = NULL, sub = NULL, xlab = NULL, ylab = NULL,
 zlab = NULL, line = NA, level = NA, floating = NULL, ...)
axis3d(edge, at = NULL, labels = TRUE, tick = TRUE, line = 0,
 pos = NULL, nticks = 5, ...)
mtext3d(text, edge, at = NULL, line = 0, level = 0,
        floating = FALSE, pos = NA, ...)
```

Arguments

`edges`	a code to describe which edge(s) of the box to use; see Details below
`labels`	whether to label the axes, or (for `axis3d`) the labels to use
`tick`	whether to use tick marks
`nticks`	suggested number of ticks
`box`	draw the full box if `"bbox"` axes are used
`expand`	how much to expand the box around the data
`main`	the main title for the plot
`sub`	the subtitle for the plot
`xlab`, `ylab`, `zlab`	the axis labels for the plot
`line`, `level`	the "line" of the plot margin to draw the label on, and "level" above or below it
`floating`	which mode of axis labels? One of `TRUE`, `FALSE` or `NA`. (`NULL` may also be used in `title3d` calls). See Details for how these are handled.
`edge`, `pos`	the position at which to draw the axis or text
`text`	the text to draw
`at`	the value of a coordinate at which to draw the axis or labels.
`...`	additional parameters which are passed to `bbox3d` or `material3d`

Details

The rectangular prism holding the 3D plot has 12 edges. They are identified using 3 character strings. The first character ('x', 'y', or 'z') selects the direction of the axis. The next two characters are each '-' or '+', selecting the lower or upper end of one of the other coordinates. If only one or two characters are given, the remaining characters normally default to '-' (but with `mtext3d(..., floating = TRUE)` the default is '+'; see below). For example `edge = 'x+'` draws an x-axis at the high level of y and the low level of z.

By default, `axes3d` uses the `bbox3d` function to draw the axes. The labels will move so that they do not obscure the data. Alternatively, a vector of arguments as described above may be used, in which case fixed axes are drawn using `axis3d`.

As of **rgl** version 0.106.21, axis drawing has changed significantly. Text drawn in the margins will adapt to the margins (see `bbox3d`). The `edge` and `floating` parameters will be recorded in the `margin` and `floating` material properties for the object.

If `floating = FALSE`, they will be drawn on the specified edge.

If `floating = TRUE`, they will move as the axis labels move when the scene is rotated. The signs on the edge specification are interpreted as agreeing with the axis ticks '+' or disagreeing '-'. For example, `"x++"` will draw text on the x axis in the same edge as the ticks, while `"x--"` will draw on the opposite edge.

The final possible value for `floating` in `mtext3d` is `NA`, which reproduces legacy **rgl** behaviour. In this case the labels are not tied to the bounding box, so they should be drawn last, or they could appear inside the box, overlapping the data.

In `title3d` `floating = NULL` (the default) indicates the main title and subtitle will be fixed while the axis labels will be floating. The default locations for title and subtitle are `line = 2` and `level = 2` on edges `"x++"` and `"x--"` respectively. The axis labels float at `line = 4` and `level = 1` on the same edge as the ticks.

The `at` parameter in `axis3d` is the location of the ticks, defaulting to `pretty` locations. In `mtext3d` the `at` parameter is the location on the specified axis at which to draw the text, defaulting to the middle of the bounding box.

The `line` parameter is the line counting out from the box in the same direction as the axis ticks, and `level` is the line out in the orthogonal direction. The ticks run from `line = 0` to `line = 1`, and the tick labels are drawn at `line = 2`. Both are drawn at level 0.

The `pos` parameter is only supported in legacy mode. If it is a numeric vector of length 3, `edge` determines the direction of the axis and the tick marks, and the values of the other two coordinates in `pos` determine the position. The `level` parameter is ignored in legacy mode.

For `mtext3d` in `floating = TRUE` or `floating = FALSE` mode, there are 3 special values for the `at` parameter: it may be `-Inf`, `NA` or `+Inf`, referring to the bottom, middle or top of the given axis respectively.

Value

These functions are called for their side effects. They return the object IDs of objects added to the scene.

Note

mtext3d is a wrapper for text3d that sets the margin and floating material properties. In fact, these properties can be set for many kinds of objects (most kinds where it would make sense), with the effect that the object will be drawn in the margin, with x coordinate corresponding to at, y corresponding to line, and z corresponding to level.

Author(s)

Duncan Murdoch

See Also

Classic graphics functions axis, box, title, mtext are related. See RGL functions bbox3d for drawing the box around the plot, and setAxisCallbacks for customized axes.

Examples

```
open3d()
points3d(rnorm(10), rnorm(10), rnorm(10))

# First add standard axes
axes3d()

# and one in the middle (the NA will be ignored, a number would
# do as well)
axis3d('x', pos = c(NA, 0, 0))

# add titles
title3d('main', 'sub', 'xlab', 'ylab', 'zlab')

# Use a log scale for z

open3d()

x <- rnorm(10)
y <- rnorm(10)
z <- exp(rnorm(10, mean = 3, sd = 2))

logz <- log10(z)
zticks <- axisTicks(range(logz), log = TRUE)
zat <- log10(zticks)

plot3d(x, y, logz, zlab = "z")
axes3d(zat = zat, zlab = zticks, box = TRUE)
```

bbox3d *Set up bounding box decoration*

Description

Set up the bounding box decoration.

Usage

```
bbox3d(xat = NULL, yat = NULL, zat = NULL,
 xunit = "pretty", yunit = "pretty", zunit = "pretty",
 expand = 1.03,
 draw_front = FALSE,
 xlab=NULL, ylab=NULL, zlab=NULL,
  xlen=5, ylen=5, zlen=5,
  marklen=15.0, marklen.rel=TRUE, ...)
```

Arguments

xat, yat, zat	vector specifying the tickmark positions
xlab, ylab, zlab	character vector specifying the tickmark labeling
xunit, yunit, zunit	value specifying the tick mark base for uniform tick mark layout
xlen, ylen, zlen	value specifying the number of tickmarks
marklen	value specifying the length of the tickmarks
marklen.rel	logical, if TRUE tick mark length is calculated using 1/marklen * axis length, otherwise tick mark length is marklen in coordinate space
expand	value specifying how much to expand the bounding box around the data
draw_front	draw the front faces of the bounding box
...	Material properties (or other rgl.bbox parameters in the case of bbox3d). See material3d for details.

Details

Four different types of tick mark layouts are possible. This description applies to the X axis; other axes are similar: If xat is not NULL, the ticks are set up at custom positions. If xunit is numeric but not zero, it defines the tick mark base. If it is "pretty" (the default in bbox3d), ticks are set at pretty locations. If xlen is not zero, it specifies the number of ticks (a suggestion if xunit is "pretty").

The first color specifies the bounding box, while the second one specifies the tick mark and font color.

bbox3d defaults to pretty locations for the axis labels and a slightly larger box, whereas rgl.bbox covers the exact range.

axes3d offers more flexibility in the specification of the axes, but they are static, unlike those drawn by bbox3d.

Value

This function is called for the side effect of setting the bounding box decoration. A shape ID is returned to allow pop3d to delete it.

See Also

material3d, axes3d

Examples

```
open3d()
points3d(rnorm(100), rnorm(100), rnorm(100))
bbox3d(color = c("#333377", "black"), emission = "#333377",
       specular = "#3333FF", shininess = 5, alpha = 0.8)
```

---

bg3d *Set up background*

Description

Set up the background of the scene.

Usage

```
bg3d(color,
    sphere=FALSE,
    back="lines",
    fogtype="none",
    fogScale = 1,
    col, ...)
```

Arguments

color, col	See Details below.
sphere	logical: if TRUE, an environmental sphere geometry is used for the background decoration.
back	Specifies the fill style of the sphere geometry. See material3d for details.
fogtype	fog type: **"none"** no fog **"linear"** linear fog function **"exp"** exponential fog function **"exp2"** squared exponential fog function Fog only applies to objects with material3d property fog set to TRUE.
fogScale	Scaling for fog. See Details.
...	Additional material properties. See material3d for details.

Details

The background color is taken from `color` or `col` if `color` is missing. The first entry is used for background clearing and as the fog color. The second (if present) is used for background sphere geometry.

If `color` and `col` are both missing, the default is found in the `r3dDefaults$bg` list, or `"white"` is used if nothing is specified there.

If `sphere` is set to `TRUE`, an environmental sphere enclosing the whole scene is drawn.

If not, but the material properties include a bitmap as a texture, the bitmap is drawn in the background of the scene. (The bitmap colors modify the general color setting.)

If neither a sphere nor a bitmap background is drawn, the background is filled with a solid color.

The `fogScale` parameter should be a positive value to change the density of the fog in the plot. For `fogtype = "linear"` it multiplies the density of the fog; for the exponential fog types it multiplies the density parameter used in the display.

See the OpenGL 2.1 reference for the formulas used in the fog calculations within R (though the `"exp2"` formula appears to be wrong, at least on my system). In WebGL displays, the following rules are used. They appear to match the rules used in R on my system.

- For `"linear"` fog, the near clipping plane is taken as $c = 0$, and the far clipping plane is taken as $c = 1$. The amount of fog is $s * c$ clamped to a 0 to 1 range, where $s = fogScale$.
- For `"exp"` and `"exp2"` fog, the observer location is negative at a distance depending on the field of view. The formula for the distance is

$$c = [1 - sin(theta)]/[1 + sin(theta)]$$

  where $theta = FOV/2$. We calculate

$$c' = d(1 - c) + c$$

  so c' runs from 0 at the observer to 1 at the far clipping plane.
- For `"exp"` fog, the amount of fog is $1 - exp(-s * c')$.
- For `"exp2"` fog, the amount of fog is $1 - exp[-(s * c')^2]$.

See Also

`material3d`, `bgplot3d` to add a 2D plot as background.

Examples

```
open3d()

# a simple white background

bg3d("white")

# the holo-globe (inspired by star trek):

bg3d(sphere = TRUE, color = c("black", "green"), lit = FALSE, back = "lines" )
```

```
# an environmental sphere with a nice texture.

bg3d(sphere = TRUE, texture = system.file("textures/sunsleep.png", package = "rgl"),
       back = "filled" )

# The same texture as a fixed background

open3d()
bg3d(texture = system.file("textures/sunsleep.png", package = "rgl"), col = "white")
```

---

bgplot3d *Use base graphics for RGL background*

Description

Add a 2D plot or a legend in the background of an RGL window.

Usage

```
bgplot3d(expression, bg.color = getr3dDefaults("bg", "color"),
         magnify = 1, ...)
legend3d(...)
```

Arguments

`expression`	Any plotting commands to produce a plot.
`bg.color`	The color to use for the background.
`magnify`	Multiplicative factor to apply to size of window when producing background plot.
`...`	For `legend3d`, arguments to pass to `bgplot3d` or `legend`; for `bgplot3d`, arguments to pass to `bg3d`.

Details

The `bgplot3d` function opens a `png` device and executes `expression`, producing a plot there. This plot is then used as a bitmap background for the current RGL subscene.

The `legend3d` function draws a standard 2D legend to the background of the current subscene by calling `bgplot3d` to open a device, and setting up a plot region there to fill the whole display.

Value

The `bgplot3d` function invisibly returns the ID of the background object that was created, with attribute `"value"` holding the value returned when the `expression` was evaluated.

The `legend3d` function does similarly. The `"value"` attribute is the result of the call to `legend`. The scaling of the coordinates runs from 0 to 1 in X and Y.

Note

Because the background plots are drawn as bitmaps, they do not resize very gracefully. It's best to size your window first, then draw the background at that size.

Author(s)

Duncan Murdoch

See Also

bg3d for other background options.

Examples

```
x <- rnorm(100)
y <- rnorm(100)
z <- rnorm(100)
open3d()
# Needs to be a bigger window than the default
par3d(windowRect = c(100, 100, 612, 612))
parent <- currentSubscene3d()
mfrow3d(2, 2)
plot3d(x, y, z)
next3d(reuse = FALSE)
bgplot3d(plot(y, z))
next3d(reuse = FALSE)
bgplot3d(plot(x, z))
next3d(reuse = FALSE)
legend3d("center", c("2D Points", "3D Points"), pch = c(1, 16))
useSubscene3d(parent)
```

Buffer *R6 Class for binary buffers in glTF files.*

Description

These files typically have one buffer holding all the binary data for a scene.

Methods

Public methods:

- Buffer$new()
- Buffer$load()
- Buffer$saveOpenBuffer()
- Buffer$getBuffer()
- Buffer$setBuffer()
- Buffer$openBuffer()

- Buffer$writeBuffer()
- Buffer$closeBuffer()
- Buffer$closeBuffers()
- Buffer$getBufferview()
- Buffer$addBufferView()
- Buffer$openBufferview()
- Buffer$setBufferview()
- Buffer$getAccessor()
- Buffer$setAccessor()
- Buffer$readAccessor()
- Buffer$readAccessor0()
- Buffer$addAccessor()
- Buffer$dataURI()
- Buffer$as.list()
- Buffer$clone()

Method `new()`**:**

Usage:

```
Buffer$new(json = NULL, binfile = NULL)
```

Arguments:

`json` list read from glTF file.

`binfile` optional External binary filename, or raw vector

Method `load()`**:** Load from file.

Usage:

```
Buffer$load(uri, buf = 0)
```

Arguments:

`uri` Which file to load.

`buf` Which buffer number to load.

Method `saveOpenBuffer()`**:** Write open buffer to connection.

Usage:

```
Buffer$saveOpenBuffer(con, buf = 0)
```

Arguments:

`con` Output connection.

`buf` Buffer number.

Method `getBuffer()`**:** Get buffer object.

Usage:

```
Buffer$getBuffer(buf, default = list(byteLength = 0))
```

Arguments:

`buf` Buffer number.

`default` Default buffer object if `buf` not found.

Returns: A list containing components described here: https://registry.khronos.org/glTF/specs/2.0/glTF-2.0.html#reference-buffer.

Method `setBuffer()`**:** Set buffer object.

Usage:

```
Buffer$setBuffer(buf, buffer)
```

Arguments:

`buf` Buffer number.

`buffer` New value to insert.

Method `openBuffer()`**:** Open a connection for the data in a buffer.

Usage:

```
Buffer$openBuffer(buf)
```

Arguments:

`buf` Buffer number.

Returns: An open binary connection.

Method `writeBuffer()`**:** Write data to buffer.

Usage:

```
Buffer$writeBuffer(values, type, size, buf = 0)
```

Arguments:

`values` Values to write.

`type` Type to write.

`size` Byte size of each value.

`buf` Which buffer to write to.

Returns: Byte offset of start of bytes written.

Method `closeBuffer()`**:** Close the connection in a buffer.

If there was a connection open, this will save the contents in the raw vector `bytes` within the buffer object.

Usage:

```
Buffer$closeBuffer(buf)
```

Arguments:

`buf` The buffer number.

Method `closeBuffers()`**:** Close any open buffers.

Call this after working with a GLTF file to avoid warnings from R about closing unused connections.

Usage:

```
Buffer$closeBuffers()
```

Method `getBufferview()`**:** Get `bufferView` object.

Usage:

```
Buffer$getBufferview(bufv)
```

Arguments:

`bufv` `bufferView` number.

Returns: A list containing components described here: https://registry.khronos.org/glTF/specs/2.0/glTF-2.0.html#reference-bufferview.

Method `addBufferView()`**:** Add a new buffer view.

Usage:

```
Buffer$addBufferView(values, type, size, target = NULL, buf = 0)
```

Arguments:

`values` Values to put in the view.
`type` Type of values.
`size` Size of values in bytes.
`target` Optional target use for values.
`buf` Which buffer to write to.

Returns: New `bufferView` number.

Method `openBufferview()`**:** Open a connection to a buffer view.

Usage:

```
Buffer$openBufferview(bufv)
```

Arguments:

`bufv` Which `bufferView`.

Returns: A connection.

Method `setBufferview()`**:** Set `bufferView` object.

Usage:

```
Buffer$setBufferview(bufv, bufferView)
```

Arguments:

`bufv` `bufferView` number.
`bufferView` New value to insert.

Method `getAccessor()`**:** Get accessor object

Usage:

```
Buffer$getAccessor(acc)
```

Arguments:

`acc` Accessor number

Returns: A list containing components described here: https://registry.khronos.org/glTF/specs/2.0/glTF-2.0.html#reference-accessor

Method `setAccessor()`**:** Set accessor object.

Usage:

```
Buffer$setAccessor(acc, accessor)
```

Arguments:

`acc` Accessor number.

`accessor` New value to insert.

Method `readAccessor()`**:** Read data given by accessor number.

Usage:

```
Buffer$readAccessor(acc)
```

Arguments:

`acc` Accessor number.

Returns: A vector or array as specified in the accessor. For the `MATn` types, the 3rd index indexes the element.

Method `readAccessor0()`**:** Read data given by accessor object.

Usage:

```
Buffer$readAccessor0(accessor)
```

Arguments:

`accessor` Accessor object

Returns: A vector or array as specified in the accessor. For the `MATn` types, the 3rd index indexes the element.

Method `addAccessor()`**:** Write values to accessor, not including `min` and `max`.

Usage:

```
Buffer$addAccessor(
  values,
  target = NULL,
  types = "anyGLTF",
  normalized = FALSE
)
```

Arguments:

`values` Values to write.

`target` Optional target use for values.

`types` Which types can be used?

`normalized` Are normalized integers allowed?

`useDouble` Whether to write doubles or singles.

Returns: New accessor number

Method `dataURI()`**:** Convert buffer to data URI.

Usage:

```
Buffer$dataURI(buf = 0)
```

Arguments:

`buf` Buffer to convert.

Returns: String containing data URI.

Method `as.list()`**:** Convert to list.

Usage:

```
Buffer$as.list()
```

Returns: List suitable for writing using JSON.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Buffer$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`callbacks` *User callbacks on mouse events*

Description

Set and get user callbacks on mouse events.

Usage

```
rgl.setMouseCallbacks(button, begin = NULL, update = NULL, end = NULL,
                      dev = cur3d(), subscene = currentSubscene3d(dev))
rgl.getMouseCallbacks(button,
                      dev = cur3d(), subscene = currentSubscene3d(dev))

rgl.setWheelCallback(rotate,
                      dev = cur3d(), subscene = currentSubscene3d(dev))

rgl.getWheelCallback(dev = cur3d(), subscene = currentSubscene3d(dev))
```

Arguments

`button`	Which button? Use 1 for left, 2 for right, 3 for middle, 4 for wheel. Use 0 to set an action when no button is pressed.
`begin`	Called when mouse down event occurs
`update`	Called when mouse moves
`end`	Called when mouse is released
`rotate`	Called when mouse wheel is rotated
`dev, subscene`	The RGL device and subscene to work with

Details

The set functions set event handlers on mouse events that occur within the current RGL window. The `begin` and `update` events should be functions taking two arguments; these will be the mouse coordinates when the event occurs. The `end` event handler takes no arguments. The `rotate` event takes a single argument, which will be equal to `1` if the user pushes the wheel away by one click, and `2` if the user pulls the wheel by one click.

Alternatively, the handlers may be set to `NULL`, the default value, in which case no action will occur.

If a subscene has multiple listeners, the user action will still only be called for the subscene that received the mouse event. It should consult `par3d("listeners")` if it makes sense to take action on the whole group of subscenes.

The get function retrieves the callbacks that are currently set.

The "no button" mouse handler may be set by specifying `button = 0`. The `begin` function will be called the first time the mouse moves within the subscene, and the `update` function will be called repeatedly as it moves. The `end` function will never be called.

Value

The set functions are called for the side effect of setting the mouse event handlers.

The `rgl.getMouseCallbacks` function returns a list containing the callback functions or `NULL` if no user callback is set. The `rgl.getWheelCallback` returns the callback function or `NULL`.

Author(s)

Duncan Murdoch

See Also

`par3d` to set built-in handlers, `setUserCallbacks` to work with `rglwidget`.

Examples

```
pan3d <- function(button, dev = cur3d(), subscene = currentSubscene3d(dev)) {
  start <- list()

  begin <- function(x, y) {
    activeSubscene <- par3d("activeSubscene", dev = dev)
    start$listeners <<- par3d("listeners", dev = dev, subscene = activeSubscene)
    for (sub in start$listeners) {
      init <- par3d(c("userProjection","viewport"), dev = dev, subscene = sub)
      init$pos <- c(x/init$viewport[3], 1 - y/init$viewport[4], 0.5)
      start[[as.character(sub)]] <<- init
    }
  }

  update <- function(x, y) {
    for (sub in start$listeners) {
      init <- start[[as.character(sub)]]
      xlat <- 2*(c(x/init$viewport[3], 1 - y/init$viewport[4], 0.5) - init$pos)
      mouseMatrix <- translationMatrix(xlat[1], xlat[2], xlat[3])
```

```
    par3d(userProjection = mouseMatrix %*% init$userProjection, dev = dev, subscene = sub )
     }
   }
   rgl.setMouseCallbacks(button, begin, update, dev = dev, subscene = subscene)
  cat("Callbacks set on button", button, "of RGL device", dev, "in subscene", subscene, "\n")
}
open3d()
shade3d(icosahedron3d(), col = "yellow")
# This only works in the internal display...
pan3d(1)
```

---

checkDeldir *Check for a compatible version of* **deldir**

Description

Version 1.0-2 of **deldir** is not compatible with **rgl**. This allows code to avoid trying to call it.

Usage

```
checkDeldir(error = FALSE)
```

Arguments

error	If TRUE, stop with an error.

Value

Returns TRUE if **deldir** is available in a compatible version.

Examples

```
checkDeldir()
```

---

clipMesh3d *Clip mesh or RGL object to general region*

Description

Modifies a mesh3d object so that values of a function are bounded.

Usage

```
clipMesh3d(mesh, fn, bound = 0, greater = TRUE,
           minVertices = 0, plot = FALSE, keepValues = FALSE,
           keepTags = FALSE)
clipObj3d(ids = tagged3d(tags), fn, bound = 0, greater = TRUE,
           minVertices = 0,
           replace = TRUE, tags)
```

Arguments

mesh	A mesh3d object.
fn	A function used to determine clipping, or a vector of values from such a function, with one value per vertex.
bound	The value(s) of fn on the clipping boundary.
greater	Logical; whether to keep fn >= bound or not.
minVertices	See Details below.
plot	Logical; whether or not to plot the mesh.
keepValues	Logical; whether to save the function values at each vertex when plot = FALSE.
keepTags	Whether to keep the "tags" component of the result; see details below.
ids	The RGL id value(s) of objects to clip.
tags	Object tags; an alternate way to specify ids. Ignored if ids is given.
replace	Should the ids objects be deleted after the clipped ones are drawn?

Details

These functions transform a mesh3d object or other RGL objects by removing parts where fn violates the bound.

For clipMesh3d the fn argument can be any of the following:

- a character vector naming a function (with special names "x", "y", and "z" corresponding to functions returning those coordinates)
- a function
- a numeric vector with one value per vertex
- NULL, indicating that the numeric values are saved in mesh$values

For clipObj3d any of the above except NULL may be used.

If fn is a numeric vector, with one value per vertex, those values will be used in the test. If it is a function with formal arguments x, y and z, it will receive the coordinates of vertices in those arguments, otherwise it will receive the coordinates in a single n x 3 matrix. The function should be vectorized and return one value per vertex, to check against the bound.

These operations are performed on the mesh:

First, all quads are converted to triangles.

Next, each vertex is checked against the condition.

Modifications to triangles depend on how many of the vertices satisfy the condition (fn >= bound or fn <= bound, depending on greater) for inclusion.

- If no vertices in a triangle satisfy the condition, the triangle is omitted.
- If one vertex satisfies the condition, the other two vertices in that triangle are shrunk towards it by assuming fn is locally linear.
- If two vertices satisfy the condition, the third vertex is shrunk along each edge towards each other vertex, forming a quadrilateral made of two new triangles.

- If all vertices satisfy the condition, they are included with no modifications.

Modifications to line segments are similar: the segment will be shortened if it crosses the boundary, or omitted if it is entirely out of bounds. Points, spheres, text and sprites will just be kept or rejected.

The `minVertices` argument is used to improve the approximation to the boundary when `fn` is a non-linear function. In that case, the interpolation described above can be inaccurate. If `minVertices` is set to a positive number (e.g. `10000`), then each object is modified by subdivision to have at least that number of vertices, so that pieces are smaller and the linear interpolation is more accurate. In the `clipObj3d` function, `minVertices` can be a vector, with entries corresponding to each of the entries in `ids`.

Value

If `plot = FALSE`, `clipMesh3d` returns new mesh3d object in which all vertices (approximately) satisfy the clipping condition. Note that the order of vertices will likely differ from the original order, and new vertices will be added near the boundary (and if `minVertices > 0`, in the interior). If in addition `keepValues = TRUE`, a component named `"values"` will be added to the mesh containing the values for each vertex. If `keepTags = TRUE`, the `tags` component described below will be added to the output mesh.

If `plot = TRUE`, the result will be plotted with shade3d and its result returned.

`clipObj3d` is called for the side effect of modifying the scene. It returns a list of new RGL id values corresponding to the `ids` passed as arguments.

The `keepTags` argument

If `keepTags = TRUE`, a `"tags"` element will be added to the result. It will be a vector with one entry per point, segment, triangle and quad in the output mesh. (These tags are not related to the tags used to identify **rgl** objects.) The mesh tags may be used to show the correspondence between the parts of the input mesh and output mesh. By default, the tags are constructed as a numerical sequence over points, segments, triangles and quads in the input mesh, in that order, starting from one. This is the same order used for colours when shading with `meshColor == "faces"`.

For example, start with a mesh with one point, two segments, three triangles and four quads, but no `tags` member. It would implicitly tag the parts from one to ten as

```
c(1,    # the point
  2:3,  # the two segments
  4:6,  # the three triangles
  7:10) # the four quads
```

If clipping deleted the segments and the first triangle, the output would contain the seven element result

```
mesh$tags <- c(1,    # the point remains
                     # no segments now
               5:6,  # the two remaining triangles
                     # were previously items 5 and 6
               7:10) # the four quads
```

The `tags` output may contain repetitions. For example, when a triangle is partially clipped and replaced by several smaller triangles, entries for all of them will contain the value corresponding to the original triangle.

The `mesh$tags` component may be supplied as part of the input mesh as any type of vector; the output will propagate values to the new mesh. The input length must match the total number of points, segments, triangles and quads in the input mesh or an error will be raised.

Author(s)

Duncan Murdoch

References

See https://stackoverflow.com/q/56242470/2554330 and https://laustep.github.io/stlahblog/posts/MeshClipping.html for a motivating example.

See Also

See `contourLines3d` and `filledContour3d` for ways to display function values without clipping.

Examples

```
# Show the problem that minVertices solves:

cube <- cube3d(col = rainbow(6), meshColor = "faces")

# This function only has one argument, so it will
# be passed x, y and z in columns of a matrix
vecnorm <- function(vals) apply(vals, 1, function(row) sqrt(sum(row^2)))

open3d()
mfrow3d(2, 2, sharedMouse = TRUE)
id1 <- shade3d(cube)
# All vertices have norm sqrt(3), so this clips nothing:
clipObj3d(id1, fn = vecnorm, bound = sqrt(2))
next3d()
id2 <- wire3d(cube, lit = FALSE)
clipObj3d(id2, fn = vecnorm, bound = sqrt(2))

# This subdivides the cube, and does proper clipping:
next3d()
id3 <- shade3d(cube)
clipObj3d(id3, fn = vecnorm, bound = sqrt(2), minVertices = 200)
next3d()
id4 <- wire3d(cube, lit = FALSE)
clipObj3d(id4, fn = vecnorm, bound = sqrt(2), minVertices = 200)
```

clipplaneControl *Sets attributes of a clipping plane*

Description

This is a function to produce actions in a web display. A `playwidget` or Shiny input control (e.g. a `sliderInput` control) sets a value which controls attributes of one or more clipping planes.

Usage

```
clipplaneControl(a = NULL, b = NULL, c = NULL, d = NULL,
                 plane = 1, clipplaneids = tagged3d(tag), tag, ...)
```

Arguments

a, b, c, d	Parameter values for the clipping planes.
plane	Which plane in the clipplane object?
clipplaneids	The id of the clipplane object.
tag	Select clipplane with matching tag. Ignored if `clipplaneid` is specified.
...	Other parameters passed to `propertyControl`.

Value

A list of class `"rglControl"` of cleaned up parameter values, to be used in an RGL widget.

Author(s)

Duncan Murdoch

See Also

The User Interaction in WebGL vignette gives more details.

Examples

```
open3d()
saveopts <- options(rgl.useNULL = TRUE)
xyz <- matrix(rnorm(300), ncol = 3)
id <- plot3d(xyz, type="s", col = "blue", zlim = c(-3,3))["clipplanes"]
dvals <- c(3, -3)
widget <- rglwidget() %>%
  playwidget(clipplaneControl(d = dvals, clipplaneids = id),
             start = 0, stop = 1, step = 0.01,
             rate = 0.5)
if (interactive() || in_pkgdown_example())
  widget
options(saveopts)
```

contourLines3d *Draw contours on a surface*

Description

contourLines3d draws contour lines on a surface; filledContour3d draws filled contours on it.

Usage

```
contourLines3d(obj, ...)
## S3 method for class 'rglId'
contourLines3d(obj, ...)
## S3 method for class 'mesh3d'
contourLines3d(obj, fn = "z",
    nlevels = 10,
    levels = NULL,
    minVertices = 0,
    plot = TRUE, ... )
filledContour3d(obj, ...)
## S3 method for class 'rglId'
filledContour3d(obj, plot = TRUE, replace = plot, ...)
## S3 method for class 'mesh3d'
filledContour3d(obj, fn = "z",
    nlevels = 20,
    levels = pretty(range(values), nlevels),
    color.palette = function(n) hcl.colors(n, "YlOrRd", rev = TRUE),
    col = color.palette(length(levels) - 1),
    minVertices = 0,
    plot = TRUE,
    keepValues = FALSE, ... )
```

Arguments

obj	The object(s) on which to draw contour lines.
fn	The function(s) to be contoured. See Details.
nlevels	Suggested number of contour levels if levels is not given.
levels	Specified contour values.
minVertices	See Details below.
plot	Whether to draw the lines or return them in a dataframe.
...	For the "mesh3d" methods, additional parameters to pass to segments3d when drawing the contour lines or to shade3d when drawing the filled contours. For the "rglId" methods, additional parameters to pass to the "mesh3d" methods.
replace	Whether to delete the objects that are being contoured.
color.palette	a color palette function to assign colors in the plot
col	the actual colors to use in the plot.
keepValues	whether to save the function values at each vertex when plot = FALSE

Details

For contourLines3d, the fn argument can be any of the following:

- a character vector naming one or more functions
- a function
- a numeric vector with one value per vertex
- NULL, indicating that the numeric values are saved in obj$values
- a list containing any of the above.

For filledContour3d, only one function can be specified.

The special names "x", "y", "z" may be used in fn to specify functions returning one of those coordinates. (If you have existing functions x(), y() or z() they will be masked by this choice; specify such functions by value rather than name, e.g. fn = x instead of fn = "x".)

Functions in fn with formal arguments x, y and z will receive the coordinates of vertices in those arguments, otherwise they will receive the coordinates in a single n x 3 matrix. They should be vectorized and return one value per vertex.

Each of the functions will be evaluated at each vertex of the surface specified by obj, and contours will be drawn assuming the function is linear between vertices. If contours of a nonlinear function are needed, you may want to increase minVertices as described below.

If levels is not specified, values will be set separately for each entry in fn, using pretty(range(values, na.rm = TRUE), nlevels) where values are the values on the vertices.

The minVertices argument is used to improve the approximation to the contour when the function is non-linear. In that case, the interpolation between vertices can be inaccurate. If minVertices is set to a positive number (e.g. 10000), then the mesh is modified by subdivision to have at least that number of vertices, so that pieces are smaller and the linear interpolation is more accurate.

Value

For both contourLines3d and filledContour3d the "rglId" method converts the given id values to a mesh, and calls the "mesh3d" method.

The "mesh3d" method returns an object of class "rglId" corresponding to what was drawn if plot is TRUE,

If plot is FALSE, contourLines3d returns a dataframe containing columns c("x", "y", "z", "fn", "level") giving the coordinates of the endpoints of each line segment, the name (or index) of the function for this contour, and the level of the contour.

If plot is FALSE, filledContour3d returns a "mesh3d" object holding the result. If keepValues is TRUE, the mesh will contain the values corresponding to each vertex (with linear approximations at the boundaries).

Note

To draw contours on a surface, the surface should be drawn with material property polygon_offset = 1 (or perhaps some larger positive value) so that the lines of the contour are not obscured by the surface.

In R versions prior to 3.6.0, the default color.palette is grDevices::cm.colors.

Author(s)

Duncan Murdoch

See Also

The **misc3d** package contains the function `contour3d` to draw contour surfaces in space instead of contour lines on surfaces.

Examples

```
# Add contourlines in "z" to a persp plot

z <- 2 * volcano        # Exaggerate the relief
x <- 10 * (1:nrow(z))   # 10 meter spacing (S to N)
y <- 10 * (1:ncol(z))   # 10 meter spacing (E to W)

open3d()
id <- persp3d(x, y, z, aspect = "iso",
      axes = FALSE, box = FALSE, polygon_offset = 1)
contourLines3d(id)     # "z" is the default function
filledContour3d(id, polygon_offset = 1, nlevels = 10, replace = TRUE)

# Draw longitude and latitude lines on a globe

lat <- matrix(seq(90, -90, length.out = 50)*pi/180, 50, 50, byrow = TRUE)
long <- matrix(seq(-180, 180, length.out = 50)*pi/180, 50, 50)

r <- 6378.1 # radius of Earth in km
x <- r*cos(lat)*cos(long)
y <- r*cos(lat)*sin(long)
z <- r*sin(lat)

open3d()
ids <- persp3d(x, y, z, col = "white",
        texture = system.file("textures/worldsmall.png", package = "rgl"),
        specular = "black", axes = FALSE, box = FALSE, xlab = "", ylab = "", zlab = "",
        normal_x = x, normal_y = y, normal_z = z, polygon_offset = 1)

contourLines3d(ids, list(latitude = function(x, y, z) asin(z/sqrt(x^2+y^2+z^2))*180/pi,
                         longitude = function(x, y, z) atan2(y, x)*180/pi))
```

---

cube3d *Sample 3D mesh objects*

Description

A collection of sample mesh objects.

Usage

```
cube3d(trans = identityMatrix(), ...)
tetrahedron3d(trans = identityMatrix(), ...)
octahedron3d(trans = identityMatrix(), ...)
icosahedron3d(trans = identityMatrix(), ...)
dodecahedron3d(trans = identityMatrix(), ...)
cuboctahedron3d(trans = identityMatrix(), ...)

oh3d(trans = identityMatrix(), ...)    # an 'o' object
```

Arguments

trans	transformation to apply to objects
...	additional parameters to pass to mesh3d

Details

These sample objects optionally take a 4x4 matrix transformation trans as an argument. This transformation is applied to all vertices of the default shape. The default is an identity transformation.

Value

Objects of class c("mesh3d", "shape3d").

See Also

mesh3d

Examples

```
# render all of the Platonic solids
open3d()
shade3d( translate3d( tetrahedron3d(col = "red"), 0, 0, 0) )
shade3d( translate3d( cube3d(col = "green"), 3, 0, 0) )
shade3d( translate3d( octahedron3d(col = "blue"), 6, 0, 0) )
shade3d( translate3d( dodecahedron3d(col = "cyan"), 9, 0, 0) )
shade3d( translate3d( icosahedron3d(col = "magenta"), 12, 0, 0) )
```

---

cylinder3d *Create cylindrical or "tube" plots*

Description

This function converts a description of a space curve into a "mesh3d" object forming a cylindrical tube around the curve.

Usage

```
cylinder3d(center, radius = 1, twist = 0, e1 = NULL, e2 = NULL, e3 = NULL,
 sides = 8, section = NULL, closed = 0,
 rotationMinimizing = is.null(e2) && is.null(e3),
 debug = FALSE, keepVars = FALSE, ...)
```

Arguments

center	An n by 3 matrix whose columns are the x, y and z coordinates of the space curve.
radius	The radius of the cross-section of the tube at each point in the center.
twist	The amount by which the polygon forming the tube is twisted at each point.
e1, e2, e3	The local coordinates to use at each point on the space curve. These default to a rotation minimizing frame or Frenet coordinates.
sides	The number of sides in the polygon cross section.
section	The polygon cross section as a two-column matrix, or NULL.
closed	Whether to treat the first and last points of the space curve as identical, and close the curve, or put caps on the ends. See the Details.
rotationMinimizing	Use a rotation minimizing local frame if TRUE, or a Frenet or user-specified frame if FALSE.
debug	If TRUE, plot the local Frenet coordinates at each point.
keepVars	If TRUE, return the local variables in attribute "vars".
...	Additional arguments to set as material properties.

Details

The number of points in the space curve is determined by the vector lengths in center, after using xyz.coords to convert it to a list. The other arguments radius, twist, e1, e2, and e3 are extended to the same length.

The closed argument controls how the ends of the cylinder are handled. If closed > 0, it represents the number of points of overlap in the coordinates. closed == TRUE is the same as closed = 1. If closed > 0 but the ends don't actually match, a warning will be given and results will be somewhat unpredictable.

Negative values of closed indicate that caps should be put on the ends of the cylinder. If closed == -1, a cap will be put on the end corresponding to center[1,]. If closed == -2, caps will be put on both ends.

If section is NULL (the default), a regular sides-sided polygon is used, and radius measures the distance from the center of the cylinder to each vertex. If not NULL, sides is ignored (and set internally to nrow(section)), and radius is used as a multiplier to the vertex coordinates. twist specifies the rotation of the polygon. Both radius and twist may be vectors, with values recycled to the number of rows in center, while sides and section are the same at every point along the curve.

The three optional arguments e1, e2, and e3 determine the local coordinate system used to create the vertices at each point in center. If missing, they are computed by simple numerical approximations. e1 should be the tangent coordinate, giving the direction of the curve at the point. The cross-section of the polygon will be orthogonal to e1. When rotationMinimizing is TRUE, e2 and e3 are chosen to give a rotation minimizing frame (see Wang et al., 2008). When it is FALSE, e2 defaults to an approximation to the normal or curvature vector; it is used as the image of the y axis of the polygon cross-section. e3 defaults to an approximation to the binormal vector, to which the x axis of the polygon maps. The vectors are orthogonalized and normalized at each point.

Value

A "mesh3d" object holding the cylinder, possibly with attribute "vars" containing the local environment of the function.

Author(s)

Duncan Murdoch

References

Wang, W., Jüttler, B., Zheng, D. and Liu, Y. (2008). Computation of rotation minimizing frames. ACM Transactions on Graphics, Vol. 27, No. 1, Article 2.

Examples

```
# A trefoil knot
open3d()
theta <- seq(0, 2*pi, length.out = 25)
knot <- cylinder3d(
      center = cbind(
        sin(theta) + 2*sin(2*theta),
        2*sin(3*theta),
        cos(theta) - 2*cos(2*theta)),
      e1 = cbind(
        cos(theta) + 4*cos(2*theta),
        6*cos(3*theta),
        sin(theta) + 4*sin(2*theta)),
      radius = 0.8,
      closed = TRUE,
      color = "green")

shade3d(addNormals(subdivision3d(knot, depth = 2)))
```

---

decorate3d *Add decorations to a 3D plot*

Description

decorate3d adds the usual decorations to a plot: labels, axes, etc.

Usage

