Package ‘rgl’

July 8, 2019

Version 0.100.26

Title 3D Visualization Using OpenGL

Author Daniel Adler <dadler@uni-goettingen.de>, Duncan Murdoch <murdoch@stats.uwo.ca>, and others (see README)

Maintainer Duncan Murdoch <murdoch@stats.uwo.ca>

Depends R (>= 3.2.0)

Suggests MASS, rmarkdown, deldir, orientlib, lattice, misc3d, rstudioapi, magick, plotrix (>= 3.7-3), tripack, interp, alphashape3d

Imports graphics, grDevices, stats, utils, htmlwidgets, htmltools, knitr, jsonlite (>= 0.9.20), shiny, magrittr, crosstalk, manipulateWidget (>= 0.9.0)

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://r-forge.r-project.org/projects/rgl/

SystemRequirements OpenGL, GLU Library, XQuartz (on OSX), zlib (optional), libpng (>=1.2.9, optional), FreeType (optional), pandoc (>=1.14, needed for vignettes)

BugReports https://r-forge.r-project.org/projects/rgl/

VignetteBuilder knitr

NeedsCompilation yes

Repository CRAN

Date/Publication 2019-07-08 18:51:03 UTC
R topics documented:

- rgl-package
- .check3d
- abclines3d
- addNormals
- ageControl
- ageSetter
- arc3d
- arrow3d
- as.mesh3d
- as.mesh3d.ashape3d
- as.mesh3d.rglId
- as.triangles3d
- aspect3d
- asRow
- axes3d
- bg3d
- bgplot3d
- clipMesh3d
- clipplaneControl
- cylinder3d
- elementId2Prefix
- ellipse3d
- extrude3d
- figWidth
- GramSchmidt
- grid3d
- hook_rgl
- identify3d
- import
- light
- matrices
- mergeVertices
- mesh3d
- mfrow3d
- observer3d
- par3d
- par3dinterp
- par3dinterpControl
- pch3d
- persp3d
- persp3d.deldir
- persp3d.function
- persp3d.triSht
- planes3d
- play3d
- playwidget
### R topics documented:

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>plot3d</td>
<td>75</td>
</tr>
<tr>
<td>plot3d.formula</td>
<td>77</td>
</tr>
<tr>
<td>plot3d.lm</td>
<td>78</td>
</tr>
<tr>
<td>plotmath3d</td>
<td>79</td>
</tr>
<tr>
<td>points3d</td>
<td>80</td>
</tr>
<tr>
<td>polygon3d</td>
<td>82</td>
</tr>
<tr>
<td>propertyControl</td>
<td>83</td>
</tr>
<tr>
<td>propertySetter</td>
<td>85</td>
</tr>
<tr>
<td>r3d</td>
<td>86</td>
</tr>
<tr>
<td>readSTL</td>
<td>89</td>
</tr>
<tr>
<td>rgl.attrib</td>
<td>91</td>
</tr>
<tr>
<td>rgl.attrib.info</td>
<td>92</td>
</tr>
<tr>
<td>rgl.bbox</td>
<td>94</td>
</tr>
<tr>
<td>rgl.bringtotop</td>
<td>95</td>
</tr>
<tr>
<td>rgl.material</td>
<td>97</td>
</tr>
<tr>
<td>rgl.open</td>
<td>97</td>
</tr>
<tr>
<td>rgl.pixels</td>
<td>101</td>
</tr>
<tr>
<td>rgl.postscript</td>
<td>102</td>
</tr>
<tr>
<td>rgl.primitive</td>
<td>103</td>
</tr>
<tr>
<td>rgl.select</td>
<td>105</td>
</tr>
<tr>
<td>rgl.setMouseCallbacks</td>
<td>106</td>
</tr>
<tr>
<td>rgl.snapshot</td>
<td>107</td>
</tr>
<tr>
<td>rglToLattice</td>
<td>108</td>
</tr>
<tr>
<td>rgl.surface</td>
<td>110</td>
</tr>
<tr>
<td>rgl.Sweave</td>
<td>111</td>
</tr>
<tr>
<td>rgl.useNULL</td>
<td>113</td>
</tr>
<tr>
<td>rgl.user2window</td>
<td>114</td>
</tr>
<tr>
<td>rglIds</td>
<td>115</td>
</tr>
<tr>
<td>rglMouseMove</td>
<td>116</td>
</tr>
<tr>
<td>rglShared</td>
<td>118</td>
</tr>
<tr>
<td>select3d</td>
<td>119</td>
</tr>
<tr>
<td>show2d</td>
<td>120</td>
</tr>
<tr>
<td>scene</td>
<td>123</td>
</tr>
<tr>
<td>scene3d</td>
<td>124</td>
</tr>
<tr>
<td>sceneChange</td>
<td>126</td>
</tr>
<tr>
<td>select3d</td>
<td>128</td>
</tr>
<tr>
<td>selectpoints3d</td>
<td>129</td>
</tr>
<tr>
<td>setUserShaders</td>
<td>131</td>
</tr>
<tr>
<td>shapelist3d</td>
<td>133</td>
</tr>
<tr>
<td>shiny</td>
<td>135</td>
</tr>
<tr>
<td>shinyGetPar3d</td>
<td>136</td>
</tr>
<tr>
<td>show3d</td>
<td>138</td>
</tr>
<tr>
<td>spheres3d</td>
<td>140</td>
</tr>
<tr>
<td>spin3d</td>
<td>141</td>
</tr>
<tr>
<td>sprites</td>
<td>142</td>
</tr>
<tr>
<td>subdivision3d</td>
<td>144</td>
</tr>
<tr>
<td>subscene3d</td>
<td>145</td>
</tr>
<tr>
<td>subsceneInfo</td>
<td>148</td>
</tr>
<tr>
<td>surface3d</td>
<td>149</td>
</tr>
</tbody>
</table>
rgl-package

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 `rgl.set()` 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.

`rgl` also includes a lower level interface which is described in the `rgl.open` help topic. We recommend that you avoid mixing `rgl.*` and `*3d` calls.

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

file.show(system.file("NEWS", package = "rgl"))
exmaple(surface3d)
exmaple(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

rl1.dev.list()
.check3d()
rl1.dev.list()
.check3d()
rl1.dev.list()
rl1.dev.list()
rl1.close()
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**

```r
gl.abclines(x, y = NULL, z = NULL, a, b = NULL, c = NULL, ...)  
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**

These functions draw the segment of a line that intersects the current bounding box of the scene using the parametrization $(x, y, z) + (a, b, c) \cdot s$ 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`, `rgl.planes` for mathematical planes.
- `segments3d` draws sections of lines that do not adapt to the bounding box.

**Examples**

```r
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, ...)

Arguments

x An object to which to add normals.

... Additional parameters which will be passed to the methods. Currently unused.