```
decorate3d(xlim = NULL, ylim = NULL, zlim = NULL,
           xlab = "x", ylab = "y", zlab = "z",
           box = TRUE, axes = TRUE,
           main = NULL, sub = NULL,
           top = TRUE, aspect = FALSE, expand = 1.03,
           tag = material3d("tag"), ...)
```

Arguments

xlim, ylim, zlim	These are used for the labels.
xlab, ylab, zlab	labels for the coordinates.
box, axes	whether to draw a box and axes.
main, sub	main title and subtitle.
top	whether to bring the window to the top when done.
aspect	either a logical indicating whether to adjust the aspect ratio, or a new ratio.
expand	how much to expand the box around the data, if it is drawn.
tag	optional label for objects being produced.
...	ignored.

Value

The RGL id values for those items.

Examples

```
open3d()
shade3d(tetrahedron3d(), col = "red")
decorate3d(main = "A Tetrahedron")
```

---

drape3d *Drape lines over a scene.*

Description

Project a line onto the surface in a scene so that it appears to drape itself onto the surface.

Usage

```
drape3d(obj, ...)
## S3 method for class 'mesh3d'
drape3d(obj, x, y = NULL, z = NULL, plot = TRUE,
                          up = c(0, 0, 1),
                          P = projectDown(up), ...)
## Default S3 method:
drape3d(obj, ...)
```

Arguments

obj	The object(s) upon which to drape lines.
x, y, z	Coordinates of the line segments to be draped. Any reasonable way of defining the coordinates is acceptable. See the function xyz.coords for details.
plot	Should the result be plotted, or returned as a data frame?
up	The direction to consider as "up".
P	The projection to use for draping, a 4x4 matrix.
...	For the "mesh3d" method, additional parameters to pass to segments3d when drawing the draped lines. For the "default" method, additional parameters to pass to the "mesh3d" method.

Details

The default method converts obj to a mesh using as.mesh3d, then uses the "mesh3d" method.

The current implementation constructs the segments to drape across the surface using the same method as lines3d uses: each successive point is joined to the previous one. Use NA coordinates to indicate breaks in the line.

The P matrix is used to project points to a plane as follows: They are transformed by P in homogeneous coordinates, then only first two (Euclidean) coordinates are kept.

Value

If plot = TRUE, plots the result and invisibly returns the object ID of the collection of segments.

If plot = FALSE, returns a matrix containing "x", "y" and "z" values for the line(s) (for use with segments3d),

Author(s)

George Helffrich and Duncan Murdoch

See Also

shadow3d, facing3d

Examples

```
#
# volcano example taken from "persp"
#

z <- 2 * volcano        # Exaggerate the relief

x <- 10 * (1:nrow(z))   # 10 meter spacing (S to N)
y <- 10 * (1:ncol(z))   # 10 meter spacing (E to W)

zlim <- range(z)
zlen <- zlim[2] - zlim[1] + 1
```

```
colorlut <- terrain.colors(zlen) # height color lookup table

col <- colorlut[ z - zlim[1] + 1 ] # assign colors to heights for each point

open3d()
id <- surface3d(x, y, z, color = col, polygon_offset = 1)

segs <- data.frame(x = range(x) + c(100, -100),
                   y = range(y) + c(150, -100), z = 325)
drape3d(id, segs, col = 'yellow', lwd = 3)
lines3d(segs, col='red', lwd=3)

p <- c(350, 205)         # (x,y) of strike & dip reading
off <- 20*c(-1, +1)      # X-marks-the-spot offset
segs <- data.frame(
    x = c(p[1] + off, NA, p[1] + off),
    y = c(p[2] + off, NA, p[2] - off),
    z = rep(350, 5)
    )
drape3d(id, segs, col = "yellow", lwd = 3)
```

`ellipse3d` *Make an ellipsoid*

Description

A generic function and several methods returning an ellipsoid or other outline of a confidence region for three parameters.

Usage

```
ellipse3d(x, ...)
## Default S3 method:
ellipse3d(x, scale = c(1, 1, 1), centre = c(0, 0, 0), level = 0.95,
 t = sqrt(qchisq(level, 3)), which = 1:3, subdivide = 3, smooth = TRUE, ...)
## S3 method for class 'lm'
ellipse3d(x, which = 1:3, level = 0.95, t = sqrt(3 * qf(level,
                                                3, x$df.residual)), ...)
## S3 method for class 'glm'
ellipse3d(x, which = 1:3, level = 0.95, t, dispersion, ...)
## S3 method for class 'nls'
ellipse3d(x, which = 1:3, level = 0.95, t = sqrt(3 * qf(level,
                                                3, s$df[2])), ...)
```

Arguments

x	An object. In the default method the parameter x should be a square positive definite matrix at least 3x3 in size. It will be treated as the correlation or covariance of a multivariate normal distribution.

...	Additional parameters to pass to the default method or to qmesh3d.
scale	If x is a correlation matrix, then the standard deviations of each parameter can be given in the scale parameter. This defaults to c(1, 1, 1), so no rescaling will be done.
centre	The centre of the ellipse will be at this position.
level	The confidence level of a simultaneous confidence region. The default is 0.95, for a 95% region. This is used to control the size of the ellipsoid.
t	The size of the ellipse may also be controlled by specifying the value of a t-statistic on its boundary. This defaults to the appropriate value for the confidence region.
which	This parameter selects which variables from the object will be plotted. The default is the first 3.
subdivide	This controls the number of subdivisions (see subdivision3d) used in constructing the ellipsoid. Higher numbers give a smoother shape.
smooth	If TRUE, smooth interpolation of normals is used; if FALSE, a faceted ellipsoid will be displayed.
dispersion	The value of dispersion to use. If specified, it is treated as fixed, and chi-square limits for t are used. If missing, it is taken from summary(x).

Value

A mesh3d object representing the ellipsoid.

Examples

```
# Plot a random sample and an ellipsoid of concentration corresponding to a 95%
# probability region for a
# trivariate normal distribution with mean 0, unit variances and
# correlation 0.8.
if (requireNamespace("MASS", quietly = TRUE)) {
  Sigma <- matrix(c(10, 3, 0, 3, 2, 0, 0, 0, 1), 3, 3)
  Mean <- 1:3
  x <- MASS::mvrnorm(1000, Mean, Sigma)

  open3d()

  plot3d(x, box = FALSE)

  plot3d( ellipse3d(Sigma, centre = Mean), col = "green", alpha = 0.5, add = TRUE)
}

# Plot the estimate and joint 90% confidence region for the displacement and cylinder
# count linear coefficients in the mtcars dataset

data(mtcars)
fit <- lm(mpg ~ disp + cyl , mtcars)

open3d()
plot3d(ellipse3d(fit, level = 0.90), col = "blue", alpha = 0.5, aspect = TRUE)
```

`expect_known_scene` *Helper for* **testthat** *testing.*

Description

Gets the current scene using `scene3d`, and compares the result to a saved value, optionally closing the window afterwards.

Usage

```
expect_known_scene(name,
                   close = TRUE,
                   file = paste0("testdata/", name, ".rds"),
                   ...)
```

Arguments

`name`	By default, the base name of the file to save results in. Not used if `file` is specified.
`close`	Whether to close the **rgl** window after the comparison.
`file`	The file in which to save the result.
`...`	Other arguments which will be passed to `expect_known_value`.

Details

This function uses `expect_known_value` to save a representation of the scene. During the comparison, the scene is modified so that non-reproducible aspects are standardized or omitted:

- object ids are changed to start at 1.
- system-specific font names and texture names are deleted.
- the window is shifted to the top left of the screen.

Calls to `expect_known_scene()` enable `testthat::local_edition(2)` for the duration of the call, so it will work in **testthat** "3rd edition".

Value

A value describing the changes to the saved object, suitable for use in `test_that()`.

Examples

```
## Not run:
# These lines can be included in testthat::test_that() code.
plot3d(1:10, 1:10, 1:10)
expect_known_scene("plot")

## End(Not run)
```

extrude3d *Generate extrusion mesh*

Description

Given a two-dimensional polygon, this generates a three-dimensional extrusion of the shape by triangulating the polygon and creating a cylinder with that shape as the end faces.

Usage

```
extrude3d(x, y = NULL, thickness = 1, smooth = FALSE, ...)
```

Arguments

x, y	A polygon description in one of the forms supported by triangulate.
thickness	The extrusion will have this thickness.
smooth	logical; should normals be added so that the edges of the extrusion appear smooth?
...	Other parameters to pass to tmesh3d when constructing the mesh.

Details

The extrusion is always constructed with the polygon in the xy plane at z = 0 and another copy at z = thickness. Use the transformation functions (e.g. rotate3d) to obtain other orientations and placements.

Value

A mesh object containing a triangulation of the polygon for each face, and quadrilaterals for the sides.

Author(s)

Duncan Murdoch

See Also

polygon3d for a simple polygon, triangulate for the triangulation, turn3d for a solid of rotation.

Examples

```
x <- c(1:10, 10:1)
y <- rev(c(rep(c(0, 2), 5), rep(c(1.5, -0.5), 5)))
plot(x, y, type = "n")
polygon(x, y)
open3d()
shade3d( extrude3d(x, y), col = "red" )
```

facing3d *Subset an object to parts facing in a particular direction*

Description

facing3d subsets an object by converting it to a triangle mesh, then subsetting to those triangles that are counterclockwise (for front = TRUE) when projected into a plane.

projectDown computes a projection that "looks down" the specified direction.

Usage

```
facing3d(obj, up = c(0, 0, 1),
         P = projectDown(up),
         front = TRUE, strict = TRUE)
projectDown(up)
```

Arguments

obj	An object that can be converted to a triangular mesh object.
up	The direction that is to be considered "up". It may be either a 3 vector in Euclidean coordinates or a 4 vector in homogeneous coordinates.
P	The projection to use for draping, a 4x4 matrix. See drape3d for details on how P is used.
front	If front = TRUE, retains triangles that are counterclockwise after projection by P, otherwise retains those that are clockwise.
strict	If TRUE, drops indeterminate triangles (those that are annihilated by P).

Details

By default the returned subset will be those triangles whose upper side matches front. Change up or use an arbitrary projection for different subsets.

drape3d and shadow3d project objects onto meshes; these functions can be used to project only onto the top or front.

Value

facing3d returns a mesh object made of those triangles which face in the desired direction.

projectDown computes a 4x4 matrix. The first two coordinates of asEuclidean(x %*% projectDown(up)) give a projection of x from above into a plane, where up determines which direction is taken to be "up".

See Also

drape3d, shadow3d

Examples

```
open3d()
d <- rnorm(3)
d <- d/sqrt(sum(d^2))
shade3d( facing3d( icosahedron3d(), up = d, strict = FALSE),
         col = "yellow")
wire3d( facing3d( icosahedron3d(), up = d, front = FALSE),
         col = "black")
# Show the direction:
arrow3d(-2*d , -d)
```

`figWidth` *Get R Markdown figure dimensions in pixels*

Description

In an R Markdown document, figure dimensions are normally specified in inches; these are translated into pixel dimensions when HTML output is requested and `rglwidget` is used. These functions reproduce that translation.

Usage

```
figWidth()
figHeight()
```

Value

When used in an R Markdown document, these functions return the requested current dimensions of figures in pixels. Outside such a document, `NULL` is returned.

Author(s)

Duncan Murdoch

Examples

```
# No useful return value outside of R Markdown:
figWidth()
figHeight()
```

getBoundary3d *Extract the boundary of a mesh*

Description

Constructs a mesh of line segments corresponding to non-shared (i.e. boundary) edges of triangles or quads in the original mesh.

Usage

```
getBoundary3d(mesh, sorted = FALSE, simplify = TRUE, ...)
```

Arguments

mesh	A mesh object.
sorted	Whether the result should have the segments sorted in sequential order.
simplify	Whether to simplify the resulting mesh, dropping all unused vertices. If FALSE, the vertices of the result will be identical to the vertices of mesh; if TRUE, they will likely be different, even if no vertices were dropped.
...	Material properties to apply to the mesh.

Value

A "mesh3d" object containing 0 or more segments.

Author(s)

Duncan Murdoch

See Also

mesh3d

Examples

```
x <- cube3d(col = "blue")
x$ib <- x$ib[,-(1:2)]
b <- getBoundary3d(x, sorted = TRUE, col = "black")

open3d()
shade3d(x, alpha=0.2)

shade3d(b)

# Show edge vertices in sequence:
text3d(t(b$vb), texts = 1:ncol(b$vb), adj = 0)
c(b$is[1,1], b$is[2,])
```

gltfTypes *Names of glTF types.*

Description

The glTF specification packs data into buffers, labelling the type of each part with an integer. The first six values in `gltfTypes` are the integers used there, with `"int"` and `"double"` added for completeness; those values are taken from OpenGL header files.

Usage

```
gltfTypes
```

Format

`gltfTypes` is simply a named vector containing integer values.

Details

These are used in the `Buffer` object.

References

https://registry.khronos.org/glTF/specs/2.0/glTF-2.0.html#_accessor_componenttype

Examples

```
gltfTypes
```

GramSchmidt *The Gram-Schmidt algorithm*

Description

Generate a 3x3 orthogonal matrix using the Gram-Schmidt algorithm.

Usage

```
GramSchmidt(v1, v2, v3, order = 1:3)
```

Arguments

`v1`, `v2`, `v3`	Three length 3 vectors (taken as row vectors).
`order`	The precedence order for the vectors; see Details.

Details

This function orthogonalizes the matrix `rbind(v1, v2, v3)` using the Gram-Schmidt algorithm. It can handle rank 2 matrices (returning a rank 3 matrix). If the original is rank 1, it is likely to fail.

The `order` vector determines the precedence of the original vectors. For example, if it is `c(i, j, k)`, then row `i` will be unchanged (other than normalization); row `j` will normally be transformed within the span of rows `i` and `j`. Row `k` will be transformed orthogonally to the span of the others.

Value

A 3x3 matrix whose rows are the orthogonalization of the original row vectors.

Author(s)

Duncan Murdoch

Examples

```
# Proceed through the rows in order
print(A <- matrix(rnorm(9), 3, 3))
GramSchmidt(A[1, ], A[2, ], A[3, ])

# Keep the middle row unchanged
print(A <- matrix(c(rnorm(2), 0, 1, 0, 0, rnorm(3)), 3, 3, byrow = TRUE))
GramSchmidt(A[1, ], A[2, ], A[3, ], order = c(2, 1, 3))
```

---

grid3d *Add a grid to a 3D plot*

Description

This function adds a reference grid to an RGL plot.

Usage

```
grid3d(side, at = NULL, col = "gray", lwd = 1, lty = 1, n = 5)
```

Arguments

`side`	Where to put the grid; see the Details section.
`at`	How to draw the grid; see the Details section.
`col`	The color of the grid lines.
`lwd`	The line width of the grid lines. (Currently only `lty = 1` is supported.)
`lty`	The line type of the grid lines.
`n`	Suggested number of grid lines; see the Details section.

Details

This function is similar to grid in classic graphics, except that it draws a 3D grid in the plot.

The grid is drawn in a plane perpendicular to the coordinate axes. The first letter of the side argument specifies the direction of the plane: "x", "y" or "z" (or uppercase versions) to specify the coordinate which is constant on the plane.

If at = NULL (the default), the grid is drawn at the limit of the box around the data. If the second letter of the side argument is "-" or is not present, it is the lower limit; if "+" then at the upper limit. The grid lines are drawn at values chosen by pretty with n suggested locations. The default locations should match those chosen by axis3d with nticks = n.

If at is a numeric vector, the grid lines are drawn at those values.

If at is a list, then the "x" component is used to specify the x location, the "y" component specifies the y location, and the "z" component specifies the z location. Missing components are handled using the default as for at = NULL.

Multiple grids may be drawn by specifying multiple values for side or for the component of at that specifies the grid location.

Value

A vector or matrix of object ids is returned invisibly.

Note

If the scene is resized, the grid will not be resized; use abclines3d to draw grid lines that will automatically resize.

Author(s)

Ben Bolker and Duncan Murdoch

See Also

axis3d

Examples

```
x <- 1:10
y <- 1:10
z <- matrix(outer(x - 5, y - 5) + rnorm(100), 10, 10)
open3d()
persp3d(x, y, z, col = "red", alpha = 0.7, aspect = c(1, 1, 0.5))
grid3d(c("x", "y+", "z"))
```

hover3d *Display hover info in plot.*

Description

This adds text to identify points within a plot when the mouse is near them.

Usage

```
hover3d(x, y = NULL, z = NULL,
        labeller = NULL,
        tolerance = 20,
        persist = c("no", "one", "yes"),
        labels = seq_along(x),
        adj = c(-0.2, 0.5),
        scene = scene3d(minimal = FALSE),
        applyToScene = TRUE,
        ...)
```

Arguments

x, y, z	Coordinates of point to identify. Any reasonable way of defining the coordinates is acceptable. See the function xyz.coords for details. Alternatively, x may be the id of a single existing object, and its vertices will be used.
labeller	A function to display information about identified points. NULL indicates the default function, described in Details.
tolerance	How close (in pixels) the mouse should be to a point to display the information.
persist	Should the label persist? If "no" (the default), it will be removed when the mouse moves away. If "one", it will be removed when another point is closer to the mouse. If "yes", it will not be removed.
labels	If the default labeller is used, these labels will be displayed.
adj	If the default labeller is used, this adjustment will be passed to text3d to display the labels.
scene, applyToScene	Arguments to pass to setUserCallbacks. The applyToDev argument to that function is always TRUE.
...	Additional arguments that will be passed to the labeller.

Details

If specified, the labeller argument should specify a function with arguments compatible with function(index, ...). It will be called with index being the index of the point that was selected. It should plot the label, and return the **rgl** ids of the objects that were plotted.

When applyToScene is TRUE, all labels or labelling objects will be created and attached to the scene. You may want to delete them (using the ids returned in idverts and idtexts) once rglwidget has been called, as they serve no purpose in the current device.

Only one hover handler is supported per scene or device.

Value

A lowlevel vector of ids is returned invisibly. If applyToScene is TRUE, it will contain the ids of the temporary objects created for Javascript. It will also have these attributes:

oldPar	Values of par3d parameters that were changed. Currently only "mouseMode".
oldDev	The value of cur3d() at the time of calling, so that oldPar can be restored to the right device.

Author(s)

Duncan Murdoch

See Also

identify3d and selectpoints3d work in the **rgl** device and return information about the selections. setUserCallbacks is the underlying function used by hover3d.

Examples

```
# Create a labeller to show the coordinates of the selected point.
labelLocation <- function(x, y = NULL, z = NULL) {
  xyz <- xyz.coords(x, y, z)
  function(sel, ...) {
    p <- with(xyz, matrix(c(x[sel], y[sel], z[sel]), ncol = 3))
    c(text3d(p, texts = sprintf("x:%.2f", p[1]),
                 adj = c(-0.2, -0.6), ...),
      text3d(p, texts = sprintf("y:%.2f", p[2]),
                 adj = c(-0.2, 0.5), ...),
      text3d(p, texts = sprintf("z:%.2f", p[3]),
                 adj = c(-0.2, 1.6), ...))
  }
}

xyz <- matrix(rnorm(30), ncol = 3)
open3d()
ids <- plot3d(xyz)
hover3d(xyz, labeller = labelLocation(xyz), col = "red", cex = 0.8)
# The same thing using the data id:
# hover3d(ids["data"],
#         labeller = labelLocation(rgl.attrib(ids["data"], "vertices")),
#         col = "red", cex = 0.8)
```

`identify3d` *Identify points in plot*

Description

Identify points in a plot, similarly to the `identify` function in base graphics.

Usage

```
identify3d(x, y = NULL, z = NULL, labels = seq_along(x), n = length(x),
 plot = TRUE, adj = c(-0.1, 0.5), tolerance = 20,
 buttons = c("right", "middle"))
```

Arguments

`x`, `y`, `z`	coordinates of points in a scatter plot. Alternatively, any object which defines coordinates (see `xyz.coords`) can be given as `x`, and `y` and `z` left missing.
`labels`	an optional character vector giving labels for the points. Will be coerced using `as.character`, and recycled if necessary to the length of `x`.
`n`	the maximum number of points to be identified.
`plot`	logical: if `plot` is `TRUE`, the labels are printed near the points and if `FALSE` they are omitted.
`adj`	numeric vector to use as `adj` parameter to `text3d` when plotting the labels.
`tolerance`	the maximal distance (in pixels) for the pointer to be 'close enough' to a point.
`buttons`	a length 1 or 2 character vector giving the buttons to use for selection and quitting.

Details

If `buttons` is length 1, the user can quit by reaching `n` selections, or by hitting the escape key, but the result will be lost if escape is used.

Value

A vector of selected indices.

Author(s)

Duncan Murdoch

See Also

`identify` for base graphics, `select3d` for selecting regions.

import *Imported from magrittr*

Description

This object is imported from **magrittr**. Follow the link to its documentation.

magrittr `%>%`

Pipes can be used to string together `rglwidget` calls and `playwidget` calls. See `ageControl` for an example.

in_pkgdown_example *Are we running in* **pkgdown** *or a* **pkgdown** *example?*

Description

This is mainly for internal use to decide whether results should be automatically included in a **pkgdown** web page.

See the Using RGL in pkgdown web sites vignette for details about using **pkgdown**.

Usage

```
in_pkgdown()
in_pkgdown_example()
```

Value

`TRUE` or `FALSE`

Examples

```
in_pkgdown_example()
```

`light` *Add light source*

Description

add a light source to the scene.

Usage

```
light3d(theta=0, phi=15,
        x=NULL, y = NULL, z = NULL,
        viewpoint.rel = TRUE,
        ambient = "#FFFFFF",
        diffuse = "#FFFFFF",
        specular = "#FFFFFF")
```

Arguments

`theta`, `phi`	direction to infinitely distant light
`x`, `y`, `z`	position of finitely distant light
`viewpoint.rel`	logical, if TRUE light is a viewpoint light that is positioned relative to the current viewpoint
`ambient`, `diffuse`, `specular`	light color values used for lighting calculation

Details

Up to 8 light sources are supported. They are positioned either in world space or relative to the camera. By providing polar coordinates to `theta` and `phi` a directional light source is used. If numerical values are given to x, y and z, a point-like light source with finite distance to the objects in the scene is set up.

If `x` is non-null, `xyz.coords` will be used to form the location values, so all three coordinates can be specified in `x`.

If no lights have been added to a subscene, lights from the parent subscene will be used.

See `material3d` for a discussion of how the components of the light affect the display of objects.

Value

This function is called for the side effect of adding a light. A light ID is returned to allow `pop3d` to remove it.

See Also

`clear3d` `pop3d`

Examples

```
#
# a lightsource moving through the scene
#
data(volcano)
z <- 2 * volcano # Exaggerate the relief
x <- 10 * (1:nrow(z)) # 10 meter spacing (S to N)
y <- 10 * (1:ncol(z)) # 10 meter spacing (E to W)
zlim <- range(z)
zlen <- zlim[2] - zlim[1] + 1
colorlut <- terrain.colors(zlen) # height color lookup table
col <- colorlut[ z - zlim[1] + 1 ] # assign colors to heights for each point

open3d()
bg3d("gray50")
surface3d(x, y, z, color = col, back = "lines")
r <- max(y) - mean(y)
lightid <- spheres3d(1, 1, 1, alpha = 0)
frame <- function(time) {
    a <- pi*(time - 1)
    save <- par3d(skipRedraw = TRUE)
    clear3d(type = "lights")
    pop3d(id = lightid)
    xyz <- matrix(c(r*sin(a) + mean(x), r*cos(a) + mean(y), max(z)), ncol = 3)
    light3d(x = xyz, diffuse = "gray75",
            specular = "gray75", viewpoint.rel = FALSE)
    light3d(diffuse = "gray10", specular = "gray25")
    lightid <<- spheres3d(xyz, emission = "white", radius = 4)
    par3d(save)
    Sys.sleep(0.02)
    NULL
}
play3d(frame, duration = 2)
```

---

makeDependency *Process Javascript for HTML dependency*

Description

A utility function to help in development of internal Javascript code, this function processes the Javascript to minify it and report on errors and bad style.

Usage

```
makeDependency(name, src, script = NULL, package,
               version = packageVersion(package),
               minifile = paste0(basename(src), ".min.js"),
               debugging = FALSE, ...)
```

Arguments

name, src, script, package, version, ...	Arguments to pass to htmltools::htmlDependency.
minifile	Basename of minified file.
debugging	See details below.

Details

This is a utility function used by RGL to process its Javascript code used when displaying rglwidget values. It may be helpful in other packages to use in their own installation.

If the **js** package version 1.2 or greater is installed, the Javascript code will be minified and stored in the file named by minifile in the src directory. Syntax errors in the code will stop the process; unused variables will be reported.

If debugging is TRUE, the locations of Javascript syntax errors will be reported, along with hints about improvements, and the original files will be used in the dependency object that is created.

If debugging is FALSE (the default), the minified file will be used in the dependency object, hints won't be given, and syntax errors will lead to an uninformative failure to minify.

Value

An object that can be included in a list of dependencies passed to htmltools::attachDependencies.

Note

The usual way to use makeDependency is to call it in a '.R' file in a package, saving the result in a variable that will be used when an HTML widget is created. This way it is only run during package installation, when it is safe to write to the R library holding the package.

Do not call it to write to the R library from code the user can run, as that is not allowed in general.

If your package uses Roxygen, you may have problems because by default Roxygen will run the code, and it is likely to fail. The current workaround is to specify Roxygen option load = "installed" which prevents it from running your '.R' code.

Author(s)

Duncan Murdoch

Examples

```
## Not run:
# This is a slightly simplified version of the code used to
# produce one of the dependencies for rglwidget().
# It writes to the system library copy of rgl so
# has been marked not to run in the example code.

makeDependency("rglwidgetClass",
               src = "htmlwidgets/lib/rglClass",
               script = c("rglClass.src.js",
                          "utils.src.js",
```

```
                        "buffer.src.js",
                        "subscenes.src.js",
                        "shaders.src.js",
                        "textures.src.js",
                        "projection.src.js",
                        "mouse.src.js",
                        "init.src.js",
                        "pieces.src.js",
                        "draw.src.js",
                        "controls.src.js",
                        "selection.src.js",
                        "rglTimer.src.js",
                        "pretty.src.js",
                        "axes.src.js",
                        "animation.src.js"),
              stylesheet = "rgl.css",
              package = "rgl",
              debugging = isTRUE(as.logical(Sys.getenv("RGL_DEBUGGING", "FALSE"))))

## End(Not run)
```

`material3d` *Get or set material properties*

Description

Get or set material properties for geometry appearance.

Usage

```
material3d(..., id = NULL)

rgl.material.names
rgl.material.readonly
```

Arguments

`...`	Material properties to set or query.
`id`	the **rgl** id of an object to query, or `NULL` to query or set the defaults.

Details

In an **rgl** scene, each object has "material properties" that control how it is rendered and (in the case of `tag`) that can be used to store a label or other information. `material3d` sets defaults for these properties and queries the defaults or specific values for an individual object.

To set values, use `name = value` settings, e.g. `material3d(color = "red")`. To query values, specify the property or properties in a character vector, e.g. `material3d("color")`.

Only one side at a time can be culled.

The `material` member of the `r3dDefaults` list may be used to set default values for material properties.

Value

`material3d()` returns values similarly to `par3d`: When setting properties, it returns the previous values invisibly in a named list. When querying multiple values, a named list is returned. When a single value is queried it is returned directly.

Material Properties

The `rgl.material.names` variable contains the full list of material names. The following read-write material properties control the appearance of objects in an **rgl** scene.

color vector of R color characters. Represents the diffuse component in case of lighting calculation (lit = TRUE), otherwise it describes the solid color characteristics.

lit logical, specifying if lighting calculation should take place on geometry

ambient, specular, emission, shininess properties for lighting calculation. ambient, specular, emission are R color character string values; shininess represents a numerical.

alpha vector of alpha values between 0.0 (fully transparent) and 1.0 (opaque). See A Note on Transparency for a discussion of some issues with transparency.

smooth logical, specifying whether smooth shading or flat shading should be used. For smooth shading, Gouraud shading is used in **rgl** windows, while Phong shading is used in WebGL.

texture path to a texture image specification. See the Textures section below for details.

textype specifies what is defined with the pixmap

- **"alpha"** alpha values
- **"luminance"** luminance
- **"luminance.alpha"** luminance and alpha
- **"rgb"** color
- **"rgba"** color and alpha texture

Note that support for these modes is slightly different in the display within R versus the WebGL display using `rglwidget()`. In particular, in WebGL `textype = "alpha"` will always take the alpha value from the luminance (i.e. the average of the R, G and B channels) of the texture, whereas the R display bases the choice on the internal format of the texture file.

texmode specifies how the texture interacts with the existing color

- **"replace"** texture value replaces existing value
- **"modulate"** default; texture value multiplies existing value
- **"decal"** for `textype = "rgba"`, texture is mixed with existing value
- **"blend"** uses the texture to blend the existing value with black
- **"add"** adds the texture value to the existing. May not be available in the R display with very old OpenGL drivers.

texmipmap Logical, specifies if the texture should be mipmapped.

texmagfilter specifies the magnification filtering type (sorted by ascending quality):

- **"nearest"** texel nearest to the center of the pixel

"linear" weighted linear average of a 2x2 array of texels

texminfilter specifies the minification filtering type (sorted by ascending quality):

"nearest" texel nearest to the center of the pixel

"linear" weighted linear average of a 2x2 array of texels

"nearest.mipmap.nearest" low quality mipmapping

"nearest.mipmap.linear" medium quality mipmapping

"linear.mipmap.nearest" medium quality mipmapping

"linear.mipmap.linear" high quality mipmapping

texenvmap logical, specifies if auto-generated texture coordinates for environment-mapping should be performed on geometry.

front, back Determines the polygon mode for the specified side:

"filled" filled polygon

"lines" wireframed polygon

"points" point polygon

"culled" culled (hidden) polygon

size numeric, specifying the size of points in pixels

lwd numeric, specifying the line width in pixels

fog logical, specifying if fog effect should be applied on the corresponding shape. Fog type is set in `bg3d`.

point_antialias, line_antialias logical, specifying if points should be round and lines should be antialiased, but see Note below.

depth_mask logical, specifying whether the object's depth should be stored.

depth_test Determines which depth test is used to see if this object is visible, depending on its apparent depth in the scene compared to the stored depth. Possible values are `"never"`, `"less"` (the default), `"equal"`, `"lequal"` (less than or equal), `"greater"`, `"notequal"`, `"gequal"` (greater than or equal), `"always"`.

polygon_offset A one or two element vector giving the '`factor`' and '`units`' values to use in a `glPolygonOffset()` call in OpenGL. If only one value is given, it is used for both elements. The '`units`' value is added to the depth of all pixels in a filled polygon, and the '`factor`' value is multiplied by an estimate of the slope of the polygon and then added to the depth. Positive values "push" polygons back slightly for the purpose of depth testing, to allow points, lines or other polygons to be drawn on the surface without being obscured due to rounding error. Negative values pull the object forward. A typical value to use is `1` (which is automatically expanded to `c(1,1)`). If values are too large, objects which should be behind the polygon will show through, and if values are too small, the objects on the surface will be partially obscured. Experimentation may be needed to get it right. The first example in ?`persp3d` uses this property to add grid lines to a surface.

margin, floating Used mainly for text to draw annotations in the margins, but supported by most kinds of objects: see `mtext3d`.

tag A length 1 string value. These may be used to identify objects, or encode other meta data about the object.

blend Two string values from the list below describing how transparent objects are blended with colors behind them. The first determines the coefficient applied to the color of the current object (the source); the second determines the coefficient applied to the existing color (the destination). The resulting color will be the sum of the two resulting colors. The allowed strings correspond to OpenGL constants:

"zero" Zero; color has no effect.

"one" One; color is added to the other term.

"src_color", "one_minus_src_color" Multiply by source color or its opposite.

"dst_color", "one_minus_dst_color" Multiply by destination color or its opposite.

"src_alpha", "one_minus_src_alpha" Multiply by source alpha or its opposite. Default values.

"dst_alpha", "one_minus_dst_alpha" Multiply by destination alpha or its opposite.

"constant_color", "one_minus_constant_color", "constant_alpha", "one_minus_constant_alpha", "src_alpha_ These are allowed, but to be useful they require other settings which **rgl** doesn't support.

col An allowed abbreviation of `color`.

The `rgl.material.readonly` variable contains the subset of material properties that are read-only so they can be queried but not set. Currently there is only one:

isTransparent Is the current color transparent?

Textures

The `texture` material property may be `NULL` or the name of a bitmap file to be displayed on the surface being rendered. Currently PNG and JPEG format files are supported. It may also be any other object for which `as.raster()` works, e.g. a matrix.

By default, the colors in the bitmap will modify the color of the object being plotted. If the color is black (a common default), you won't see anything, so a warning may be issued. You can suppress the warning by specifying the color explicitly, or calling `options{rgl.warnBlackTexture = FALSE}`.

Other aspects of texture display are controlled by the material properties `textype`, `texmode`, `texmipmap`, `texmagfilter`,`texminfilter` and `texenvmap` described above.

For an extensive discussion of textures, see the Textures section of the rgl Overview vignette.

Display of objects

Object display colors are determined as follows:

- If `lit = FALSE`, an element of the `color` vector property is displayed without modification. See documentation for individual objects for information on which element is chosen.
- If `lit = TRUE`, the color is determined as follows.
  1. The color is set to the `emission` property of the object.
  2. For each defined light, the following are added:
     - the product of the `ambient` color of the light and the `ambient` color of the object is added.

- the color of the object is multiplied by the diffuse color of the light and by a constant depending on the angle between the surface and the direction to the light, and added.
- the specular color of the object is multiplied by the specular color of the light and a constant depending on the shininess of the object and the direction to the light, and added. The shininess property mainly determines the size of the shiny highlight; adjust one or both of the specular colors to change its brightness.

If point_antialias is TRUE, points will be drawn as circles in WebGL; otherwise, they will be drawn as squares. Within R, the behaviour depends on your graphics hardware: for example, I see circles for both settings on my laptop.

Within R, lines tend to appear heavier with line_antialias == TRUE. There's no difference at all in WebGL.

See Also

bbox3d, bg3d, light3d

Examples

```
save <- material3d("color")
material3d(color = "red")
material3d("color")
material3d(color = save)

# this illustrates the effect of depth_test
x <- c(1:3); xmid <- mean(x)
y <- c(2, 1, 3); ymid <- mean(y)
z <- 1
open3d()
tests <- c("never", "less", "equal", "lequal", "greater",
                  "notequal", "gequal", "always")
for (i in 1:8) {
  triangles3d(x, y, z + i, col = heat.colors(8)[i])
  texts3d(xmid, ymid, z + i, paste(i, tests[i], sep = ". "), depth_test = tests[i])
}
highlevel()  # To trigger display

# this illustrates additive blending
open3d()
bg3d("darkgray")
quad <- cbind(c(-1, 1, 1, -1), 1, c(-1, -1, 1, 1))
quads3d(rbind(translate3d(quad, -0.5, 0, -0.5),
              translate3d(quad, 0.5,  0.5, -0.5),
              translate3d(quad, 0, 1, 0.5)),
        col = rep(c("red", "green", "blue"), each = 4),
        alpha = 0.5,
        blend = c("src_alpha", "one"))
```

matrices *Work with homogeneous coordinates*

Description

These functions construct 4x4 matrices for transformations in the homogeneous coordinate system used by OpenGL, and translate vectors between homogeneous and Euclidean coordinates.

Usage

```
identityMatrix()
scaleMatrix(x, y, z)
translationMatrix(x, y, z)
rotationMatrix(angle, x, y, z, matrix)
asHomogeneous(x)
asEuclidean(x)
asHomogeneous2(x)
asEuclidean2(x)

scale3d(obj, x, y, z, ...)
translate3d(obj, x, y, z, ...)
rotate3d(obj, angle, x, y, z, matrix, ...)

transform3d(obj, matrix, ...)
```

Arguments

x, y, z, angle, matrix	See details
obj	An object to be transformed
...	Additional parameters to be passed to methods

Details

OpenGL uses homogeneous coordinates to handle perspective and affine transformations. The homogeneous point (x, y, z, w) corresponds to the Euclidean point (x/w, y/w, z/w). The matrices produced by the functions scaleMatrix, translationMatrix, and rotationMatrix are to be left-multiplied by a row vector of homogeneous coordinates; alternatively, the transpose of the result can be right-multiplied by a column vector. The generic functions scale3d, translate3d and rotate3d apply these transformations to the obj argument. The transform3d function is a synonym for rotate3d(obj, matrix = matrix).

By default, it is assumed that obj is a row vector (or a matrix of row vectors) which will be multiplied on the right by the corresponding matrix, but users may write methods for these generics which operate differently. Methods are supplied for mesh3d objects.

To compose transformations, use matrix multiplication. The effect is to apply the matrix on the left first, followed by the one on the right.

`identityMatrix` returns an identity matrix.

`scaleMatrix` scales each coordinate by the given factor. In Euclidean coordinates, `(u, v, w)` is transformed to `(x*u, y*v, z*w)`.

`translationMatrix` translates each coordinate by the given translation, i.e. `(u, v, w)` is transformed to `(u + x, v + y, w + z)`.

`rotationMatrix` can be called in three ways. With arguments `angle, x, y, z` it represents a rotation of `angle` radians about the axis `x, y, z`. If `matrix` is a 3x3 rotation matrix, it will be converted into the corresponding matrix in 4x4 homogeneous coordinates. Finally, if a 4x4 matrix is given, it will be returned unchanged. (The latter behaviour is used to allow `transform3d` to act like a generic function, even though it is not.)

Use `asHomogeneous(x)` to convert the Euclidean vector `x` to homogeneous coordinates, and `asEuclidean(x)` for the reverse transformation. These functions accept the following inputs:

- n x 3 matrices: rows are assumed to be Euclidean
- n x 4 matrices: rows are assumed to be homogeneous
- vectors of length 3n or 4n: assumed to be vectors concatenated. For the ambiguous case of vectors that are length 12n (so both 3n and 4n are possible), the assumption is that the conversion is necessary: `asEuclidean` assumes the vectors are homogeneous, and `asHomogeneous` assumes the vectors are Euclidean.

Outputs are n x 4 or n x 3 matrices for `asHomogeneous` and `asEuclidean` respectively.

The functions `asHomogeneous2` and `asEuclidean2` act similarly, but they assume inputs are 3 x n or 4 x n and outputs are in similar shapes.

Value

`identityMatrix`, `scaleMatrix`, `translationMatrix`, and `rotationMatrix` produce a 4x4 matrix representing the requested transformation in homogeneous coordinates.

`scale3d`, `translate3d` and `rotate3d` transform the object and produce a new object of the same class.

Author(s)

Duncan Murdoch

See Also

`par3d` for a description of how RGL uses matrices in rendering.

Examples

```
# A 90 degree rotation about the x axis:

rotationMatrix(pi/2, 1, 0, 0)

# Find what happens when you rotate (2, 0, 0) by 45 degrees about the y axis:
```

```
x <- asHomogeneous(c(2, 0, 0))
y <- x %*% rotationMatrix(pi/4, 0, 1, 0)
asEuclidean(y)

# or more simply...

rotate3d(c(2, 0, 0), pi/4, 0, 1, 0)
```

`merge.mesh3d` *Merge RGL mesh objects*

Description

Attempts to merge `"mesh3d"` objects. Objects need to be similar enough; see Details.

Usage