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

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, 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.
age Chosen ages at which the following attributes will apply.
objids Object ids to which the changes apply.
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

Examples

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

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

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, 1, 0, 1)),
    start = 0, stop = 4*pi,
    step = 0.1, rate = 4)
if (interactive())
    widget
    options(saveopts)
```

Description

Many `rgl` shapes contain lists of vertices with various attributes (available via `rgl.attrib`). This
function modifies the data for those attributes in a WebGL scene.

Usage

```r
ageSetter(births, ages,
    colors = NULL, alpha = NULL, radii = NULL,
    vertices = NULL, normals = NULL, origins = NULL,
    texcoords = NULL,
    objids, prefixes = "", digits = 7,
    param = seq(floor(min(births)), ceiling(max(births))))
```

Arguments

- **births** Numeric vector with one value per vertex, used to determine the “age” of the
  vertex when displaying it.
- **ages** A non-decreasing sequence of “ages”.
- **colors, alpha, radii, vertices, normals, origins, texcoords**
  Attributes of the vertices. Non-NULL attributes will be interpolated from these
  values. See the Details section below.
The object ids and scene prefixes to modify. These are recycled to the same length.

- **digits**: How many digits to output in the generated Javascript code.
- **param**: Default values to be used by a slider control calling the generated function.

**Details**

The vertex attributes are specified as follows:

- **colors**: A vector of colors in a format suitable for input to `col2rgb`.
- **alpha**: A numeric vector of alpha values between 0 and 1.
- **radii**: A numeric vector of sphere radii.
- **vertices**: A 3-column matrix of vertex coordinates.
- **normals**: A 3-column matrix of vertex normals.
- **origins**: A 2-column matrix of origins for text or sprites.
- **texcoords**: A 2-column matrix of texture coordinates.

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 listed here; if one is chosen that is not a property of the corresponding object, a Javascript `alert()` will be generated.

**Value**

A character vector of class `c("ageSetter", "propertySetter")` containing Javascript code defining a function suitable for use in a `propertySlider`.

The function takes a single argument, `time`, and uses it to compute the “age” of vertex `i` as `time - births[i]`. Those are then used with the `ages` argument to linearly interpolate settings of the specified attributes. Extrapolation is constant. Repeated values in `ages` can be used to obtain discontinuities in the settings.

**Author(s)**

Duncan Murdoch

**See Also**

`propertySlider`; more detailed control is available in `vertexSetter`.

**Examples**

```r
propertySlider(ageSetter(births = 1:10, ages = c(-1, 0, 1),
               alpha = c(0, 1, 1), objids = 123))
```
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.

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

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

arrow3d

Draw an arrow in a scene.

Description

Draws various types of arrows in a scene.

Usage

```r
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

```r
exyz <- 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")
```

Description

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

The default method works 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 according to the `triangles` 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

```r
as.mesh3d(x, ...)  
## Default S3 method:  
as.mesh3d(x, y = NULL, z = NULL,
  triangles = length(x) %% 3 == 0,
  smooth = FALSE,
  tolerance = sqrt(.Machine$double.eps),
  notEqual = NULL,
  merge = TRUE,
  ...
```

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.

- **triangles** Logical value indicating whether the coordinates are for triangles or quadrilaterals.

- **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`.

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

```r
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, triangles = FALSE, col = "red")
mesh$vb
mesh$ib
open3d()
shade3d(mesh)
```
Convert alpha-shape surface of a cloud of points to mesh3d object.

Description

The \texttt{alphashape3d::ashape3d} function computes the 3D $\alpha$-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 \texttt{rgl} \texttt{tmesh3d} object.

Usage

\begin{verbatim}
## S3 method for class 'ashape3d'
as.mesh3d(x,
  alpha = x$alpha[1],
  col = "gray",
  smooth = FALSE, normals = NULL,
  texcoords = NULL, ...)
\end{verbatim}

Arguments

- \texttt{x} An object of class "ashape3d".
- \texttt{alpha} Which alpha value stored in \texttt{x} should be converted?
- \texttt{col} The surface colour.
- \texttt{smooth} Whether to attempt to add normals to make the surface look smooth. See the Details below.
- \texttt{normals, texcoords} Normals and texture coordinates at each vertex can be specified.
- \texttt{...} Additional arguments to pass to use as \texttt{material3d} properties on the resulting mesh.

Details

Edelsbrunner and Mucke’s (1994) $\alpha$-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 \texttt{smooth} = \texttt{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**


**Examples**

```r
if (requireNamespace("alphashape3d", quietly = TRUE)) {
  set.seed(123)
  xyz <- rbind(cbind(runif(1000), runif(1000), runif(1000)),
               cbind(runif(125, 1, 1.5),
               runif(125, 0.25, 0.75),
               runif(125, 0.25, 0.75)))
  ash <- suppressMessages(alphashape3d::asshape3d(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 mesh3d object.

**Description**

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

```r
## 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 only handle objects of types from `c("triangles", "quads", "planes", "surface")`. Since this method only produces triangular meshes, 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 triangular 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

```r
# 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)

colorlut <- terrain.colors(zlen)  # height color lookup table
```
as.triangles3d

```r
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()
rgl.close()

open3d()
shade3d(m)
```

### Description

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

### Usage

```r
as.triangles3d(obj, ...)
```

### 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 $n \times 3$ 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
aspect3d

See Also

\texttt{as.mesh3d} to also capture material properties.

Examples

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

Description

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

Usage

```r
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 \texttt{aspect3d} has been called.) Partial matches to "iso" are allowed. \texttt{aspect3d} works by modifying \texttt{par3d("scale").}

Value

The previous value of the scale is returned invisibly.

Author(s)

Duncan Murdoch

See Also

\texttt{plot3d, par3d}
asRow

Examples

```r
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()
```

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

```r
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

`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.

Examples

library(crosstalk)
sd <- SharedDataFrame$new(mtcars)
ids <- plot3d(sd$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())
  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, ...)

axis3d(edge, at = NULL, labels = TRUE, tick = TRUE, line = 0,
  pos = NULL, nticks = 5, ...)

mtext3d(text, edge, line = 0, at = NULL, pos = NA, ...)
**Axes3d**

**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**
  - the “line” of the plot margin to draw the label on
- **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
- **...**
  - 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 default to ‘-‘. 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**.

If **pos** 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. See the examples.

**Value**

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

**Author(s)**

Duncan Murdoch

**See Also**

Classic graphics functions **axis**, **box**, **title**, **mtext**, and **rgl** function **bbox3d**.
Examples

```r
code
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')

gl().bringtototop()

code
open3d()
points3d(rnorm(10), rnorm(10), rnorm(10))

# Use fixed axes
axes3d(c('x', 'y', 'z'))

# Put 4 x-axes on the plot
axes3d(c('x-', 'x+', 'x--', 'x++'))
axis3d('x', pos = c(NA, 0, 0))
title3d('main', 'sub', 'xlab', 'ylab', 'zlab')
```

---

**bg3d**

Set up Background

**Description**

Set up the background of the scene.

**Usage**

```r
code
bg3d(...)

gl().bg( sphere = FALSE, fogtype = "none", color = c("black", "white"), back = "lines", ...)
```

**Arguments**

- **fogtype**
  - fog type:
    - "none" no fog
    - "linear" linear fog function
"exp" exponential fog function
"exp2" squared exponential fog function

sphere logical, if true, an environmental sphere geometry is used for the background decoration.
color Primary color is used for background clearing and as fog color. Secondary color is used for background sphere geometry. See material3d for details.
back Specifies the fill style of the sphere geometry. See material3d for details.

Material properties. See material3d for details.

Details
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.

Note
The writeWebGL function only supports solid color backgrounds.

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")
Description

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

Usage

```r
bgplot3d(expression)
legend3d(...)```

Arguments

- `expression`: Any plotting commands to produce a plot.
- `...`: Arguments to pass to the `legend` function.

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

```r
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))
Sys.sleep(0.1) # Allow sluggish window managers to catch up
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)
```

---

**clipMesh3d**

*Clip mesh to general region*

**Description**

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

**Usage**

```r
clipMesh3d(mesh, fn, bound = 0, greater = TRUE, attribute = "vertices")
```

**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 in the mesh.
- `bound`: The value(s) of `fn` on the clipping boundary.
- `greater`: Logical; whether to keep `fn` >= `bound` or not.
- `attribute`: Which attribute(s) to pass to `fn`? Possible values are c("vertices", "normals", "texcoords", "index").

**Details**

This function transforms a mesh3d object.

First, all quads are converted to triangles. Next, each vertex is checked against the condition. If `fn` is a numeric vector, with one value per vertex, those values will be used in the test. If it is a function, it will be passed a matrix, whose columns
are the specified attribute(s), with one row per vertex. It should return a vector of values, one per
vertex, to check against the bound. The "vertices" and "normals" values will be converted to
Euclidean coordinates. "index" will be an integer from 1 to the number of vertices.

Modifications to the 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.

Value

A 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.

Author(s)

Duncan Murdoch

References

See https://stackoverflow.com/q/56242470/2554330 for the motivating example.

Examples

```r
if (requireNamespace("misc3d")) {
  # Togliatti surface equation: f(x,y,z) = 0
  # Due to Stephane Laurent
  f <- function(x,y,z){
    w <- 1
    64*(x-w)*
    (x^4-4*x^3*w-10*x^2*y^2-4*x^2*w^2+16*x*w^3-20*x*y^2*w+5*y^4+16*w^4-4-20*y^2*w^2) -
    5*(5-sqrt(5))*(2*z-sqrt(5))^2*(4*(x^2+y^2-z^2)+(1+3*sqrt(5))*w^2))^2
  }

  # make grid
  # The original had RRP instead of RPL this is coarse to be quicker
  nx <- 20; ny <- 20; nz <- 20
  x <- seq(-5, 5, length=nx)
  y <- seq(-5, 5, length=ny)
  z <- seq(-4, 4, length=nz)
  g <- expand.grid(x=x, y=y, z=z)
  # calculate voxel
  voxel <- array(with(g, f(x,y,z)), dim = c(nx,ny,nz))

  # compute isosurface
  open3d(useNULL = TRUE)
}
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, ...)

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.
... 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
Examples

```r
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())
  widget
options(saveopts)
```

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

```r
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".
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 approx-
imations. 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 envi-
ronment of the function.

Author(s)
Duncan Murdoch

References
ACM Transactions on Graphics, Vol. 27, No. 1, Article 2.

Examples
# A trefoil knot
open3d()
theta <- seq(0, 2*pi, len = 25)
knot <- cylinder3d(
  center = cbind(
    sin(theta) + 2*sin(2*theta),
    cos(theta) + 2*cos(2*theta),
    theta
  ),
  radius = 0.1,
  twist = 0,
  closed = 2,
  section = NULL,
  e1 = NULL,
  e2 = NULL,
  e3 = NULL
)
\[
2 \sin(3 \theta), \\
\cos(\theta) - 2 \cos(2 \theta), \\
e_1 = \text{cbind(} \\
\cos(\theta) + 4 \cos(2 \theta), \\
6 \cos(3 \theta), \\
\sin(\theta) + 4 \sin(2 \theta), \\
\text{radius} = 0.8, \\
\text{closed} = \text{TRUE})
\]

\text{shade3d}(\text{addNormals}(\text{subdivision3d}(\text{knot, depth} = 2)), \text{col} = \text{"green"})

---

**elementId2Prefix**

*Use widget with old-style controls.*

---

**Description**

The `rglwidget` control is designed to work in the `htmlwidgets` framework. Older `rgl` web pages that used `writeWebGL` or `knitr` used a different method of linking the controls to the scene. This is a partial bridge between the two systems.

**Usage**

```r
elementId2Prefix(elementId, prefix = elementId)
```

**Arguments**

- `elementId` An element identifier from a `rglwidget` call.
- `prefix` The prefix to use in the old-style control.

**Details**

Because of the order of initialization, this isn’t a perfect bridge. The old-style control will not set the scene to the initial value from the control, but subsequent changes to the control should be passed to the widget.

**Value**

This function generates Javascript code, so it should be used in an `results = "asis"` block in a `knitr` document.

**Author(s)**

Duncan Murdoch
ellipse3d

Examples

```r
## Not run:
grlwidget(elementId = "thewidget")
  elementId2Prefix("thewidget", "theprefix")
subsetSlider(subsets = as.list(1:5),
  prefixes = "theprefix", subscenes = 1)

## End(Not run)
```

**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**

```r
ellipse3d(x, ...)
```

## Default S3 method:

```r
elementId2Prefix("thewidget", "theprefix")
subsetSlider(subsets = as.list(1:5),
  prefixes = "theprefix", subscenes = 1)
```

## S3 method for class 'lm'

```r
elementId2Prefix("thewidget", "theprefix")
subsetSlider(subsets = as.list(1:5),
  prefixes = "theprefix", subscenes = 1)
```

## S3 method for class 'g1lm'

```r
elementId2Prefix("thewidget", "theprefix")
subsetSlider(subsets = as.list(1:5),
  prefixes = "theprefix", subscenes = 1)
```

## S3 method for class 'n1s'

```r
elementId2Prefix("thewidget", "theprefix")
subsetSlider(subsets = as.list(1:5),
  prefixes = "theprefix", subscenes = 1)
```

## 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.
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.

This parameter selects which variables from the object will be plotted. The default is the first 3.

This controls the number of subdivisions (see subdivision3d) used in constructing the ellipsoid. Higher numbers give a smoother shape.

If TRUE, smooth interpolation of normals is used; if FALSE, a faceted ellipsoid will be displayed.

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

```r
# 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")) {
  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)
```

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**

\[ \text{extrude3d}(x, y = \text{NULL}, \text{thickness} = 1, \text{smooth} = \text{FALSE}, ...) \]

**Arguments**

- \( x, y \)  
  A polygon description in one of the forms supported by `triangulate`.

- \( \text{thickness} \)  
  The extrusion will have this thickness.

- \( \text{smooth} \)  
  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 \( z = \text{thickness} \). 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**

\[
\begin{align*}
\text{x} & \leftarrow \text{c}(1:10, 10:1) \\
\text{y} & \leftarrow \text{rev(c(rep(c(0, 2), 5), rep(c(1.5, -0.5), 5)))} \\
\text{plot(x, y, type = "n")} \\
\text{polygon(x, y)} \\
\text{open3d()} \\
\text{shade3d( extrude3d(x, y), col = "red" )}
\end{align*}
\]
**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**

```r
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**

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

---

**GramSchmidt**

*The Gram-Schmidt algorithm*

**Description**

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

**Usage**

```r
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(1, 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 n = 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"))
Hook functions to use with knitr

Description

These functions allow rgl graphics to be embedded in knitr documents, 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).

Usage

setupKnitr()

hook_rgl(before, options, envir)

hook_webgl(before, options, envir)

Arguments

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 and to initialize hook_webgl.

The following chunk options are supported:

- rgl.keepopen: no longer used. Ignored with a warning.
- rgl.newwindow (default FALSE): Whether to open a new window for the display.
- rgl.margin (default 100): number of pixels by which to indent the WebGL window.
- dpi, fig.retina, fig.width, fig.height: standard knitr chunk options used to set the size of the output.
- dev: used by hook_rgl to set the output format. May be "eps", "postscript", "pdf" or "png" (default: "png").

Value

A string to be embedded into the output, or NULL if called when no output is available.

Author(s)

Originally by Yihui Xie in the knitr package; modified by Duncan Murdoch.

See Also

rgl.Sweave embeds fixed images in Sweave documents.
identify3d

Identify points in plot.

Description

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

Usage

```r
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.

light

add light source

Description

add a light source to the scene.