```
## S3 method for class 'mesh3d'
merge(x, y, ..., attributesMustMatch = FALSE)
```

Arguments

`x, y`	`"mesh3d"` objects to merge.
`...`	Optional additional objects.
`attributesMustMatch`	See Details.

Details

To allow objects to be merged, they need to be similar enough in terms of having the same list of material properties, normals, texture coordinates, etc.

If `attributesMustMatch` is `TRUE`, it is an error to have attributes in one mesh but not in another, and those attributes that only specify a single value must have equal values in all meshes.

If `attributesMustMatch` is `FALSE`, any non-matching attributes will be dropped from the final result.

Value

A single `"mesh3d"` object merging the contents of the arguments.

Author(s)

Duncan Murdoch

Examples

```
open3d()
# Notice that the alpha setting for the cube is dropped, because
# the other shapes don't specify alpha.
shade3d(merge(cube3d(col="red", alpha = 0.5),
              translate3d(tetrahedron3d(col="green"), 2, 0, 0),
              translate3d(octahedron3d(col="blue"), 4, 0, 0)))
```

`mergeVertices` *Merge duplicate vertices in mesh object*

Description

A mesh object can have the same vertex listed twice. Each copy is allowed to have separate normals, texture coordinates, and color. However, it is more efficient to have just a single copy if those differences aren't needed. For automatic smoothing using `addNormals`, triangles and quads need to share vertices. This function merges identical (or similar) vertices to achieve this.

Usage

```
mergeVertices(mesh,
              notEqual = NULL,
              attribute = "vertices",
              tolerance = sqrt(.Machine$double.eps))
```

Arguments

`mesh`	A `mesh3d` object.
`notEqual`	A logical matrix indicating that certain pairs should not be merged even if they appear identical.
`attribute`	Which attribute(s) should be considered in comparing vertices? A vector chosen from `c("vertices", "colors", "normals", "texcoords")`
`tolerance`	When comparing vertices using `all.equal`, this tolerance will be used to ignore rounding error.

Value

A new mesh object.

Author(s)

Duncan Murdoch

See Also

`as.mesh3d.rglId`, which often constructs mesh objects containing a lot of duplication.

Examples

```
open3d()
(mesh1 <- cuboctahedron3d(col = rainbow(14), meshColor = "face"))
id <- shade3d(mesh1)
(mesh2 <- as.mesh3d(id))
shade3d(translate3d(mesh2, 3, 0, 0))
(mesh3 <- mergeVertices(mesh2))
shade3d(translate3d(mesh3, 6, 0, 0))
```

`mesh3d` *Construct 3D mesh objects*

Description

Creates meshes containing points, segments, triangles and quads.

Usage

```
mesh3d( x, y = NULL, z = NULL, vertices,
                  material = NULL,
                  normals = NULL, texcoords = NULL,
                  points = NULL, segments = NULL,
                  triangles = NULL, quads = NULL,
                  meshColor = c("vertices", "edges", "faces", "legacy"))
qmesh3d(vertices, indices, homogeneous = TRUE, material = NULL,
       normals = NULL, texcoords = NULL,
       meshColor = c("vertices", "edges", "faces", "legacy"))
tmesh3d(vertices, indices, homogeneous = TRUE, material = NULL,
       normals = NULL, texcoords = NULL,
       meshColor = c("vertices", "edges", "faces", "legacy"))
```

Arguments

`x`, `y`, `z`	coordinates. Any reasonable way of defining the coordinates is acceptable. See the function `xyz.coords` for details.
`vertices`	A 4 row matrix of homogeneous coordinates; takes precedence over `x`, `y`, `z`.
`material`	material properties for later rendering
`normals`	normals at each vertex
`texcoords`	texture coordinates at each vertex
`points`	vector of indices of vertices to draw as points
`segments`	2 x n matrix of indices of vertices to draw as segments
`triangles`	3 x n matrix of indices of vertices to draw as triangles
`quads`	4 x n matrix of indices of vertices to draw as quads

indices	(obsolete) 3 or 4 x n matrix of vertex indices
homogeneous	(obsolete) should tmesh3d and qmesh3d vertices be assumed to be homogeneous?
meshColor	how should colours be interpreted? See details in shade3d.

Details

These functions create mesh3d objects, which consist of a matrix of vertex coordinates together with a matrices of indices indicating how the vertices should be displayed, and material properties.

The "shape3d" class is a general class for shapes that can be plotted by dot3d, wire3d or shade3d.

The "mesh3d" class is a class of objects that form meshes: the vertices are in member vb, as a 4 by n matrix using homogeneous coordinates. Indices of these vertices are contained in optional components ip for points, is for line segments, it for triangles, and ib for quads. Individual meshes may have any combination of these.

The functions tmesh3d and qmesh3d are included for back-compatibility; they produce meshes of triangles and quads respectively.

Value

Objects of class c("mesh3d", "shape3d").

See points3d for a discussion of texture coordinates.

See Also

shade3d, shapelist3d for multiple shapes

Examples

```
# generate a quad mesh object

vertices <- c(
   -1.0, -1.0, 0,
    1.0, -1.0, 0,
    1.0,  1.0, 0,
   -1.0,  1.0, 0
)
indices <- c( 1, 2, 3, 4 )

open3d()
wire3d( mesh3d(vertices = vertices, quads = indices) )
```

mfrow3d *Set up multiple figure layouts*

Description

The mfrow3d and layout3d functions provide functionality in RGL similar to par("mfrow") and layout in classic R graphics.

Usage

```
subsceneList(value, window = cur3d())

mfrow3d(nr, nc, byrow = TRUE, parent = NA, sharedMouse = FALSE, ...)
layout3d(mat, widths = rep.int(1, ncol(mat)),
              heights = rep.int(1, nrow(mat)),
              parent = NA, sharedMouse = FALSE,
              ...)
next3d(current = NA, clear = TRUE, reuse = TRUE)
clearSubsceneList(delete = currentSubscene3d() %in% subsceneList(),
                  window = cur3d())
```

Arguments

value	A new subscene list to set. If missing, return the current one (or NULL).
window	Which window to operate on.
nr, nc	Number of rows and columns of figures.
byrow	Whether figures progress by row (as with par("mfrow")) or by column (as with par("mfcol")).
mat, widths, heights	Layout parameters; see layout for their interpretation.
parent	The parent subscene. NA indicates the current subscene. See Details below.
sharedMouse	Whether to make all subscenes par3d("listeners") to each other.
...	Additional parameters to pass to newSubscene3d as each subscene is created.
current	The subscene to move away from. NA indicates the current subscene.
clear	Whether the newly entered subscene should be cleared upon entry.
reuse	Whether to skip advancing if the current subscene has no objects in it.
delete	If TRUE, delete the subscenes in the current window.

Details

rgl can maintain a list of subscenes; the mfrow3d and layout3d functions create that list. When the list is in place, next3d causes RGL to move to the next scene in the list, or cycle back to the first one.

Unlike the classic R graphics versions of these functions, these functions are completely compatible with each other. You can mix them within a single RGL window.

In the default case where `parent` is missing, `mfrow3d` and `layout3d` will call `clearSubsceneList()` at the start.

By default `clearSubsceneList()` checks whether the current subscene is in the current subscene list; if so, it will delete all subscenes in the list, and call `gc3d` to delete any objects that are no longer shown. The subscene list will be set to a previous value if one was recorded, or `NULL` if not.

If `parent` is specified in `mfrow3d` or `layout3d` (even as `NA`), the new subscenes will be created within the parent.

The `next3d()` function first finds out if the current subscene is in the current list. If not, it moves to the previous list, and looks there. Once it finds a list containing the current subscene, it moves to the next entry in that list. If it can't find one, it creates a length one list containing just the current subscene.

Value

`mfrow3d` and `layout3d` return a vector of subscene id values that have just been created. If a previous subscene list was in effect and was not automatically cleared, it is attached as an attribute `"prev"`.

Author(s)

Duncan Murdoch

See Also

`newSubscene3d`, `par`, `layout`.

Examples

```
shapes <- list(Tetrahedron = tetrahedron3d(), Cube = cube3d(), Octahedron = octahedron3d(),
               Icosahedron = icosahedron3d(), Dodecahedron = dodecahedron3d(),
               Cuboctahedron = cuboctahedron3d())
col <- rainbow(6)
open3d()
mfrow3d(3, 2)
for (i in 1:6) {
  next3d()   # won't advance the first time, since it is empty
  shade3d(shapes[[i]], col = col[i])
}
highlevel(integer()) # To trigger display as rglwidget

open3d()
mat <- matrix(1:4, 2, 2)
mat <- rbind(mat, mat + 4, mat + 8)
layout3d(mat, height = rep(c(3, 1), 3), sharedMouse = TRUE)
for (i in 1:6) {
  next3d()
  shade3d(shapes[[i]], col = col[i])
  next3d()
```

```
  text3d(0, 0, 0, names(shapes)[i])
}
highlevel(integer())
```

---

observer3d *Set the observer location*

Description

This function sets the location of the viewer.

Usage

```
observer3d(x, y = NULL, z = NULL, auto = FALSE)
```

Arguments

x, y, z	The location as a 3 vector, using the usual xyz.coords conventions for specification. If x is missing or any coordinate is NA, no change will be made to the location.
auto	If TRUE, the location will be set automatically by RGL to make the whole bounding box visible.

Details

This function sets the location of the viewer relative to the scene, after the model transformations (scaling, rotation) have been done, but before lighting or projection have been applied. (See par3d for details on the rendering pipeline.)

The coordinate system is a slightly strange one: the X coordinate moves the observer location from left to right, and the Y coordinate moves up and down. The Z coordinate changes the depth from the viewer. All are measured relative to the center of the bounding box (par("bbox")) of the subscene. The observer always looks in the positive Z direction after the model rotation have been done. The coordinates are in post-scaling units.

Value

Invisibly returns the previous value.

Note

This function is likely to change in future versions of RGL, to allow more flexibility in the specification of the observer's location and orientation.

Author(s)

Duncan Murdoch

Examples

```
example(surface3d)  # The volcano data
observer3d(0, 0, 440) # Viewed from very close up
```

---

open3d *Work with RGL windows*

Description

open3d opens a new RGL window; cur3d returns the device number of the current window; close3d closes one or more windows.

Usage

```
open3d(..., params = getr3dDefaults(),
     useNULL = rgl.useNULL(), silent = FALSE)

close3d(dev = cur3d(), silent = TRUE)

cur3d()

rgl.dev.list()

set3d(dev, silent = FALSE)

getr3dDefaults(class = NULL, value = NULL)

r3dDefaults

rgl.quit()
```

Arguments

...	arguments in name = value form, or a list of named values. The names must come from the graphical parameters described in par3d.
params	a list of graphical parameters
useNULL	whether to use the null graphics device
dev	which device to close or use
silent	whether report on what was done
class, value	names of components to retrieve

Details

open3d opens a new RGL device, and sets the parameters as requested. The r3dDefaults list returned by the getr3dDefaults function will be used as default values for parameters. As installed this sets the point of view to 'world coordinates' (i.e. x running from left to right, y from front to back, z from bottom to top), the mouseMode to (zAxis, zoom, fov), and the field of view to 30 degrees. useFreeType defaults to FALSE on Windows; on other systems it indicates the availability of FreeType. Users may create their own variable named r3dDefaults in the global environment and it will override the installed one. If there is a bg element in the list or the arguments, it should be a list of arguments to pass to the bg3d function to set the background.

The arguments to open3d may include material, a list of material properties as in r3dDefaults, but note that high level functions such as plot3d normally use the r3dDefaults values in preference to this setting.

If useNULL is TRUE, RGL will use a "null" device. This device records objects as they are plotted, but displays nothing. It is intended for use with rglwidget.

Value

The open3d function returns the device that was opened. If silent = TRUE, it is returned invisibly.

The cur3d function returns the current device, or the value 0 if there isn't one. rgl.dev.list returns a vector of all open devices. Items are named according to the type of device: null for a hidden null device, wgl for a Windows device, and glX for an X windows device.

set3d returns the device number of the previously active device.

The close3d function returns the new current device, invisibly.

The r3dDefaults variable is a list containing default settings. The getr3dDefaults function searches the user's global environment for r3dDefaults and returns the one in the RGL namespace if it was not found there. The components of the list may include any settable par3d parameter, or "material", which should include a list of default material3d properties, or "bg", which is a list of defaults to pass to the bg3d function.

rgl.quit attempts to unload **rgl** and then returns NULL invisibly.

See Also

rgl.useNULL for default usage of null device.

Examples

```
r3dDefaults
open3d()
shade3d(cube3d(color = rainbow(6), meshColor = "faces"))
cur3d()
```

par3d *Set or query RGL parameters*

Description

par3d can be used to set or query graphical parameters in RGL. Parameters can be set by specifying them as arguments to par3d in name = value form, or by passing them as a list of named values.

Usage

```
par3d(..., no.readonly = FALSE, dev = cur3d(),
      subscene = currentSubscene3d(dev))
rgl.par3d.names
rgl.par3d.readonly
```

Arguments

...	arguments in name = value form, or a list of tagged values. The names must come from the graphical parameters described below.
no.readonly	logical; if TRUE and there are no other arguments, only those parameters which can be set by a subsequent par3d() call are returned.
dev	integer; the RGL device.
subscene	integer; the subscene.

Details

Parameters are queried by giving one or more character vectors to par3d.

par3d() (no arguments) or par3d(no.readonly = TRUE) is used to get *all* the graphical parameters (as a named list).

By default, queries and modifications apply to the current subscene on the current device; specify dev and/or subscene to change this. Some parameters apply to the device as a whole; these are marked in the list below.

Value

When parameters are set, their former values are returned in an invisible named list. Such a list can be passed as an argument to par3d to restore the parameter values. Use par3d(no.readonly = TRUE) for the full list of parameters that can be restored.

When just one parameter is queried, its value is returned directly. When two or more parameters are queried, the result is a list of values, with the list names giving the parameters.

Note the inconsistency: setting one parameter returns a list, but querying one parameter returns an object.

Parameters

The `rgl.par3d.names` variable contains the full list of names of `par3d` properties. `rgl.par3d.readonly` contains the list of read-only properties.

In the list below, ***R.O.*** indicates the read-only arguments: These may only be used in queries, they do not set anything.

`activeSubscene` ***R.O.*** integer. Used with `rgl.setMouseCallbacks`: during a callback, indicates the id of the subscene that was clicked.

`antialias` ***R.O.*** in `par3d`, may be set in `open3d`. The (requested) number of hardware antialiasing planes to use (with multisample antialiasing). The OpenGL driver may not support the requested number, in which case `par3d("antialias")` will report what was actually set. Applies to the whole device.

`cex` real. The default size for text.

`family` character. The default device independent family name; see `text3d`. Applies to the whole device.

`font` integer. The default font number (from 1 to 4; see `text3d`). Applies to the whole device.

`useFreeType` logical. Should FreeType fonts be used? Applies to the whole device.

`fontname` ***R.O.***; the system-dependent name of the current font. Applies to the whole device.

`FOV` real. The field of view, from 0 to 179 degrees. This controls the degree of parallax in the perspective view. Isometric perspective corresponds to `FOV = 0`.

`ignoreExtent` logical. Set to `TRUE` so that subsequently plotted objects will be ignored in calculating the bounding box of the scene. Applies to the whole device.

`maxClipPlanes` ***R.O.***; an integer giving the maximum number of clip planes that can be defined in the current system. Applies to the whole device.

`modelMatrix` ***R.O.***; a 4 by 4 matrix describing the position of the user data. See the Note below.

`listeners` integer. A vector of subscene id values. If a subscene receives a mouse event (see `mouseMode` just below), the same action will be carried out on all subscenes in this list. (The subscene itself is normally listed as a listener. If it is not listed, it will not respond to its own mouse events.)

`mouseMode` character. A vector of 5 strings describing mouse actions. The 5 entries are named `c("none", "left", "right", "middle", "wheel")`, corresponding to actions for no button, the left, right or middle button, and the mouse wheel. Partial matching to action names is used. Possible values for the actions are:

- `"none"` No action for this button.
- `"trackball"` Mouse acts as a virtual trackball, rotating the scene.
- `"xAxis"` Similar to `"trackball"`, but restricted to X axis rotation.
- `"yAxis"` Y axis rotation.
- `"zAxis"` Z axis rotation.
- `"polar"` Mouse rotates the scene by moving in polar coordinates.
- `"selecting"` Mouse is used for selection. This is not normally set by the user, but is used internally by the `select3d` function.
- `"zoom"` Mouse is used to zoom the display.
- `"fov"` Mouse changes the field of view of the display.

"user" Used when a user handler is set by rgl.setMouseCallbacks.

Possible values for the last entry corresponding to the mouse wheel also include

"pull" Pulling on the mouse wheel increases magnification, i.e. "pulls the scene closer".

"push" Pulling on the mouse wheel decreases magnification, i.e. "pushes the scene away".

"user2" Used when a user handler is set by rgl.setWheelCallback.

A common default on Mac OSX is to convert a two finger drag on a trackpad to a mouse wheel rotation.

The first entry is for actions to take when no mouse button is pressed. Legal values are the same as for the mouse buttons.

The first entry was added after **rgl** version 0.106.8. For back compatibility, if the vector of actions is less than 5 entries, "none" will be added at the start of it.

observer ***R.O.***; the position of the observer relative to the model. Set by observer3d. See the Note below.

projMatrix ***R.O.***; a 4 by 4 matrix describing the current projection of the scene.

scale real. A vector of 3 values indicating the amount by which to rescale each axis before display. Set by aspect3d.

skipRedraw whether to update the display. Set to TRUE to suspend updating while making multiple changes to the scene. See demo(hist3d) for an example. Applies to the whole device.

userMatrix a 4 by 4 matrix describing user actions to display the scene.

userProjection a 4 by 4 matrix describing changes to the projection.

viewport real. A vector giving the dimensions of the window in pixels. The entries are taken to be c(x, y, width, height) where c(x, y) are the coordinates in pixels of the lower left corner within the window.

zoom real. A positive value indicating the current magnification of the scene.

bbox ***R.O.***; real. A vector of six values indicating the current values of the bounding box of the scene (xmin, xmax, ymin, ymax, zmin, zmax)

windowRect integer. A vector of four values indicating the left, top, right and bottom of the displayed window (in pixels). Applies to the whole device.

Rendering

The parameters returned by par3d are sufficient to determine where RGL would render a point on the screen. Given a column vector (x, y, z) in a subscene s, it performs the equivalent of the following operations:

1. It converts the point to homogeneous coordinates by appending w = 1, giving the vector v = (x, y, z, 1).
2. It calculates the M = par3d("modelMatrix") as a product from right to left of several matrices:
   - A matrix to translate the centre of the bounding box to the origin.
   - A matrix to rescale according to par3d("scale").
   - The par3d("userMatrix") as set by the user.
   - A matrix which may be set by mouse movements.

- The description above applies to the usual case where there is just one subscene, or where the subscene's `"model"` is set to `"replace"`. If it is set to `"modify"`, the first step is skipped, and at the end the procedure is followed for the parent subscene. If it is set to `"inherit"` only the parent settings are used.

3. It multiplies the point by `M` giving `u = M %*% v`.
4. It multiplies that point by a matrix based on the observer position to translate the origin to the centre of the viewing region.
5. Using this location and information on the normals (which have been similarly transformed), it performs lighting calculations.
6. It obtains the projection matrix `P = par3d("projMatrix")` based on the bounding box and field of view or observer location, multiplies that by the `userProjection` matrix to give `P`. It multiplies the point by it giving `P %*% u = (x2, y2, z2, w2)`.
7. It converts back to Euclidean coordinates by dividing the first 3 coordinates by `w2`.
8. The new value `z2/w2` represents the depth into the scene of the point. Depending on what has already been plotted, this depth might be obscured, in which case nothing more is plotted.
9. If the point is not culled due to depth, the `x2` and `y2` values are used to determine the point in the image. The `par3d("viewport")` values are used to translate from the range `(-1, 1)` to pixel locations, and the point is plotted.
10. If hardware antialiasing is enabled, then the whole process is repeated multiple times (at least conceptually) with different locations in each pixel sampled to determine what is plotted there, and then the images are combined into what is displayed.

See ?matrices for more information on homogeneous and Euclidean coordinates.

Note that many of these calculations are done on the graphics card using single precision; you will likely see signs of rounding error if your scene requires more than 4 or 5 digit precision to distinguish values in any coordinate.

Note

The `"xAxis"`, `"yAxis"` and `"zAxis"` mouse modes rotate relative to the coordinate system of the data, regardless of the current orientation of the scene.

When multiple parameters are set, they are set in the order given. In some cases this may lead to warnings and ignored values; for example, some font families only support `cex = 1`, so changing both `cex` and `family` needs to be done in the right order. For example, when using the `"bitmap"` family on Windows, `par3d(family = "sans", cex = 2)` will work, but `par3d(cex = 2, family = "sans")` will leave `cex` at 1 (with a warning that the `"bitmap"` family only supports that size).

Although `par3d("viewport")` names the entries of the reported vector, names are ignored when setting the viewport and entries must be specified in the standard order.

In **rgl** versions 0.94.x the `modelMatrix` entry had a changed meaning; before and after that it contains a copy of the OpenGL MODELVIEW matrix.

As of version 0.100.32, when changing the `"windowRect"` parameter, the `"viewport"` for the root (or specified) subscene is changed immediately. This fixes a bug where in earlier versions it would only be changed when the window was redrawn, potentially after another command making use of the value.

Default values are not described here, as several of them are changed by the `r3dDefaults` variable when the window is opened by `open3d`.

References

OpenGL Architecture Review Board (1997). OpenGL Programming Guide. Addison-Wesley.

See Also

view3d to set FOV and zoom.

open3d for how to open a new window with default settings for these parameters.

Examples

```
open3d()
shade3d(cube3d(color = rainbow(6), meshColor = "faces"))
save <- par3d(userMatrix = rotationMatrix(90*pi/180, 1, 0, 0))
highlevel()  # To trigger display
save
par3d("userMatrix")
par3d(save)
highlevel()
par3d("userMatrix")
```

---

par3dinterp *Interpolator for par3d parameters*

Description

Returns a function which interpolates par3d parameter values, suitable for use in animations.

Usage

```
par3dinterp(times = NULL, userMatrix, scale, zoom, FOV,
            method = c("spline", "linear"),
            extrapolate = c("oscillate", "cycle", "constant", "natural"),
            dev = cur3d(), subscene = par3d("listeners", dev = dev))
```

Arguments

times	Times at which values are recorded or a list; see below
userMatrix	Values of par3d("userMatrix")
scale	Values of par3d("scale")
zoom	Values of par3d("zoom")
FOV	Values of par3d("FOV")
method	Method of interpolation
extrapolate	How to extrapolate outside the time range
dev	Which RGL device to use
subscene	Which subscene to use

Details

This function is intended to be used in constructing animations. It produces a function that returns a list suitable to pass to par3d, to set the viewpoint at a given point in time.

All of the parameters are optional. Only those par3d parameters that are specified will be returned.

The input values other than times may each be specified as lists, giving the parameter value settings at a fixed time, or as matrices or arrays. If not lists, the following formats should be used: userMatrix can be a 4 x 4 x n array, or a 4 x 4n matrix; scale should be an n x 3 matrix; zoom and FOV should be length n vectors.

An alternative form of input is to put all of the above arguments into a list (i.e. a list of lists, or a list of arrays/matrices/vectors), and pass it as the first argument. This is the most convenient way to use this function with the function tkpar3dsave.

Interpolation is by cubic spline or linear interpolation in an appropriate coordinate-wise fashion. Extrapolation may oscillate (repeat the sequence forward, backward, forward, etc.), cycle (repeat it forward), be constant (no repetition outside the specified time range), or be natural (linear on an appropriate scale). In the case of cycling, the first and last specified values should be equal, or the last one will be dropped. Natural extrapolation is only supported with spline interpolation.

Value

A function is returned. The function takes one argument, and returns a list of par3d settings interpolated to that time.

Note

Prior to **rgl** version 0.95.1476, the subscene argument defaulted to the current subscene, and any additional entries would be ignored by play3d. The current default value of par3d("listeners", dev = dev) means that all subscenes that share mouse responses will also share modifications by this function.

Author(s)

Duncan Murdoch

See Also

play3d to play the animation.

Examples

```
f <- par3dinterp( zoom = c(1, 2, 3, 1) )
f(0)
f(1)
f(0.5)
## Not run:
play3d(f)

## End(Not run)
```

`par3dinterpControl` *Control RGL widget like par3dinterp()*

Description

This control works with `playwidget` to change settings in a WebGL display in the same way as `par3dinterp` does within R.

Usage

```
par3dinterpControl(fn, from, to, steps, subscene = NULL, omitConstant = TRUE, ...)
```

Arguments

`fn`	A function returned from `par3dinterp`.
`from`, `to`, `steps`	Values where `fn` should be evaluated.
`subscene`	Which subscene's properties should be modified?
`omitConstant`	If `TRUE`, do not set values that are constant across the range.
`...`	Additional parameters which will be passed to `propertyControl`.

Details

`par3dinterpSetter` sets parameters corresponding to values produced by the result of `par3dinterp`.

Value

Returns controller data in a list of class "rglControl".

Author(s)

Duncan Murdoch

See Also

The User Interaction in WebGL vignette gives more details.

Examples

```
example(plot3d)
M <- r3dDefaults$userMatrix
fn <- par3dinterp(times = (0:2)*0.75, userMatrix = list(M,
                                      rotate3d(M, pi/2, 1, 0, 0),
                                      rotate3d(M, pi/2, 0, 1, 0)),
                                      scale = c(0.5, 1, 2))
control <- par3dinterpControl(fn, 0, 3, steps = 15)
control
if (interactive() || in_pkgdown_example())
  rglwidget(width = 500, height = 250) %>%
```

```
playwidget(control,
     step = 0.01, loop = TRUE, rate = 0.5)
```

---

pch3d *Plot symbols similar to base graphics*

Description

This function plots symbols similarly to what the base graphics function `points` does when `pch` is specified.

Usage

```
pch3d(x, y = NULL, z = NULL, pch = 1,
      bg = material3d("color")[1], cex = 1, radius,
      color = "black", lit = FALSE, ...)
```

Arguments

`x`, `y`, `z`	The locations at which to plot in a form suitable for use in `xyz.coords`.
`pch`	A vector of integers or single characters describing the symbols to plot.
`bg`	The fill color(s) to use for `pch` from 21 to 25.
`cex`	A relative size of the symbol to plot.
`radius`	An absolute size of the symbol to plot in user coordinates.
`color`	The color(s) to use for symbols.
`lit`	Whether the object responds to lighting or just shows the displayed color directly.
`...`	Other material properties.

Details

The list of symbols encoded by numerical `pch` values is given in the `points` help page.

Value

A vector of object id values is returned invisibly. Separate objects will be drawn for each different combination of `pch` value from 0 to 25, `color` and `bg`, and another holding all the character symbols.

Note

This function is not a perfect match to how the points function works due to limitations in RGL and OpenGL. In particular:

Symbols with numbers from 1 to 25 are drawn as 3D sprites (see sprites3d), so they will resize as the window is zoomed. Letters and numbers from 32 to 255 (which are mapped to letters) are drawn using text3d, so they maintain a fixed size.

A calculation somewhat like the one in plot3d that sets the size of spheres is used to choose the size of sprites based on cex and the current scaling. This will likely need manual tweaking. Use the radius argument for a fixed size.

No special handling is done for the case of pch = ".". Use points3d for small dots.

As of **rgl** version 0.100.10, background and foreground colors can vary from symbol to symbol.

Author(s)

Duncan Murdoch

See Also

points3d, text3d and sprites3d for other ways to label points, points for the base graphics symbol definitions.

Examples

```
open3d()
i <- 0:25; x <- i %% 5; y <- rep(0, 26); z <- i %/% 5
pch3d(x, y, z, pch = i, bg = "gray", color = rainbow(26))
text3d(x, y, z + 0.3, i)
pch3d(x + 5, y, z, pch = i+65)
text3d(x + 5, y, z + 0.3, i+65)
```

---

persp3d *Surface plots*

Description

This function draws plots of surfaces in 3-space. persp3d is a generic function.

Usage

```
persp3d(x, ...)

## Default S3 method:
persp3d(x = seq(0, 1, length.out = nrow(z)),
    y = seq(0, 1, length.out = ncol(z)), z,
    xlim = NULL, ylim = NULL, zlim = NULL,
    xlab = NULL, ylab = NULL, zlab = NULL, add = FALSE, aspect = !add,
    forceClipregion = FALSE, ...)
```

Arguments

x, y, z	points to plot on surface. See Details below.
xlim, ylim, zlim	x-, y- and z-limits. If present, the plot is clipped to this region.
xlab, ylab, zlab	titles for the axes. N.B. These must be character strings; expressions are not accepted. Numbers will be coerced to character strings.
add	whether to add the points to an existing plot.
aspect	either a logical indicating whether to adjust the aspect ratio, or a new ratio.
forceClipregion	force a clipping region to be used, whether or not limits are given.
...	additional material parameters to be passed to surface3d and decorate3d.

Details

The default method plots a surface defined as a grid of (x,y,z) locations in space. The grid may be specified in several ways:

- As with persp, x and y may be given as vectors in ascending order, with z given as a matrix. There should be one x value for each row of z and one y value for each column. The surface drawn will have x constant across rows and y constant across columns. This is the most convenient format when z is a function of x and y which are measured on a regular grid.
- x and y may also be given as matrices, in which case they should have the same dimensions as z. The surface will combine corresponding points in each matrix into locations (x,y,z) and draw the surface through those. This allows general surfaces to be drawn, as in the example of a spherical Earth shown below.
- If x is a list, its components xx, xy and x$z are used for x, y and z respectively, though an explicitly specified z value will have priority.

One difference from persp is that colors are specified on each vertex, rather than on each facet of the surface. To emulate the persp color handling, you need to do the following. First, convert the color vector to an (nx - 1) by (ny - 1) matrix; then add an extra row before row 1, and an extra column after the last column, to convert it to nx by ny. (These extra colors will not be used). For example, col <- rbind(1, cbind(matrix(col, nx - 1, ny - 1), 1)). Finally, call persp3d with material property smooth = FALSE.

See the "Clipping" section in plot3d for more details on xlim, ylim, zlim and forceClipregion.

Value

This function is called for the side effect of drawing the plot. A vector of shape IDs is returned invisibly.

Author(s)

Duncan Murdoch

See Also

plot3d, persp. There is a persp3d.function method for drawing functions, and persp3d.deldir can be used to draw surfaces defined by an irregular collection of points. A formula method persp3d.formula draws surfaces using this method.

The surface3d function is used to draw the surface without the axes etc.

Examples

```
# (1) The Obligatory Mathematical surface.
#     Rotated sinc function.

x <- seq(-10, 10, length.out = 20)
y <- x
f <- function(x, y) { r <- sqrt(x^2 + y^2); 10 * sin(r)/r }
z <- outer(x, y, f)
z[is.na(z)] <- 1
open3d()

# Draw the surface twice:  the first draws the solid part,
# the second draws the grid.  Offset the first so it doesn't
# obscure the lines.

persp3d(x, y, z, aspect = c(1, 1, 0.5), col = "lightblue",
        xlab = "X", ylab = "Y", zlab = "Sinc( r )",
        polygon_offset = 1)
persp3d(x, y, z, front = "lines", back = "lines",
        lit = FALSE, add = TRUE)
highlevel()   # trigger the plot

# (2) Add to existing persp plot:

xE <- c(-10, 10); xy <- expand.grid(xE, xE)
points3d(xy[, 1], xy[, 2], 6, col = "red")
lines3d(x, y = 10, z = 6 + sin(x), col = "green")

phi <- seq(0, 2*pi, length.out = 201)
r1 <- 7.725 # radius of 2nd maximum
xr <- r1 * cos(phi)
yr <- r1 * sin(phi)
lines3d(xr, yr, f(xr, yr), col = "pink", lwd = 2)

# (3) Visualizing a simple DEM model

z <- 2 * volcano        # Exaggerate the relief
x <- 10 * (1:nrow(z))   # 10 meter spacing (S to N)
y <- 10 * (1:ncol(z))   # 10 meter spacing (E to W)

open3d()
invisible(bg3d("slategray")) # suppress display
material3d(col = "black")
persp3d(x, y, z, col = "green3", aspect = "iso",
```

```
      axes = FALSE, box = FALSE)

# (4) A globe

lat <- matrix(seq(90, -90, length.out = 50)*pi/180, 50, 50, byrow = TRUE)
long <- matrix(seq(-180, 180, length.out = 50)*pi/180, 50, 50)

r <- 6378.1 # radius of Earth in km
x <- r*cos(lat)*cos(long)
y <- r*cos(lat)*sin(long)
z <- r*sin(lat)

open3d()
persp3d(x, y, z, col = "white",
       texture = system.file("textures/worldsmall.png", package = "rgl"),
       specular = "black", axes = FALSE, box = FALSE, xlab = "", ylab = "", zlab = "",
       normal_x = x, normal_y = y, normal_z = z)

## Not run:
# This looks much better, but is slow because the texture is very big
persp3d(x, y, z, col = "white",
       texture = system.file("textures/world.png", package = "rgl"),
       specular = "black", axes = FALSE, box = FALSE, xlab = "", ylab = "", zlab = "",
       normal_x = x, normal_y = y, normal_z = z)

## End(Not run)
```

`persp3d.deldir` *Plot a Delaunay triangulation*

Description

The `deldir()` function in the **deldir** package computes a Delaunay triangulation of a set of points. These functions display it as a surface.

Usage

```
## S3 method for class 'deldir'
plot3d(x, ...)
## S3 method for class 'deldir'
persp3d(x, ..., add = FALSE)
## S3 method for class 'deldir'
as.mesh3d(x, col = "gray", coords = c("x", "y", "z"),
    smooth = TRUE, normals = NULL, texcoords = NULL, ...)
```

Arguments

x	A "deldir" object, produced by the deldir() function. It must contain z values.
add	Whether to add surface to existing plot (add = TRUE) or create a new plot (add = FALSE, the default).
col	Colors to apply to each vertex in the triangulation. Will be recycled as needed.
coords	See Details below.
smooth	Whether to average normals at vertices for a smooth appearance.
normals	User-specified normals at each vertex. Requires smooth = FALSE.
texcoords	Texture coordinates at each vertex.
...	See Details below.

Details

These functions construct a mesh3d object corresponding to the triangulation in x. The plot3d and persp3d methods plot it.

The coords parameter allows surfaces to be plotted over any coordinate plane. It should be a permutation of the column names c("x", "y", "z") from the "deldir" object. The first will be used as the x coordinate, the second as the y coordinate, and the third as the z coordinate.

The ... parameters in plot3d.deldir are passed to persp3d.deldir; in persp3d.deldir they are passed to both as.mesh3d.deldir and persp3d.mesh3d; in as.mesh3d.deldir they are used as material parameters in a tmesh3d call.

Examples

```
x <- rnorm(200, sd = 5)
y <- rnorm(200, sd = 5)
r <- sqrt(x^2 + y^2)
z <- 10 * sin(r)/r
col <- cm.colors(20)[1 + round(19*(z - min(z))/diff(range(z)))]

save <- options(rgl.meshColorWarning = FALSE)

# This code is awkward:  to work with demo(rglExamples),
# we need auto-printing of the plots.  This means we
# have to repeat the test for deldir.

haveDeldir <- checkDeldir()

if (haveDeldir) {
  dxyz <- deldir::deldir(x, y, z = z, suppressMsge = TRUE)
  persp3d(dxyz, col = col)
}

if (haveDeldir) {
  open3d()
  # Do it without smoothing and with a different orientation.
  persp3d(dxyz, col = col, coords = c("z", "x", "y"), smooth = FALSE)
```

```
}

options(save)
```

`persp3d.function` *Plot a function of two variables*

Description

Plot a function `z(x, y)` or a parametric function `(x(s, t), y(s, t), z(s, t))`.

Usage

```
## S3 method for class '`function`'
persp3d(x,
  xlim = c(0, 1), ylim = c(0, 1),
  slim = NULL, tlim = NULL,
  n = 101,
  xvals = seq.int(min(xlim), max(xlim), length.out = n[1]),
  yvals = seq.int(min(ylim), max(ylim), length.out = n[2]),
  svals = seq.int(min(slim), max(slim), length.out = n[1]),
  tvals = seq.int(min(tlim), max(tlim), length.out = n[2]),
  xlab, ylab, zlab,
  col = "gray", otherargs = list(),
  normal = NULL, texcoords = NULL, ...)
## S3 method for class '`function`'
plot3d(x, ...)
```

Arguments

`x`	A function of two arguments. See the details below.
`xlim`, `ylim`	By default, the range of x and y values. For a parametric surface, if these are not missing, they are used as limits on the displayed x and y values.
`slim`, `tlim`	If not `NULL`, these give the range of s and t in the parametric specification of the surface. If only one is given, the other defaults to `c(0, 1)`.
`n`	A one or two element vector giving the number of steps in the x and y (or s and t) grid.
`xvals`, `yvals`	The values at which to evaluate x and y. Ignored for a parametric surface. If used, `xlim` and/or `ylim` are ignored.
`svals`, `tvals`	The values at which to evaluate s and t for a parametric surface. Only used if `slim` or `tlim` is not `NULL`. As with `xvals` and `yvals`, these override the corresponding `slim` or `tlim` specification.
`xlab`, `ylab`, `zlab`	The axis labels. See the details below for the defaults.
`col`	The color to use for the plot. See the details below.
`otherargs`	Additional arguments to pass to the function.

`normal`, `texcoords`
: Functions to set surface normals or texture coordinates. See the details below.

`...`
: Additional arguments to pass to `persp3d`.

Details

The `"function"` method for `plot3d` simply passes all arguments to `persp3d`. Thus this description applies to both.

The first argument `x` is required to be a function. It is named `x` only because of the requirements of the S3 system; in the remainder of this help page, we will assume that the assignment `f <- x` has been made, and will refer to the function `f()`.

`persp3d.function` evaluates `f()` on a two-dimensional grid of values, and displays the resulting surface. The values on the grid will be passed in as vectors in the first two arguments to the function, so `f()` needs to be vectorized. Other optional arguments to `f()` can be specified in the `otherargs` list.

In the default form where `slim` and `tlim` are both `NULL`, it is assumed that `f(x, y)` returns heights, which will be plotted in the z coordinate. The default axis labels will be taken from the argument names to `f()` and the expression passed as argument `x` to this function.

If `slim` or `tlim` is specified, a parametric surface is plotted. The function `f(s, t)` must return a 3-column matrix, giving x, y and z coordinates of points on the surface. The default axis labels will be the column names if those are present. In this case `xlim`, `ylim` and `zlim` are used to define a clipping region only if specified; the defaults are ignored.

The color of the surface may be specified as the name of a color, or a vector or matrix of color names. In this case the colors will be recycled across the points on the grid of values.

Alternatively, a function may be given: it should be a function like `rainbow` that takes an integer argument and returns a vector of colors. In this case the colors are mapped to z values.

The `normal` argument allows specification of a function to compute normal vectors to the surface. This function is passed the same arguments as `f()` (including `otherargs` if present), and should produce a 3-column matrix containing the x, y and z coordinates of the normals.

The `texcoords` argument is a function similar to `normal`, but it produces a 2-column matrix containing texture coordinates.

Both `normal` and `texcoords` may also contain matrices, with 3 and 2 columns respectively, and rows corresponding to the points that were passed to `f()`.

Value

This function constructs a call to `persp3d` and returns the value from that function.

Author(s)

Duncan Murdoch

See Also

The `curve` function in base graphics does something similar for functions of one variable. See the example below for space curves.

Examples

```
# (1) The Obligatory Mathematical surface.
#     Rotated sinc function, with colors

f <- function(x, y) {
  r <- sqrt(x^2 + y^2)
  ifelse(r == 0, 10, 10 * sin(r)/r)
}
open3d()
plot3d(f, col = colorRampPalette(c("blue", "white", "red")),
       xlab = "X", ylab = "Y", zlab = "Sinc( r )",
       xlim = c(-10, 10), ylim = c(-10, 10),
       aspect = c(1, 1, 0.5))