Usage

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

Arguments

theta, phi      polar coordinates, used by default
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
x, y, z      cartesian coordinates, optional
...         generic arguments passed through to RGL-specific (or other) functions

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.
Value

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

See Also

`rgl.clear` `rgl.pop`

Examples

```r
# 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)


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")
  rgl.pop(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)
```

---

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

```r
identityMatrix()
scaleMatrix(x, y, z)
translationMatrix(x, y, z)
rotationMatrix(angle, x, y, z, matrix)
asHomogeneous(x)
asEuclidean(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.

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
asEuclidean(y)

# or more simply...
rotate3d(c(2, 0, 0), pi/4, 0, 1, 0)
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

```r
(mesh1 <- cube3d())
id <- shade3d(mesh1, col = rainbow(6), meshColor = "face")
(mesh2 <- as.mesh3d(id))
(mesh3 <- mergeVertices(mesh2))
```

mesh3d 3D Mesh objects

Description

3D triangle and quadrangle mesh object creation and a collection of sample objects.
Usage

qmesh3d(verticies, indices, homogeneous = TRUE, material = NULL,
   normals = NULL, texcoords = NULL)

tmesh3d(verticies, indices, homogeneous = TRUE, material = NULL,
   normals = NULL, texcoords = NULL)

cube3d(trans = identityMatrix(), ...)

tetrahedron3d(trans = identityMatrix(), ...)

octahedron3d(trans = identityMatrix(), ...)

cuboctahedron3d(trans = identityMatrix(), ...)

dodecahedron3d(trans = identityMatrix(), ...)

oxahedron3d(trans = identityMatrix(), ...)

dot3d(x, ...) # draw dots at the vertices of an object
   ## S3 method for class 'mesh3d'

dot3d(x, override = TRUE, ...)

ewire3d(x, ...) # draw a wireframe object
   ## S3 method for class 'mesh3d'

ewire3d(x, override = TRUE, meshColor = c("vertices", "edges", "faces", "legacy"), ...)

shade3d(x, ...) # draw a shaded object
   ## S3 method for class 'mesh3d'

shade3d(x, override = TRUE, meshColor = c("vertices", "faces", "legacy"), ...)

Arguments

- **x**: a mesh3d object (class qmesh3d or tmesh3d).
- **verticies**: 3- or 4-component vector of coordinates
- **indices**: 4-component vector of vertex indices
- **homogeneous**: logical indicating if homogeneous (four component) coordinates are used.
- **material**: material properties for later rendering
- **normals**: normals at each vertex
- **texcoords**: texture coordinates at each vertex
- **trans**: transformation to apply to objects; see below for defaults
- **...**: additional rendering parameters
- **override**: should the parameters specified here override those stored in the object?
- **meshColor**: how should colours be interpreted? See details below

Details

These functions create and work with mesh3d objects, which consist of a matrix of vertex coordinates together with a matrix of indices indicating which vertex is part of which face. Such objects may have triangular faces, planar quadrilateral faces, or both.
The sample objects optionally take a matrix transformation `trans` as an argument. This transformation is applied to all vertices of the default shape. The default is an identity transformation.

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 3 or 4 by `n` matrix. Meshes with triangular faces will contain `i`, a `3 * n` matrix giving the indices of the vertices in each face. Quad meshes will have vertex indices in `ib`, a `4 * n` matrix. Individual meshes may have both types of faces.

The `meshColor` argument to `wire3d` and `shade3d` controls how material colours are interpreted. This parameter was added in `rgl` version 0.100.1. Possible values are:

"vertices" Colours are applied by vertex, in the order they appear in the `vb` matrix.
"edges" Colours are applied to each edge: first to the 3 edges of each triangle in the `i` matrix, then the 4 edges of each quad in the `ib` matrix.
"faces" Colours are applied to each face: first to the triangles in the `i` matrix, then to the quads in the `ib` matrix.
"legacy" Colours 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, currently a warning will by default be given that their interpretation may have changed. This warning can be suppressed by setting `options(rgl.meshColorWarning = FALSE)`. In some future release of `rgl`, the default will change to no warning.

Note that the `shade3d` function doesn’t support `meshColor = "edges"`, and the `wire3d` function will draw edges twice if they are common edges between two faces; which copy is visible depends on the `material3d("depth_test")` setting.

Value

`qmesh3d`, `cube3d`, `oh3d`, `tmesh3d`, `tetrahedron3d`, `octahedron3d`, `icosahedron3d` and `dodecahedron3d` return objects of class `c("mesh3d", "shape3d")`. The first three of these are quad meshes, the rest are triangle meshes.

`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, for some classes) invisibly.

See `rgl.primitive` for a discussion of texture coordinates.

See Also

`r3d`, `par3d`, `shapelist3d` for multiple shapes

Examples

```r
# generate a quad mesh object

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

open3d()
wire3d(qmesh3d(vertices, 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))

---

**mfrow3d**

*Set up multiple figure layouts in rgl.*

**Description**

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

**Usage**

```
subscenelist(value, window = rgl.cur())

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 = rgl.cur())
```
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("mfcoll")`).
- **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.

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 \texttt{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 (\texttt{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 \texttt{rgl}, to allow more flexibility in the specification of the observer’s location and orientation.

Author(s)

Duncan Murdoch

Examples

\begin{verbatim}
example(surface3d)  # The volcano data
observer3d(0, 0, 440) # Viewed from very close up
\end{verbatim}

---

\textbf{par3d \hspace{1cm} Set or Query RGL Parameters}

\textbf{Description}

\texttt{par3d} can be used to set or query graphical parameters in \texttt{rgl}. Parameters can be set by specifying them as arguments to \texttt{par3d in \texttt{tag = value}} form, or by passing them as a list of tagged values.

\textbf{Usage}

\begin{verbatim}
par3d(..., no.readonly = FALSE, dev = rgl.cur(),
       subscene = currentSubscene3d(dev))
open3d(..., params = getr3dDefaults(),
       useNULL = rgl.useNULL())
getr3dDefaults()
rg3dDefaults
\end{verbatim}
Arguments

arguments in tag = value form, or a list of tagged values. The tags 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.

params a list of graphical parameters

useNULL whether to use the null graphics device

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.

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. 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 writeWebGL and similar functions.

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.

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.
**Parameters**

*R.O.* indicates *read-only arguments*: These may only be used in queries, i.e., 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.

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

font integer. The default font number (from 1 to 5; 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.

F0V real. The field of view, from 0 to 179 degrees. This controls the degree of parallax in the perspective view. Isometric perspective corresponds to `F0V = 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 4 strings describing what the 3 mouse buttons and the mouse wheel do. Partial matching is used. Possible values for the first 3 entries of `mouseMode` (corresponding to the mouse buttons) 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 4th entry corresponding to the mouse wheel are...
"none" No action.
"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”.
"user" Used when a user handler is set by \texttt{rgl.setWheelCallback}.

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

\texttt{observer R.O.}; the position of the observer relative to the model. Set by \texttt{observer3d}. See the Note below.

\texttt{projMatrix R.O.}; a 4 by 4 matrix describing the current projection of the scene.

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

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

\texttt{userMatrix} a 4 by 4 matrix describing user actions to display the scene.

\texttt{userProjection} a 4 by 4 matrix describing changes to the projection.

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

\texttt{zoom real.} A positive value indicating the current magnification of the scene.

\texttt{bbox R.O.}; real. A vector of six values indicating the current values of the bounding box of the scene ($\text{xmin}$, $\text{xmax}$, $\text{ymin}$, $\text{ymax}$, $\text{zmin}$, $\text{zmax}$)

\texttt{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.

\textbf{Rendering}

The parameters returned by \texttt{par3d} are sufficient to determine where \texttt{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 $M = \texttt{par3d("modelMatrix")}$ as a product from right to left of the following matrices:
   - A matrix to translate the centre of the bounding box to the origin.
   - A matrix to rescale according to \texttt{par3d("scale")}.
   - The \texttt{par3d("userMatrix")} as set by the user.
   - A matrix which may be set by mouse movements.
   - If $s$ has the "model" set to "modify", a similar collection of matrices using parameters from the parent subscene.
3. It multiplies the point by $M$ giving $u = M \times v$.
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 $P = \text{par3d}("\text{projMatrix}")$ 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 $P \times u = (x2, y2, z2, w2)$.

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 $\text{par3d}("\text{viewport}")$ 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 $\text{cex} = 1$, so changing both $\text{cex}$ and $\text{family}$ needs to be done in the right order. For example, when using the "bitmap" family on Windows, $\text{par3d}(\text{family} = \"sans\", \text{cex} = 2)$ will work, but $\text{par3d}(\text{cex} = 2, \text{family} = \"sans\") will leave $\text{cex}$ at 1 (with a warning that the "bitmap" family only supports that size).

Although $\text{par3d}("\text{viewport}")$ 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 $\text{modelMatrix}$ entry had a changed meaning; before and after that it contains a copy of the OpenGL $\text{MODELVIEW}$ matrix.

References


See Also

`rgl.viewpoint` to set FOV and zoom.

`rgl.useNULL` for default usage of null device.
Examples

```r
r3dDefaults
open3d()
shade3d(cube3d(color = rep(rainbow(6), rep(4, 6))))
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

```r
par3dinterp(times = NULL, userMatrix, scale, zoom, FOV,
method = c("spline", "linear"),
extrapolate = c("oscillate", "cycle", "constant", "natural"),
dev = rgl.cur(), 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 $4 \times 4 \times n$ array, or a $4 \times 4n$ matrix; scale should be an $n \times 3$ matrix; zoom and F0V 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 tkrgl function par3dsave.

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

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

Examples

```r
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())
  playwidget(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

```r
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, plot3d, points`.

Examples

```r
codes <- c(0:25, z <- rep(0, 26, i <- seq(0, 5, i + 5)
> pch3d(x, y, z, pch = i, bg = "gray", color = rainbow(26))
> text3d(x + 5, y, z, pch = i+6)
> text3d(x + 5, y, z + 0.3, i+6)
```

Description

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

Usage

```r
persp3d(x, ...)
```

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 \texttt{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 \(x\$x, x\$y\) and \(x\$z\) are used for \(x, y\) and \(z\) respectively, though an explicitly specified \(z\) value will have priority.

One difference from \texttt{persp} is that colors are specified on each vertex, rather than on each facet of the surface. To emulate the \texttt{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, \(\text{col} \leftarrow \text{rbind}(1, \text{cbind}(\text{matrix(\text{col}, nx - 1, ny - 1), 1}))\). Finally, call \texttt{persp3d} with material property \texttt{smooth = FALSE}.

See the “Clipping” section in \texttt{plot3d} for more details on \texttt{xlim}, \texttt{ylim}, \texttt{zlim} and \texttt{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

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

The \texttt{surface3d} function is used to draw the surface without the axes etc.

Examples

```r
# (1) The Obligatory Mathematical surface.
#     Rotated sinc function.
```
x <- seq(-10, 10, length = 30)
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()
bg3d("white")
material3d(col = "black")

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

# (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, len = 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()
bg3d("slategray")
material3d(col = "black")
persp3d(x, y, z, col = "green3", aspect = "iso",
        axes = FALSE, box = FALSE)

# (4) A globe
lat <- matrix(seq(90, -90, len = 50)*pi/180, 50, 50, byrow = TRUE)
long <- matrix(seq(-180, 180, len = 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)
persp3d.deldir

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

```r
## 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

```r
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)))]

if (requireNamespace("deldir", quietly = TRUE)) {
  save <- options(rgl.meshColorWarning = FALSE)
dxyz <- deldir::deldir(x, y, z = z, suppressMsge = TRUE)
persp3d(dxyz, col = col)
open3d()
  # Do it without smoothing and with a different orientation.
persp3d(dxyz, col = col, coords = c("z", "x", "y"), smooth = FALSE)
  options(save)
}
```

Description

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

Usage

```r
## S3 method for class 'function'
persp3d(x, 
  xlim = c(0, 1), ylim = c(0, 1), 
  slim = NULL, tlim = NULL, 
  n = 101,
```

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

```r
## 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 corre-
  sponding 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

```r
# (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,
                      (s + t - 1)^2,
                      (s + t + 1)^2))
  x <- r * cos(t)
  y <- r * sin(t)
  z <- s
  return(cbind(x, y, z))
}
open3d()
plot3d(f, col = colorRampPalette(c("blue", "white", "red")),
     xlab = "X", ylab = "Y", zlab = "Cylinder",
     xlim = c(-10, 10), ylim = c(-10, 10),
     aspect = c(1, 1, 0.5))
```

persp3d.triSht

(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 or tripack Delaunay triangulation.

Description

The tri.mesh() functions in the interp and tripack packages compute 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, ...)
## S3 method for class 'tri'
plot3d(x, z, ...)
## S3 method for class 'tri'
persp3d(x, z, ..., add = FALSE)
## S3 method for class 'tri'
as.mesh3d(x, z, col = "gray", coords = c("x", "y", "z"),
         smooth = TRUE, normals = NULL, texcoords = NULL, ...)

Arguments

x       A "triSht" or "tri" object, produced by the tri.mesh() function in the interp or tripack packages respectively.
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

```r
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(10*(z - min(z))/diff(range(z)))]
save <- NULL
if (requireNamespace("interp", quietly = TRUE)) {
  save <- options(rgl.meshColorWarning = FALSE)
  dxy <- interp::tri.mesh(x, y)
  open3d()
  persp3d(dxy, z, col = col, meshColor = "vertices")
  open3d()
  # Do it without smoothing and with a different orientation.
  persp3d(dxy, z, col = col, coords = c("z", "x", "y"), smooth = FALSE)
}
if (requireNamespace("tripack", quietly = TRUE)) {
  if (is.null(save))
    save <- options(rgl.meshColorWarning = FALSE)

  # Leave a circular hole around (3, 0)
  theta <- seq(0, 2*pi, len = 30)[-1]
  cx <- 2*cos(theta) + 3
  cy <- 2*sin(theta)
  keep <- (x - 3)^2 + y^2 > 4
```
planes3d

```
dxy2 <- tripack::tri.mesh(x[keep], y[keep])
dxy2 <- tripack::add.constraint(dxy2, cx, cy)
z <- dxy2$x^2 - dxy2$y^2
col <- terrain.colors(20)[1 + round(19*(z - min(z))/diff(range(z)))]
open3d()
persp3d(dxy2, z, col = col)
```

planes3d

**Description**

planes3d and rgl.planes add mathematical planes to a scene. Their intersection with the current bounding box will be drawn. clipplanes3d and rgl.clipplanes add clipping planes to a scene.

**Usage**

```r
planes3d(a, b = NULL, c = NULL, d = 0, ...)
rgl.planes(a, b = NULL, c = NULL, d = 0, ...)
clipplanes3d(a, b = NULL, c = NULL, d = 0)
rgl.clipplanes(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 `rgl.material` for details.

**Details**

planes3d and rgl.planes draw 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 and rgl.clipplanes define 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, rgl.abclines- for mathematical lines.
-triangles3d, rgl.triangles- 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)

coeffs <- coef(fit)
a <- coeff["x"]
b <- coeff["y"]
c <- -1
d <- coeff["(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 = rgl.cur(), ..., startTime = 0)
movie3d(f, duration, dev = rgl.cur(), ..., fps = 10,
       movie = "movie", frames = movie, dir = tempdir(),
       convert = NULL, clean = TRUE, verbose = TRUE,
       top = TRUE, type = "gif", startTime = 0)
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.

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())
  play3d( par3dinterp(time = (0:2)*0.75, userMatrix = list(M, rotate3d(M, pi/2, 1, 0, 0), rotate3d(M, pi/2, 0, 1, 0)),
               duration = 3 )
## 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**

```r
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 auto-generated 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 playwidgets. 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.