# (2) A cylindrical plot

f <- function(s, t) {
  r <- 1 + exp( -pmin( (s - t)^2,
                       (s - t - 1)^2,
                       (s - t + 1)^2 )/0.01 )
  cbind(r*cos(t*2*pi), r*sin(t*2*pi), s)
}

open3d()
plot3d(f, slim = c(0, 1), tlim = c(0, 1), col = "red", alpha = 0.8)

# Add a curve to the plot, fixing s at 0.5.

plot3d(f(0.5, seq.int(0, 1, length.out = 100)), type = "l", add = TRUE,
       lwd = 3, depth_test = "lequal")
```

`persp3d.triSht` *Plot an interp Delaunay triangulation*

Description

The `tri.mesh()` function in the **interp** package computes a Delaunay triangulation of a set of points. These functions display it as a surface.

Usage

```
## S3 method for class 'triSht'
plot3d(x, z, ...)
## S3 method for class 'triSht'
persp3d(x, z, ..., add = FALSE)
## S3 method for class 'triSht'
as.mesh3d(x, z, col = "gray", coords = c("x", "y", "z"),
    smooth = TRUE, normals = NULL, texcoords = NULL, ...)
```

Arguments

x	A "triSht" object, produced by the tri.mesh() function in the **interp** package.
z	z coordinate values corresponding to each of the nodes in x.
add	Whether to add surface to existing plot (add = TRUE) or create a new plot (add = FALSE, the default).
col	Colors to apply to each vertex in the triangulation. Will be recycled as needed.
coords	See Details below.
smooth	Whether to average normals at vertices for a smooth appearance.
normals	User-specified normals at each vertex. Requires smooth = FALSE.
texcoords	Texture coordinates at each vertex.
...	See Details below.

Details

These functions construct a mesh3d object corresponding to the triangulation in x. The plot3d and persp3d methods plot it.

The coords parameter allows surfaces to be plotted over any coordinate plane. It should be a permutation of the column names c("x", "y", "z"). The first will be used as the x coordinate, the second as the y coordinate, and the third as the z coordinate.

The ... parameters in plot3d.triSht and plot3d.tri are passed to persp3d; in persp3d.triSht and persp3d.tri they are passed to both as.mesh3d and persp3d.mesh3d; in as.mesh3d.triSht and as.mesh3d.tri they are used as material parameters in a tmesh3d call.

"tri" objects may contain constraints. These appear internally as extra nodes, representing either the inside or outside of boundaries on the region being triangulated. Each of these nodes should also have a z value, but triangles corresponding entirely to constraint nodes will not be drawn. In this way complex, non-convex regions can be triangulated. See the second example below.

Note

If there are duplicate points, the tri.mesh() functions will optionally delete some of them. If you choose this option, the z values must correspond to the nodes *after* deletion, not before.

Examples

```
x <- rnorm(200, sd = 5)
y <- rnorm(200, sd = 5)
r <- sqrt(x^2 + y^2)
z <- 10 * sin(r)/r
col <- cm.colors(20)[1 + round(19*(z - min(z))/diff(range(z)))]
save <- NULL
if ((haveinterp <- requireNamespace("interp", quietly = TRUE))) {
  save <- options(rgl.meshColorWarning = FALSE)
  dxy <- interp::tri.mesh(x, y)
  open3d()
  persp3d(dxy, z, col = col, meshColor = "vertices")
```

```
}
if (haveinterp) {
  open3d()
  # Do it without smoothing and with a different orientation.
  persp3d(dxy, z, col = col, coords = c("z", "x", "y"), smooth = FALSE)
}

options(save)
```

---

planes3d *Add planes*

Description

planes3d adds mathematical planes to a scene. Their intersection with the current bounding box will be drawn. clipplanes3d adds clipping planes to a scene.

Usage

```
planes3d(a, b = NULL, c = NULL, d = 0, ...)
clipplanes3d(a, b = NULL, c = NULL, d = 0)
```

Arguments

a, b, c	Coordinates of the normal to the plane. Any reasonable way of defining the coordinates is acceptable. See the function xyz.coords for details.
d	Coordinates of the "offset". See the details.
...	Material properties. See material3d for details.

Details

planes3d draws planes using the parametrization $ax + by + cz + d = 0$. Multiple planes may be specified by giving multiple values for any of a, b, c, d; the other values will be recycled as necessary.

clipplanes3d defines clipping planes using the same equations. Clipping planes suppress the display of other objects (or parts of them) in the subscene, based on their coordinates. Points (or parts of lines or surfaces) where the coordinates x, y, z satisfy $ax + by + cz + d < 0$ will be suppressed.

The number of clipping planes supported by the OpenGL driver is implementation dependent; use par3d("maxClipPlanes") to find the limit.

Value

A shape ID of the planes or clipplanes object is returned invisibly.

See Also

abclines3d for mathematical lines.

triangles3d or the corresponding functions for quadrilaterals may be used to draw sections of planes that do not adapt to the bounding box.

The example in subscene3d shows how to combine clipping planes to suppress complex shapes.

Examples

```
# Show regression plane with z as dependent variable

open3d()
x <- rnorm(100)
y <- rnorm(100)
z <- 0.2*x - 0.3*y + rnorm(100, sd = 0.3)
fit <- lm(z ~ x + y)
plot3d(x, y, z, type = "s", col = "red", size = 1)

coefs <- coef(fit)
a <- coefs["x"]
b <- coefs["y"]
c <- -1
d <- coefs["(Intercept)"]
planes3d(a, b, c, d, alpha = 0.5)

open3d()
ids <- plot3d(x, y, z, type = "s", col = "red", size = 1, forceClipregion = TRUE)
oldid <- useSubscene3d(ids["clipregion"])
clipplanes3d(a, b, c, d)
useSubscene3d(oldid)
```

---

play3d *Play animation of RGL scene*

Description

play3d calls a function repeatedly, passing it the elapsed time in seconds, and using the result of the function to reset the viewpoint. movie3d does the same, but records each frame to a file to make a movie.

Usage

```
play3d(f, duration = Inf, dev = cur3d(), ..., startTime = 0)
movie3d(f, duration, dev = cur3d(), ..., fps = 10,
                     movie = "movie", frames = movie, dir = tempdir(),
                     convert = NULL, clean = TRUE, verbose = TRUE,
                     top = !rgl.useNULL(), type = "gif", startTime = 0,
                     webshot = TRUE)
```

Arguments

f	A function returning a list that may be passed to par3d
duration	The duration of the animation
dev	Which RGL device to select
...	Additional parameters to pass to f.
startTime	Initial time at which to start the animation
fps	Number of frames per second
movie	The base of the output filename, not including .gif
frames	The base of the name for each frame
dir	A directory in which to create temporary files for each frame of the movie
convert	How to convert to a GIF movie; see Details
clean	If convert is NULL or TRUE, whether to delete the individual frames
verbose	Whether to report the convert command and the output filename
top	Whether to call rgl.bringtotop before each frame
type	What type of movie to create. See Details.
webshot	Whether to use the **webshot2** package for snapshots of frames. See snapshot3d.

Details

The function f will be called in a loop with the first argument being the startTime plus the time in seconds since the start (where the start is measured after all arguments have been evaluated).

play3d is likely to place a high load on the CPU; if this is a problem, calls to Sys.sleep should be made within the function to release time to other processes.

play3d will run for the specified duration (in seconds), but can be interrupted by pressing ESC while the RGL window has the focus.

movie3d saves each frame to disk in a filename of the form 'framesXXX.png', where XXX is the frame number, starting from 0. If convert is NULL (the default) and the **magick** package is installed, it will be used to convert the frames to a GIF movie (or other format if supported). If **magick** is not installed or convert is TRUE, movie3d will attempt to use the external ImageMagick program to convert the frames to a movie. The newer magick executable is tried first, then convert if that fails. The type argument will be passed to ImageMagick to use as a file extension to choose the file type.

Finally, convert can be a template for a command to execute in the standard shell (wildcards are allowed). The template is converted to a command using
sprintf(convert, fps, frames, movie, type, duration, dir)
For example, convert = TRUE uses the template "magick -delay 1x%d %s*.png %s.%s". All work is done in the directory dir, so paths should not be needed in the command. (Note that sprintf does not require all arguments to be used, and supports formats that use them in an arbitrary order.)

The top = TRUE default is designed to work around an OpenGL limitation: in some implementations, rgl.snapshot will fail if the window is not topmost.

As of **rgl** version 0.94, the dev argument is not needed: the function f can specify its device, as spin3d does, for example. However, if dev is specified, it will be selected as the current device as each update is played.

As of **rgl** version 0.95.1476, `f` can include multiple values in a `"subscene"` component, and `par3d()` will be called for each of them.

Value

`play3d` is called for the side effect of its repeated calls to `f`. It returns `NULL` invisibly.

`movie3d` is also normally called for the side effect of producing the output movie. It invisibly returns

Author(s)

Duncan Murdoch, based on code by Michael Friendly

See Also

`spin3d` and `par3dinterp` return functions suitable to use as `f`. See `demo(flag)` for an example that modifies the scene in `f`.

Examples

```
open3d()
plot3d( cube3d(col = "green") )
M <- par3d("userMatrix")
if (!rgl.useNULL() && interactive())
  play3d( par3dinterp(times = (0:2)*0.5, userMatrix = list(M,
                                    rotate3d(M, pi/2, 1, 0, 0),
                                    rotate3d(M, pi/2, 0, 1, 0) ) ),
        duration = 2 )
## Not run:
movie3d( spin3d(), duration = 5 )

## End(Not run)
```

`playwidget` *Add a widget to play animations*

Description

This is a widget that can be put in a web page to allow animations with or without Shiny.

Usage

```
playwidget(sceneId, controls,
            start = 0, stop = Inf, interval = 0.05, rate = 1,
            components = c("Reverse", "Play", "Slower", "Faster",
                           "Reset", "Slider", "Label"),
            loop = TRUE,
            step = 1, labels = NULL,
```

```
precision = 3,
elementId = NULL, respondTo = NULL,
reinit = NULL,
buttonLabels = components, pause = "Pause",
height = 40,
...)
```

Arguments

sceneId	The HTML id of the RGL scene being controlled, or an object. See the Details below.
controls	A single "rglControl" object, e.g. propertyControl, or a list of several.
start, stop	The starting and stopping values of the animation. If labels is supplied stop will default to step through the labels.
interval	The requested interval (in seconds) between updates. Updates may occur at longer intervals.
rate	The number of units of "nominal" time per real world second.
components	Which components should be displayed? See Details below.
loop	When the player reaches the end of the interval, should it loop back to the beginning?
step	Step size in the slider.
labels	Optional labels to use, corresponding to slider steps. Set to NULL for autogenerated labels.
precision	If labels=NULL, the precision to use when displaying timer values.
elementId	The HTML id of the generated widget, containing buttons, slider, etc.
respondTo	The HTML ID of a Shiny input control (e.g. a sliderInput control) to respond to.
reinit	A vector of ids that will need re-initialization before being drawn again.
buttonLabels, pause	These are the labels that will be shown on the buttons if they are displayed. pause will be shown on the "Play" button while playing.
height	The height of the widget in pixels. In a pipe, this is a relative height.
...	Additional arguments to pass to to htmlwidgets::createWidget.

Details

The components are buttons to control the animation, a slider for manual control, and a label to show the current value. They will be displayed in the order given in components. Not all need be included.

The buttons have the following behaviour:

Reverse Reverse the direction.

Play Play the animation.

Slower Decrease the playing speed.

Faster Increase the playing speed.

Reset Stop the animation and reset to the start value.

If `respondTo` is used, no `components` are shown, as it is assumed Shiny (or whatever control is being referenced) will provide the UI components.

The `sceneId` component can be another `playwidget`, a `rglwidget` result, or a result of `htmltools::tags` or `htmltools::tagList`. This allows you to use a **magrittr**-style "pipe" command to join an `rglwidget` with one or more `playwidget`s. If a `playwidget` comes first, `sceneId` should be set to `NA`. If the `rglwidget` does not come first, previous values should be piped into its `controllers` argument. Other HTML code (including other widgets) can be used in the chain if wrapped in `htmltools::tagList`.

Each control should inherit from `"rglControl"`. They can have the following components in addition to any private ones:

`labels` default labels for the slider.

`param` values to include on the slider.

`dependencies` additional HTML dependencies to include, after the default `rglwidgetClass`.

Value

A widget suitable for use in an **Rmarkdown**-generated web page, or elsewhere.

Appearance

The appearance of the controls is set by the stylesheet in `system.file("htmlwidgets/lib/rglClass/rgl.css")`.

The overall widget is of class `rglPlayer`, with id set according to `elementId`.

The buttons are of HTML class `rgl-button`, the slider is of class `rgl-slider`, and the label is of class `rgl-label`. Each element has an id prefixed by the widget id, e.g. `elementId-button-Reverse`, `elementId-slider`, etc. (where `elementId` should be replaced by the actual id).

The `reinit` parameter handles the case where an object needs re-initialization after each change. For example, plane objects may need this if their intersection with the bounding box changes shape. Note that re-initialization is generally incompatible with the `vertexControl` as it modifies values which are set during initialization.

Author(s)

Duncan Murdoch

See Also

`subsetControl`, `propertyControl`, `ageControl` and `vertexControl` are possible controls to use.

`toggleWidget` is a wrapper for `playwidget` and `subsetControl` to insert a single button to toggle some elements in a display.

Examples

```
saveopts <- options(rgl.useNULL = TRUE)

objid <- plot3d(1:10, 1:10, rnorm(10), col=c("red", "red"), type = "s")["data"]

control <- ageControl(value=0,
             births=1:10,
             ages = c(-5,0,5),
             colors = c("green", "yellow", "red"),
             objids = objid)


# This example uses explicit names
rglwidget(elementId = "theplot", controllers = "theplayer",
          height = 300, width = 300)
playwidget("theplot", control, start = -5, stop = 5,
           rate = 3, elementId = "theplayer",
           components = c("Play", "Slider"))


# This example uses pipes, and can skip the names

widget <- rglwidget(height = 300, width = 300) %>%
playwidget(control, start = -5, stop = 5,
           rate = 3, components = c("Play", "Slider"))
if (interactive() || in_pkgdown_example())
  widget

options(saveopts)
```

---

plot3d *3D scatterplot*

Description

Draws a 3D scatterplot.

Usage

```
plot3d(x, ...)
## Default S3 method:
plot3d(x, y, z,
 xlab, ylab, zlab, type = "p",
 col, size, lwd, radius,
 add = FALSE, aspect = !add,
 xlim = NULL, ylim = NULL, zlim = NULL,
 forceClipregion = FALSE,
 decorate = !add, ...)
## S3 method for class 'mesh3d'
```

```
plot3d(x, xlab = "x", ylab = "y", zlab = "z", type = c("shade", "wire", "dots"),
 add = FALSE, aspect = !add, ...)
```

Arguments

x, y, z	vectors of points to be plotted. Any reasonable way of defining the coordinates is acceptable. See the function xyz.coords for details.
xlab, ylab, zlab	labels for the coordinates.
type	For the default method, a single character indicating the type of item to plot. Supported types are: 'p' for points, 's' for spheres, 'l' for lines, 'h' for line segments from z = 0, and 'n' for nothing. For the mesh3d method, one of 'shade', 'wire', or 'dots'. Partial matching is used.
col	the color to be used for plotted items.
size	the size for plotted points.
lwd	the line width for plotted items.
radius	the radius of spheres: see Details below.
add	whether to add the points to an existing plot.
aspect	either a logical indicating whether to adjust the aspect ratio, or a new ratio.
xlim, ylim, zlim	If not NULL, set clipping limits for the plot.
forceClipregion	Force a clipping region to be used, whether or not limits are given.
decorate	Whether to add bounding axes and other decorations.
...	additional parameters which will be passed to par3d, material3d or decorate3d.

Details

plot3d is a partial 3D analogue of plot.default.

Missing values in the data are skipped, as in standard graphics.

If aspect is TRUE, aspect ratios of c(1, 1, 1) are passed to aspect3d. If FALSE, no aspect adjustment is done. In other cases, the value is passed to aspect3d.

With type = "s", spheres are drawn centered at the specified locations. The radius may be controlled by size (specifying the size relative to the plot display, with the default size = 3 giving a radius about 1/20 of the plot region) or radius (specifying it on the data scale if an isometric aspect ratio is chosen, or on an average scale if not).

Value

plot3d is called for the side effect of drawing the plot; a vector of object IDs is returned.

Clipping

If any of xlim, ylim or zlim are specified, they should be length two vectors giving lower and upper clipping limits for the corresponding coordinate. NA limits will be ignored.

If any clipping limits are given, then the data will be plotted in a newly created subscene within the current one; otherwise plotting will take place directly in the current subscene. This subscene

is named "clipregion" in the results. This may affect the appearance of transparent objects if some are drawn in the plot3d call and others after, as RGL will not attempt to depth-sort objects if they are in different subscenes. It is best to draw all overlapping transparent objects in the same subscene. See the example in planes3d. It will also affect the use of clipplanes3d; clipping planes need to be in the same subscene as the objects being clipped.

Use forceClipregion = TRUE to force creation of this subscene even without specifying limits.

Author(s)

Duncan Murdoch

See Also

plot.default, open3d, par3d. There are plot3d.function and plot3d.deldir methods for plotting surfaces.

Examples

```
open3d()
x <- sort(rnorm(1000))
y <- rnorm(1000)
z <- rnorm(1000) + atan2(x, y)
plot3d(x, y, z, col = rainbow(1000))
```

---

plot3d.formula *Methods for formulas*

Description

These functions provide a simple formula-based interface to plot3d and persp3d.

Usage

```
## S3 method for class 'formula'
plot3d(x, data = NULL, xlab, ylab, zlab, ...)
## S3 method for class 'formula'
persp3d(x, data = NULL, xlab, ylab, zlab, ...)
```

Arguments

x	A formula like z ~ x + y.
data	An optional dataframe or list in which to find the components of the formula.
xlab, ylab, zlab	Optional axis labels to override the ones automatically obtained from the formula.
...	Additional arguments to pass to the default plot3d method, or the persp3d method for "deldir" objects.

Details

Only simple formulas (the ones handled by the `xyz.coords` function) are supported: a single variable on the left hand side (which will be plotted on the Z axis), and a sum of two variables on the right hand side (which will be the X and Y axis variables in the plot.)

Value

These functions are called for the side effect of drawing the plots. The `plot3d` method draws a scatterplot. The `persp3d` method draws a surface plot.

Return values are as given by the `plot3d.default` method or the `persp3d.deldir` methods.

Note

The `persp3d` method requires that the suggested package **deldir** is installed.

Author(s)

Duncan Murdoch

Examples

```
open3d()
mfrow3d(1, 2, sharedMouse = TRUE)
plot3d(mpg ~ wt + qsec, data = mtcars)
if (checkDeldir())
  persp3d(mpg ~ wt + qsec, data = mtcars)
```

---

plot3d.lm *Method for plotting simple linear fit*

Description

This function provides several plots of the result of fitting a two-predictor model.

Usage

```
## S3 method for class 'lm'
plot3d(x,
       which = 1,
       plane.col = "gray", plane.alpha = 0.5,
       sharedMouse = TRUE,
       use_surface3d,
       do_grid = TRUE,
       grid.col = "black",
       grid.alpha = 1,
       grid.steps = 5,
       sub.steps = 4,
```

```
        vars = get_all_vars(terms(x), x$model),
        clip_to_density = 0,
        ...)
```

Arguments

x	An object inheriting from class "lm" obtained by fitting a two-predictor model.
which	Which plot to show? See Details below.
plane.col, plane.alpha	These parameters control the colour and transparency of a plane or surface.
sharedMouse	If multiple plots are requested, should they share mouse controls, so that they move in sync?
use_surface3d	Use the surface3d function to plot the surface rather than planes3d. This allows curved surfaces to be shown. The default is FALSE if the model looks like a simple 2 parameter linear fit, otherwise TRUE.
do_grid	Plot a grid.
grid.col, grid.alpha, grid.steps	Characteristics of the grid.
sub.steps	If use_surface3d is TRUE, use an internal grid of grid.steps*sub.steps to draw the surface. sub.steps > 1 allows curvature within facets. Similarly, if do_grid is TRUE, it allows curvature within grid lines.
vars	A dataframe containing the variables to plot in the first three columns, with the response assumed to be in column 1. See the Note below.
clip_to_density	If positive, the surface, plane or grid will be clipped to a region with sufficient data.
...	Other parameters to pass to the default plot3d method, to control the appearance of aspects of the plot other than the plane.

Details

Three plots are possible, depending on the value(s) in which:

1. (default) Show the points and the fitted plane or surface.
2. Show the residuals and the plane at z = 0.
3. Show the predicted values on the fitted plane or surface.

If clip_to_density is positive, then the surface, plane or grid will be clipped to the region where a non-parametric density estimate (using MASS::kde2d), normalized to have a maximum value of 1, is greater than the given value. This will suppress parts of the plot that aren't supported by the observed data.

Value

Called for the side effect of drawing one or more plots.

Invisibly returns a high-level vector of object ids. Names of object ids have the plot number (in drawing order) appended.

Note

The default value for the `vars` argument will handle simple linear models with a response and two predictors, and some models with functions of those two predictors. For models that fail (e.g. models using `poly`), you can include the observed values as in the third example below.

If `clip_to_density > 0`,

1. The clipping is approximate, so it may not agree perfectly between surfaces, planes and grids.
2. This option requires the suggested packages **MASS** and **interp**, and will be ignored with a warning if either is not installed.

Author(s)

Duncan Murdoch

Examples

```
open3d()
ids <- plot3d(lm(mpg ~ wt + qsec, data = mtcars), which = 1:3)
names(ids)

open3d()
plot3d(lm(mpg ~ wt + I(wt^2) + qsec, data = mtcars))

open3d()
# Specify vars in the order:  response, pred1, pred2.
plot3d(lm(mpg ~ poly(wt, 3) + qsec, data = mtcars),
       vars = mtcars[,c("mpg", "wt", "qsec")])

open3d()
# Clip parts of the plot with few (wt, qsec) points
plot3d(lm(mpg ~ poly(wt, 3) + qsec, data = mtcars),
       vars = mtcars[,c("mpg", "wt", "qsec")],
       clip_to_density = 0.1)
```

`plotmath3d` *Draw text using base graphics math plotting or LaTeX*

Description

To plot mathematical text, `plotmath3d` uses base graphics functions to plot it to a '`.png`' file, then uses that file as a texture in a sprite. `latex3d` uses the **xdvir** package to render LaTeX code, then uses the same approach to display it in **rgl**.

Usage

```
plotmath3d(x, y = NULL, z = NULL, text, cex = par3d("cex"),
           adj = 0.5, pos = NULL, offset = 0.5,
           fixedSize = TRUE, startsize = 480, initCex = 5,
           margin = "", floating = FALSE, tag = "",
           polygon_offset = material3d("polygon_offset"), ...)
latex3d(x, y = NULL, z = NULL, text, cex = par3d("cex"),
           adj = 0.5, pos = NULL, offset = 0.5,
           fixedSize = TRUE, startsize = 480, initCex = 5,
           margin = "", floating = FALSE, tag = "",
           polygon_offset = material3d("polygon_offset"),
           verbose = FALSE, ...)
```

Arguments

x, y, z	Coordinates. Any reasonable way of defining the coordinates is acceptable. See the function xyz.coords for details.
text	A character vector or (in plotmath3d) expression. See plotmath for how expressions are interpreted in plotmath3d. In latex3d each text entry should be a LaTeX fragment.
cex	Character size expansion.
adj	One value specifying the horizontal adjustment, or two, specifying horizontal and vertical adjustment respectively, or three, for depth as well.
pos, offset	Alternate way to specify adj; see text3d
fixedSize	Should the resulting sprite behave like the default ones, and resize with the scene, or like text, and stay at a fixed size?
startsize, initCex	These parameters are unlikely to be needed by users. startsize is an overestimate of the size (in pixels) of the largest expression. Increase this if large expressions are cut off. initCex is the size of text used to form the bitmap. Increase this if letters look too blurry at the desired size.
margin, floating, tag, polygon_offset	material3d properties for the sprites.
verbose	If TRUE, latex3d will print intermediate results for debugging.
...	For plotmath3d, additional arguments to pass to text when drawing the text. For latex3d, additional arguments to pass to xdvir::latexGrob.

Value

Called for the side effect of displaying the sprites. The shape ID of the displayed object is returned.

Note

The text3d function passes calls to plotmath3d if its usePlotmath argument is TRUE. This is the default value if its texts argument looks like an expression.

The latex3d function produces nicer looking results than plotmath3d, but it is much slower, especially on the first run.

Author(s)

Duncan Murdoch

See Also

text3d

Examples

```
open3d()
plotmath3d(1:3, 1:3, 1:3, expression(x[1] == 1, x[2] == 2, x[3] == 3))
# This lets the text resize with the plot
text3d(4, 4, 4, "resizeable text", usePlotmath = TRUE, fixedSize = FALSE)

if (requireNamespace("xdvir")) {

# Do the same plot using latex3d(). This example runs slowly!

  open3d()
  latex3d(1:3, 1:3, 1:3, c("$x_1 = 1$", "$x_2 = 2$", "$x_3 = 3$"))
  latex3d(4, 4, 4, "resizeable text", fixedSize = FALSE)

}
```

---

polygon3d *Draw a polygon in three dimensions*

Description

This function takes a description of a flat polygon in x, y and z coordinates, and draws it in three dimensions.

Usage

```
polygon3d(x, y = NULL, z = NULL, fill = TRUE, plot = TRUE,
          coords, random = TRUE, ...)
```

Arguments

x, y, z	Vertices of the polygon in a form accepted by xyz.coords.
fill	logical; should the polygon be filled?
plot	logical; should the polygon be displayed?
coords	Which two coordinates (x = 1, y = 2, z = 3) describe the polygon. If missing, triangulate makes an automatic choice.
random	Currently ignored. The triangulation is deterministic.
...	Other parameters to pass to lines3d or shade3d if plot = TRUE.

Details

The function triangulates the two dimensional polygon described by `coords`, then applies the triangulation to all three coordinates. No check is made that the polygon is actually all in one plane, but the results may be somewhat unpredictable (especially if `random = TRUE`) if it is not.

Polygons need not be simple; use `NA` to indicate separate closed pieces. For `fill = FALSE` there are no other restrictions on the pieces, but for `fill = TRUE` the resulting two-dimensional polygon needs to be one that `triangulate` can handle.

Value

If `plot = TRUE`, the id number of the lines (for `fill = FALSE`) or triangles (for `fill = TRUE`) that have been plotted.

If `plot = FALSE`, then for `fill = FALSE`, a vector of indices into the XYZ matrix that could be used to draw the polygon. For `fill = TRUE`, a triangular mesh object representing the triangulation.

Author(s)

Duncan Murdoch

See Also

`extrude3d` for a solid extrusion of a polygon, `triangulate` for the triangulation.

Examples

```
theta <- seq(0, 4*pi, length.out = 50)
r <- theta + 1
r <- c(r[-50], rev(theta*0.8) + 1)
theta <- c(theta[-50], rev(theta))
x <- r*cos(theta)
y <- r*sin(theta)
open3d()
plot(x, y, type = "n")
polygon(x, y)
polygon3d(x, y, x + y, col = "blue")
```

---

primitives *Add primitive shape*

Description

Adds a shape node to the current scene.

Usage

```
points3d(x, y = NULL, z = NULL,  ...)
lines3d(x, y = NULL, z = NULL,  ...)
segments3d(x, y = NULL, z = NULL, ...)
triangles3d(x, y = NULL, z = NULL, ...)
quads3d(x, y = NULL, z = NULL, ...)
```

Arguments

x, y, z	coordinates. Any reasonable way of defining the coordinates is acceptable. See the function xyz.coords for details.
...	Material properties (see material3d), normals, texcoords or indices; see details below.

Details

The functions points3d, lines3d, segments3d, triangles3d and quads3d add points, joined lines, line segments, filled triangles or quadrilaterals to the plots. They correspond to the OpenGL types GL_POINTS, GL_LINE_STRIP, GL_LINES, GL_TRIANGLES and GL_QUADS respectively.

Points are taken in pairs by segments3d, triplets as the vertices of the triangles, and quadruplets for the quadrilaterals. Colors are applied vertex by vertex; if different at each end of a line segment, or each vertex of a polygon, the colors are blended over the extent of the object. Polygons must be non-degenerate and quadrilaterals must be entirely in one plane and convex, or the results are undefined.

The appearance of the new objects are defined by the material properties. See material3d for details.

For triangles and quads, the normals at each vertex may be specified using normals. These may be given in any way that would be acceptable as a single argument to xyz.coords. These need not match the actual normals to the polygon: curved surfaces can be simulated by using other choices of normals.

Texture coordinates may also be specified. These may be given in any way that would be acceptable as a single argument to xy.coords, and are interpreted in terms of the bitmap specified as the material texture, with (0, 0) at the lower left, (1, 1) at the upper right. The texture is used to modulate the color of the polygon.

All of these functions support an argument called indices, which allows vertices (and other attributes) to be re-used, as they are in objects created by mesh3d and related functions. This is intended to be used on smooth surfaces, where each shared vertex has just one value for normals, colors and texture coordinates.

For shapes with flat-looking faces (e.g. polyhedra like cube3d), the vertices **must** be duplicated to be rendered properly.

Value

Each function returns the integer object ID of the shape that was added to the scene. These can be passed to pop3d to remove the object from the scene.

Author(s)

Ming Chen and Duncan Murdoch

Examples

```
# Show 12 random vertices in various ways.

M <- matrix(rnorm(36), 3, 12, dimnames = list(c('x', 'y', 'z'),
                                       rep(LETTERS[1:4], 3)))

# Force 4-tuples to be convex in planes so that quads3d works.

for (i in c(1, 5, 9)) {
    quad <- as.data.frame(M[, i + 0:3])
    coeffs <- runif(2, 0, 3)
    if (mean(coeffs) < 1) coeffs <- coeffs + 1 - mean(coeffs)
    quad$C <- with(quad, coeffs[1]*(B - A) + coeffs[2]*(D - A) + A)
    M[, i + 0:3] <- as.matrix(quad)
}

open3d()

# Rows of M are x, y, z coords; transpose to plot

M <- t(M)
shift <- matrix(c(-3, 3, 0), 12, 3, byrow = TRUE)

points3d(M)
lines3d(M + shift)
segments3d(M + 2*shift)
triangles3d(M + 3*shift, col = 'red')
quads3d(M + 4*shift, col = 'green')
text3d(M + 5*shift, texts = 1:12)

# Add labels

shift <- outer(0:5, shift[1, ])
shift[, 1] <- shift[, 1] + 3
text3d(shift,
       texts = c('points3d', 'lines3d', 'segments3d',
         'triangles3d', 'quads3d', 'text3d'),
       adj = 0)
 rgl.bringtotop()
```

---

propertyControl *Controls to use with playwidget()*

Description

These are setter functions to produce actions in a Shiny app, or in an animation.

Usage

```
subsetControl(value = 1, subsets, subscenes = NULL,
                         fullset = Reduce(union, subsets),
                         accumulate = FALSE)
propertyControl(value = 0, entries, properties,
                objids = tagged3d(tags), tags, values = NULL,
                param = seq_len(NROW(values)) - 1, interp = TRUE)
```

Arguments

value	The value to use for input (typically input$value in a Shiny app.)
subsets	A list of vectors of object identifiers; the value will choose among them.
fullset	Objects in the subscene which are not in fullset will not be touched.
subscenes	The subscenes to be controlled. If NULL, the root subscene.
accumulate	If TRUE, the subsets will accumulate (by union) as the value increases.
entries, properties, objids	Which properties to set.
tags	Select objects with matching tags. Ignored if objids is specified.
values	Values to set.
param	Parameter values corresponding to the rows of value
interp	Whether to use linear interpolation between param values

Details

subsetControl produces data for playwidget to display subsets of the object in one or more subscenes. This code will not touch objects in the subscenes if they are not in fullset. fullset defaults to the union of all the object ids mentioned in subsets, so by default if an id is not mentioned in one of the subsets, it will not be controlled by the slider. If value is specified in R code, it will be a 1-based index into the subsets list; when specified internally in Javascript, 0-based indexing into the corresponding array will be used.

propertyControl sets individual properties. Here the row of values is determined by the position of value in param.

Value

These functions return controller data in a list of class "rglControl".

Author(s)

Duncan Murdoch

See Also

subsetSetter for a way to embed a pure Javascript control, and playwidget for a way to use these in animations (including Shiny), rglShared for linking using the **crosstalk** package. The User Interaction in WebGL vignette gives more details.

r3d *Generic 3D interface*

Description

Generic 3D interface for 3D rendering and computational geometry.

Details

R3d is a design for an interface for 3d rendering and computation without dependency on a specific rendering implementation. R3d includes a collection of 3D objects and geometry algorithms. All r3d interface functions are named `*3d`. They represent generic functions that delegate to implementation functions.

The interface can be grouped into 8 categories: Scene Management, Primitive Shapes, High-level Shapes, Geometry Objects, Visualization, Interaction, Transformation, Subdivision.

The rendering interface gives an abstraction to the underlying rendering model. It can be grouped into four categories:

Scene Management: A 3D scene consists of shapes, lights and background environment.

Primitive Shapes: Generic primitive 3D graphics shapes such as points, lines, triangles, quadrangles and texts.

High-level Shapes: Generic high-level 3D graphics shapes such as spheres, sprites and terrain.

Interaction: Generic interface to select points in 3D space using the pointer device.

In this package we include an implementation of r3d using the underlying `rgl.*` functions.

3D computation is supported through the use of object structures that live entirely in R.

Geometry Objects: Geometry and mesh objects allow to define high-level geometry for computational purpose such as triangle or quadrangle meshes (see `mesh3d`).

Transformation: Generic interface to transform 3d objects.

Visualization: Generic rendering of 3d objects such as dotted, wired or shaded.

Computation: Generic subdivision of 3d objects.

At present, the main practical differences between the r3d functions and the `rgl.*` functions are as follows.

The r3d functions call `open3d` if there is no device open, and the `rgl.*` functions call `rgl.open`. By default `open3d` sets the initial orientation of the coordinate system in 'world coordinates', i.e. a right-handed coordinate system in which the x-axis increases from left to right, the y-axis increases with depth into the scene, and the z-axis increases from bottom to top of the screen. `rgl.*` functions, on the other hand, use a right-handed coordinate system similar to that used in OpenGL. The x-axis matches that of r3d, but the y-axis increases from bottom to top, and the z-axis decreases with depth into the scene. Since the user can manipulate the scene, either system can be rotated into the other one.

The r3d functions also preserve the `rgl.material` setting across calls (except for texture elements, in the current implementation), whereas the deprecated `rgl.*` functions leave it as set by the last call.

The example code below illustrates the two coordinate systems.

See Also

points3d, lines3d, segments3d, triangles3d, quads3d, text3d, spheres3d, sprites3d, terrain3d, select3d, dot3d, wire3d, shade3d, transform3d, rotate3d, subdivision3d, mesh3d, cube3d, rgl

Examples

```
x <- c(0, 1, 0, 0)
y <- c(0, 0, 1, 0)
z <- c(0, 0, 0, 1)
labels <- c("Origin", "X", "Y", "Z")
i <- c(1, 2, 1, 3, 1, 4)

# *3d interface

open3d()
text3d(x, y, z, labels)
text3d(1, 1, 1, "*3d coordinates")
segments3d(x[i], y[i], z[i])
```

---

readSTL *Read and write STL (stereolithography) format files*

Description

These functions read and write STL files. This is a simple file format that is commonly used in 3D printing. It does not represent text, only triangles. The writeSTL function converts some RGL object types to triangles.

Usage

```
readSTL(con, ascii = NA, plot = TRUE, ...)
writeSTL(con, ascii = FALSE,
         pointRadius = 0.005,
         pointShape = icosahedron3d(),
         lineRadius = pointRadius,
         lineSides = 20,
         ids = tagged3d(tags),
         tags = NULL)
```

Arguments

con	A connection or filename.
ascii	Whether to use the ASCII format or the binary format. The default NA setting for readSTL() causes it to detect the format. This only works for files, not other connections, which default to binary.
plot	On reading, should the object be plotted?

`...`	If plotting, other parameters to pass to `triangles3d`
`pointRadius, lineRadius`	The radius of points and lines relative to the overall scale of the figure.
`pointShape`	A mesh shape to use for points. It is scaled by the `pointRadius`.
`lineSides`	Lines are rendered as cylinders with this many sides.
`ids`	The identifiers (from `ids3d`) of the objects to write. If `NULL`, try to write everything.
`tags`	Alternate way to specify `ids`. Ignored if `ids` is given.

Details

The current implementation is limited. For reading, it ignores normals and color information. For writing, it only outputs triangles, quads, planes, spheres, points, line segments, line strips and surfaces, and does not write color information. Lines and points are rendered in an isometric scale: if your data scales vary, they will look strange.

Since the STL format only allows one object per file, all RGL objects are combined into a single object when output.

The output file is readable by Blender and Meshlab; the latter can write in a number of other formats, including U3D, suitable for import into a PDF document.

Value

`readSTL` invisibly returns the object id if `plot = TRUE`, or (visibly) a matrix of vertices of the triangles if not.

`writeSTL` invisibly returns the name of the connection to which the data was written.

Author(s)

Duncan Murdoch

References

The file format was found on Wikipedia on October 25, 2012. I learned about the STL file format from David Smith's blog reporting on Ian Walker's `r2stl` function.

See Also

`scene3d` saves a copy of a scene to an R variable; `rglwidget`, `writeASY`, `writePLY`, `writeOBJ` and `writeSTL` write the scene to a file in various other formats.

Examples

```
filename <- tempfile(fileext = ".stl")
open3d()
shade3d( icosahedron3d(col = "magenta") )
writeSTL(filename)
open3d()
readSTL(filename, col = "red")
```

rgl.attrib *Get information about shapes*

Description

Retrieves information about the shapes in a scene.

Usage

```
rgl.attrib(id, attrib, first = 1,
 last = rgl.attrib.count(id, attrib))
```

Arguments

id	A shape identifier, as returned by ids3d.
attrib	An attribute of a shape. Currently supported: one of "vertices", "normals", "colors", "texcoords", "dim", "texts", "cex", "adj", "radii", "centers", "ids", "usermatrix", "types", "flags", "offsets", "family", "font", "pos" or unique prefixes to one of those.
first, last	Specify these to retrieve only those rows of the result.

Details

If the identifier is not found or is not a shape that has the given attribute, zero will be returned by rgl.attrib.count, and an empty matrix will be returned by rgl.attrib.

The first four attrib names correspond to the usual OpenGL properties; "dim" is used just for surfaces, defining the rows and columns in the rectangular grid; "cex", "adj", "family", "font" and "pos" apply only to text objects.

Value

rgl.attrib returns the values of the attribute. Attributes are mostly real-valued, with the following sizes:

"vertices"	3 values	x, y, z
"normals"	3 values	x, y, z
"centers"	3 values	x, y, z
"colors"	4 values	r, g, b, a
"texcoords"	2 values	s, t
"dim"	2 values	r, c
"cex"	1 value	cex
"adj"	2 values	x, y
"radii"	1 value	r
"ids"	1 value	id
"usermatrix"	4 values	x, y, z, w

"texts"	1 value	text
"types"	1 value	type
"flags"	1 value	flag
"family"	1 value	family
"font"	1 value	font
"pos"	1 value	pos

The "texts", "types" and "family" attributes are character-valued; the "flags" attribute is logical valued, with named rows.

These are returned as matrices with the row count equal to the count for the attribute, and the columns as listed above.

Author(s)

Duncan Murdoch

See Also

ids3d, rgl.attrib.info

Examples