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())
  widget
options(saveopts)

---

plot3d

### 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, ...)

## S3 method for class 'mesh3d'
plot3d(x, xlab = "x", ylab = "y", zlab = "z", type = c("shade", "wire", "dots"),
  add = FALSE, aspect = !add, ...)
decorate3d(xlim, ylim, zlim,
  xlab = "x", ylab = "y", zlab = "z",
  box = TRUE, axes = TRUE, main = NULL, sub = NULL,
  top = TRUE, aspect = FALSE, expand = 1.03,
  ...)

---

3D Scatterplot
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.
- **expand** how much to expand the box around the data, if it is drawn.
- **xlim, ylim, zlim** In `plot3d`, if not NULL, set clipping limits for the plot. In `decorate3d`, these are used for the labels.
- **forceClipregion** Force a clipping region to be used, whether or not limits are given.
- **...** additional parameters which will be passed to `par3d, material3d` or `decorate3d`.
- **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.

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.
- `decorate3d` adds the usual decorations to a plot: labels, axes, etc.
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 \( z \sim x + y \).
- **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

```r
open3d()
mfrow3d(1, 2, sharedMouse = TRUE)
plot3d(mpg ~ wt + qsec, data = mtcars)
if (requireNamespace("deldir"))
  persp3d(mpg ~ wt + qsec, data = mtcars)
```

Description

This function provides several plots of the result of fitting a two-predictor model.
Usage

```r
## 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),
...)
```

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.
- `...` 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.
2. Show the residuals and the plane at $z = 0$.
3. Show the predicted values on the fitted plane.
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.

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")])

plotmath3d
Generate sprites using base graphics math plotting.

Description
To plot mathematical text, this function uses base graphics functions to plot it to a `.png` file, then uses that file as a texture in a sprite.

Usage
plotmath3d(x, y = NULL, z = NULL, text, cex = par("cex"), adj = par("adj"),
pos = NULL, offset = 0.5,
fixedSize = TRUE, startsize = 480, initCex = 5, ...)
plotmath3d

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 expression. See `plotmath` for how expressions are interpreted.
- **cex** Character size expansion.
- **adj** One value specifying the horizontal adjustment, or two, specifying horizontal and vertical adjustment respectively.
- **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?

  These parameters are unlikely to be needed by users. `startsize` is an over-estimate 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.

  Additional arguments to pass to `text` when drawing the text.

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 this function if its `usePlotmath` argument is `TRUE`. The default value is determined by examining its `texts` argument; if it looks like an expression, `plotmath3d` is used.

Author(s)

Duncan Murdoch

See Also

- `text3d`

Examples

```r
open3d()
# This lets the text resize with the plot
text3d(4, 4, 4L, "resizeable text", usePlotmath = TRUE, fixedSize = FALSE)
```
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 rgl.material). For normals use normals and for texture coordinates use texcoords; see rgl.primitive for details.

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.

These functions call the lower level functions rgl.points, rgl.linestrips, and so on, and are provided for convenience.

The appearance of the new objects are defined by the material properties. See rgl.material for details.

The two principal differences between the rgl.* functions and the *3d functions are that the former set all unspecified material properties to defaults, whereas the latter use current values as defaults; the former make persistent changes to material properties with each call, whereas the latter make temporary changes only for the duration of the call.

Value

Each function returns the integer object ID of the shape that was added to the scene. These can be passed to rgl.pop to remove the object from the scene.
polygon3d

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()

---

polygon3d

Triangulate and 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

```r
polygon3d(x, y = NULL, z = NULL, fill = TRUE, plot = TRUE, coords = 1:2, 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.
- `random`  Should a random triangulation be used?
- `...`  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

```r
theta <- seq(0, 4*pi, len = 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)
plot(x, y, type = "n")
polygon(x, y)
polygon3d(x, y, x + y, col = "blue")
```
propertyControl | Controls to use with `playwidget()`.

**Description**

These are setter functions to produce actions in a Shiny app, or in an animation.

**Usage**

```r
subsetControl(value = 1, subsets, subscenes = NULL,
               fullset = Reduce(union, subsets),
               accumulate = FALSE)
propertyControl(value = 0, entries, properties, objids, 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.
- `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`. By default, `fullset` will be 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
propertySetter

Write HTML/Javascript code to control a WebGL display.

Description

propertyslider writes out HTML code to control WebGL displays on the same page via a slider; par3dinterpsetter and propertySetter return Javascript code to be used in HTML controls.

Usage

propertyslider(setter = propertySetter, 
minS = NULL, maxS = NULL, step = 1, init = NULL, 
labels, 
id = basename(tempfile("input")), name = id, 
outputid = paste0(id, "text"), 
index = NULL, 
...)

propertySetter(values = NULL, entries, properties, objids, prefixes = "", 
param = seq_len(NROW(values)), interp = TRUE, digits = 7)

par3dinterpsetter(fn, from, to, steps, subscene, omitconstant = TRUE, 
rename = character(), ...)

matrixsetter(fns, from, to, steps, subscene = currentSubscene3d(), 
matrix = "userMatrix", omitConstant = TRUE, prefix = "", ...)

vertexsetter(values, vertices = 1, attributes, objid, prefix = "", 
param = seq_len(NROW(values)), interp = TRUE, 
digits = 7)

Arguments

setter A function to write Javascript code, or its output, or a list containing several of these.
minS, maxS, step, init Slider values to be displayed. Reasonable defaults are used if missing.
labels Labels to display for each slider value. The defaults are calculated using internal variables. If NULL, no labels will be shown.
id The id of the input control that will be generated.
name The name of the input control that will be generated.
outputid  The id of the output control that will display the slider value, or NULL for none.
index    The 1-based index of this slider: it controls the corresponding entry in an indexed setter such as matrixSetter.
...      See Details below.
values   An array of values; rows correspond to slider positions. Alternatively, NULL; the generated function takes a single value or array of values and applies them directly.

entries, properties, objids, prefixes
    Vectors describing the columns of values. See the details below.
param    Parameter values corresponding to each row of values.
interp   Whether to interpolate values. If FALSE, the Javascript function will expect non-negative integer values. Ignored if values is NULL.
digits   How many significant digits to emit in the Javascript code.
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.
rename   A named character vector of names of Javascript properties to modify. See the details.
fns      A list containing functions returned from par3dinterp.
matrix   A character string giving the Javascript property name of the matrix to modify.
prefix   The prefix of the scene containing matrix.
vertices A vector of vertex numbers (1-based) within an object.
attributes A vector of attributes of a vertex, from c("x", "y", "z", "r", "g", "b", "a", "nx", "ny", "nz"), See Details.
objid    The object containing the vertices to be modified.

Details

The ... parameters to propertySlider will be passed to setter if the latter is a function, otherwise ignored.
The ... parameters to par3dinterpSetter will be passed to propertySetter.
The ... parameters to matrixSetter will be passed to the par3dinterpSetter functions used for each of the functions in fns.

propertySetter is a low-level general purpose function for modifying properties of objects in the scene. It is mainly for internal use. propertySlider uses it to generate Javascript for a slider control to manipulate those properties.

vertexSetter 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, the are: coordinates of the vertex "x", "y", "z", color of the vertex "r", "g", "b", "a", 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".

`propertySetter` and `vertexSetter` allow values to be specified in two ways. The normal way when used with a slider is to interpolate between specified values indexed by the slider. If `values = NULL`, the value of the slider is used directly (and only one entry can be set). Multiple entries can be set directly by passing an array of values in custom Javascript code.

`par3dinterpSetter` uses `propertySetter` to set parameters corresponding to values produced by the result of `par3dinterp`. Its `rename` argument allows translation of names, e.g. `rename = c(userMatrix = "myMatrix")` would cause the "userMatrix" result from `par3dinterp` to be used to modify the Javascript `myMatrix` property.

`matrixSetter` is used in the situation where multiple controls (e.g. sliders) are used to determine the value of a matrix, typically `userMatrix`. It will generate one `par3dinterpSetter` function for each of the entries in `fns`; these will be called when a `propertySlider` with the corresponding (1-based) index is changed, and the results multiplied together from right to left to produce a new value for whichever property is named in `matrix`.

The rows of the `values` matrix correspond to different settings for numeric properties. The columns are values to insert into those properties.

Argument `entries` gives the numeric (zero based) index into the Javascript property named by `properties`, for the object id `objids`, in the display with prefix `prefixes`. All of these may be vectors, corresponding to the columns of `values`. All but `entries` will be recycled to the appropriate length; its length needs to match the number of columns in `values`.

There are two modes for determining the values to substitute. In the simplest mode (`interp = FALSE` in `propertySetter`), each row of `values` corresponds to a location for the slider, and the values are simply copied into place. This requires that `param`, `min`, `max` and `step` take on their default values.

In other cases, linear interpolation is used between successive rows of `values`, with extrapolation outside the range of `param` repeating the first or last row. `param` should then contain the values that correspond to exact rows.

In both cases, `param` must be a strictly increasing vector.

**Value**

`propertySlider` prints the full code to generate the control, and returns the id of the control that was generated.

`propertySetter` returns a single element character vector containing the Javascript source for a function to set the appropriate properties. It does not assign the function to a variable or include any of the HTML wrapper text that `propertySlider` adds.

The character vector has class "propertySetter", and an attribute named "env" which gives access to the local environment where it was created, so for example `attr(value, "env")$prefixes` will give access to the `prefixes` argument if value was produced by "propertySetter".

`par3dinterpSetter` returns a `property Setter` result.

`matrixSetter` is similar to `propertySetter`, but the Javascript function takes arguments `value`, `index`, and the class of the result is `c("matrixSetter","indexedSetter","propertySetter")`.

`vertexSetter` is similar to `propertySetter`, but the class of the result is `c("vertexSetter","propertySetter")`. 
r3d

Author(s)
Duncan Murdoch

See Also
writeWebGL.clipplaneSlider makes use of propertySlider.
ageSetter can be used as the setter argument to propertySlider to modify objects according to a linear (age) scale.

Examples

```r
# Just the setter function
cat(propertySetter(1:4, entries = 12, properties = "values", objids = 13))

# A 4-position slider
propertySlider(values = 1:4, entries = 12, properties = "values", objids = 13, interp = FALSE)

# A 10-position slider interpolating the 4-position slider
propertySlider(values = 1:4, entries = 12, properties = "values", objids = 13,
              step = (4-1)/9)

# The userMatrix interpolation from example(play3d)
M <- r3dDefaults$userMatrix
fn <- par3dinterp(time = (0:2)*0.75, userMatrix = list(M,
               rotate3d(M, pi/2, 1, 0, 0),
               rotate3d(M, pi/2, 0, 1, 0)))
cat(par3dinterpSetter(fn, 0, 3, steps=10))
```

---------------------------

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 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

```r
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)

# rgl.* interface

rgl.open()
rgl.texts(x, y, z, labels)
rgl.texts(1, 1, 1, "rgl.* coordinates")
```
readSTL

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

```r
readSTL(con, ascii = FALSE, plot = TRUE, ...) writeSTL(con, ascii = FALSE,
  pointRadius = 0.005,
  pointShape = icosahedron3d(),
  lineRadius = pointRadius,
  lineSides = 20,
  ids = NULL)
```

Arguments

- **con** A connection or filename.
- **ascii** Whether to use the ASCII format or the binary format.
- **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 `rgl.id$`) of the objects to write. If NULL, try to write everything.
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; writeWebGL, writeASY, writePLY, writeOBJ and writeSTL write the scene to a file in various other formats.

Examples

```r
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 rgl.ids.

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 log-
ical 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

rgl.ids, rgl.attrib.info

Examples

```r
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

```r
rgl.attrib.info(id = rgl.ids("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.
rgl.bbox

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(), rgl.ids("all"))

rgl.bbox

Set up Bounding Box decoration

Description

Set up the bounding box decoration.

Usage

rgl.bbox(
  xat = NULL, xlab = NULL, xunit = 0, xlen = 5,
  yat = NULL, ylab = NULL, yunit = 0, ylen = 5,
  zat = NULL, zlab = NULL, zunit = 0, zlen = 5,
  marklen = 15.0, marklen.rel = TRUE, expand = 1,
  draw_front = FALSE, ...)
bbox3d(xat = NULL, yat = NULL, zat = NULL,
  xunit = "pretty", yunit = "pretty", zunit = "pretty",
  expand = 1.03,
  draw_front = FALSE, ...)

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 rgl.material 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 rgl.bbox and bbox3d.

Value

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

See Also

rgl.material, axes3d

Examples

rgl.open()
rgl.points(rnorm(100), rnorm(100), rnorm(100))
rgl.bbox(color = c("#333377", "white"), emission = "#333377",
specular = "#3333FF", shininess = 5, alpha = 0.8 )

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

rgl.open()
rgl.points(rnorm(1000), rnorm(1000), rnorm(1000), color = heat.colors(1000))
rgl.bringtotop(stay = TRUE)

rgl.material  Generic Appearance setup

Description

Set material properties for geometry appearance.
Usage

```r
gl.material(
  color = c("white"),
  alpha = c(1.0),
  lit = TRUE,
  ambient = "black",
  specular = "white",
  emission = "black",
  shininess = 50.0,
  smooth = TRUE,
  texture = NULL,
  textype = "rgb",
  texmipmap = FALSE,
  texminfilter = "linear",
  texmagfilter = "linear",
  texenvmap = FALSE,
  front = "fill",
  back = "fill",
  size = 3.0,
  lwd = 1.0,
  fog = TRUE,
  point_antialias = FALSE,
  line_antialias = FALSE,
  depth_mask = TRUE,
  depth_test = "less",
  polygon_offset = c(0.0, 0.0),
  ...
)
material3d(...)
```

Arguments

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) .. 1.0 (opaque).

smooth logical, specifying whether Gouraud shading (smooth) or flat shading should be used.

texture path to a texture image file. Supported formats: png.

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
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
point_antialias, line_antialias logical, specifying if points and lines should be antialiased
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 If non-zero, offsets are added to the recorded depth of filled polygons. See Details below.
... Any of the arguments above can be passed to material3d; see Details below. rgl.material will ignore others.

Details

Values can be queried by specifying their names in a character vector, e.g. material3d("color"). There is one read-only property that can be queried but not set:
isTransparency Is the current colour transparent?
Only one side at a time can be culled.

The `polygon_offset` property is a 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.

`material3d` is an alternate interface to the material properties, modelled after `par3d`: rather than setting defaults for parameters that are not specified, they will be left unchanged. `material3d` may also be used to query the material properties; see the examples below.

The current implementation does not return parameters for textures.

If `point_antialias` is `TRUE`, points will be drawn as circles; otherwise, they will be drawn as squares. Lines tend to appear heavier with `line_antialias = TRUE`.

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

The `...` parameter to `rgl.material` is ignored.

### Value

`rgl.material()` is called for the side effect of setting the material properties. It returns a value invisibly which is not intended for use by the user.

Users should use `material3d()` to query material properties. It returns values similarly to `par3d` as follows: When setting properties, it returns the previous values in a named list. A named list is also returned when more than one value is queried. When a single value is queried it is returned directly.

### See Also

`rgl.primitive, rgl.bbox, rgl.bg, rgl.light`

### Examples

```r
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

rgl.open  3D visualization device system

Description

3D real-time rendering system.