```
p <- plot3d(rnorm(100), rnorm(100), rnorm(100), type = "s", col = "red")
rgl.attrib(p["data"], "vertices", last = 10)
```

---

rgl.attrib.info *Get information about attributes of objects*

Description

These functions give information about the attributes of RGL objects. rgl.attrib.info is the more "user-friendly" function; rgl.attrib.count is a lower-level function more likely to be used in programming.

Usage

```
rgl.attrib.info(id = ids3d("all", 0)$id, attribs = NULL, showAll = FALSE)
rgl.attrib.count(id, attrib)
```

Arguments

id	One or more RGL object ids.
attribs	A character vector of one or more attribute names.
showAll	Should attributes with zero entries be shown?
attrib	A single attribute name.

Details

See the first example below to get the full list of attribute names.

Value

A dataframe containing the following columns:

`id`	The id of the object.
`attrib`	The full name of the attribute.
`nrow`, `ncol`	The size of matrix that would be returned by `rgl.attrib` for this attribute.

Author(s)

Duncan Murdoch

See Also

`rgl.attrib` to obtain the attribute values.

Examples

```
open3d()
id <- points3d(rnorm(100), rnorm(100), rnorm(100), col = "green")
rgl.attrib.info(id, showAll = TRUE)
rgl.attrib.count(id, "vertices")

merge(rgl.attrib.info(), ids3d("all"))
```

`rgl.bringtotop` *Assign focus to an RGL window*

Description

'rgl.bringtotop' brings the current RGL window to the front of the window stack (and gives it focus).

Usage

```
rgl.bringtotop(stay = FALSE)
```

Arguments

`stay`	whether to make the window stay on top.

Details

If `stay` is `TRUE`, then the window will stay on top of normal windows.

Note

not completely implemented for X11 graphics (stay not implemented; window managers such as KDE may block this action (set "Focus stealing prevention level" to None in Control Center/Window Behavior/Advanced)). Not currently implemented under OS/X.

Author(s)

Ming Chen/Duncan Murdoch

Examples

```
open3d()
points3d(rnorm(1000), rnorm(1000), rnorm(1000), color = heat.colors(1000))
rgl.bringtotop(stay = TRUE)
```

---

rgl.getAxisCallback *Get user-defined axis labelling callbacks.*

Description

This function gets a user-defined axis labelling callback in R.

Usage

```
rgl.getAxisCallback(axis, dev = cur3d(), subscene = currentSubscene3d(dev))
```

Arguments

axis	Which axis? Can be value from 1:3.
dev, subscene	The RGL device and subscene to work with.

Value

The callback function.

Author(s)

Duncan Murdoch

See Also

setAxisCallbacks to work with rglwidget.

rgl.incrementID *Increment ID*

Description

This function is mainly for internal use. It simply increments the internal object ID number and returns the new value. Negative values have no effect.

Usage

```
rgl.incrementID(n = 1L)
```

Arguments

n	An integer increment to use.

Value

The resulting ID value.

Examples

```
# Get the current ID value
rgl.incrementID(0)

# Increment it
rgl.incrementID()
```

rgl.init *Initializing RGL*

Description

Initializing the RGL system.

Usage

```
rgl.init(initValue = 0, onlyNULL = FALSE,
         debug = getOption("rgl.debug", FALSE))
```

Arguments

initValue	value for internal use only
onlyNULL	only initialize the null (no display) device
debug	enable some debugging messages

Details

If useNULL is TRUE, RGL will use a "null" device. This device records objects as they are plotted, but displays nothing. It is intended for use with rglwidget and similar functions.

Currently debug only controls messages printed by the OpenGL library during initialization. In future debug = TRUE may become more verbose.

For display within an OpenGL window in R, RGL requires the OpenGL system to be installed and available. If there is a problem initializing it, you may see the message 'rgl.init' failed, running with 'rgl.useNULL = There are several causes and remedies:

- On any system, the OpenGL libraries need to be present for RGL to be able to start an OpenGL device.
  - On macOS, you need to install XQuartz. It is available from https://www.xquartz.org.
  - On Linux, you need to install Mesa 3D. One of these commands may work, depending on your system:

```
zypper source-install --build-deps-only Mesa # openSUSE/SLED/SLES
yum-builddep mesa      # yum Fedora, OpenSuse(?)
dnf builddep mesa      # dnf Fedora
apt-get build-dep mesa # Debian, Ubuntu and related
```

  - Windows should have OpenGL installed by default.
- On Unix-alike systems (macOS and Linux, for example), RGL normally uses the GLX system for creating displays. If the graphic is created on a remote machine, it may need to use "Indirect GLX" (IGLX). Due to security concerns, this is often disabled by default. See https://www.x.org/wiki/Development/Security/Advisory-2014-12-09/ for a discussion of the security issues, and https://unix.stackexchange.com/q/317954 for ways to re-enable IGLX.
- The https://www.virtualgl.org project is intended to be a way to avoid IGLX, by rendering remotely and sending bitmaps to the local machine. It's not a simple install...
- If you don't need to see RGL displays on screen, you can use the "NULL device". See rgl.useNULL.
- If you can't build the **rgl** package with OpenGL support, you can disable it and use the NULL device. (This may happen automatically during configuration, but you'll get a tested result if you specify it explicitly.) See the instructions in the 'README' file in the source tarball.

Value

Normally the user doesn't call rgl.init at all: it is called when the package is loaded. It returns no useful value.

---

rgl.pixels *Extract pixel information from window*

Description

This function extracts single components of the pixel information from the topmost window.

Usage

```
rgl.pixels(component = c("red", "green", "blue"),
        viewport = par3d("viewport"), top = TRUE)
```

Arguments

component	Which component(s)?
viewport	Lower left corner and size of desired region.
top	Whether to bring window to top before reading.

Details

The possible components are "red", "green", "blue", "alpha", "depth", and "luminance" (the sum of the three colors). All are scaled from 0 to 1.

Note that the luminance is kept below 1 by truncating the sum; this is the definition used for the GL_LUMINANCE component in OpenGL.

Value

A vector, matrix or array containing the desired components. If one component is requested, a vector or matrix will be returned depending on the size of block requested (length 1 dimensions are dropped); if more, an array, whose last dimension is the list of components.

Author(s)

Duncan Murdoch

See Also

rgl.snapshot to write a copy to a file, demo("stereo") for functions that make use of this to draw a random dot stereogram and an anaglyph.

Examples

```
example(surface3d)
depth <- rgl.pixels(component = "depth")
if (length(depth) && is.matrix(depth)) # Protect against empty or single pixel windows
    contour(depth)
```

`rgl.postscript` *Export vector graphics*

Description

Saves the screenshot to a file in PostScript or other vector graphics format.

Usage

```
rgl.postscript( filename, fmt = "eps", drawText = TRUE )
```

Arguments

`filename`	full path to filename.
`fmt`	export format, currently supported: ps, eps, tex, pdf, svg, pgf
`drawText`	logical, whether to draw text

Details

Animations can be created in a loop modifying the scene and saving a screenshot to a file. (See example below)

This function is a wrapper for the GL2PS library by Christophe Geuzaine, and has the same limitations as that library: not all OpenGL features are supported, and some are only supported in some formats. See the reference for full details.

Author(s)

Christophe Geuzaine / Albrecht Gebhardt

References

GL2PS: an OpenGL to PostScript printing library by Christophe Geuzaine, https://www.geuz.org/gl2ps/, version 1.4.2.

See Also

`view3d`, `snapshot3d`

Examples

```
# Create new files in tempdir
savedir <- setwd(tempdir())

x <- y <- seq(-10, 10, length.out = 20)
z <- outer(x, y, function(x, y) x^2 + y^2)
persp3d(x, y, z, col = 'lightblue')

title3d("Using LaTeX text", col = 'red', line = 3)
```

```
rgl.postscript("persp3da.ps", "ps", drawText = FALSE)
rgl.postscript("persp3da.pdf", "pdf", drawText = FALSE)
rgl.postscript("persp3da.tex", "tex")
pop3d()
title3d("Using ps/pdf text", col = 'red', line = 3)
rgl.postscript("persp3db.ps", "ps")
rgl.postscript("persp3db.pdf", "pdf")
rgl.postscript("persp3db.tex", "tex", drawText = FALSE)

setwd(savedir)

## Not run:

#
# create a series of frames for an animation
#

open3d()
shade3d(oh3d(), color = "red")
view3d(0, 20)

for (i in 1:45) {
  view3d(i, 20)
  filename <- paste("pic", formatC(i, digits = 1, flag = "0"), ".eps", sep = "")
  rgl.postscript(filename, fmt = "eps")
}


## End(Not run)
```

`rgl.select` *Switch to select mode, and return the mouse position selected*

Description

Mostly for internal use, this function temporarily installs a handler on a button of the mouse that will return the mouse coordinates of one click and drag rectangle.

Usage

```
rgl.select(button = c("left", "middle", "right"),
           dev = cur3d(), subscene = currentSubscene3d(dev))
```

Arguments

`button`	Which button to use?
`dev, subscene`	The RGL device and subscene to work with

Value

A vector of four coordinates: the X and Y coordinates of the start and end of the dragged rectangle.

Author(s)

Duncan Murdoch

See Also

`select3d`, a version that allows the selection region to be used to select points in the scene.

`rgl.Sweave` *Integrating RGL with Sweave*

Description

As of R 2.13.0, it is possible to include RGL graphics into a Sweave document. These functions support that integration.

Usage

```
Sweave.snapshot()
rgl.Sweave(name, width, height, options, ...)
rgl.Sweave.off()
```

Arguments

`name`, `width`, `height`, `options`, `...`
: These arguments are passed by `Sweave` to `rgl.Sweave` when it opens the device.

Details

The `rgl.Sweave` function is not normally called by the user. The user specifies it as the graphics driver when opening the code chunk, e.g. by using

```
<<fig = TRUE, pdf = FALSE, grdevice = rgl.Sweave, resolution = 100>>=
```

When the RGL device is closed at the end of the code chunk, `rgl.Sweave.off()` will be called automatically. It will save a snapshot of the last image (by default in '`.png`' format) for inclusion in the Sweave document and (by default) close the device. Alternatively, the `Sweave.snapshot()` function can be called to save the image before the end of the chunk. Only one snapshot will be taken per chunk.

Several chunk options are used by the `rgl.Sweave` device:

stayopen (default `FALSE`). If `TRUE` then the RGL device will *not* be closed at the end of the chunk, instead a call to `Sweave.snapshot()` will be used if it has not been called explicitly. Subsequent chunks can add to the scene.

outputtype (default png). The output may be specified as `outputtype = pdf` or `outputtype = eps` instead, in which case the `rgl.postscript` function will be used to write output in the specified format. Note that `rgl.postscript` has limitations and does not always render scenes correctly.

delay (default 0.1). After creating the display window, `Sys.sleep` will be called to delay this many seconds, to allow the display system to initialize. This is needed in X11 systems which open the display asynchronously. If the default time is too short, `rgl.Sweave` may falsely report that the window is too large to open.

Value

These functions are called for their side effects.

Note

We recommend turning off all other graphics drivers in a chunk that uses `grdevice = rgl.Sweave`. The RGL functions do not write to a standard graphics device.

Note

The **rgl** package relies on your graphics hardware to render OpenGL scenes, and the default '`.png`' output copies a bitmap from the hardware device. All such devices have limitations on the size of the bitmap, but they do not always signal these limitations in a way that RGL will detect. If you find that images are not being produced properly, try reducing the size using the `resolution`, `width` or `height` chunk options.

Author(s)

Duncan Murdoch

See Also

`RweaveLatex` for a description of alternate graphics drivers in Sweave, and standard options that can be used in code chunks.

`hook_rgl` and `hook_webgl` allow fixed or interactive RGL scenes to be embedded in **knitr** documents.

`rgl.useNULL` *Report default use of null device*

Description

This function checks the `"rgl.useNULL"` option if present, or the `RGL_USE_NULL` environment variable if it is not. If the value is `TRUE` or a string which matches "yes" or "true" in a case-insensitive test, `TRUE` is returned.

Usage

```
rgl.useNULL()
```

Value

A logical value indicating the current default for use of the null device.

Note

This function is checked by the initialization code when the **rgl** package is loaded. Thus if you want to run RGL on a system where there is no graphics support, you should run `options(rgl.useNULL = TRUE)` or set the environment variable `RGL_USE_NULL=TRUE` *before* calling `library(rgl)` (or other code that loads **rgl**), and it will not fail in its attempt at initialization.

Author(s)

Duncan Murdoch

See Also

`open3d` and `rgl.open`.

Examples

```
rgl.useNULL()
```

---

rgl.user2window *Convert between RGL user and window coordinates*

Description

This function converts from 3-dimensional user coordinates to 3-dimensional window coordinates.

Usage

```
rgl.user2window(x, y = NULL, z = NULL, projection = rgl.projection())
rgl.window2user(x, y = NULL, z = 0, projection = rgl.projection())
rgl.projection(dev = cur3d(), subscene = currentSubscene3d(dev))
```

Arguments

x, y, z	Input coordinates. Any reasonable way of defining the coordinates is acceptable. See the function `xyz.coords` for details.
projection	The RGL projection to use
dev, subscene	The RGL device and subscene to work with

Details

These functions convert between user coordinates and window coordinates.

Window coordinates run from 0 to 1 in X, Y, and Z. X runs from 0 on the left to 1 on the right; Y runs from 0 at the bottom to 1 at the top; Z runs from 0 foremost to 1 in the background. RGL does not currently display vertices plotted outside of this range, but in normal circumstances will automatically resize the display to show them. In the example below this has been suppressed.

Value

The coordinate conversion functions produce a matrix with columns corresponding to the X, Y, and Z coordinates.

`rgl.projection()` returns a list containing the following components:

`model`	the modelview matrix
`projection`	the projection matrix
`viewport`	the viewport vector

See `par3d` for more details.

Author(s)

Ming Chen / Duncan Murdoch

See Also

`select3d`

Examples

```
open3d()
points3d(rnorm(100), rnorm(100), rnorm(100))
if (interactive() || !.Platform$OS == "unix") {
# Calculate a square in the middle of the display and plot it
square <- rgl.window2user(c(0.25, 0.25, 0.75, 0.75, 0.25),
                          c(0.25, 0.75, 0.75, 0.25, 0.25), 0.5)
par3d(ignoreExtent = TRUE)
lines3d(square)
par3d(ignoreExtent = FALSE)
}
```

`rglExtrafonts` *Register extra fonts*

Description

This function uses the **extrafont** package to help register system fonts for use with FreeType in **rgl**.

Usage

```
rglExtrafonts(..., quiet = TRUE)
```

Arguments

`...`	Vectors of fonts to try. See the Details.
`quiet`	Whether to print information on progress.

Details

The **extrafont** package collects information on installed fonts from the system. When you first install **extrafont**, or after new fonts have been installed on your system, run `extrafont::font_import()` to build its database of system fonts.

Fonts can be installed in **rgl** using `rglExtrafonts(rglname = familyname)` or `rglExtrafonts(familyname)`. In this call `familyname` is a vector of family names to look for in the **extrafont** database using `extrafont::choose_font(familyname)`; the first one found will be registered with **rgl**. The optional name `rglname` will also be usable to refer to the font family.

If none of the given family names is found, no change will be made to the registered fonts in **rgl**.

During startup, **rgl** detects whether **extrafont** is installed, and if so runs

```
rglExtrafonts(sans = c("Helvetica", "Arial"),
              serif = c("Times", "Times New Roman"),
              mono = c("Courier", "Courier New"))
```

to attempt to set up the default fonts.

Fonts found by **extrafont** can also be used in some other graphics devices besides **rgl**; see the **extrafont** documentation for details.

Value

Invisibly returns a vector giving the **rgl** name and the family name for the newly installed font.

Note

Each font in a display needs a unique **rgl** name; if the associated font for a given name is changed, all previously plotted text will also change.

Currently `rglwidget` displays will not respect the new definitions.

Author(s)

Duncan Murdoch

See Also

text3d, rglFonts

Examples

```
if (requireNamespace("extrafont") && !in_pkgdown_example()) {

  open3d()
  text3d(1,1,1, "Default", family = "sans", cex = 2)

  # Attempt to register new sans-serif font:
  newfamily <- rglExtrafonts(newsans = c("Comic Sans MS", "Impact",
                                         "Verdana", "Tahoma"))

  text3d(2,2,2, newfamily, family = "newsans", cex = 2)

}
```

---

rglFonts *Specify FreeType fonts*

Description

Specify FreeType fonts for use in **rgl** graphics.

Usage

```
rglFonts(...)
```

Arguments

...	Device dependent font definitions for use with FreeType.

Details

FreeType fonts are specified using the `rglFonts` function. This function takes a vector of four filenames of TrueType font files which will be used for the four styles regular, bold, italic and bold italic. The vector is passed with a name to be used as the family name, e.g. `rglFonts(sans = c("/path/to/FreeSans.ttf", ...))`. In order to limit the file size, the **rgl** package ships with just 3 font files, for regular versions of the `serif`, `sans` and `mono` families. Additional free font files were available in the past from the Amaya project, though currently the `rglExtrafonts` function provides an easier way to register new fonts.

On Windows the system fonts are acceptable and are used when `useFreeType = FALSE` (the current default in `r3dDefaults`). Mappings to `family` names are controlled by the `grDevices::windowsFonts()` function.

Full pathnames should normally be used to specify font files. If relative paths are used, they are interpreted differently by platform. Currently Windows fonts are looked for in the Windows fonts folder, while other platforms use the current working directory.

If FreeType fonts are not used, then bitmapped fonts will be used instead. On Windows these will be based on the fonts specified using the `windowsFonts` function, and are resizable. Other platforms will use the default bitmapped font which is not resizable.

Bitmapped fonts have a limited number of characters supported; if any unsupported characters are used, an error will be thrown.

Value

the current set of font definitions.

See Also

`text3d`

Examples

```
## Not run:
# These FreeType fonts are available from the Amaya project, and are not shipped
# with rgl.  You would normally install them to the rgl/fonts directory
# and use fully qualified pathnames, e.g.
# system.file("fonts/FreeSerif.ttf", package = "rgl")

rglFonts(serif = c("FreeSerif.ttf", "FreeSerifBold.ttf", "FreeSerifItalic.ttf",
                   "FreeSerifBoldItalic.ttf"),
         sans  = c("FreeSans.ttf", "FreeSansBold.ttf", "FreeSansOblique.ttf",
                   "FreeSansBoldOblique.ttf"),
         mono  = c("FreeMono.ttf", "FreeMonoBold.ttf", "FreeMonoOblique.ttf",
                   "FreeMonoBoldOblique.ttf"),
         symbol= c("ESSTIX10.TTF", "ESSTIX12.TTF", "ESSTIX9_.TTF",
                   "ESSTIX11.TTF"))

## End(Not run)
```

---

rglIds *RGL id values*

Description

All objects in an RGL scene have a numerical id. These ids are normally stored in vectors of class `c("rglIds", "numeric")`, which will also have class `"rglHighlevel"` or `"rglLowlevel"` depending on whether a high level function like `plot3d` or `persp3d`, or a low level function created the objects.

Usage

```
rglId(ids = integer())
lowlevel(ids = integer())
highlevel(ids = integer())
## S3 method for class 'rglId'
print(x,
      rglwidget = getOption("rgl.printRglwidget", FALSE),
      ...)
```

Arguments

ids	A vector of object ids.
x	An "rglId" object to print.
rglwidget	Whether to create and print an RGL widget. If false, nothing is printed.
...	Other arguments which will be passed to rglwidget if it is used.

Details

These functions and classes are intended to allow RGL scenes to be automatically displayed in R Markdown documents. See setupKnitr for details on enabling auto-printing.

Note that *all* objects in the current scene will be printed by default, not just the ids in x. (One reason for this is that lights are also objects; printing objects without lights would rarely make sense.)

Value

Objects of class "rglId", c("rglHighlevel", "rglId","numeric") or c("rglLowlevel", "rglId","numeric") for rglId, lowlevel or highlevel respectively.

Author(s)

Duncan Murdoch

Examples

```
x <- matrix(rnorm(30), ncol = 3, dimnames = list(NULL, c("x", "y", "z")))
p <- plot3d(x, type = "s")
str(p)
if (interactive() || in_pkgdown_example())
  print(p, rglwidget = TRUE)
```

rglMouse *Generate HTML code to select mouse mode*

Description

This generates an HTML `select` element to choose among the mouse modes supported by `rglwidget`.

Usage

```
rglMouse(sceneId,
         choices = c("trackball", "selecting",
                     "xAxis", "yAxis", "zAxis",
                     "polar", "zoom", "fov",
                     "none"),
         labels = choices,
         button = 1,
         dev = cur3d(),
         subscene = currentSubscene3d(dev),
        default = par3d("mouseMode", dev = dev, subscene = subscene)[button + 1],
         stayActive = FALSE,
         height = 40,
         ...)
```

Arguments

`sceneId`	Either an `rglwidget` or the `elementId` from one of them.
`choices`	Which mouse modes to support?
`labels`	How to label each mouse mode.
`button`	Which mouse button is being controlled.
`dev`	The RGL device used for defaults.
`subscene`	Which subscene is being modified.
`default`	What is the default entry to show in the control.
`stayActive`	Whether a selection brush should stay active if the mouse mode is changed.
`height`	The (relative) height of the item in the output display.
`...`	Additional arguments to pass to `htmltools::tags$select()`, e.g. `id` or `class`.

Details

A result of an `rglwidget` call can be passed as the `sceneId` argument. This allows the widget to be "piped" into the `rglMouse` call. The widget will appear first, the selector next in a `tagList`.

If the `sceneId` is a character string, it should be the `elementId` of a separately constructed `rglwidget` result.

Finally, the `sceneId` can be omitted. In this case the `rglMouse` result needs to be passed into an `rglwidget` call as part of the `controllers` argument. This will place the selector before the widget on the resulting display.

If the mouse mode is changed while brushing the scene, by default the brush will be removed (and so the selection will be cleared too). If this is not desired, set `stayActive = TRUE`.

Value

A browsable value to put in a web page.

Author(s)

Duncan Murdoch

See Also

The User Interaction in WebGL vignette gives more details.

Examples

```
if (interactive() || in_pkgdown_example()) {
  open3d()
  xyz <- matrix(rnorm(300), ncol = 3)
  id <- plot3d(xyz, col = "red", type = "s")["data"]
  par3d(mouseMode = "selecting")
  share <- rglShared(id)

# This puts the selector below the widget.
  rglwidget(shared = share, width = 300, height = 300) %>% rglMouse()

# This puts the selector above the widget.
  rglMouse() %>% rglwidget(shared = share, width = 300, height = 300, controllers = .)
}
```

---

rglShared *Create shared data from an RGL object*

Description

The **crosstalk** package provides a way for different parts of an interactive display to communicate about datasets, using "shared data" objects. When selection or filtering is performed in one view, the result is mirrored in all other views.

This function allows vertices of RGL objects to be treated as shared data.

Usage

```
rglShared(id, key = NULL, group = NULL,
          deselectedFade = 0.1,
          deselectedColor = NULL,
   selectedColor = NULL,
          selectedIgnoreNone = TRUE,
   filteredFade = 0,
          filteredColor = NULL)
```

Arguments

`id`	An existing RGL id.
`key`	Optional unique labels to apply to each vertex. If missing, numerical keys will be used.
`group`	Optional name of the shared group to which this data belongs. If missing, a random name will be generated.
`deselectedFade`, `deselectedColor`	Appearance of points that are not selected. See Details.
`selectedColor`	Appearance of points that are selected.
`selectedIgnoreNone`	If no points are selected, should the points be shown in their original colors (`TRUE`), or in the deselected colors (`FALSE`)?
`filteredFade`, `filteredColor`	Appearance of points that have been filtered out.

Details

Some functions which normally work on dataframe-like datasets will accept shared data objects in their place.

If a selection is in progress, the alpha value for unselected points is multiplied by `deselectedFade`. If `deselectedColor` is `NULL`, the color is left as originally specified; if not, the point is changed to the color given by `deselectedColor`.

If no points have been selected, then by default points are shown in their original colors. However, if `selectedIgnoreNone = FALSE`, all points are displayed as if unselected.

The `selectedColor` argument is similarly used to change the color (or not) of selected points, and `filteredFade` and `filteredColor` are used for points that have been filtered out of the display.

Value

An object of class `"SharedData"` (from the optional **crosstalk** package) which contains the `x`, `y` and `z` coordinates of the RGL object with the given `id`.

Author(s)

Duncan Murdoch

References

https://rstudio.github.io/crosstalk/index.html

See Also

The User Interaction in WebGL vignette gives more details.

Examples

```
save <- options(rgl.useNULL = TRUE)

#  rglShared requires the crosstalk package,
#  and the slider and rglMouse require manipulateWidget

if (requireNamespace("crosstalk", quietly = TRUE) &&
    requireNamespace("manipulateWidget", quietly = TRUE)) {
  open3d()
  x <- sort(rnorm(100))
  y <- rnorm(100)
  z <- rnorm(100) + atan2(x, y)
  ids <- plot3d(x, y, z, col = rainbow(100))

  # The data will be selected and filtered, not the axes.
  sharedData <- rglShared(ids["data"])

  # Also add some labels that are only displayed
  # when points are selected

  sharedLabel <- rglShared(text3d(x, y, z, text = 1:100,
                                  adj = -0.5),
                           group = sharedData$groupName(),
                           deselectedFade = 0,
                           selectedIgnoreNone = FALSE)
  if (interactive() || in_pkgdown_example())
    crosstalk::filter_slider("x", "x", sharedData, ~x) %>%
    rglwidget(shared = list(sharedData, sharedLabel), controller = .) %>%
    rglMouse()
}
options(save)
```

`rglToLattice` *Convert RGL userMatrix to lattice or base angles*

Description

These functions take a user orientation matrix from an RGL scene and approximate the parameters to either **lattice** or base graphics functions.

Usage

```
rglToLattice(rotm = par3d("userMatrix"))
rglToBase(rotm = par3d("userMatrix"))
```

Arguments

rotm	A matrix in homogeneous coordinates to convert.

Details

The **lattice** package can use Euler angles in the ZYX scheme to describe the rotation of a scene in its wireframe or cloud functions. The rglToLattice function computes these angles based on rotm, which defaults to the current user matrix. This allows RGL to be used to interactively find a decent viewpoint and then reproduce it in **lattice**.

The base graphics persp function does not use full Euler angles; it uses a viewpoint angle, and assume the z axis remains vertical. The rglToBase function computes the viewpoint angle accurately if the RGL scene is displayed with a vertical z axis, and does an approximation otherwise.

Value

rglToLattice returns a list suitable to be used as the screen argument to wireframe.

rglToBase returns a list containing theta and phi components which can be used as corresponding arguments in persp.

Author(s)

Duncan Murdoch

Examples

```
persp3d(volcano, col = "green")
if ((hasorientlib <- requireNamespace("orientlib", quietly = TRUE)) &&
    requireNamespace("lattice", quietly = TRUE))
    lattice::wireframe(volcano, screen = rglToLattice())
if (hasorientlib) {
  angles <- rglToBase()
  persp(volcano, col = "green", border = NA, shade = 0.5,
        theta = angles$theta, phi = angles$phi)
}
```

---

rglwidget *An htmlwidget to hold an RGL scene*

Description

The **htmlwidgets** package provides a framework for embedding graphical displays in HTML documents of various types. This function provides the necessities to embed an RGL scene in one.

Usage

```
rglwidget(x = scene3d(minimal), width = figWidth(), height = figHeight(),
          controllers = NULL,
          elementId = NULL,
          reuse = FALSE,
          webGLoptions = list(preserveDrawingBuffer = TRUE),
          shared = NULL, minimal = TRUE,
          webgl, snapshot,
          shinyBrush = NULL,
          altText = "3D plot",
          ...,
          oldConvertBBox = FALSE,
          fastTransparency = getOption("rgl.fastTransparency", TRUE))
```

Arguments

x	An RGL scene produced by the scene3d function.
width, height	The width and height of the display in pixels.
controllers	Names of playwidget objects associated with this scene, or objects (typically piped in). See Details below.
snapshot, webgl	Control of mode of display of scene. See Details below.
elementId	The id to use on the HTML div component that will hold the scene.
reuse	Ignored. See Details below.
webGLoptions	A list of options to pass to WebGL when the drawing context is created. See the Details below.
shared	An object produced by rglShared, or a list of such objects.
minimal	Should attributes be skipped if they currently have no effect? See scene3d.
shinyBrush	The name of a Shiny input element to receive information about mouse selections.
altText	Text to include for screen-readers or browsers that don't handle WebGL. See Details below.
oldConvertBBox, fastTransparency	See Details below.
...	Additional arguments to pass to htmlwidgets::createWidget.

Details

This produces a WebGL version of an RGL scene using the **htmlwidgets** framework. This allows display of the scene in the RStudio IDE, a browser, an **rmarkdown** document or in a **shiny** app.

`options(rgl.printRglwidget = TRUE)` will cause `rglwidget()` to be called and displayed when the result of an RGL call that changes the scene is printed.

In RMarkdown or in standalone code, you can use a **magrittr**-style "pipe" command to join an `rglwidget` with a `playwidget` or `toggleWidget`. If the control widget comes first, it should be piped into the `controllers` argument. If the `rglwidget` comes first, it can be piped into the first argument of `playwidget` or `toggleWidget`.

In earlier versions, the reuse argument let one output scene share data from earlier ones. This is no longer supported.

If elementId is NULL and we are not in a Shiny app, elementId is set to a random value to facilitate re-use of information.

To save the display to a file, use htmlwidgets::saveWidget. This requires pandoc to be installed. For a snapshot, you can use htmltools::save_html(img(src=rglwidget(snapshot=TRUE)), file = ...).

The webGLoptions argument is a list which will be passed when the WebGL context is created. See the WebGL 1.0 specification on https://registry.khronos.org/webgl/ for possible settings. The default in rglwidget differs from the WebGL default by setting preserveDrawingBuffer = TRUE in order to allow other tools to read the image, but please note that some implementations of WebGL contain bugs with this setting. We have attempted to work around them, but may change our default in the future if this proves unsatisfactory.

The webgl argument controls whether a dynamic plot is displayed in HTML. In LaTeX and some other formats dynamic plots can't be displayed, so if the snapshot argument is TRUE, webgl must be FALSE. (In previous versions of the **rgl** package, both webgl and snapshot could be TRUE; that hasn't worked for a while and is no longer allowed as of version 0.105.6.)

The snapshot argument controls whether a snapshot is displayed: it must be !webgl if both are specified.

Prior to **rgl** 0.106.21, rglwidget converted bounding box decorations into separate objects: a box, text for the labels, segments for the ticks. By default it now generates these in Javascript, allowing axis labels to move as they do in the display in R. If you prefer the old conversion, set oldConvertBBox = TRUE.

In version 1.3.4, the handling of transparent objects was changed to match the **rgl** device more closely. The new method of rendering is quite a bit faster, though sometimes less accurate. To get the older drawing method set fastTransparency = FALSE.

Value

An object of class "htmlwidget" (or "shiny.tag.list" if pipes are used) that will intelligently print itself into HTML in a variety of contexts including the R console, within R Markdown documents, and within Shiny output bindings.

If objects are passed in the shared argument, then the widget will respond to selection and filtering applied to those as shared datasets. See rglShared for more details and an example.

R Markdown specifics

In an R Markdown document, you would normally call setupKnitr(autoprint = TRUE) and would not make explicit calls to rglwidget(). If you do make such calls, the graphics will be inserted into the document.

In **knitr** versions greater than 1.42.5, the altText argument will be ignored and the alternate text will be set from chunk option fig.alt or fig.cap as with other graphics.

Shiny specifics

This widget is designed to work with Shiny for interactive displays linked to a server running R.

In a Shiny app, there will often be one or more playwidget objects in the app, taking input from the user. In order to be sure that the initial value of the user control is reflected in the scene, you should list all players in the controllers argument. See the sample application in system.file("shinyDemo", package = "rglwidget") for an example.

In Shiny, it is possible to find out information about mouse selections by specifying the name of an input item in the shinyBrush argument. For example, with shinyBrush = "brush3d", each change to the mouse selection will send data to input$brush3d in an object of class "rglMouseSelection" with the following components:

subscene The ID of the subscene where the mouse is selecting.

state Either "changing" or "inactive".

region The coordinates of the corners of the selected region in the window, in order c(x1, y1, x2, y2).

model, proj, view The model matrix, projection matrix and viewport in effect at that location.

This object can be used as the first argument to selectionFunction3d to produce a test function for whether a particular location is in the selected region. If the brush becomes inactive, an object containing only the state field will be sent, with value "inactive".

Appearance

The appearance of the display is set by the stylesheet in system.file("htmlwidgets/lib/rglClass/rgl.css").

The widget is of class rglWebGL, with id set according to elementId. (As of this writing, no special settings are given for class rglWebGL, but you can add your own.)

Author(s)

Duncan Murdoch

See Also

hook_webgl for an earlier approach to this problem. rglwidgetOutput for Shiny details. The User Interaction in WebGL vignette gives more details.

Examples

```
save <- options(rgl.useNULL=TRUE)
example("plot3d", "rgl")
widget <- rglwidget()
if (interactive() || in_pkgdown_example())
  widget


if (interactive() && !in_pkgdown_example()) {
  # Save it to a file.  This requires pandoc
  filename <- tempfile(fileext = ".html")
  htmlwidgets::saveWidget(rglwidget(), filename)
  browseURL(filename)
}
```

```
options(save)
```

`safe.dev.off` *Close graphics device in a safe way.*

Description

The `dev.off` function in **grDevices** doesn't restore the previous graphics device when called. This function does.

Usage

```
safe.dev.off(which = dev.cur(), prev = dev.prev())
```

Arguments

`which`	Which device to close.
`prev`	Which device to set as current after closing.

Details

This function closes device `which` if it is not device 1, then calls `dev.set(prev)` if there are any devices still open.

Value

The number and name of the new active device. It will not necessarily be `prev` if that device isn't already open.

Author(s)

Duncan Murdoch

References

https://bugs.r-project.org/show_bug.cgi?id=18604

Examples

```
# Open a graphics device
dev.new()
first <- dev.cur()

# Open a second graphics device
dev.new()
second <- dev.cur()
second
```

```
# Open another one, and close it using dev.off()
dev.new()
dev.off()
dev.cur() == second # Not the same as second!

# Try again with safe.dev.off()
dev.set(second)
dev.new()
safe.dev.off()
dev.cur() == second

# Close the other two devs
safe.dev.off()
safe.dev.off()
```

---

scene *Scene management*

Description

Clear shapes, lights, bbox

Usage

```
clear3d( type = c("shapes", "bboxdeco", "material"), defaults, subscene = 0 )
pop3d( type = "shapes", id = 0, tag = NULL)
ids3d( type = "shapes", subscene = NA, tags = FALSE )
```

Arguments

type	Select subtype(s): **"shapes"** shape stack **"lights"** light stack **"bboxdeco"** bounding box **"userviewpoint"** user viewpoint **"modelviewpoint"** model viewpoint **"material"** material properties **"background"** scene background **"subscene"** subscene list **"all"** all of the above
defaults	default values to use after clearing
subscene	which subscene to work with. NA means the current one, 0 means the whole scene
id	vector of ID numbers of items to remove
tag	override id with objects matching these tag material properties
tags	logical; whether to return tag column.

Details

RGL holds several lists of objects in each scene. There are lists for shapes, lights, bounding box decorations, subscenes, etc. `clear3d` clears the specified stack, or restores the defaults for the bounding box (not visible) or viewpoint. With `id = 0` `pop3d` removes the last added node on the list (except for subscenes: there it removes the active subscene). The `id` argument may be used to specify arbitrary item(s) to remove; if `id != 0`, the `type` argument is ignored.

`clear3d` may also be used to clear material properties back to their defaults.

`clear3d` has an optional `defaults` argument, which defaults to `r3dDefaults`. Only the `materials` component of this argument is currently used by `clear3d`.

`ids3d` returns a dataframe containing the IDs in the currently active subscene by default, or a specified subscene, or if `subscene = 0`, in the whole rgl window along with an indicator of their type and if `tags = TRUE`, the `tag` value for each.

Note that clearing the light stack leaves the scene in darkness; it should normally be followed by a call to `light3d`.

See Also

`rgl`, `bbox3d`, `light3d`, `open3d` to open a new window.

Examples

```
x <- rnorm(100)
y <- rnorm(100)
z <- rnorm(100)
p <- plot3d(x, y, z, type = 's', tag = "plot")
ids3d()
lines3d(x, y, z)
ids3d(tags = TRUE)
if (interactive() && !rgl.useNULL() && !in_pkgdown_example()) {
  readline("Hit enter to change spheres")
  pop3d(id = p["data"])
  spheres3d(x, y, z, col = "red", radius = 1/5)
  box3d()
}
```

---

scene3d *Saves the current scene to a variable, and displays such variables*

Description

This function saves a large part of the RGL state associated with the current window to a variable.

Usage

```
scene3d(minimal = TRUE)
## S3 method for class 'rglscene'
plot3d(x, add = FALSE, open3dParams = getr3dDefaults(), ...)
## S3 method for class 'rglobject'
plot3d(x, ...)
## S3 method for class 'rglscene'
print(x, ...)
## S3 method for class 'rglobject'
print(x, ...)
```

Arguments

minimal	Should attributes be skipped if they currently have no effect? See Details.
x	An object of class "rglscene"
add	Whether to open a new window, or add to the existing one.
open3dParams	Default parameters for open3d
...	Additional parameters passed to open3d by plot3d(..., add = FALSE). These override open3dParams.

Details

The components saved are: the par3d settings, the material3d settings, the bg3d settings, the lights and the objects in the scene.

In most cases, calling plot3d on that variable will duplicate the scene. (There are likely to be small differences, mostly internal, but some aspects of the scene are not currently available.) If textures are used, the name of the texture will be saved, rather than the contents of the texture file.

Other than saving the code to recreate a scene, saving the result of scene3d to a file will allow it to be reproduced later most accurately. In roughly decreasing order of fidelity, writeWebGL (now deprecated), writePLY, writeOBJ and writeSTL write the scene to a file in formats readable by other software.

If minimal = TRUE (the default), then attributes of objects will not be saved if they currently have no effect on the display, thereby reducing the file size. Set minimal = FALSE if the scene is intended to be used in a context where the appearance could be changed. Currently this only affects the inclusion of normals; with minimal = TRUE they are omitted for objects when the material is not lit.

Value

The scene3d function returns an object of class "rglscene". This is a list with some or all of the components:

material	The results returned from a material3d call.
rootSubscene	A list containing information about the main ("root") subscene. This may include: **id** The scene id. **type** "subscene"

par3d The par3d settings for the subscene.

embeddings The subsceneInfo()$embeddings for the main subscene.

objects The ids for objects in the subscene.

subscenes A recursive list of child subscenes.

`objects` A list containing the RGL lights, background and objects in the scene.

The objects in the `objects` component are of class `"rglobject"`. They are lists containing some or all of the components

`id` The RGL identifier of the object in the original scene.

`type` A character variable identifying the type of object.

`material` Components of the material that differ from the scene material.

`vertices`, `normals`, `etc.`
Any of the attributes of the object retrievable by rgl.attrib.

`ignoreExtent` A logical value indicating whether this object contributes to the bounding box. Currently this may differ from the object in the original scene.

`objects` Sprites may contain other objects; they will be stored here as a list of `"rglobject"`s.

Lights in the scene are stored similarly, mixed into the `objects` list.

The `plot3d` methods invisibly return a vector of RGL object ids that were plotted. The `print` methods invisibly return the object that was printed.

Author(s)

Duncan Murdoch

See Also

rglwidget, writePLY, writeOBJ and writeSTL write the scene to a file in various formats.

Examples

```
open3d()
z <- 2 * volcano        # Exaggerate the relief
x <- 10 * (1:nrow(z))   # 10 meter spacing (S to N)
y <- 10 * (1:ncol(z))   # 10 meter spacing (E to W)
persp3d(x, y, z, col = "green3", aspect = "iso")

s <- scene3d()
# Make it bigger
s$par3d$windowRect <- 1.5*s$par3d$windowRect
# and draw it again
plot3d(s)
```

sceneChange *Make large change to a scene from Shiny*

Description

These functions allow Shiny apps to make relatively large changes to a scene, adding and removing objects from it.

Usage

```
sceneChange(elementId, x = scene3d(minimal),
            delete = NULL, add = NULL, replace = NULL,
            material = FALSE, rootSubscene = FALSE,
            delfromSubscenes = NULL, skipRedraw = FALSE,
            minimal = TRUE)
registerSceneChange()
```

Arguments

elementId	The id of the element holding the rglClass instance.
x	The new scene to use as a source for objects to add.
delete, add, replace	Object ids to modify in the scene. The delete and replace ids must be present in the old scene in the browser; the add and replace ids must be present in x.
material	Logical to indicate whether default material should be updated.
rootSubscene	Logical to indicate whether root subscene should be updated.
delfromSubscenes	A vector of subscene ids that may have been changed by deletions. By default, all subscenes in x are used, but the objects may be included in subscenes in the browser that are different.
skipRedraw	If TRUE, stop the scene from redrawing until skipRedraw=FALSE is sent. If NA, don't redraw this time, but don't change the state of the skipRedraw flag.
minimal	See scene3d.

Details

registerSceneChange must be called in the UI component of a Shiny app to register the "sceneChange" custom message.

Value

registerSceneChange returns the HTML code to register the message.

sceneChange returns a list to be used as the "sceneChange" message to change the scene. Use shiny::session$sendCustomMessage to send it.

Author(s)

Duncan Murdoch

See Also

playwidget for a different approach to modifying scenes that can be much faster, but may be less flexible. The Shiny demo in this package makes use of all of these approaches.

Examples

```
## Not run:
shinyUI(fluidPage(
  registerSceneChange(),
  actionButton("thebutton", "Change")
))

shinyServer(function(input, output, session) {
  observeEvent(input$thebutton, {
    session$sendCustomMessage("sceneChange",
      sceneChange("thewidget", delete = deletes, add = adds))
  })
})

## End(Not run)
```

---

select3d *Select a rectangle in an RGL scene*

Description

This function allows the user to use the mouse to select a region in an RGL scene.

Usage

```
select3d(button = c("left", "middle", "right"),
             dev = cur3d(), subscene = currentSubscene3d(dev))
selectionFunction3d(proj, region = proj$region)
```

Arguments

button	Which button to use for selection.
dev, subscene	The RGL device and subscene to work with
proj	An object returned from rgl.projection containing details of the current projection.
region	Corners of a rectangular region in the display.

Details

`select3d` selects 3-dimensional regions by allowing the user to use a mouse to draw a rectangle showing the projection of the region onto the screen. It returns a function which tests points for inclusion in the selected region.

`selectionFunction3d` constructs such a test function given coordinates and current transformation matrices.

If the scene is later moved or rotated, the selected region will remain the same, though no longer corresponding to a rectangle on the screen.

Value

These return a function `f(x, y, z)` which tests whether each of the points `(x, y, z)` is in the selected region, returning a logical vector. This function accepts input in a wide variety of formats as it uses `xyz.coords` to interpret its parameters.

Author(s)

Ming Chen / Duncan Murdoch

See Also

`selectpoints3d`, `locator`

Examples

```
# Allow the user to select some points, and then redraw them
# in a different color

if (interactive() && !in_pkgdown_example()) {
 x <- rnorm(1000)
 y <- rnorm(1000)
 z <- rnorm(1000)
 open3d()
 points3d(x, y, z)
 f <- select3d()
 if (!is.null(f)) {
   keep <- f(x, y, z)
   pop3d()
   points3d(x[keep], y[keep], z[keep], color = 'red')
   points3d(x[!keep], y[!keep], z[!keep])
 }
}
```

`selectpoints3d` *Select points from a scene*

Description

This function uses the `select3d` function to allow the user to choose a point or region in the scene, then reports on all the vertices in or near that selection.

Usage

```
selectpoints3d(objects = ids3d()$id, value = TRUE, closest = TRUE,
               multiple = FALSE, ...)
```

Arguments

`objects`	A vector of object id values to use for the search.
`value`	If `TRUE`, return the coordinates of the points; otherwise, return their indices.
`closest`	If `TRUE`, return the points closest to the selection of no points are exactly within it.
`multiple`	If `TRUE` or a function, do multiple selections. See the Details below.
`...`	Other parameters to pass to `select3d`.

Details

The `multiple` argument may be a logical value or a function. If logical, it controls whether multiple selections will be performed. If `multiple` is `FALSE`, a single selection will be performed; it might contain multiple points. If `TRUE`, multiple selections will occur and the results will be combined into a single matrix.

If `multiple` is a function, it should take a single argument. This function will be called with the argument set to a matrix containing newly added rows to the value, i.e. it will contain coordinates of the newly selected points (if `value = TRUE`), or the indices of the points (if `value = FALSE`). It should return a logical value, `TRUE` to indicate that selection should continue, `FALSE` to indicate that it should stop.

In either case, if multiple selections are being performed, the `ESC` key will stop the process.

Value

If `value` is `TRUE`, a 3-column matrix giving the coordinates of the selected points. All rows in the matrix will be unique even if multiple vertices have the same coordinates.

If `value` is `FALSE`, a 2-column matrix containing columns:

`id`	The object id containing the point.
`index`	The index of the point within `rgl.attrib(id, "vertices")`. If multiple points have the same coordinates, all indices will be returned.

Note

This function selects points, not areas. For example, if the selection region is in the interior of the triangle, that will count as a miss for all of the triangle's vertices.

Author(s)

Duncan Murdoch

See Also

`select3d` to return a selection function.

Examples

```
xyz <- cbind(rnorm(20), rnorm(20), rnorm(20))
ids <- plot3d( xyz )

if (interactive() && !in_pkgdown_example()) {
  # Click near a point to select it and put a sphere there.
  # Press ESC to quit...

  # This version returns coordinates
  selectpoints3d(ids["data"],
     multiple = function(x) {
        spheres3d(x, color = "red", alpha = 0.3, radius = 0.2)
        TRUE
     })

  # This one returns indices
  selectpoints3d(ids["data"], value = FALSE,
     multiple = function(ids) {
        spheres3d(xyz[ids[, "index"], , drop = FALSE], color = "blue",
                  alpha = 0.3, radius = 0.2)
        TRUE
     })
}
```

`setAxisCallbacks` *User-defined axis labelling callbacks.*

Description

This function sets user callbacks to construct axes in R or `rglwidget` displays.

Usage

```
setAxisCallbacks(axes, fns,
                 javascript = NULL,
                 subscene = scene$rootSubscene$id,
                 scene = scene3d(minimal = FALSE),
                 applyToScene = TRUE,
                 applyToDev = missing(scene))
```

Arguments

axes	Which axes? Specify as number in 1:3 or letter in c("x", "y", "z").
fns	Function or list of functions or character vector giving names of functions.
javascript	Optional block of Javascript code to be included (at the global level).
subscene	Which subscene do these callbacks apply to?
scene	Which scene?
applyToScene	Should these changes apply to the scene object?
applyToDev	Should these changes apply to the current device?

Details

If applyToScene is TRUE, this function adds Javascript callbacks to the scene object. If applyToDev is TRUE, it adds R callbacks to the current RGL device.

For Javascript, the callbacks are specified as strings; these will be evaluated within the browser in the global context to define the functions, which will then be called with the Javascript this object set to the current rglwidgetClass object.

For R, they may be strings or R functions.

Both options may be TRUE, in which case the callbacks must be specified as strings which are both valid Javascript and valid R. The usual way to do this is to give just a function name, with the function defined elsewhere, as in the Example below.

The functions should have a header of the form function(margin). The margin argument will be a string like "x++" indicating which margin would be chosen by R. If RGL would not choose to draw any axis annotations (which happens with rglwidget, though not currently in R itself), only the letter will be passed, e.g. "x".

Value

Invisibly returns an rglScene object. This object will record the changes if applyToScene is TRUE.

If applyToDev is TRUE, it will also have the side effect of attempting to install the callbacks.

Author(s)

Duncan Murdoch

See Also

setUserCallbacks for mouse callbacks.

Examples

```
# Draw arrows instead of tick marks on axes

arrowAxis <- local({
  ids <- c(NA, NA, NA)
  bbox <- c(NA, NA, NA, NA, NA, NA)
  function(margin) {
    dim <- if (grepl("x", margin)) 1 else
           if (grepl("y", margin)) 2 else
           3
    inds <- 2*dim + (-1):0
    range <- par3d("bbox")[inds]
    if (!identical(bbox[inds], range)) {
      if (!is.na(ids[dim]))
        pop3d(id = ids[dim])

      bbox[inds] <<- range
      center <- mean(range)
      from <- mean(c(range[1], center))
      to <- mean(c(center, range[2]))
      # margin should agree with suggestion, so use "x++" etc.
      margin <- gsub("-", "+", margin)
      ids[dim] <<- arrow3d(p0 = c(from, 1, 1),
                           p1 = c(to, 1, 1),
                           n = 4,
                           type = "lines",
                           margin = margin,
                           floating = TRUE)
    }
  }
})

# Define the Javascript function with the same name to use in WebGL
# Since Javascript won't change the bounding box, this function
# doesn't need to do anything.

js <- "
window.arrowAxis = function(margin) {} ;
"

xyz <- matrix(rnorm(60), ncol = 3)
plot3d(xyz, xlab = "x", ylab = "y", zlab = "z")
setAxisCallbacks(1:3, "arrowAxis", javascript = js)
rglwidget()
```

`setGraphicsDelay` *Set a one-time slowdown on opening standard graphics*

Description

This function is mainly for internal use, to work around a bug in macOS Catalina: if base plotting happens too quickly after opening RGL and the first call to quartz, R crashes.

This inserts a delay after the first call to open the graphics device. The default is no delay, unless on Catalina with no graphics device currently open but the `quartz` device set as the default, when a 1 second delay will be added. Use environment variable "RGL_SLOW_DEV = value" to set a different default delay.

It works by changing the value of `options("device")`, so explicit calls to the device will not be affected.

It is called automatically when the **rgl** package is loaded.

Usage

```
setGraphicsDelay(delay = Sys.getenv("RGL_SLOW_DEV", 0),
                 unixos = "none")
```

Arguments

`delay`	Number of seconds for the delay.
`unixos`	The name of the Unix OS. If set to `"Darwin"`, and the version is 19.0.0 or greater, the default delay is changed to 1 second.

Value

Called for the side effect of adding the delay to the first opening of the graphics device.

`setupKnitr` *Displaying RGL scenes in* **knitr** *documents*

Description

These functions allow RGL graphics to be embedded in **knitr** documents.

The simplest method is to run `setupKnitr(autoprint = TRUE)` early in the document. That way RGL commands act a lot like base graphics commands: plots will be automatically inserted where appropriate, according to the `fig.keep` chunk option. By default (`fig.keep = "high"`), only high-level plots are kept, after low-level changes have been merged into them. See the **knitr** documentation https://yihui.org/knitr/options/#plots for more details. To suppress auto-printing, the RGL calls can be wrapped in `invisible()`. Similarly to **grid** graphics (used by **lattice** and **ggplot2**), automatic inclusion requires the object to be printed: only the last statement in a code block in braces is automatically printed. Unlike those packages, auto-printing is the only way to get this to work: calling `print` explicitly doesn't work.

Other functions allow embedding either as bitmaps (`hook_rgl` with format `"png"`), fixed vector graphics (`hook_rgl` with format `"eps"`, `"pdf"` or `"postscript"`), or interactive WebGL graphics (`hook_webgl`). `hook_rglchunk` is not normally invoked by the user; it is the hook that supports automatic creation and deletion of RGL scenes.

Usage

```
setupKnitr(autoprint = FALSE,
           rgl.newwindow = autoprint,
           rgl.closewindows = autoprint)
hook_rgl(before, options, envir)
hook_webgl(before, options, envir)
hook_rglchunk(before, options, envir)
```

Arguments

`autoprint`	If true, RGL commands automatically plot (with low level plots suppressed by the default value of the `fig.keep` chunk option.)
`rgl.newwindow`, `rgl.closewindows`	Default values for the **knitr** chunk options.
`before`, `options`, `envir`	Standard **knitr** hook function arguments.

Details

The `setupKnitr()` function needs to be called once at the start of the document to install the **knitr** hooks. If it is called twice in the same session the second call will override the first.

The following chunk options are supported:

- `rgl.newwindow`: Whether to open a new window for the chunk. Default is set by `setupKnitr` argument.
- `rgl.closewindows`: Whether to close windows at the end of the chunk. Default is set by `setupKnitr` argument.
- `rgl.margin` (default 100): number of pixels by which to indent the WebGL window.
- `snapshot`: Logical value: when autoprinting in HTML, should a snapshot be used instead of the dynamic WebGL display? Corresponds to `rglwidget(snapshot = TRUE, webgl = FALSE)`. Ignored in LaTeX, where a snapshot will always be produced (unless `fig.keep` specifies no figure at all).
- `dpi`, `fig.retina`, `fig.width`, `fig.height`: standard **knitr** chunk options used to set the size of the output.
- `fig.keep`, `fig.hold`, `fig.beforecode`: standard **knitr** chunk options used to control the display of plots.
- `dev`: used by `hook_rgl` to set the output format. May be `"eps"`, `"postscript"`, `"pdf"` or `"png"` (default: `"png"`).
- `rgl.keepopen`: no longer used. Ignored with a warning.
- `fig.alt` is partially supported: **rgl** will always use the first entry if `fig.alt` is a vector. Other graphics types match the entries in `fig.alt` to successive plots within the chunk. (This is due to a limitation in **knitr**, and may change in the future.)

Value

A string to be embedded into the output, or `NULL` if called when no output is available.

Note

The `setupKnitr(autoprint = TRUE)` method assumes *all* printing of RGL objects happens through auto-printing of objects produced by the `lowlevel` or `highlevel` functions. All RGL functions that produce graphics do this, but functions in other packages that call them may not return values appropriately.

Mixing explicit calls to `rglwidget` with auto-printing is likely to lead to failure of some scenes to display. To avoid this, set `options(rgl.printRglwidget = FALSE)` before using such explicit calls. Similarly, use that option before calling the `example` function in a code chunk if the example prints RGL objects.

Author(s)

The `hook*` functions are originally by Yihui Xie in the **knitr** package; and have been modified by Duncan Murdoch. Some parts of the `setupKnitr` function duplicate source code from **knitr**.

---

setUserCallbacks *Set mouse callbacks in R or Javascript code*

Description

This function sets user mouse callbacks in R or `rglwidget` displays.

Usage

```
setUserCallbacks(button,
                 begin = NULL,
                 update = NULL,
                 end = NULL,
                 rotate = NULL,
                 javascript = NULL,
                 subscene = scene$rootSubscene$id,
                 scene = scene3d(minimal = FALSE),
                 applyToScene = TRUE,
               applyToDev = missing(scene))
```

Arguments

`button`	Which button should this callback apply to? Can be numeric from `0:4`, or character from `"none"`, `"left"`, `"right"`, `"center"`, `"wheel"`.
`begin`, `update`, `end`, `rotate`	Functions to call when events occur. See Details.
`javascript`	Optional block of Javascript code to be included (at the global level).
`subscene`	Which subscene do these callbacks apply to?
`scene`	Which scene?
`applyToScene`	Should these changes apply to the scene object?
`applyToDev`	Should these changes apply to the current device?

Details

If applyToScene is TRUE, this function adds Javascript callbacks to the scene object. If applyToDev is TRUE, it adds R callbacks to the current RGL device.

For Javascript, the callbacks are specified as strings; these will be evaluated within the browser in the global context to define the functions, which will then be called with the Javascript this object set to the current rglwidgetClass object.

For R, they may be strings or R functions.

Both options may be TRUE, in which case the callbacks must be specified as strings which are both valid Javascript and valid R. The usual way to do this is to give just a function name, with the function defined elsewhere, as in the Example below.

The begin and update functions should be of the form function(x, y) { ... }. The end function will be called with no arguments.

The rotate callback can only be set on the mouse wheel. It is called when the mouse wheel is rotated. It should be of the form function(away), where away will be 1 while rotating the wheel "away" from you, and 2 while rotating it towards you. If rotate is not set but other callbacks are set on the wheel "button", then each click of the mouse wheel will trigger all start, update, then end calls in sequence.

The javascript argument is an optional block of code which will be evaluated once during the initialization of the widget. It can define functions and assign them as members of the window object, and then the names of those functions can be given in the callback arguments; this allows the callbacks to share information.

Value

Invisibly returns an rglScene object. This object will record the changes if applyToScene is TRUE.

If applyToDev is TRUE, it will also have the side effect of attempting to install the callbacks using rgl.setMouseCallbacks and rgl.setWheelCallback.

Author(s)

Duncan Murdoch

See Also

setAxisCallbacks for user defined axes.

Examples

```
# This example identifies points in both the rgl window and
# in WebGL

verts <- cbind(rnorm(11), rnorm(11), rnorm(11))
idverts <- plot3d(verts, type = "s", col = "blue")["data"]

# Plot some invisible text; the Javascript will move it
idtext <- text3d(verts[1,,drop = FALSE], texts = 1, adj = c(0.5, -1.5), alpha = 0)
```

```
# Define the R functions to use within R
fns <- local({
  idverts <- idverts
  idtext <- idtext
  closest <- -1
  update <- function(x, y) {
    save <- par3d(skipRedraw = TRUE)
    on.exit(par3d(save))
    rect <- par3d("windowRect")
    size <- rect[3:4] - rect[1:2]
    x <- x / size[1];
    y <- 1 - y / size[2];
    verts <- rgl.attrib(idverts, "vertices")
    # Put in window coordinates
    vw <- rgl.user2window(verts)
    dists <- sqrt((x - vw[,1])^2 + (y - vw[,2])^2)
    newclosest <- which.min(dists)
    if (newclosest != closest) {
      if (idtext > 0)
        pop3d(id = idtext)
      closest <<- newclosest
     idtext <<- text3d(verts[closest,,drop = FALSE], texts = closest, adj = c(0.5, -1.5))
     }
  }
  end <- function() {
    if (idtext > 0)
      pop3d(id = idtext)
    closest <<- -1
    idtext <<- -1
  }
  list(rglupdate = update, rglend = end)
})
rglupdate <- fns$rglupdate
rglend <- fns$rglend

# Define the Javascript functions with the same names to use in WebGL
js <-
 ' var idverts = %id%, idtext = %idtext%, closest = -1,
       subid = %subid%;

   window.rglupdate = function(x, y) {
     var   obj = this.getObj(idverts), i, newdist, dist = Infinity, pt, newclosest;
     x = x/this.canvas.width;
     y = y/this.canvas.height;

     for (i = 0; i < obj.vertices.length; i++) {
       pt = obj.vertices[i].concat(1);
       pt = this.user2window(pt, subid);
       pt[0] = x - pt[0];
       pt[1] = y - pt[1];
       pt[2] = 0;
       newdist = rglwidgetClass.vlen(pt);
       if (newdist < dist) {
```

```
          dist = newdist;
          newclosest = i;
        }
      }

      if (newclosest !== closest) {
        closest = newclosest
        var text = this.getObj(idtext);
        text.vertices[0] = obj.vertices[closest];
        text.colors[0][3] = 1; // alpha is here!
        text.texts[0] = (closest + 1).toString();
        text.initialized = false;
        this.drawScene();
      }
    };
    window.rglend = function() {
      var text = this.getObj(idtext);
      closest = -1;
      text.colors[0][3] = 0;
      text.initialized = false;
      this.drawScene();
    }'
js <- sub("%id%", idverts, js)
js <- sub("%subid%", subsceneInfo()$id, js)
js <- sub("%idtext%", idtext, js)

# Install both
setUserCallbacks("left",
                 begin = "rglupdate",
                 update = "rglupdate",
                 end = "rglend",
                 javascript = js)
rglwidget()

# This example doesn't affect the rgl window, it only modifies
# the scene object to implement panning

# Define the Javascript functions to use in WebGL
js <-
'  window.subid = %subid%;

   window.panbegin = function(x, y) {
     var activeSub = this.getObj(subid),
         viewport = activeSub.par3d.viewport,
         activeModel = this.getObj(this.useid(activeSub.id, "model")),
         l = activeModel.par3d.listeners, i;

      this.userSave = {x:x, y:y, viewport:viewport,
                       cursor:this.canvas.style.cursor};
      for (i = 0; i < l.length; i++) {
        activeSub = this.getObj(l[i]);
        activeSub.userSaveMat = new CanvasMatrix4(activeSub.par3d.userMatrix);
      }
```

```
        this.canvas.style.cursor = "grabbing";
      };

      window.panupdate = function(x, y) {
        var objects = this.scene.objects,
            activeSub = this.getObj(subid),
            activeModel = this.getObj(this.useid(activeSub.id, "model")),
            l = activeModel.par3d.listeners,
            viewport = this.userSave.viewport,
            par3d, i, zoom;
        if (x === this.userSave.x && y === this.userSave.y)
          return;
        x = (x - this.userSave.x)/this.canvas.width;
        y = (y - this.userSave.y)/this.canvas.height;
        for (i = 0; i < l.length; i++) {
          activeSub = this.getObj(l[i]);
          par3d = activeSub.par3d;
          /* NB:  The right amount of zoom depends on the scaling of the data
                  and the position of the observer.  This might
                  need tweaking.
          */
          zoom = rglwidgetClass.vlen(par3d.observer)*par3d.zoom;
          activeSub.par3d.userMatrix.load(objects[l[i]].userSaveMat);
          activeSub.par3d.userMatrix.translate(zoom*x, zoom*y, 0);
        }
        this.drawScene();
      };

      window.panend = function() {
        this.canvas.style.cursor = this.userSave.cursor;
      };
'

js <- sub("%subid%", subsceneInfo()$id, js)

scene <- setUserCallbacks("left",
                  begin = "panbegin",
                  update = "panupdate",
                  end = "panend",
                  applyToDev = FALSE, javascript = js)
rglwidget(scene)
```

---

setUserShaders *Set user-defined shaders for RGL objects, or get shaders.*

Description

setUserShaders sets user-defined shaders (programs written in GLSL) for customized display of RGL objects. Currently only supported in WebGL displays, as the regular displays do not support GLSL. getShaders gets the user defined shader, or if it is not present, the automatically generated one.

Usage

```
setUserShaders(ids, vertexShader = NULL, fragmentShader = NULL,
               attributes = NULL, uniforms = NULL, textures = NULL,
               scene = scene3d(minimal), minimal = TRUE)

getShaders(id, scene = scene3d(minimal), minimal = TRUE)
```

Arguments

ids, id	Which objects should receive the shaders, or which object should be queried?
vertexShader, fragmentShader	The vertex and fragment shader source code. If NULL, the automatically generated shader will be used instead.
attributes	A named list of "attributes" to attach to each vertex.
uniforms	A named list of "uniforms".
textures	A named list of textures.
scene	A scene3d object to work with.
minimal	See scene3d.

Details

Modern versions of OpenGL work with "shaders", programs written to run on the graphics processor. The vertex shader does the calculations to move vertices and set their intrinsic colours. The fragment shader computes how each pixel in the display will be shown, taking into account lighting, material properties, etc. (More precisely, it does the computation for each "fragment"; a fragment is a pixel within an object to display. There may be many objects at a particular location, and each will result in a fragment calculation unless culled by z-buffering or being discarded in some other way.)

Normally the WebGL Javascript code uses the default shaders stored in `system.file("htmlwidgets/lib/rglClass/shade = "rgl")`. This function allows them to be written by hand, for testing new features, hand optimization, etc. The defines used by the default shaders will also be prepended to user shaders, which can use them for customization on an object-by-object basis.

The names used for the attributes, uniforms and textures should match names in the shaders for corresponding variables. (The texture names should be names of uniform sampler2D variables.)

Value

A modified version of the scene.

Note

The getShaders function requires the **V8** package to extract auto-generated shaders, since the defines are generated by Javascript code.

Author(s)

Duncan Murdoch

See Also

rglwidget for display of the scene in WebGL.

Examples

```
open3d()
id <- shade3d(octahedron3d(), col = "red")

# For each triangle, set weights on the 3 vertices.
# This will be replicated to the appropriate size in Javascript.
wts <- diag(3)

# This leaves out the centres of each face
vs <- "
  attribute vec3 aPos;
  attribute vec4 aCol;
  uniform mat4 mvMatrix;
  uniform mat4 prMatrix;
  varying vec4 vCol;
  varying vec4 vPosition;
  attribute vec3 aNorm;
  uniform mat4 normMatrix;
  varying vec3 vNormal;
  attribute vec3 wts;
  varying vec3 vwts;
  void main(void) {
    vPosition = mvMatrix * vec4(aPos, 1.);
    gl_Position = prMatrix * vPosition;
    vCol = aCol;
    vNormal = normalize((normMatrix * vec4(aNorm, 1.)).xyz);
    vwts = wts;
  }
"
fs <- "
  #ifdef GL_ES
  precision highp float;
  #endif
  varying vec4 vCol; // carries alpha
  varying vec4 vPosition;
  varying vec3 vNormal;
  uniform mat4 mvMatrix;
  uniform vec3 emission;
  uniform float shininess;
  uniform vec3 ambient[NLIGHTS];
  uniform vec3 specular[NLIGHTS]; // light*material
  uniform vec3 diffuse[NLIGHTS];
  uniform vec3 lightDir[NLIGHTS];
  uniform bool viewpoint[NLIGHTS];
```

```
  uniform bool finite[NLIGHTS];
  varying vec3 vwts;
  uniform vec2 wtrange;
  void main(void) {
    float minwt = min(vwts.x, min(vwts.y, vwts.z));
    if (minwt < wtrange.x || minwt > wtrange.y) discard;
    vec3 eye = normalize(-vPosition.xyz);
    vec3 lightdir;
    vec4 colDiff;
    vec3 halfVec;
    vec4 lighteffect = vec4(emission, 0.);
    vec3 col;
    float nDotL;
    vec3 n = normalize(vNormal);
    n = -faceforward(n, n, eye);
    colDiff = vec4(vCol.rgb * diffuse[0], vCol.a);
    lightdir = lightDir[0];
    if (!viewpoint[0])
      lightdir = (mvMatrix * vec4(lightdir, 1.)).xyz;
    if (!finite[0]) {
      halfVec = normalize(lightdir + eye);
    } else {
      lightdir = normalize(lightdir - vPosition.xyz);
      halfVec = normalize(lightdir + eye);
    }
    col = ambient[0];
    nDotL = dot(n, lightdir);
    col = col + max(nDotL, 0.) * colDiff.rgb;
    col = col + pow(max(dot(halfVec, n), 0.), shininess) * specular[0];
    lighteffect = lighteffect + vec4(col, colDiff.a);
    gl_FragColor = lighteffect;
  }
"
x <- setUserShaders(id, vs, fs, attributes = list(wts=wts),
                    uniforms = list(wtrange = c(-0.01, 0.15)))
if (interactive() || in_pkgdown_example())
  rglwidget(x)
```

---

shade3d *Draw 3D mesh objects*

Description

Draws 3D mesh objects in full, or just the edges, or just the vertices.

Usage

```
  dot3d(x, ...)   # draw dots at the vertices of an object
  ## S3 method for class 'mesh3d'
dot3d(x, ...,
```

```
                          front = "points", back = "points")
  wire3d(x, ...)  # draw a wireframe object
  ## S3 method for class 'mesh3d'
wire3d(x, ...,
                           front = "lines", back = "lines")
  shade3d(x, ...) # draw a shaded object
  ## S3 method for class 'mesh3d'
shade3d(x, override = TRUE,
                          meshColor = c("vertices", "edges", "faces", "legacy"),
                            texcoords = NULL, ...,
                            front = "filled", back = "filled")
```

Arguments

x	a mesh3d object.
...	additional rendering parameters, or for dots3d and wire3d, parameters to pass to shade3d
override	should the parameters specified here override those stored in the object?
meshColor	how should colours be interpreted? See details below
texcoords	texture coordinates at each vertex.
front, back	Material properties for rendering.

Details

The meshColor argument controls how material colours and textures are interpreted. This parameter was added in **rgl** version 0.100.1 (0.100.27 for dot3d). Possible values are:

"vertices" Colours and texture coordinates are applied by vertex, in the order they appear in the x$vb matrix.

"edges" Colours are applied to each edge: first to the segments in the x$is matrix, then the 3 edges of each triangle in the x$it matrix, then the 4 edges of each quad in the x$ib matrix. This mode is only supported if both front and back materials are "lines", and the mesh contains no points.

"faces" Colours are applied to each object in the mesh: first to the points, then the segments, triangles and finally quads. The entire whole face (or point or segment) receives one colour from the specified colours.

"legacy" Colours and textures are applied in the same way as in **rgl** versions earlier than 0.100.1.

Unique partial matches of these values will be recognized.

If colours are specified but meshColor is not and options(rgl.meshColorWarning = TRUE), a warning will be given that their interpretation may have changed. In versions 0.100.1 to 0.100.26 of **rgl**, the default was to give the warning; now the default is for no warning.

Note that since version 0.102.10, meshColor = "edges" is only allowed when drawing lines (the wire3d default), and it may draw edges more than once. In general, if any rendering draws twice at the same location, which copy is visible depends on the order of drawing and the material3d("depth_test") setting.

Whether points, lines or solid faces are drawn is determined in 3 steps:

1. If arguments "front" or "back" are specified in the call, those are used.
2. If one or both of those arguments are not specified, but the material properties are present in the object, those are used.
3. If values are not specified in either of those places, shade3d draws filled surfaces, wire3d draws lines, and dot3d draws points.

Note: For some versions of rgl up to version 0.107.15, rule 2 above was not respected.

Value

dot3d, wire3d, and shade3d are called for their side effect of drawing an object into the scene; they return an object ID (or vector of IDs) invisibly.

See primitives for a discussion of texture coordinates.

See Also

mesh3d, par3d, shapelist3d for multiple shapes

Examples

```
# generate a quad mesh object

vertices <- c(
   -1.0, -1.0, 0,
    1.0, -1.0, 0,
    1.0,  1.0, 0,
   -1.0,  1.0, 0
)
indices <- c( 1, 2, 3, 4 )

open3d()
wire3d( mesh3d(vertices = vertices, quads = indices) )

# render 4 meshes vertically in the current view

open3d()
bg3d("gray")
l0 <- oh3d(tran = par3d("userMatrix"), color = "green" )
shade3d( translate3d( l0, -6, 0, 0 ))
l1 <- subdivision3d( l0 )
shade3d( translate3d( l1 , -2, 0, 0 ), color = "red", override = FALSE )
l2 <- subdivision3d( l1 )
shade3d( translate3d( l2 , 2, 0, 0 ), color = "red", override = TRUE )
l3 <- subdivision3d( l2 )
shade3d( translate3d( l3 , 6, 0, 0 ), color = "red" )

# render all of the Platonic solids
open3d()
shade3d( translate3d( tetrahedron3d(col = "red"), 0, 0, 0) )
shade3d( translate3d( cube3d(col = "green"), 3, 0, 0) )
shade3d( translate3d( octahedron3d(col = "blue"), 6, 0, 0) )
```

```
shade3d( translate3d( dodecahedron3d(col = "cyan"), 9, 0, 0) )
shade3d( translate3d( icosahedron3d(col = "magenta"), 12, 0, 0) )
```

---

shadow3d *Project shadows of mesh onto object.*

Description

Project a mesh onto a surface in a scene so that it appears to cast a shadow onto the surface.

Usage

```
shadow3d(obj, mesh, plot = TRUE, up = c(0, 0, 1),
         P = projectDown(up), outside = FALSE,
         ...)
```

Arguments

obj	The target object which will show the shadow.
mesh	The mesh which will cast the shadow.
plot	Whether to plot the result.
up	Which direction is "up"?
P	The projection to use for draping, a 4x4 matrix. See drape3d for details on how P is used.
outside	Should the function compute and (possibly) plot the region outside of the shadow?
...	Other arguments to pass to filledContour3d, which will do the boundary calculations and plotting.

Details

shadow3d internally constructs a function that is zero on the boundary of the shadow and positive inside, then draws filled contours of that function. Because the function is nonlinear, the boundaries will be approximate, with the best approximation resulting from a large value of filledContour3d parameter minVertices.

If outside = TRUE, the first color used by filledContour3d will indicate the inside of the shadow, and the second color will indicate the exterior.

Value

The returned value from filledContour3d.

Author(s)

Duncan Murdoch

See Also

drape3d, facing3d

Examples

```
open3d()
obj <- translate3d(scale3d(oh3d(), 0.3, 0.3, 0.3), 0,0,2)
shade3d(obj, col = "red")
target <- icosahedron3d()

# We offset the target using polygon_offset = 1 so that the
# shadow on its surface will appear clearly.

shade3d(target, col = "white", polygon_offset = 1)

# minVertices = 1000 leaves noticeable artifacts on the edges
# of the shadow.  A larger value gives a better result, but is
# slower.

# We use facing3d(target) so the shadow and outside part only
# appear on the upper side of the target

shadow3d(facing3d(target), obj, minVertices = 1000, plot=TRUE,
         col = c("yellow", "blue"), outside = TRUE)
```

---

shapelist3d *Create and plot a list of shapes*

Description

These functions create and plot a list of shapes.

Usage

```
shapelist3d(shapes, x = 0, y = NULL, z = NULL, size = 1, matrix = NULL, override = TRUE,
            ..., plot = TRUE)
```

Arguments

shapes	A single shape3d object, or a list of them.
x, y, z	Translation(s) to apply
size	Scaling(s) to apply
matrix	A single matrix transformation, or a list of them.
override	Whether the material properties should override the ones in the shapes.
...	Material properties to apply.
plot	Whether to plot the result.

Details

shapelist3d is a quick way to create a complex object made up of simpler ones. Each of the arguments shapes through override may be a vector of values (a list in the case of shapes or matrix). All values will be recycled to produce a list of shapes as long as the longest of them.

The xyz.coords function will be used to process the x, y and z arguments, so a matrix may be used as x to specify all three. If a vector is used for x but y or z is missing, default values of 0 will be used.

The "shapelist3d" class is simply a list of "shape3d" objects.

Methods for dot3d, wire3d, shade3d, translate3d, scale3d, and rotate3d are defined for these objects.

Value

An object of class c("shapelist3d", "shape3d").

Author(s)

Duncan Murdoch

See Also

mesh3d

Examples

```
open3d()
shapelist3d(icosahedron3d(), x = rnorm(10), y = rnorm(10), z = rnorm(10), col = 1:5, size = 0.3)
```

---

shiny *Functions for integration of RGL widgets into Shiny app*

Description

These functions allow an RGL scene to be embedded in a Shiny app.

Usage

```
rglwidgetOutput(outputId, width = "512px", height = "512px")
renderRglwidget(expr, env = parent.frame(), quoted = FALSE, outputArgs = list())

playwidgetOutput(outputId, width = "0px", height = "0px")
renderPlaywidget(expr, env = parent.frame(), quoted = FALSE, outputArgs = list())
```

Arguments

outputId	The name for the control.
width, height	Width and height to display the control.
expr	An R expression returning a rglwidget (for renderRglwidget) or a playwidget (for renderPlaywidget) as output.
env	The environment in which to evaluate expr.
quoted	Is the expression already quoted?
outputArgs	A list containing arguments; see details below.

Details

Use rglwidgetOutput or playwidgetOutput as an output object in a Shiny user interface section; use renderRglwidget or renderPlaywidget as the render function in the server section.

In a dynamic R Markdown document with runtime: shiny, you only call the render function, and may optionally pass width and height to the output function by putting them in a list in outputArgs. See the example below.

Value

Used internally by Shiny.

Author(s)

Duncan Murdoch

Examples

```
## Not run:
# This could be used in a dynamic R Markdown document.  See
# demo("shinyDemo") and demo("simpleShinyRgl") for Shiny apps.

inputPanel(
  sliderInput("n", label = "n", min = 10, max = 100, value = 10, step = 10)
)

renderRglwidget({
    n <- input$n
    try(close3d())
    plot3d(rnorm(n), rnorm(n), rnorm(n))
    rglwidget()
  }, outputArgs = list(width = "auto", height = "300px"))


## End(Not run)
```

shinyGetPar3d *Communicate RGL parameters between R and Javascript in Shiny*

Description

These functions allow Shiny apps to read and write the par3d settings that may have been modified by user interaction in the browser.

Usage

```
shinyGetPar3d(parameters, session, subscene = currentSubscene3d(cur3d()), tag = "")
shinySetPar3d(..., session, subscene = currentSubscene3d(cur3d()))
shinyResetBrush(session, brush)
```

Arguments

parameters	A character vector naming the parameters to get.
session	The Shiny session object.
subscene	The subscene to which the parameters apply. Defaults to the currently active subscene in the R session.
tag	An arbitrary string or value which will be sent as part of the response.
...	A number of name = value pairs to be modified, or a single named list of parameters. Entries named tag or subscene will be ignored.
brush	The name of a Shiny input element corresponding to the shinyBrush argument to rglwidget.

Details

Requesting information from the browser is a complicated process. The shinyGetPar3d function doesn't return the requested value, it just submits a request for the value to be returned later in input$par3d, a reactive input. No action will result except when a reactive observer depends on input$par3d. See the example code below.

The shinySetPar3d function sends a message to the browser asking it to change a particular parameter. The change will be made immediately, without sending the full scene to the browser, so should be reasonably fast.

Value

These functions are called for their side effects, and don't return useful values.

The side effect of shinyGetPar3d is to cause input$par3d to be updated sometime later. Besides the requested parameter values, input$par3d will contain a copy of the subscene and tag arguments.

The side effect of shinySetPar3d is to send a message to the browser to update its copy of the par3d parameters immediately.

Note

R and the browser don't maintain a perfect match between the way parameters are stored internally. The browser version of parameters will be returned by `shinyGetPar3d` and should be supplied to `shinySetPar3d`.

Author(s)

Duncan Murdoch

References

https://shiny.rstudio.com/articles/communicating-with-js.html describes the underlying mechanisms used by these two functions.

See Also

The `rglwidget` argument `shinySelectionInput` allows information about mouse selections to be returned to R.

Examples

```
## Not run:

# This example shows how to retrieve the result of user
# interaction back into R from WebGL

# Draw an rgl plot

xyz <- matrix(rnorm(300), ncol = 3)
plot3d(xyz, type = "s")

# This function displays the plot in WebGL, with a
# button to capture the userMatrix and zoom values

CapturePar3dVals <- function(thescene = scene3d()) {
  ui <- shiny::fluidPage(
    shiny::sidebarLayout(
      shiny::sidebarPanel(
        shiny::actionButton("get_vals", "Capture par3d")
      ),
      shiny::mainPanel(
        rglwidgetOutput("the_plot")
      )
    )
  )

  server <- function(input, output, session) {

    output$the_plot <- renderRglwidget({
      rglwidget(thescene)
    })
```

```
    shiny::observeEvent(input$get_vals, {
      shinyGetPar3d(c("zoom", "userMatrix"), session)
    })

    shiny::observe({
      shiny::req(input$par3d)
      shiny::stopApp(returnValue = input$par3d[c("userMatrix", "zoom")])
    })
  }

  shiny::runApp(shiny::shinyApp(ui, server))
}

# Run it to capture the parameters

par3dvals <- CapturePar3dVals()

# Mirror them in the local plot
par3d(par3dvals)

## End(Not run)
```

---

show2d *Draw a 2D plot on a rectangle in a 3D scene*

Description

This function uses a bitmap of a standard 2D graphics plot as a texture on a quadrilateral. Default arguments are set up so that it will appear on the face of the bounding box of the current 3D plot, but optional arguments allow it to be placed anywhere in the scene.

Usage

```
show2d(expression,
  face = "z-", line = 0,
  reverse = FALSE, rotate = 0,
  x = NULL, y = NULL, z = NULL,
  width = 480, height = 480,
  filename = NULL,
  ignoreExtent = TRUE,
  color = "white", specular = "black", lit = FALSE,
  texmipmap = TRUE, texminfilter = "linear.mipmap.linear",
  expand = 1.03,
  texcoords = matrix(c(0, 1, 1, 0, 0, 0, 1, 1), ncol = 2),  ...)
```

Arguments

expression	Any plotting commands to produce a plot in standard graphics. Ignored if filename is not NULL.
face	A character string defining which face of the bounding box to use. See Details below.
line	How far out from the bounding box should the quadrilateral be placed? Uses same convention as mtext3d: not lines of text, but fraction of the bounding box size.
reverse, rotate	Should the image be reversed or rotated? See Details below.
x, y, z	Specific values to use to override face.
width, height	Parameters to pass to png when creating the bitmap. See Details below.
filename	A '.png' file image to use as the texture.
ignoreExtent	Whether the quadrilateral should be ignored when computing the bounding box of the scene.
color, specular, lit, texmipmap, texminfilter, ...	Material properties to use for the quadrilateral.
expand	Amount by which the quadrilateral is expanded outside the bounding box of the data.
texcoords	Coordinates on the image. Lower left of the bitmap is c(0,0), upper right is c(1,1).

Details

The default arguments are chosen to make it easy to place a 2D image on the face of the bounding box. If x, y and z are NULL (the defaults), face will be used as a code for one of the six faces of the bounding box. The first letter should be "x", "y" or "z"; this defines the axis perpendicular to the desired face. If the second letter is "-" or is missing, the face will be chosen to be the face with the lower value on that axis. Any other letter will use the opposite face.

If any of x, y or z is given, the specified value will be used to replace the value calculated above. Usually four values should be given, corresponding to the coordinates of the lower left, lower right, upper right and upper left of the destination for the image before reverse and rotate are used. Fewer values can be used for one or two coordinates; cbind will be used to put together all 3 coordinates into a 4 by 3 matrix (which will be returned as an attribute of the result).

The bitmap plot will by default be oriented so that it is properly oriented when viewed from the direction of the higher values of the perpendicular coordinate, and its lower left corner is at the lower value of the two remaining coordinates. The argument reverse causes the orientation to be mirrored, and rotate causes it to be rotated by multiples of 90 degrees. rotate should be an integer, with 0 for no rotation, 1 for a 90 degree counter-clockwise rotation, etc.

The width and height arguments control the shape and resolution of the bitmap. The defaults give a square bitmap, which is appropriate with the usual c(1,1,1) aspect ratios (see aspect3d). Some tuning may be needed to choose the resolution. The plot will look best when displayed at its original size; shrinking it smaller tends to make it look faded, while expanding it bigger will make it look blurry. If filename is given, the width and height will be taken from the file, and width and height arguments will be ignored.

Value

Invisibly returns the id value of the quadrilateral, with the following attributes:

value	The value returned by expression.
xyz	A 4 by 3 matrix giving the coordinates of the corners as used in plotting.
texcoords	A 4 by 2 matrix giving the texture coordinates of the image.
filename	The filename for the temporary file holding the bitmap image.

Author(s)

Duncan Murdoch

See Also

bgplot3d uses a plot as the background for the window.

Examples

```
example(plot3d, ask = FALSE)
show2d({
  par(mar=c(0,0,0,0))
  plot(x, y, col = rainbow(1000), axes=FALSE)
  })
```

---

snapshot3d *Export screenshot*

Description

Saves the screenshot to a file.

Usage

```
rgl.snapshot( filename, fmt = "png", top = TRUE )
snapshot3d( filename = tempfile(fileext = ".png"),
            fmt = "png", top = TRUE,
            ..., scene, width = NULL, height = NULL,
            webshot = as.logical(Sys.getenv("RGL_USE_WEBSHOT", "TRUE")) )
```

Arguments

filename	path to file to save.
fmt	image export format, currently supported: png. Ignored if webshot = TRUE.
top	whether to call rgl.bringtotop. Ignored if webshot = TRUE.
...	arguments to pass to webshot2::webshot
scene	an optional result of scene3d or rglwidget to plot
width, height	optional specifications of output size in pixels
webshot	Use the **webshot2** package to take the snapshot

Details

`rgl.snapshot()` is a low-level function that copies the current RGL window from the screen. Users should usually use `snapshot3d()` instead; it is more flexible, and (if **webshot2** is installed) can take images even if no window is showing, and they can be larger than the physical screen. On some systems **webshot2** doesn't work reliably; if you find `snapshot3d()` failing or taking a very long time I'd recommend using `snapshot3d(..., webshot = FALSE)`. See the note below about `RGL_USE_WEBSHOT` to make this the default.

Animations can be created in a loop modifying the scene and saving each screenshot to a file. Various graphics programs (e.g. ImageMagick) can put these together into a single animation. (See `movie3d` or the example below.)

Value

These functions are mainly called for the side effects. The filename of the saved file is returned invisibly.

Note

When `rgl.useNULL()` is `TRUE`, only `webshot = TRUE` will produce a snapshot. It requires the **webshot2** package and a Chrome browser. If no suitable browser is found, `snapshot3d()` will revert to `rgl.snapshot()`. To override the automatic search, set environment variable `CHROMOTE_CHROME` to the path to a suitable browser.

`rgl.snapshot` works by taking an image from the displayed window on-screen. On some systems, the snapshot will include content from other windows if they cover the active RGL window. Setting `top = TRUE` (the default) will use `rgl.bringtotop` before the snapshot to avoid this.

There are likely limits to how large `width` and `height` can be set based on the display hardware; if these are exceeded the results are undefined. A typical result is that the snapshot will still be made but at a smaller size.

There are slight differences between the displays with `webshot = TRUE` and `webshot = FALSE`, as the former are rendered using WebGL while the latter are rendered using OpenGL. Often the `webshot = TRUE` displays have better quality, but they are usually slower to produce, sometimes drastically so.

Set the environment variable `RGL_USE_WEBSHOT` to `"FALSE"` if you want `rgl.snapshot` to be used by default.

See Also

`movie3d`, `view3d`

Examples

```
if (interactive() && !in_pkgdown_example()) {
  saveopts <- options(rgl.useNULL = TRUE)
  plot3d(matrix(rnorm(300), ncol = 3, dimnames = list(NULL, c("x", "y", "z"))),
         col = "red")
  options(saveopts)
  browseURL(snapshot3d())
}
```

```
## Not run:

#
# create animation
#

shade3d(oh3d(), color = "red")
rgl.bringtotop()
view3d(0, 20)

olddir <- setwd(tempdir())
for (i in 1:45) {
  view3d(i, 20)
  filename <- paste("pic", formatC(i, digits = 1, flag = "0"), ".png", sep = "")
  snapshot3d(filename)
}
## Now run ImageMagick in tempdir().  Use 'convert' instead of 'magick'
## if you have an older version of ImageMagick:
##    magick -delay 10 *.png -loop 0 pic.gif
setwd(olddir)

## End(Not run)
```

`spheres3d` *Add spheres*

Description

Adds a sphere set shape node to the scene

Usage

```
spheres3d(x, y = NULL, z = NULL, radius = 1, fastTransparency = TRUE, ...)
```

Arguments

x, y, z	Numeric vector of point coordinates corresponding to the center of each sphere. Any reasonable way of defining the coordinates is acceptable. See the function `xyz.coords` for details.
`radius`	Vector or single value defining the sphere radius/radii
`fastTransparency`	logical value indicating whether fast sorting should be used for transparency. See the Details.
`...`	Material properties. See `material3d` for details.

Details

If a non-isometric aspect ratio is chosen, these functions will still draw objects that appear to the viewer to be spheres. Use `ellipse3d` to draw shapes that are spherical in the data scale.

When the scale is not isometric, the radius is measured in an average scale. In this case the bounding box calculation is iterative, since rescaling the plot changes the shape of the spheres in user-coordinates, which changes the bounding box. Versions of **rgl** prior to 0.92.802 did not do this iterative adjustment.

If any coordinate or radius is `NA`, the sphere is not plotted.

If a texture is used, its bitmap is wrapped around the sphere, with the top edge at the maximum y coordinate, and the left-right edges joined at the maximum in the z coordinate, centred in x.

If the `alpha` material value of the spheres is less than the default `1`, they need to be drawn in order from back to front. When `fastTransparency` is `TRUE`, this is approximated by sorting the centers and drawing complete spheres in that order. This produces acceptable results in most cases, but artifacts may be visible, especially if the `radius` values vary, or they intersect other transparent objects. Setting `fastTransparency = FALSE` will cause the sorting to apply to each of the 480 facets of individual spheres. This is much slower, but may produce better output.

Value

A shape ID of the spheres object is returned.

See Also

`material3d`, `aspect3d` for setting non-isometric scales

Examples

```
open3d()
spheres3d(rnorm(10), rnorm(10), rnorm(10),
          radius = runif(10), color = rainbow(10))
```

---

spin3d *Create a function to spin a scene at a fixed rate*

Description

This creates a function to use with `play3d` to spin an RGL scene at a fixed rate.

Usage

```
spin3d(axis = c(0, 0, 1), rpm = 5,
       dev = cur3d(), subscene = par3d("listeners", dev = dev))
```

Arguments

axis	The desired axis of rotation
rpm	The rotation speed in rotations per minute
dev	Which RGL device to use
subscene	Which subscene to use

Value

A function with header function(time, base = M), where M is the result of par3d("userMatrix") at the time the function is created. This function calculates and returns a list containing userMatrix updated by spinning the base matrix for time seconds at rpm revolutions per minute about the specified axis.

Note

Prior to **rgl** version 0.95.1476, the subscene argument defaulted to the current subscene, and any additional entries would be ignored by play3d. The current default value of par3d("listeners", dev = dev) means that all subscenes that share mouse responses will also share modifications by this function.

Author(s)

Duncan Murdoch

See Also

play3d to play the animation

Examples

```
# Spin one object
open3d()
plot3d(oh3d(col = "lightblue", alpha = 0.5))
if (!rgl.useNULL()  && interactive())
  play3d(spin3d(axis = c(1, 0, 0), rpm = 30), duration = 2)

# Show spinning sprites, and rotate the whole view
open3d()
spriteid <- NULL

spin1 <- spin3d(rpm = 4.5 ) # the scene spinner
spin2 <- spin3d(rpm = 9 ) # the sprite spinner

f <- function(time) {
    par3d(skipRedraw = TRUE) # stops intermediate redraws
    on.exit(par3d(skipRedraw = FALSE)) # redraw at the end

    pop3d(id = spriteid) # delete the old sprite
    cubeid <- shade3d(cube3d(), col = "red")
    spriteid <<- sprites3d(0:1, 0:1, 0:1, shape = cubeid,
```

```
                    userMatrix = spin2(time,
                      base = spin1(time)$userMatrix)$userMatrix)
    spin1(time)
}
if (!rgl.useNULL() && interactive())
  play3d(f, duration = 2)
```

---

sprites *Add sprites*

Description

Adds a sprite set shape node to the scene.

Usage

```
sprites3d(x, y = NULL, z = NULL, radius = 1,
          shapes = NULL, userMatrix,
          fixedSize = FALSE,
          adj = 0.5, pos = NULL, offset = 0.25,
          rotating = FALSE, ...)

particles3d(x, y = NULL, z = NULL, radius = 1, ...)
```

Arguments

x, y, z	point coordinates. Any reasonable way of defining the coordinates is acceptable. See the function xyz.coords for details.
radius	vector or single value defining the sprite radius
shapes	NULL for a simple square, a vector of identifiers of shapes in the scene, or a list of such vectors. See Details.
userMatrix	if shape is not NULL, the transformation matrix for the shapes
fixedSize	should sprites remain at a fixed size, or resize with the scene?
adj, pos, offset	positioning arguments; see Details
rotating	should sprites remain at a fixed orientation, or rotate with the scene?
...	material properties when shapes = NULL, texture mapping is supported

Details

Simple sprites (used when shapes is NULL) are 1 by 1 squares that are directed towards the viewpoint. Their primary use is for fast (and faked) atmospherical effects, e.g. particles and clouds using alpha blended textures. Particles are sprites using an alpha-blended particle texture giving the illusion of clouds and gases. The centre of each square will by default be at the coordinates given by x, y, z. This may be adjusted using the adj or pos parameters.

adj and pos are treated similarly to the same parameters for text3d. adj has 3 entries, for adjustment to the x, y and z coordinates respectively. For x, a value of 0 puts the sprite to the right of the

specified point, 0.5 centers it there, and 1 puts it to the left. The other coordinates are similar. By default, each value is 0.5 and the sprites are centered at the points given by `(x, y, z)`.

The `pos` parameter overrides `adj`. It should be an integer or vector of integers (one per point), interpreted as in `text3d` to position the sprite relative to the `(x, y, z)` point: 0 is centered on it, 1 is below, 2 is to the left, 3 is above, 4 is to the right, 5 is in front, and 6 is behind. `offset` is the fraction of the sprite size to separate it from the point.

When `shapes` is not `NULL`, it should be a vector of identifiers of objects to plot in the scene (e.g. as returned by plotting functions or by `ids3d`), or a list of such vectors. The referenced objects will be removed from the scene and duplicated as sprite images in a constant orientation, as specified by `userMatrix`. By default the origin `(0, 0, 0)` will be plotted at the coordinates given by `(x, y, z)`, perhaps modified by `adj` or `pos`.

If `shapes` is a vector all entries in it will be plotted at every location. If `shapes` is a list, different shapes will be plotted at each location. All entries in list entry 1 will be plotted at location 1, all entries in entry 2 at location 2, etc. Entries will be recycled as needed.

The `userMatrix` argument is ignored for `shapes = NULL`. For shapes, `sprites3d` defaults the matrix to `r3dDefaults$userMatrix`.

If any coordinate is `NA`, the sprite is not plotted.

The id values of the shapes may be retrieved after plotting using `rgl.attrib(id, "ids")`, the associated entry in `shapes` is retrievable in `rgl.attrib(id, "shapenum")`, and the user matrix is retrieved using `rgl.attrib(id, "usermatrix")`.

Value

These functions are called for the side effect of displaying the sprites. The shape ID of the displayed object is returned.

Note

While any rgl objects can be used as 3D sprites, spheres produced by `spheres3d` and other objects that adapt to the coordinate system may not render properly. To plot spheres, construct them as mesh objects as shown in the example.

See Also

`material3d`, `text3d`, `pch3d`

Examples

```
open3d()
particles3d( rnorm(100), rnorm(100), rnorm(100), color = rainbow(100) )
# is the same as
sprites3d( rnorm(100), rnorm(100), rnorm(100), color = rainbow(100),
  lit = FALSE, alpha = .2,
  textype = "alpha", texture = system.file("textures/particle.png", package = "rgl") )
sprites3d( rnorm(10) + 6, rnorm(10), rnorm(10), shape = shade3d(tetrahedron3d(), col = "red") )


open3d()
```

```
# Since the symbols are objects in the scene, they need
# to be added to the scene after calling plot3d()

plot3d(iris, type = "n")

# Use list(...) to apply different symbols to different points.

symbols <- list(shade3d(cube3d(), col = "red"),
                shade3d(tetrahedron3d(), col = "blue"),
                # Construct spheres
                shade3d(addNormals(subdivision3d(icosahedron3d(), 2)),
                        col = "yellow"))

sprites3d(iris, shape = symbols[iris$Species], radius = 0.1)
```

`subdivision3d` *Subdivide a mesh*

Description

The subdivision surface algorithm divides and refines (deforms) a given mesh recursively to certain degree (depth). The mesh3d algorithm consists of two stages: divide and deform. The divide step generates for each triangle or quad four new triangles or quads, the deform step drags the points (refinement step).

Usage

```
  subdivision3d( x,  ...)
  ## S3 method for class 'mesh3d'
subdivision3d( x, depth = 1, normalize = FALSE,
                deform = TRUE, keepTags = FALSE, ... )
  divide.mesh3d(mesh,  vb = mesh$vb,
                ib = mesh$ib, it = mesh$it, is = mesh$is,
                keepTags = FALSE)
  normalize.mesh3d(mesh)
  deform.mesh3d(mesh, vb = mesh$vb, ib = mesh$ib, it = mesh$it,
                is = mesh$is)
```

Arguments

`x`	3d geometry mesh
`mesh`	3d geometry mesh
`depth`	recursion depth
`normalize`	normalize mesh3d coordinates after division if `deform` is `TRUE`
`deform`	deform mesh
`keepTags`	if `TRUE`, add a `"tags"` component to the output.

is	indices for segments
it	indices for triangular faces
ib	indices for quad faces
vb	matrix of vertices: 4 x n matrix (rows x, y, z, h) or equivalent vector, where h indicates scaling of each plotted quad
...	other arguments (unused)

Details

subdivision3d takes a mesh object and replaces each segment with two new ones, and each triangle or quad with 4 new ones by adding vertices half-way along the edges (and one in the centre of a quad). The positions of the vertices are deformed so that the resulting surface is smoother than the original. These operations are repeated depth times.

The other functions do the individual steps of the subdivision. divide.mesh3d adds the extra vertices. deform.mesh3d does the smoothing by replacing each vertex with the average of each of its neighbours. normalize.mesh3d normalizes the homogeneous coordinates, by setting the 4th coordinate to 1. (The 4th coordinate is used as a weight in the deform step.)

Value

A modified mesh3d object. If keepTags is TRUE, it will contain a tags component. For details, see the clipMesh3d help topic.

See Also

r3d mesh3d

Examples

```
open3d()
shade3d( subdivision3d( cube3d(), depth = 3 ), color = "red", alpha = 0.5 )
```

---

subscene3d *Create, select or modify a subscene*

Description

This creates a new subscene, or selects one by id value, or adds objects to one.

Usage

```
newSubscene3d(viewport = "replace",
              projection = "replace",
              model = "replace",
              mouseMode = "inherit",
              parent = currentSubscene3d(),
```

```
                copyLights = TRUE,
                copyShapes = FALSE,
                copyBBoxDeco = copyShapes,
                copyBackground = FALSE, newviewport,
                ignoreExtent)
currentSubscene3d(dev = cur3d())
useSubscene3d(subscene)
addToSubscene3d(ids = tagged3d(tags), tags, subscene = currentSubscene3d())
delFromSubscene3d(ids = tagged3d(tags), tags, subscene = currentSubscene3d())
gc3d(protect = NULL)
```

Arguments

`viewport`, `projection`, `model`, `mouseMode`	How should the new subscene be embedded? Possible values are `c("inherit", "modify", "replace")`. See Details below.
`parent`	The parent subscene (defaults to the current subscene).
`copyLights`, `copyShapes`, `copyBBoxDeco`, `copyBackground`	Whether lights, shapes, bounding box decorations and background should be copied to the new subscene.
`newviewport`	Optionally specify the new subscene's viewport (in pixels).
`ignoreExtent`	Whether to ignore the subscene's bounding box when calculating the parent bounding box. Defaults to `TRUE` if `model` is not `"inherit"`.
`dev`	Which RGL device to query for the current subscene.
`subscene`	Which subscene to use or modify.
`ids`	A vector of integer object ids to add to the subscene.
`tags`	Alternate way to specify `ids`. Ignored if `ids` is given.
`protect`	Object ids to protect from this garbage collection.

Details

The **rgl** package allows multiple windows to be open; each one corresponds to a "scene". Within each scene there are one or more "subscenes". Each subscene corresponds to a rectangular region in the window, and may have its own projection, transformation and behaviour in response to the mouse.

There is always a current subscene: most graphic operations make changes there, e.g. by adding an object to it.

The scene "owns" objects; `addToSubscene3d` and `delFromSubscene3d` put their ids into or remove them from the list being displayed within a particular subscene. The `gc3d` function deletes objects from the scene if they are not visible in any subscene, unless they are protected by having their id included in `protect`.

The `viewport`, `projection` and `model` parameters each have three possible settings: `c("inherit", "modify", "replace")`. `"inherit"` means that the corresponding value from the parent subscene will be used. `"replace"` means that the new subscene will have its own value of the value, independent of its parent. `"modify"` means that the child value will be applied first, and then the parent

value will be applied. For viewport, this means that if the parent viewport is changed, the child will maintain its relative position. For the two matrices, "modify" is unlikely to give satisfactory results, but it is available for possible use.

The mouseMode parameter can only be one of c("inherit", "replace"). If it is "inherit", the subscene will use the mouse controls of the parent, and any change to them will affect the parent and all children that inherit from it. This is the behaviour that was present before **rgl** version 0.100.13. If it is "replace", then it will receive a copy of the parent mouse controls, but modifications to them will affect only this subscene, not the parent. Note that this is orthogonal to the par3d("listeners") setting: if another subscene is listed as a listener, it will respond to mouse actions using the same mode as the one receiving them.

The viewport parameter controls the rectangular region in which the subscene is displayed. It is specified using newviewport (in pixels relative to the whole window), or set to match the parent viewport.

The projection parameter controls settings corresponding to the observer. These include the field of view and the zoom; they also include the position of the observer relative to the model. The par3d("projMatrix") matrix is determined by the projection.

The model parameter controls settings corresponding to the model. Mouse rotations affect the model, as does scaling. The par3d("modelMatrix") matrix is determined by these as well as by the position of the observer (since OpenGL assumes that the observer is at (0, 0, 0) after the MODELVIEW transformation). Only those parts concerning the model are inherited when model specifies inheritance, the observer setting is controlled by projection.

If copyBackground is TRUE, the background of the newly created child will overwrite anything displayed in the parent subscene, regardless of depth.

Value

If successful, each function returns the object id of the subscene, with the exception of gc3d, which returns the count of objects which have been deleted, and useSubscene3d, which returns the previously active subscene id.

Author(s)

Duncan Murdoch and Fang He.

See Also

subsceneInfo for information about a subscene, mfrow3d and layout3d to set up multiple panes of subscenes.

Examples

```
# Show the Earth with a cutout by using clipplanes in subscenes

lat <- matrix(seq(90, -90, length.out = 50)*pi/180, 50, 50, byrow = TRUE)
long <- matrix(seq(-180, 180, length.out = 50)*pi/180, 50, 50)

r <- 6378.1 # radius of Earth in km
x <- r*cos(lat)*cos(long)
```

```
y <- r*cos(lat)*sin(long)
z <- r*sin(lat)

open3d()
obj <- surface3d(x, y, z, col = "white",
       texture = system.file("textures/worldsmall.png", package = "rgl"),
       specular = "black", axes = FALSE, box = FALSE, xlab = "", ylab = "", zlab = "",
       normal_x = x, normal_y = y, normal_z = z)

cols <- c(rep("chocolate4", 4), rep("burlywood1", 4), "darkgoldenrod1")
rs <- c(6350, 5639, 4928.5, 4207, 3486,
                        (3486 + 2351)/2, 2351, (2351 + 1216)/2, 1216)
for (i in seq_along(rs))
  obj <- c(obj, spheres3d(0, 0, col = cols[i], radius = rs[i]))

root <- currentSubscene3d()

newSubscene3d("inherit", "inherit", "inherit", copyShapes = TRUE, parent = root)
clipplanes3d(1, 0, 0, 0)

newSubscene3d("inherit", "inherit", "inherit", copyShapes = TRUE, parent = root)
clipplanes3d(0, 1, 0, 0)

newSubscene3d("inherit", "inherit", "inherit", copyShapes = TRUE, parent = root)
clipplanes3d(0, 0, 1, 0)

# Now delete the objects from the root subscene, to reveal the clipping planes
useSubscene3d(root)
delFromSubscene3d(obj)
```

`subsceneInfo` *Get information on subscenes*

Description

This function retrieves information about the tree of subscenes shown in the active window.

Usage

```
subsceneInfo(id = NA, embeddings, recursive = FALSE)
```

Arguments

`id`	Which subscene to report on; `NA` is the current subscene. Set to `"root"` for the root.
`embeddings`	Optional new setting for the embeddings for this subscene.
`recursive`	Whether to report on children recursively.

Details

In RGL, each window contains a tree of "subscenes", each containing views of a subset of the objects defined in the window.

Rendering in each subscene depends on the viewport, the projection, and the model transformation. Each of these characteristics may be inherited from the parent (`embedding[i] = "inherit"`), may modify the parent (`embedding[i] = "modify"`), or may replace the parent (`embedding[i] == "replace"`). All three must be specified if `embeddings` is used.

Value

`id`	The object id of the subscene
`parent`	The object id of the parent subscene, if any
`children`	If `recursive`, a list of the information for the children, otherwise just their object ids.
`embedding`	A vector of 3 components describing how this subscene is embedded in its parent.

Author(s)

Duncan Murdoch

See Also

`newSubscene3d`

Examples

```
example(plot3d)
subsceneInfo()
```

---

surface3d *Add surface*

Description

Adds a surface to the current scene. The surface is defined by a matrix defining the height of each grid point and two vectors or matrices defining the grid.

Usage

```
surface3d(x, y, z, ...,
          normal_x = NULL, normal_y = NULL, normal_z = NULL,
          texture_s=NULL, texture_t=NULL, flip = FALSE)
```

Arguments

x, y, z	vectors or matrices of values. See Details.
...	Material properties. See material3d for details.
normal_x, normal_y, normal_z	matrices giving the coordinates of normals at each grid point
texture_s, texture_t	matrices giving the texture coordinates at each grid point
flip	flip definition of "up"

Details

Adds a surface mesh to the current scene. The surface is typically defined by a matrix of height values in z (as in persp), but any of x, y, or z may be matrices or vectors, as long as at least one is a matrix. (One historical exception is allowed: if all are vectors but the length of z is the product of the lengths of x and y, z is converted to a matrix.)

Dimensions of all matrices must match.

If any of the coordinates are vectors, they are interpreted as follows:

- If x is a vector, it corresponds to rows of the matrix.
- If y is a vector, it corresponds to columns of the matrix.
- If z is a vector, it corresponds to columns unless y is also a vector, in which case it corresponds to rows.

If the normals are not supplied, they will be calculated automatically based on neighbouring points.

Texture coordinates run from 0 to 1 over each dimension of the texture bitmap. If texture coordinates are not supplied, they will be calculated to render the texture exactly once over the grid. Values greater than 1 can be used to repeat the texture over the surface.

surface3d always tries to draw the surface with the 'front' upwards (typically towards higher z values). This can be used to render the top and bottom differently; see material3d and the example below. If you don't like its choice, set flip = TRUE to use the opposition definition.

NA values in the height matrix are not drawn.

See Also

See persp3d for a higher level interface.

Examples

```
#
# volcano example taken from "persp"
#

z <- 2 * volcano        # Exaggerate the relief

x <- 10 * (1:nrow(z))   # 10 meter spacing (S to N)
y <- 10 * (1:ncol(z))   # 10 meter spacing (E to W)
```

```
zlim <- range(z)
zlen <- zlim[2] - zlim[1] + 1

colorlut <- terrain.colors(zlen) # height color lookup table

col <- colorlut[ z - zlim[1] + 1 ] # assign colors to heights for each point

open3d()
surface3d(x, y, z, color = col, back = "lines")
```

`tagged3d` *Find tags on rgl objects.*

Description

Objects with material properties may have an arbitrary string set as a tag. This function retrieves the id values associated with a given tag, or the tags set on given ids.

Usage

```
tagged3d(tags = NULL, ids = NULL, full = FALSE, subscene = 0)
```

Arguments

`tags`	A vector of tags to use for selection.
`ids`	A vector of ids to report the tags on.
`full`	logical; whether to return a dataframe containing `id`, `type`, `tag`, or a vector of ids or tags.
`subscene`	Where to look: by default, the whole scene is searched. `NA` restricts the search to the current subscene, or a subscene id can be given.

Details

Exactly one of `tags` and `ids` must be specified.

Value

A dataframe is constructed with columns

`id`	item id
`type`	item type
`tag`	item tag

matching the specified `tags` or `ids` value. If `full = TRUE`, the full dataframe is returned, otherwise just the requested ids or tags.

If `ids` is specified, the return value will be in the same order as `ids`).

Author(s)

Duncan Murdoch

Examples

```
open3d()
ids <- plot3d(rnorm(10), rnorm(10), rnorm(10), tag = "plot")
unclass(ids)
tagged3d("plot")
tagged3d(ids = ids, full = TRUE)
```

---

text3d *Add text to plot*

Description

Adds text to the scene. The text is positioned in 3D space. Text is always oriented towards the camera.

Usage

```
text3d(x, y = NULL, z = NULL, texts, adj = 0.5, pos = NULL, offset = 0.5,
       usePlotmath = is.language(texts),
       family = par3d("family"), font = par3d("font"),
    cex = par3d("cex"), useFreeType = par3d("useFreeType"),
   ...)
texts3d(x, y = NULL, z = NULL, texts, adj = 0.5, pos = NULL, offset = 0.5,
       usePlotmath = is.language(texts),
       family = par3d("family"), font = par3d("font"),
    cex = par3d("cex"), useFreeType = par3d("useFreeType"),
   ...)
```

Arguments

x, y, z	point coordinates. Any reasonable way of defining the coordinates is acceptable. See the function xyz.coords for details.
texts	text character vector to draw
adj	one value specifying the horizontal adjustment, or two, specifying horizontal and vertical adjustment respectively, or three, specifying adjustment in all three directions.
pos	a position specifier for the text. If specified, this overrides any adj value given. Values of 0, 1, 2, 3, 4, 5, 6 respectively indicate positions at, below, to the left of, above, to the right of, in front of, and behind the specified coordinates.
offset	when pos is specified, this value gives the offset of the label from the specified coordinate in fractions of a character width.
usePlotmath	logical. Should plotmath3d be used for the text?

family	A device-independent font family name, or ""
font	A numeric font number from 1 to 4
cex	A numeric character expansion value
useFreeType	logical. Should FreeType be used to draw text? (See details below.)
...	Material properties; see material3d for details.

Details

The adj parameter determines the position of the text relative to the specified coordinate. Use adj = c(0, 0) to place the left bottom corner at (x, y, z), adj = c(0.5, 0.5) to center the text there, and adj = c(1, 1) to put the right top corner there. The optional second coordinate for vertical adjustment defaults to 0.5. Placement is done using the "advance" of the string and the "ascent" of the font relative to the baseline, when these metrics are known.

text3d and texts3d draw text using the r3d conventions. These are synonyms; the former is singular to be consistent with the classic 2-D graphics functions, and the latter is plural to be consistent with all the other graphics primitives. Take your choice!

If any coordinate or text is NA, that text is not plotted.

If usePlotmath is TRUE, the work will be done by the plotmath3d function. This is the default if the texts parameter is "language", e.g. the result of a call to expression or quote.

Value

The text drawing functions return the object ID of the text object (or sprites, in case of usePlotmath = TRUE) invisibly.

Fonts

Fonts are specified using the family, font, cex, and useFreeType arguments. Defaults for the currently active device may be set using par3d, or for future devices using r3dDefaults.

The family specification is the same as for standard graphics, i.e. families
c("serif", "sans", "mono", "symbol")
are normally available, but users may add additional families. font numbers are restricted to the range 1 to 4 for standard, bold, italic and bold italic respectively. Font 5 is recoded as family "symbol" font 1, but that is not supported unless specifically installed, so should be avoided.

Using an unrecognized value for "family" will result in the system standard font as used in RGL up to version 0.76. That font is not resizable and font values are ignored.

If useFreeType is TRUE, then RGL will use the FreeType anti-aliased fonts for drawing. This is generally desirable, and it is the default on non-Windows systems if RGL was built to support FreeType.

FreeType fonts are specified using the rglFonts function.

See Also

r3d, plotmath3d, rglExtrafonts for adding fonts

Examples

```
open3d()
famnum <- rep(1:3, 8)
family <- c("serif", "sans", "mono")[famnum]
font <- rep(rep(1:4, each = 3), 2)
cex <- rep(1:2, each = 12)
text3d(font, cex, famnum, texts = paste(family, font), adj = 0.5,
       color = "blue", family = family, font = font, cex = cex)
```

---

textureSource *Retrieve source code used to produce texture file.*

Description

Internally, **rgl** works with PNG files for textures. If a texture is requested using a different format, a temporary PNG file of the image will be saved. This function allows you to retrieve the original expression used to produce the texture.

Usage

```
textureSource(texture)
```

Arguments

`texture`	The filename of a texture file. If missing, the directory where texture files are stored will be returned.

Details

rgl creates a new file in the temporary directory whenever a non-PNG texture is used. It will delete them when it knows there are no references and at the end of the session, but conceivably there will be situations where you need to delete them earlier. Calling `textureSource()` with no arguments will give you the directory holding the textures so that they can be deleted sooner.

Value

If `texture` is specified and it is the name of a temporary PNG texture file produced by **rgl**, the expression used to specify the texture will be returned. If it is the name of some other file, `texture` will be returned.

If no argument is given, the session-specific directory holding the temporary texture files will be returned.

See Also

`material3d`

Examples

```
xyz <- cbind(c(0,1,1,0), c(0,0,1,1), c(0,0,0,0))
st <- xyz[,1:2]

open3d()
id <- quads3d(xyz, texcoords = st,
              texture = as.raster(matrix(colors()[1:120], ncol = 10)),
              col="white")
material3d(id = id, "texture")
textureSource(material3d(id = id, "texture"))
```

`thigmophobe3d` *Find the direction away from the closest point in a 3d projection*

Description

Jim Lemon's `thigmophobe` function in the `plotrix` package computes good directions for labels in a 2D plot. This function does the same for a particular projection in a 3D plot by projecting down to 2D and calling his function.

Usage

```
thigmophobe3d(x, y = NULL, z = NULL,
              P = par3d("projMatrix"),
              M = par3d("modelMatrix"),
              windowRect = par3d("windowRect"))
```

Arguments

`x`, `y`, `z`	point coordinates. Any reasonable way of defining the coordinates is acceptable. See the function `xyz.coords` for details.
`P`, `M`, `windowRect`	The projection and modelview matrices, and the size and position of the display in pixels.

Details

Since `thigmophobe3d` projects using fixed `P` and `M`, it will not necessarily choose good directions if the user rotates the display or makes any other change to the projection.

Value

A vector of values from 1 to 4 to be used as the `pos` argument in `text3d`.

Note

The example below shows how to update the directions during an animation; I find that the moving labels are distracting, and prefer to live with fixed ones.

Author(s)

Duncan Murdoch

References

plotrix

See Also

text3d

Examples

```
if (requireNamespace("plotrix", quietly = TRUE)) {
  # Simulate some data
  xyz <- matrix(rnorm(30), ncol = 3)

  # Plot the data first, to establish the projection
  plot3d(xyz)

  # Now thigmophobe3d can choose directions
  textid <- text3d(xyz, texts = 1:10, pos = thigmophobe3d(xyz))

  # Update the label positions during an animation
  if (interactive() && !rgl.useNULL()) {
    spin <- spin3d(rpm = 5)
    f <- function(time) {
      par3d(skipRedraw = TRUE)
      on.exit(par3d(skipRedraw = FALSE))
      pop3d(id = textid)
      # Need to rotate before thigmophobe3d is called
      result <- spin(time)
      par3d(userMatrix = result$userMatrix)
      textid <<- text3d(xyz, texts = 1:10, pos = thigmophobe3d(xyz))
      result
    }
    play3d(f, duration = 5)
  } else
    textid   # just print the static display
}
```

---

tkpar3dsave *Modal dialog for saving par3d settings*

Description

This function opens a TCL/TK modal dialog to allow particular views of an RGL scene to be saved.

Usage

```
tkpar3dsave(params = c("userMatrix", "scale", "zoom", "FOV"),
          times = FALSE, dev = cur3d(), ...)
```

Arguments

params	Which parameters to save
times	Should times be saved as well?
dev	Which RGL device to work with
...	Additional parameters to pass to tktoplevel

Details

This opens a TCL/TK modal dialog box with Record and Quit buttons. Each time Record is clicked, a snapshot is taken of current par3d settings. When Quit is clicked, the dialog closes and the values are returned in a list.

If times == TRUE, then the times at which the views are recorded will also be saved, so that the play3d function will play back with the same timing.

Value

A list of the requested components. Each one will consist of a list of values that were current when the Record button was clicked. These are suitable to be passed directly to the par3dinterp function.

Author(s)

Duncan Murdoch

See Also

par3d, par3dinterp

Examples

```
if (interactive() && !in_pkgdown_example()) {

  # Record a series of positions, and then play them back immediately
  # at evenly spaced times, in an oscillating loop
  example(plot3d)
  play3d( par3dinterp( tkpar3dsave() ) )

  # As above, but preserve the click timings

  # play3d( par3dinterp( tkpar3dsave(times=TRUE) ) )
}
```

tkrgl *The former tkrgl package*

Description

Functions from the former **tkrgl** package.

Details

The **tkrgl** package contained functions to use TCL/TK to control an RGL scene on screen. These functions have now been merged into the **rgl** package, and the **tkrgl** package has been archived.

To avoid conflicts with RGL names and to indicate the TCL/TK nature of these functions, they have all been prefixed with `tk`:

`tkpar3dsave` Formerly `tkrgl::par3dsave`, allows interactive saving of scene parameters.

`tkspin3d`, `tkspinControl` Formerly `tkrgl::spin3d` and `tkrgl::spinControl`, create buttons to spin the scene.

History:

0.2-2	First public release
0.3	Added possibility to control multiple windows
0.4	Compatibility with 2.0.0 tcltk package
0.5	Added continuous rotation
0.6	Added par3dsave
0.7	Added parameters to `tkspinControl`, fixed startup
0.8	Minor fixes to pass checks
0.9	Merge functions into **rgl**

tkspin3d *Create TCL/TK controller for RGL window*

Description

This function creates a TCL/TK window containing buttons to spin and resize one or more RGL windows.

Usage

```
tkspin3d(dev = cur3d(), ...)
```

Arguments

dev	A vector of one or more RGL device numbers to control
...	Named parameters in that match named formal arguments to tkspinControl are passed there, while others are passed to tktoplevel

Author(s)

Ming Chen and Duncan Murdoch

See Also

tkspinControl

Examples

```
if (interactive() && !in_pkgdown_example()) {
    open3d()
    points3d(rnorm(100), rnorm(100), rnorm(100), size=3)
    axes3d()
    box3d()
    tkspin3d()
}
```

---

tkspinControl *Create a spin control in a TCL/TK window*

Description

This function may be used to embed a spin control in a TCL/TK window.

Usage

```
tkspinControl(base, dev = cur3d(),
 continue=FALSE, speed=30, scale=100, ... )
```

Arguments

base	The TCL/TK frame in which to insert this control.
dev	A vector of one or more RGL device numbers to control.
continue	Initial setting for continuous rotation checkbox.
speed	Initial setting for speed slider.
scale	Initial setting for scale slider.
...	Additional parameters to pass to tkframe

Author(s)

Ming Chen and Duncan Murdoch

See Also

spin3d

Examples

```
if (interactive() && !in_pkgdown_example()) {
    library(tcltk)
    open3d()
    win1 <- cur3d()
    plot3d(rexp(100), rexp(100), rexp(100), size=3, col='green')

    open3d()
    win2 <- cur3d()
    plot3d(rt(100,2), rt(100,2), rt(100, 2), size=3, col='yellow')

    open3d()
    win3 <- cur3d()
    plot3d(rexp(100), rexp(100), rexp(100), size=3, col='red')

    open3d()
    win4 <- cur3d()
   plot3d(rbinom(100,10,0.5), rbinom(100,10,0.5), rbinom(100,10,0.5), size=3, col='cyan')

    base <- tktoplevel()
    tkwm.title(base, "Spinners")
    con1 <- tkspinControl(base, dev=c(win1,win2))
    con2 <- tkspinControl(base, dev=c(win3,win4))
    tkpack(con1, con2)
}
```

---

toggleWidget *An HTML widget to toggle display of elements of a scene*

Description

This function produces a button in an HTML scene that will toggle the display of items in the scene.

Usage

```
toggleWidget(sceneId,
             ids = tagged3d(tags), tags = NULL, hidden = integer(),
             subscenes = NULL,
             label,
             ...)
```

Arguments

sceneId	The HTML id of the RGL scene being controlled, or an object as in playwidget.
ids, hidden	The RGL id numbers of the objects to toggle. Those in ids are initially shown; those in hidden are initially hidden.
tags	Alternate way to specify ids. Ignored if ids is given.
subscenes	The subscenes in which to toggle the objects.
label	The label to put on the button. The default is set from the expression passed to ids or the value of tags.
...	Additional arguments to pass to playwidget.

Details

Like playwidget, this function is designed to work within the **htmlwidgets** framework. If the value is printed, the button will be inserted into the output.

It is also designed to work with **magrittr**-style pipes: the result of rglwidget or other widgets can be piped into it to add it to a display. It can also appear first in the pipeline, if sceneId is set to NA.

Value

A widget suitable for use in an **Rmarkdown**-generated web page, or elsewhere.

Author(s)

Duncan Murdoch

See Also

toggleButton for the older style of HTML control. The User Interaction in WebGL vignette gives more details.

Examples

```
theplot <- plot3d(rnorm(100), rnorm(100), rnorm(100), col = "red")
widget <- rglwidget(height = 300, width = 300) %>%
  toggleWidget(theplot["data"],
               hidden = theplot[c("xlab", "ylab", "zlab")],
               label = "Points")
if (interactive() || in_pkgdown_example())
  widget
```

`triangulate` *Triangulate a two-dimensional polygon*

Description

This algorithm decomposes a general polygon into simple polygons and uses the "ear-clipping" algorithm to triangulate it. Polygons with holes are supported.

Usage

```
triangulate(x, y = NULL, z = NULL, random = TRUE, plot = FALSE, partial = NA)
```

Arguments

`x`, `y`, `z`	Coordinates of a two-dimensional polygon in a format supported by `xyz.coords`. See Details for a description of proper input and how `z` is handled.
`random`	Currently ignored, the triangulation is deterministic.
`plot`	Whether to plot the triangulation; mainly for debugging purposes.
`partial`	Currently ignored. Improper input will lead to undefined results.

Details

Normally `triangulate` looks only at the `x` and `y` coordinates. However, if one of those is constant, it is replaced with the `z` coordinate if present.

The algorithm works as follows. First, it breaks the polygon into pieces separated by `NA` values in `x` or `y`. Each of these pieces should be a simple, non-self-intersecting polygon, not intersecting the other pieces. (Though some minor exceptions to this rule may work, none are guaranteed). The nesting of these pieces is determined: polygons may contain holes, and the holes may contain other polygons.

Vertex order around the polygons does not affect the results: whether a polygon is on the outside or inside of a region is determined by nesting.

Polygons should not repeat vertices. An attempt is made to detect if the final vertex matches the first one. If so, it will be deleted with a warning.

The "outer" polygon(s) are then merged with the polygons that they immediately contain, and each of these pieces is triangulated using the ear-clipping algorithm from the references.

Finally, all the triangulated pieces are put together into one result.

Value

A three-by-n array giving the indices of the vertices of each triangle. (No vertices are added; only the original vertices are used in the triangulation.)

The array has an integer vector attribute `"nextvert"` with one entry per vertex, giving the index of the next vertex to proceed counter-clockwise around outer polygon boundaries, clockwise around inner boundaries.

Note

Not all inputs will succeed, though inputs satisfying the rules listed in the Details section should.

Author(s)

R wrapper code written by Duncan Murdoch; the `earcut` library has numerous authors.

References

This function uses the C++ version of the `earcut` library from https://github.com/mapbox/earcut.hpp.

See Also

`extrude3d` for a solid extrusion of a polygon, `polygon3d` for a flat display; both use `triangulate`.

Examples

```
theta <- seq(0, 2*pi, length.out = 25)[-25]
theta <- c(theta, NA, theta, NA, theta, NA, theta, NA, theta)
r <- c(rep(1.5, 24), NA, rep(0.5, 24), NA, rep(0.5, 24), NA, rep(0.3, 24), NA, rep(0.1, 24))
dx <- c(rep(0, 24), NA, rep(0.6, 24), NA, rep(-0.6, 24), NA, rep(-0.6, 24), NA, rep(-0.6, 24))
x <- r*cos(theta) + dx
y <- r*sin(theta)
plot(x, y, type = "n")
polygon(x, y)
triangulate(x, y, plot = TRUE)
open3d()
polygon3d(x, y, x - y, col = "red")
```

---

turn3d *Create a solid of rotation from a two-dimensional curve*

Description

This function "turns" the curve (as on a lathe) to form a solid of rotation along the x axis.

Usage

```
turn3d(x, y = NULL, n = 12, smooth = FALSE, ...)
```

Arguments

x, y	Points on the curve, in a form suitable for `xy.coords`. The `y` values must be non-negative.
n	How many steps in the rotation?
smooth	logical; whether to add normals for a smooth appearance.
...	Additional parameters to pass to `tmesh3d`.

Value

A mesh object containing triangles and/or quadrilaterals.

Author(s)

Fang He and Duncan Murdoch

See Also

extrude3d

Examples

```
x <- 1:10
y <- rnorm(10)^2
open3d()
shade3d(turn3d(x, y), col = "green")
```

---

vertexControl *Set attributes of vertices*

Description

This is a function to produce actions in a web display. A playwidget or Shiny input control (e.g. a sliderInput control) sets a value which controls attributes of a selection of vertices.

Usage

```
vertexControl(value = 0, values = NULL, vertices = 1, attributes,
              objid = tagged3d(tag), tag,
              param = seq_len(NROW(values)) - 1, interp = TRUE)
```

Arguments

value	The value to use for input (typically input$value in a Shiny app.) Not needed with playwidget.
values	A matrix of values, each row corresponding to an input value.
vertices	Which vertices are being controlled? Specify vertices as a number from 1 to the number of vertices in the objid.
attributes	A vector of attributes of a vertex, from c("x", "y", "z", "red", "green", "blue", "alpha", "nx", "ny", "nz", "radii", "ox", "oy", "oz", "ts", "tt", "offset"). See Details.
objid	A single RGL object id.
tag	An alternate way to specify objid.
param	Parameter values corresponding to each row of values.
interp	Whether to interpolate between rows of values.

Details

This function modifies attributes of vertices in a single object. The `attributes` are properties of each vertex in a scene; not all are applicable to all objects. In order, they are: coordinates of the vertex `"x"`, `"y"`, `"z"`, color of the vertex `"red"`, `"green"`, `"blue"`, `"alpha"`, normal at the vertex `"nx"`, `"ny"`, `"nz"`, radius of a sphere at the vertex `"radius"`, origin within a texture `"ox"`, `"oy"` and perhaps `"oz"`, texture coordinates `"ts"`, `"tt"`.

Planes are handled specially. The coefficients `a`, `b`, `c` in the `planes3d` or `clipplanes3d` specification are controlled using `"nx"`, `"ny"`, `"nz"`, and `d` is handled as `"offset"`. The `vertices` argument is interpreted as the indices of the planes when these attributes are set.

If only one attribute of one vertex is specified, `values` may be given as a vector and will be treated as a one-column matrix. Otherwise `values` must be given as a matrix with `ncol(values) == max(length(vertices),length(attributes))`. The `vertices` and `attributes` vectors will be recycled to the same length, and entries from column `j` of `values` will be applied to vertex `vertices[j]`, attribute `attributes[j]`.

The `value` argument is translated into a row (or two rows if `interp = TRUE`) of `values` by finding its location in `param`.

Value

A list of class `"rglControl"` of cleaned up parameter values, to be used in an RGL widget.

Author(s)

Duncan Murdoch

See Also

The User Interaction in WebGL vignette gives more details.

Examples

```
saveopts <- options(rgl.useNULL = TRUE)

theta <- seq(0, 6*pi, length.out = 100)
xyz <- cbind(sin(theta), cos(theta), theta)
plot3d(xyz, type="l")
id <- spheres3d(xyz[1,,drop=FALSE], col="red")

widget <- rglwidget(width=500, height=300) %>%
playwidget(vertexControl(values=xyz,
                         attributes=c("x", "y", "z"),
                         objid = id, param=1:100),
           start = 1, stop = 100, rate=10)
if (interactive() || in_pkgdown_example())
  widget
options(saveopts)
```

`viewpoint` *Set up viewpoint*

Description

Set the viewpoint orientation.

Usage

```
view3d( theta = 0, phi = 15, fov = 60, zoom = 1,
        scale = par3d("scale"), interactive = TRUE, userMatrix,
        type = c("userviewpoint", "modelviewpoint") )
```

Arguments

`theta`, `phi`	polar coordinates in degrees. `theta` rotates round the vertical axis. `phi` rotates round the horizontal axis.
`fov`	field-of-view angle in degrees
`zoom`	zoom factor
`scale`	real length 3 vector specifying the rescaling to apply to each axis
`interactive`	logical, specifying if interactive navigation is allowed
`userMatrix`	4x4 matrix specifying user point of view
`type`	which viewpoint to set?

Details

The data model can be rotated using the polar coordinates `theta` and `phi`. Alternatively, it can be set in a completely general way using the 4x4 matrix `userMatrix`. If `userMatrix` is specified, `theta` and `phi` are ignored.

The pointing device of your graphics user-interface can also be used to set the viewpoint interactively. With the pointing device the buttons are by default set as follows:

left adjust viewpoint position

middle adjust field of view angle

right or wheel adjust zoom factor

The user's view can be set with `fov` and `zoom`.

If the `fov` angle is set to 0, a parallel or orthogonal projection is used. Small non-zero values (e.g. 0.01 or less, but not 0.0) are likely to lead to rendering errors due to OpenGL limitations.

Prior to version 0.94, all of these characteristics were stored in one viewpoint object. With that release the characteristics are split into those that affect the projection (the user viewpoint) and those that affect the model (the model viewpoint). By default, this function sets both, but the `type` argument can be used to limit the effect.

See Also

par3d

Examples

```
## Not run:
# animated round trip tour for 10 seconds

open3d()
shade3d(oh3d(), color = "red")

start <- proc.time()[3]
while ((i <- 36*(proc.time()[3] - start)) < 360) {
  view3d(i, i/4);
}

## End(Not run)
```

---

writeASY *Write Asymptote code for an RGL scene*

Description

Asymptote is a language for 3D graphics that is highly integrated with LaTeX. This is an experimental function to write an Asymptote program to approximate an RGL scene.

Usage

```
writeASY(scene = scene3d(),
         title = "scene",
         outtype = c("pdf", "eps", "asy", "latex", "pdflatex"),
         prc = TRUE,
         runAsy = "asy %filename%",
         defaultFontsize = 12,
         width = 7, height = 7,
         ppi = 100,
         ids = tagged3d(tags),
         tags = NULL,
         version = "2.65")
```

Arguments

scene	RGL scene object
outtype	What type of file to write? See Details.
prc	Whether to produce an interactive PRC scene.
title	The base of the filename to produce.
runAsy	Code to run the Asymptote program.

`defaultFontsize`
: The default fontsize for text.

`width, height`
: Width and height of the output image, in inches.

`ppi`
: "Pixels per inch" to assume when converting line widths and point sizes (which RGL measures in pixels).

`ids`
: If not `NULL`, write out just these RGL objects.

`tags`
: Alternate way to specify `ids`. Ignored if `ids` is given.

`version`
: Asymptote version 2.44 had a definition for its "light()" function that was incompatibly changed in versions 2.47 and 2.50. The current code has been tested with version 2.65. If you are using an older version, set `version` to your version number and it may work better.

Details

Asymptote is both a language describing a 2D or 3D graphic, and a program to interpret that language and produce output in a variety of formats including EPS, PDF (interactive or static), etc.

The interactive scene produced with `prc = TRUE` requires `outtype = "pdf"`, and (as of this writing) has a number of limitations:

- As far as we know, only Adobe Acrobat Reader of a sufficiently recent version can display these scenes.
- Current versions ignore lighting settings.

Value

The filename of the output file is returned invisibly.

Note

This function is currently under development and limited in the quality of output it produces. Arguments will likely change.

There are a number of differences between the interactive display in Asymptote and the display in RGL. In particular, many objects that are a fixed size in RGL will scale with the image in Asymptote. Defaults have been chosen somewhat arbitrarily; tweaking will likely be needed.

Material properties of surfaces are not yet implemented.

On some systems, the program `asy` used to process the output has bugs and may fail. Run the example at your own risk!

Author(s)

Duncan Murdoch

References

J. C. Bowman and A. Hammerlindl (2008). Asymptote: A vector graphics language, TUGBOAT: The Communications of the TeX Users Group, 29:2, 288-294.

See Also

scene3d saves a copy of a scene to an R variable; rglwidget, writePLY, writeOBJ and writeSTL write the scene to a file in various other formats.

Examples

```
## Not run:
# On some systems, the program  "asy"  used
# to process the output has bugs, so this may fail.
x <- rnorm(20)
y <- rnorm(20)
z <- rnorm(20)
plot3d(x, y, z, type = "s", col = "red")
olddir <- setwd(tempdir())
writeASY(title = "interactive")  # Produces interactive.pdf
writeASY(title = "noninteractive", prc = FALSE) # Produces noninteractive.pdf
setwd(olddir)

## End(Not run)
```

---

writeOBJ *Read and write Wavefront OBJ format files*

Description

writeOBJ writes OBJ files. This is a file format that is commonly used in 3D graphics applications. It does not represent text, but does represent points, lines, polygons (and many other types that RGL doesn't support). readOBJ reads only some parts of OBJ files.

Usage

```
writeOBJ(con,
         pointRadius = 0.005, pointShape = icosahedron3d(),
         lineRadius = pointRadius, lineSides = 20,
         pointsAsPoints = FALSE, linesAsLines = FALSE,
         withNormals = TRUE, withTextures = TRUE,
         separateObjects = TRUE,
         ids = tagged3d(tags),
         tags = NULL)
readOBJ(con, ...)
```

Arguments

con	A connection or filename.
pointRadius, lineRadius	The radius of points and lines relative to the overall scale of the figure, if they are converted to polyhedra.
pointShape	A mesh shape to use for points if they are converted. It is scaled by the pointRadius.

lineSides	Lines are rendered as cylinders with this many sides.
pointsAsPoints, linesAsLines	Whether to convert points and lines to "point" and "line" records in the OBJ output.
withNormals	Whether to output vertex normals for smooth shading.
separateObjects	Whether to mark each RGL object as a separate object in the file.
withTextures	Whether to output texture coordinates.
ids	The identifiers (from ids3d) of the objects to write. If NULL, try to write everything.
tags	Alternate way to specify ids. Ignored if ids is given.
...	Additional arguments (typically just material) to pass to tmesh3d.

Details

The current writeOBJ implementation only outputs triangles, quads, planes, spheres, points, line segments, line strips and surfaces. It does not output material properties such as colors, since the OBJ format does not support the per-vertex colors that RGL uses.

The readOBJ implementation can read faces, normals, and textures coordinates, but ignores material properties (including the specification of the texture file to use). To read a file that uses a single texture, specify it in the material argument, e.g. readOBJ("model.OBJ", material = list(color = "white", texture = "texture.png")). There is no support for files that use multiple textures.

The defaults for pointsAsPoints and linesAsLines have been chosen because Blender does not import points or lines, only polygons. If you are exporting to other software you may want to change them.

If present, texture coordinates are output by default, but the textures themselves are not.

Individual RGL objects are output as separate objects in the file when separateObjects = TRUE, the default.

The output file should be readable by Blender and Meshlab; the latter can write in a number of other formats, including U3D, suitable for import into a PDF document.

Value

writeObj invisibly returns the name of the connection to which the data was written.

readObj returns a mesh object constructed from the input file.

Author(s)

Duncan Murdoch

References

The file format was found at http://www.martinreddy.net/gfx/3d/OBJ.spec on November 11, 2012.

Blender is an open source 3D object editor. See www.blender.org for details.

See Also

scene3d saves a copy of a scene to an R variable; rglwidget, writeASY, writePLY and writeSTL write the scene to a file in various other formats.

Examples

```
filename <- tempfile(fileext = ".obj")
open3d()
shade3d( icosahedron3d() )
writeOBJ(filename)

# The motivation for writing readObj() was to read a shape
# file of Comet 67P/Churyumov-Gerasimenko, from the ESA.
# The file no longer appears to be online, but may still be
# available on archive.org.  Here was the original URL:
#   cometurl <- "http://sci.esa.int/science-e/www/object/doc.cfm?fobjectid=54726"
# This code would read and display it:
#   open3d()
#   shade3d(readOBJ(url(cometurl),
#                   material = list(col = "gray")))


# Textures are used in a realistic hand image available from
# https://free3d.com/3d-model/freerealsichand-85561.html
# Thanks to Monte Shaffer for pointing this out.
# Decompress the files into the current directory, convert
# hand_mapNew.jpg to hand_mapNew.png, then use
## Not run:
open3d()
shade3d(readOBJ("hand.OBJ", material = list(color = "white",
shininess = 1, texture = "hand_mapNew.png")))

## End(Not run)
```

---

writePLY *Write Stanford PLY format files*

Description

This function writes PLY files. This is a simple file format that is commonly used in 3D printing. It does not represent text, only edges and polygons. The writePLY function does the necessary conversions.

Usage

```
writePLY(con, format = c("little_endian", "big_endian", "ascii"),
         pointRadius = 0.005, pointShape = icosahedron3d(),
         lineRadius = pointRadius, lineSides = 20,
         pointsAsEdges = FALSE, linesAsEdges = pointsAsEdges,
```

```
withColors = TRUE, withNormals = !(pointsAsEdges || linesAsEdges),
ids = tagged3d(tags), tags = NULL)
```

Arguments

con	A connection or filename.
format	Which output format. Defaults to little-endian binary.
pointRadius, lineRadius	The radius of points and lines relative to the overall scale of the figure, if they are converted to polyhedra.
pointShape	A mesh shape to use for points if they are converted. It is scaled by the pointRadius.
lineSides	Lines are rendered as cylinders with this many sides.
pointsAsEdges, linesAsEdges	Whether to convert points and lines to "Edge" records in the PLY output.
withColors	Whether to output vertex color information.
withNormals	Whether to output vertex normals for smooth shading.
ids	The identifiers (from ids3d) of the objects to write. If NULL, try to write everything.
tags	Select objects with matching tags. Ignored if ids is specified.

Details

The current implementation only outputs triangles, quads, planes, spheres, points, line segments, line strips and surfaces.

The defaults for pointsAsEdges and linesAsEdges have been chosen because Blender does not import lines, only polygons. If you are exporting to other software you may want to change them.

Since the PLY format only allows one object per file, all RGL objects are combined into a single object when output.

The output file is readable by Blender and Meshlab; the latter can write in a number of other formats, including U3D, suitable for import into a PDF document.

Value

Invisibly returns the name of the connection to which the data was written.

Author(s)

Duncan Murdoch

References

The file format was found on www.mathworks.com on November 10, 2012 at a URL that no longer exists; currently the format is described at www.mathworks.com/help/vision/ug/the-ply-format.html.

Blender is an open source 3D object editor. See www.blender.org for details.

See Also

scene3d saves a copy of a scene to an R variable; rglwidget, writeASY, writeOBJ and writeSTL write the scene to a file in various other formats.

Examples

```
filename <- tempfile(fileext = ".ply")
open3d()
shade3d( icosahedron3d(col = "magenta") )
writePLY(filename)
```

Index

∗ **FOV**
par3d, 85
∗ **activeSubscene**
par3d, 85
∗ **alpha**
material3d, 69
∗ **ambient**
material3d, 69
∗ **antialias**
par3d, 85
∗ **back**
material3d, 69
∗ **bbox**
par3d, 85
∗ **blend**
material3d, 69
∗ **cex**
par3d, 85
∗ **color**
material3d, 69
∗ **depth_mask**
material3d, 69
∗ **depth_test**
material3d, 69
∗ **dplot**
ellipse3d, 52
par3dinterp, 89
play3d, 103
spin3d, 183
∗ **dynamic**
abclines3d, 7
addNormals, 8
aspect3d, 22
axes3d, 25
bbox3d, 28
bg3d, 29
callbacks, 37
cube3d, 46
cylinder3d, 47
grid3d, 60
light, 66
material3d, 69
matrices, 74
mesh3d, 78
open3d, 83
par3d, 85
persp3d, 93
planes3d, 102
plot3d, 108
primitives, 116
r3d, 120
rgl-package, 5
rgl.bringtotop, 125
rgl.pixels, 128
rgl.postscript, 130
rgl.user2window, 134
scene, 149
select3d, 154
shade3d, 169
shapelist3d, 173
snapshot3d, 180
spheres3d, 182
sprites, 185
subdivision3d, 187
surface3d, 192
text3d, 195
viewpoint, 209
∗ **emission**
material3d, 69
∗ **family**
par3d, 85
∗ **floating**
material3d, 69
∗ **fog**
material3d, 69
∗ **fontname**
par3d, 85
∗ **font**

par3d, 85
∗ **front**
material3d, 69
∗ **graphics**
bgplot3d, 31
extrude3d, 55
identify3d, 64
mfrow3d, 80
observer3d, 82
persp3d, 93
persp3d.deldir, 96
persp3d.function, 98
persp3d.triSht, 100
polygon3d, 115
readSTL, 121
rgl.attrib, 123
scene3d, 150
selectpoints3d, 156
subscene3d, 188
subsceneInfo, 191
triangulate, 205
turn3d, 206
writeOBJ, 212
writePLY, 214
∗ **ignoreExtent**
par3d, 85
∗ **isTransparent**
material3d, 69
∗ **line_antialias**
material3d, 69
∗ **listeners**
par3d, 85
∗ **lit**
material3d, 69
∗ **lwd**
material3d, 69
∗ **margin**
material3d, 69
∗ **maxClipPlanes**
par3d, 85
∗ **modelMatrix**
par3d, 85
∗ **mouseMode**
par3d, 85
∗ **point_antialias**
material3d, 69
∗ **polygon_offset**
material3d, 69
∗ **projMatrix**
par3d, 85
∗ **rgl.warnBlackTexture**
material3d, 69
∗ **scale**
par3d, 85
∗ **shininess**
material3d, 69
∗ **size**
material3d, 69
∗ **skipRedraw**
par3d, 85
∗ **smooth**
material3d, 69
∗ **specular**
material3d, 69
∗ **tag**
material3d, 69
∗ **texenvmap**
material3d, 69
∗ **texmagfilter**
material3d, 69
∗ **texminfilter**
material3d, 69
∗ **texmipmap**
material3d, 69
∗ **texmode**
material3d, 69
∗ **texture**
material3d, 69
∗ **textype**
material3d, 69
∗ **useFreeType**
par3d, 85
∗ **userMatrix**
par3d, 85
∗ **userProjection**
par3d, 85
∗ **utilities**
rgl.Sweave, 132
setupKnitr, 160
∗ **viewport**
par3d, 85
∗ **windowRect**
par3d, 85
∗ **zoom**
par3d, 85
.check3d, 6

%>% (import), 65
%>%, *65*
3dobjects (primitives), 116

abclines3d, 7, *61*, *103*
addNormals, 8, *15*, *77*
addToSubscene3d (subscene3d), 188
ageControl, 9, *65*, *107*
all.equal, *10*, *15*, *77*
all.equal.mesh3d, 10
alphashape3d::ashape3d, *16, 17*
arc3d, 11
arrow3d, 13
arrows, *13*
as.character, *64*
as.mesh3d, 14, *17*, *19, 20*, *22*, *51*
as.mesh3d.ashape3d, 16
as.mesh3d.deldir (persp3d.deldir), 96
as.mesh3d.rglId, *14*, 18, *77*
as.mesh3d.triSht (persp3d.triSht), 100
as.raster, *72*
as.rglscene, 20
as.tmesh3d, 20
as.triangles3d, 21
as.triangles3d.rglId, *19*
asEuclidean (matrices), 74
asEuclidean2 (matrices), 74
ashape3d, *17*
asHomogeneous (matrices), 74
asHomogeneous2 (matrices), 74
aspect3d, 22, *87*, *109*, *183*
asRow, 23
attachDependencies, *68*
axes3d, 25, *28, 29*
axis, *27*
axis3d, *61*
axis3d (axes3d), 25

bbox3d, *25–27*, 28, *28*, *73*, *150*
bg3d, 29, *31, 32*, *71*, *73*, *84*, *151*
bgplot3d, *30*, 31, *180*
box, *27*
box3d (axes3d), 25
Buffer, 32, *59*

callbacks, 37
cbind, *179*
check3d (.check3d), 6
checkDeldir, 39
clear3d, *66*
clear3d (scene), 149
clearSubsceneList (mfrow3d), 80
clipMesh3d, *19*, *21*, 39, *188*
clipObj3d (clipMesh3d), 39
clipplaneControl, 43
clipplanes3d, *110*, *208*
clipplanes3d (planes3d), 102
close3d (open3d), 83
cloud, *144*
compare_proxy.mesh3d
(all.equal.mesh3d), 10
contour3d, *46*
contourLines3d, *42*, 44
createWidget, *106*, *145*
cube3d, *5*, 46, *117*, *121*
cuboctahedron3d (cube3d), 46
cur3d (open3d), 83
currentSubscene3d (subscene3d), 188
curve, *99*
cylinder3d, 47

decorate3d, 49, *94*, *109*
deform.mesh3d (subdivision3d), 187
deldir, *96, 97*
delFromSubscene3d (subscene3d), 188
dev.off, *148*
dev.set, *148*
divide.mesh3d (subdivision3d), 187
dodecahedron3d (cube3d), 46
dot3d, *121*, *174*
dot3d (shade3d), 169
drape3d, 50, *56*, *172, 173*

ellipse3d, 52, *183*
example, *162*
expect_known_scene, 54
expect_known_value, *54*
expression, *196*
extrude3d, 55, *116*, *206, 207*

facing3d, *51*, 56, *173*
figHeight (figWidth), 57
figWidth, 57
filledContour3d, *42*, *172*
filledContour3d (contourLines3d), 44

gc3d, *81*
gc3d (subscene3d), 188

getBoundary3d, 58
getr3dDefaults *(*open3d*)*, 83
getShaders *(*setUserShaders*)*, 166
getWidgetId *(*asRow*)*, 23
gltfTypes, 59
GramSchmidt, 59
grid, *61*
grid3d, 60

highlevel, *162*
highlevel *(*rglIds*)*, 138
hook_rgl, *133*
hook_rgl *(*setupKnitr*)*, 160
hook_rglchunk *(*setupKnitr*)*, 160
hook_webgl, *133*, *147*
hook_webgl *(*setupKnitr*)*, 160
hover3d, 62
htmlDependency, *68*

icosahedron3d *(*cube3d*)*, 46
identify, *64*
identify3d, *63*, 64
identityMatrix *(*matrices*)*, 74
ids3d, *122–124*, *186*, *213*, *215*
ids3d *(*scene*)*, 149
import, 65
in_pkgdown *(*in_pkgdown_example*)*, 65
in_pkgdown_example, 65
invisible, *160*

kde2d, *112*

latex3d *(*plotmath3d*)*, 113
latexGrob, *114*
layout, *80, 81*
layout3d, *190*
layout3d *(*mfrow3d*)*, 80
legend, *31*
legend3d *(*bgplot3d*)*, 31
light, 66
light3d, *73*, *150*
light3d *(*light*)*, 66
lines3d, *51*, *115*, *121*
lines3d *(*primitives*)*, 116
local_edition, *54*
locator, *155*
lowlevel, *63*, *162*
lowlevel *(*rglIds*)*, 138

magick, *104*
makeDependency, 67
material3d, *17*, *25*, *28–30*, *66*, 69, *84*, *102*, *109*, *114*, *117*, *151*, *170*, *182, 183*, *186*, *193*, *196, 197*
matrices, 74, *88*
merge.mesh3d, 76
mergeVertices, 77
mesh3d, *11*, *14–16*, *40*, *45*, *47*, *53*, *58*, *74*, *77*, 78, *97*, *101*, *117*, *120, 121*, *171*, *174*, *188*
mfrow3d, 80, *190*
movie3d, *181*
movie3d *(*play3d*)*, 103
mtext, *27*
mtext3d, *71*, *179*
mtext3d *(*axes3d*)*, 25

newSubscene3d, *80, 81*, *192*
newSubscene3d *(*subscene3d*)*, 188
next3d *(*mfrow3d*)*, 80
normalize.mesh3d *(*subdivision3d*)*, 187

observer3d, 82, *87*
octahedron3d *(*cube3d*)*, 46
oh3d *(*cube3d*)*, 46
open3d, *5, 6*, 83, *88, 89*, *110*, *120*, *134*, *150*
options, *160*

par, *80, 81*
par3d, *5*, *23*, *38*, *63*, *70*, *75*, *80*, *82, 83*, 85, *90*, *104*, *109, 110*, *135*, *151, 152*, *171*, *190*, *196*, *200*, *210*
par3dinterp, 89, *91*, *105*, *200*
par3dinterpControl, 91
par3dsave *(*tkpar3dsave*)*, 199
particles3d *(*sprites*)*, 185
pch3d, 92, *186*
persp, *94, 95*, *144*, *193*
persp3d, *17*, *71*, 93, *99*, *110*, *138*, *193*
persp3d.deldir, *95*, 96, *111*
persp3d.formula, *95*
persp3d.formula *(*plot3d.formula*)*, 110
persp3d.function, *95*, 98
persp3d.triSht, 100
pipe, *24*
pipe *(*import*)*, 65
planes3d, *7*, 102, *110*, *112*, *208*
play3d, *90*, 103, *183, 184*, *200*

playwidget, *9*, *43*, *65*, *91*, 105, *107*, *119*, *145*, *147*, *154*, *175*, *204*, *207*
playwidgetOutput *(*shiny*)*, 174
plot.default, *110*
plot3d, *5*, *17*, *23*, *84*, *93–95*, 108, *110*, *112*, *138*, *151*
plot3d.default, *111*
plot3d.deldir, *110*
plot3d.deldir *(*persp3d.deldir*)*, 96
plot3d.formula, 110
plot3d.function, *110*
plot3d.function *(*persp3d.function*)*, 98
plot3d.lm, 111
plot3d.rglobject *(*scene3d*)*, 150
plot3d.rglscene *(*scene3d*)*, 150
plot3d.triSht *(*persp3d.triSht*)*, 100
plotmath, *114*
plotmath3d, 113, *195*, *196*
plotrix, *198*
png, *31*, *179*
points, *92*, *93*
points3d, *11*, *79*, *93*, *121*
points3d *(*primitives*)*, 116
poly, *113*
polygon3d, *13*, *55*, 115, *206*
pop3d, *29*, *66*, *117*
pop3d *(*scene*)*, 149
pretty, *26*, *28*, *61*
primitives, 116, *171*
print, *160*
print.rglId *(*rglIds*)*, 138
print.rglobject *(*scene3d*)*, 150
print.rglscene *(*scene3d*)*, 150
projectDown *(*facing3d*)*, 56
propertyControl, *43*, *91*, *106*, *107*, 118

qmesh3d, *15*, *53*
qmesh3d *(*mesh3d*)*, 78
quads3d, *121*
quads3d *(*primitives*)*, 116
quartz, *160*
quote, *196*

r3d, *5*, 120, *188*, *196*
r3dDefaults, *30*, *70*, *84*, *88*, *138*, *150*, *196*
r3dDefaults *(*open3d*)*, 83
rainbow, *99*
readOBJ *(*writeOBJ*)*, 212
readSTL, 121
registerSceneChange *(*sceneChange*)*, 153
renderPlaywidget *(*shiny*)*, 174
renderRglwidget *(*shiny*)*, 174
RGL *(*rgl-package*)*, 5
rgl, *121*, *150*
rgl *(*rgl-package*)*, 5
rgl-package, 5
rgl.attrib, 123, *125*, *152*, *156*
rgl.attrib.count *(*rgl.attrib.info*)*, 124
rgl.attrib.info, 124, *124*
rgl.bringtotop, *104*, 125, *180*, *181*
rgl.dev.list *(*open3d*)*, 83
rgl.getAxisCallback, 126
rgl.getMouseCallbacks *(*callbacks*)*, 37
rgl.getWheelCallback *(*callbacks*)*, 37
rgl.incrementID, 127
rgl.init, 127
rgl.material.names *(*material3d*)*, 69
rgl.material.readonly *(*material3d*)*, 69
rgl.open, *120*, *134*
rgl.par3d.names *(*par3d*)*, 85
rgl.par3d.readonly *(*par3d*)*, 85
rgl.pixels, 128
rgl.postscript, 130, *133*
rgl.printRglwidget *(*rglwidget*)*, 144
rgl.projection, *154*
rgl.projection *(*rgl.user2window*)*, 134
rgl.quit *(*open3d*)*, 83
rgl.select, 131
rgl.setMouseCallbacks, *86*, *87*, *163*
rgl.setMouseCallbacks *(*callbacks*)*, 37
rgl.setWheelCallback, *87*, *163*
rgl.setWheelCallback *(*callbacks*)*, 37
rgl.snapshot, *104*, *129*
rgl.snapshot *(*snapshot3d*)*, 180
rgl.Sweave, 132
rgl.useNULL, *5*, *84*, *128*, 133
rgl.user2window, 134
rgl.window2user *(*rgl.user2window*)*, 134
RGL_DEBUGGING *(*makeDependency*)*, 67
RGL_SLOW_DEV *(*setGraphicsDelay*)*, 159
RGL_USE_NULL *(*rgl.useNULL*)*, 133
RGL_USE_WEBSHOT *(*snapshot3d*)*, 180
rglExtrafonts, 136, *137*, *196*
rglFonts, 137, *137*, *196*
rglHighlevel *(*rglIds*)*, 138
rglId *(*rglIds*)*, 138
rglIds, 138

rglLowlevel *(rglIds)*, 138
rglMouse, 140
rglobject-class *(scene3d)*, 150
rglscene-class *(scene3d)*, 150
rglShared, *119*, 141, *145, 146*
rglToBase *(rglToLattice)*, 143
rglToLattice, 143
rglwidget, *38*, *57*, *63*, *65*, *68*, *84*, *107*, *122*, *126*, *128*, *136*, *139–141*, 144, *152*, *157, 158*, *162*, *168*, *175–177*, *180*, *204*, *212*, *214*, *216*
rglwidgetOutput, *147*
rglwidgetOutput *(shiny)*, 174
rotate3d, *55*, *121*, *174*
rotate3d *(matrices)*, 74
rotationMatrix *(matrices)*, 74
RweaveLatex, *133*

safe.dev.off, 148
saveWidget, *146*
scale3d, *174*
scale3d *(matrices)*, 74
scaleMatrix *(matrices)*, 74
scene, 149
scene3d, *20*, *54*, *122*, *145*, 150, *153*, *167*, *180*, *212*, *214*, *216*
sceneChange, 153
segments3d, *7*, *13*, *44*, *51*, *121*
segments3d *(primitives)*, 116
select3d, *64*, *86*, *121*, *132*, *135*, 154, *156, 157*
selectionFunction3d, *147*
selectionFunction3d *(select3d)*, 154
selectpoints3d, *63*, *155*, 156
set3d, *5*
set3d *(open3d)*, 83
setAxisCallbacks, *27*, *126*, 157, *163*
setGraphicsDelay, 159
setupKnitr, *139*, *146*, 160
setUserCallbacks, *38*, *62, 63*, *158*, 162
setUserShaders, 166
shade3d, *13, 14*, *41*, *44*, *79*, *115*, *121*, 169, *174*
shadow3d, *51*, *56*, 172
shape3d *(mesh3d)*, 78
shapelist3d, *79*, *171*, 173
shiny, 174
shiny::session$sendCustomMessage, *153*
shinyGetPar3d, 176
shinyResetBrush *(shinyGetPar3d)*, 176
shinySetPar3d *(shinyGetPar3d)*, 176
show2d, 178
sliderInput, *43*, *106*, *207*
snapshot3d, *104*, *130*, 180
spheres3d, *121*, 182, *186*
spin3d, *104, 105*, 183, *203*
sprintf, *104*
sprites, 185
sprites3d, *93*, *121*
sprites3d *(sprites)*, 185
subdivision3d, *53*, *121*, 187
subscene3d, *103*, 188
subsceneInfo, *152*, *190*, 191
subsceneList *(mfrow3d)*, 80
subsetControl, *107*
subsetControl *(propertyControl)*, 118
subsetSetter, *119*
surface3d, *94, 95*, *112*, 192
Sweave, *132*
Sweave.snapshot *(rgl.Sweave)*, 132
Sys.sleep, *104*, *133*

tagged3d, 194
tagList, *107*, *140*
tags, *107*
terrain3d, *121*
test_that, *54*
tetrahedron3d *(cube3d)*, 46
text, *114*
text3d, *27*, *62*, *64*, *86*, *93*, *114, 115*, *121*, *137, 138*, *185, 186*, 195, *198, 199*
texts3d *(text3d)*, 195
textureSource, 197
thigmophobe, *198*
thigmophobe3d, 198
title, *27*
title3d *(axes3d)*, 25
tkframe, *202*
tkpar3dsave, *90*, 199, *201*
tkrgl, 201
tkspin3d, 201, *201*
tkspinControl, *201*, 202, *202*
tktoplevel, *200*, *202*
tmesh3d, *15, 16*, *21*, *55*, *97*, *101*, *206*, *213*
tmesh3d *(mesh3d)*, 78
toggleButton, *204*
toggleWidget, *107*, *145*, 203
transform3d, *121*
transform3d *(matrices)*, 74
translate3d, *174*

translate3d (matrices), 74
translationMatrix (matrices), 74
tri.mesh, *100, 101*
triangles3d, *21*, *103*, *121, 122*
triangles3d (primitives), 116
triangulate, *55*, *115, 116*, 205
turn3d, *55*, 206

useSubscene3d, *5*
useSubscene3d (subscene3d), 188

vertexControl, *107*, 207
view3d, *89*, *130*, *181*
view3d (viewpoint), 209
viewpoint, 209

wire3d, *121*, *174*
wire3d (shade3d), 169
wireframe, *144*
writeASY, *122*, 210, *214*, *216*
writeOBJ, *122*, *151, 152*, 212, *212*, *216*
writePLY, *122*, *151, 152*, *212*, 214, *214*
writeSTL, *122*, *151, 152*, *212*, *214*, *216*
writeSTL (readSTL), 121
writeWebGL, *151*

xy.coords, *117*, *206*
xyz.coords, *7*, *14, 15*, *48*, *51*, *62*, *64*, *66*, *78*, *92*, *102*, *109*, *111*, *114, 115*, *117*, *134*, *155*, *174*, *182*, *185*, *195*, *198*, *205*