Usage

# Low level rgl.* interface
rgl.open(useNULL = rgl.useNULL())  # open new device
rgl.close()  # close current device
rgl.cur()  # returns active device ID
rgl.dev.list()  # returns all device IDs
rgl.set(which, silent = FALSE)  # set device as active
rgl.quit()  # shutdown rgl device system
rgl.init(initValue = 0, onlyNULL = FALSE)  # re-initialize rgl

Arguments

useNULL  whether to open the “null” device
which  device ID
silent  whether to suppress update of window titles
initValue  value for internal use only
onlyNULL  only initialize the null (no display) device

Details

The rgl 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 rgl.set(). rgl.quit() shuts down the rgl subsystem and all open devices, detaches the package including the shared library and additional system libraries.

The rgl.open() function attempts to open a new RGL window. If the ”rgl.antialias” option is set, it will be used to select the requested antialiasing. (See open3d for more description of antialiasing and an alternative way to set the value.)

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 writeWebGL and similar functions.

If rgl.open() fails (e.g. because X windows is not running, or its DISPLAY variable is not set properly), then you can retry the initialization by calling rgl.init(). Do not do this when windows
have already been successfully opened: they will be orphaned, with no way to remove them other
than closing R. In fact, it’s probably a good idea not to do this at all: quitting R and restarting it is a
better solution.

This package also includes a higher level interface which is described in the r3d help topic. That
interface is designed to act more like classic 2D R graphics. We recommend that you avoid mixing
rgl.* and *3d calls.

See the first example below to display the ChangeLog.

Value

rgl.open, rgl.close and rgl.set are called for their side effects and return no useful value.
Similarly rgl.init and rgl.quit are not designed to return useful values; in fact, users shouldn’t
call them at all!

rgl.dev.cur returns the currently active devices, or 0 if none is active; rgl.dev.list returns a vector
of all open devices. Both functions name the items according to the type of device: null for a
hidden null device, wgl for a Windows device, and glx for an X windows device.

See Also

r3d, rgl.clear, rgl.pop, rgl.viewpoint, rgl.light, rgl.bg, rglbbox, rgl.points, rgl.lines,
rgl.triangles, rgl.quads, rgl.texts, rgl.surface, rgl.spheres, rgl.sprites, rgl.snapshot,
rgl.useNULL

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

eexample(surface3d)
depth <- rgl.pixels(component = "depth")
if (length(depth) && is.matrix(depth)) # Protect against empty or single pixel windows
    contour(depth)

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 limita-
tions 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 1.4.0.

See Also
rgl.viewpoint, rgl.snapshot

Examples
# Create new files in tempdir
savedir <- setwd(tempdir())

x <- y <- seq(-10, 10, length = 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")
rgl.pop()
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
#

rgl.open()
shade3d(oh3d(), color = "red")
rgl.viewpoint(0, 20)

for (i in 1:45) {
    rgl.viewpoint(i, 20)
    filename <- paste("pic", formatC(i, digits = 1, flag = "0"), ".eps", sep = "")
    rgl.postscript(filename, fmt = "eps")
}

## End(Not run)
rgl.primitive

**Description**

Adds a shape node to the current scene

**Usage**

```r
rgl.points(x, y = NULL, z = NULL, ...)  
rgl.lines(x, y = NULL, z = NULL, ...)  
rgl.linestrips(x, y = NULL, z = NULL, ...)  
rgl.triangles(x, y = NULL, z = NULL, normals = NULL, texcoords = NULL, ...)  
rgl.quads(x, y = NULL, z = NULL, normals = NULL, texcoords = NULL, ...)  
```

**Arguments**

- **x, y, z**  coordinates. Any reasonable way of defining the coordinates is acceptable. See the function `xyz.coords` for details.
- **normals**  Normals at each point.
- **texcoords**  Texture coordinates at each point.
- **...**  Material properties. See `rgl.material` for details.

**Details**

Adds a shape node to the scene. The appearance is defined by the material properties. See `rgl.material` for details.

The names of these functions correspond to OpenGL primitives. They all take a sequence of vertices in `x, y, z`. The only non-obvious ones are `rgl.lines` which draws line segments based on pairs of vertices, and `rgl.linestrips` which joins the vertices.

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.

These are the lower level functions called by `points3d, segments3d, lines3d`, etc. The two principal differences between the `rgl.*` functions and the `*3d` functions are that the former set all unspecified material properties to defaults, whereas the latter use current values as defaults; the former make persistent changes to material properties with each call, whereas the latter make temporary changes only for the duration of the call.
Each primitive function returns the integer object ID of the shape that was added to the scene. These can be passed to `rgl.pop` to remove the object from the scene.

**See Also**

`rgl.material`, `rgl.spheres`, `rgl.texts`, `rgl.surface`, `rgl.sprites`

**Examples**

```r
g1l.open()
rgl.points(rnorm(1000), rnorm(1000), rnorm(1000), color = heat.colors(1000))
```

---

**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**

```r
g1l.select(button = c("left", "middle", "right"),
           dev = g1l.cur(), 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**

`rgl.select3d`, a version that allows the selection region to be used to select points in the scene.
rgl.setMouseCallbacks  User callbacks on mouse events

Description

Set and get user callbacks on mouse events.

Usage

```r
go. setMouseCallbacks(button, begin = NULL, update = NULL, end = NULL,
  dev = go.l.cur(), subscene = currentSubscene3d(dev))
go. getMouseCallbacks(button,
  dev = go.l.cur(), subscene = currentSubscene3d(dev))
go. setWheelCallback(rotate,
  dev = go.l.cur(), subscene = currentSubscene3d(dev))
go. getWheelCallback(dev = go.l.cur(), subscene = currentSubscene3d(dev))
```

Arguments

- **button**  Which button?
- **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 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` function returns the callback function or NULL.

**Value**

Author(s)

Duncan Murdoch

See Also

`par3d` to set built-in handlers

**Examples**

```r
pan3d <- function(button, dev = rgl.cur(), 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.setMouseCallables(button, begin, update, dev = dev, subscene = subscene)
  cat("Callables set on button", button, "of rgl device", dev, "in subscene", subscene, "\n")
}
```

```r
dev <- rgl.open3d()

shade3d(icosahedron3d(), col = "yellow")
pan3d()
```

---

**Description**

Saves the screenshot as png file.
rgl.snapshot

Usage

rgl.snapshot(filename, fmt = "png", top = TRUE)
snapshot3d(...)

Arguments

filename full path to filename.
fmt image export format, currently supported: png
top whether to call rgl.bringtotop
... arguments to pass to rgl.snapshot

Details

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.)

Note

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.

See Also

movie3d, rgl.viewpoint

Examples

## Not run:

# create animation
#

shade3d(oh3d(), color = "red")
rgl.bringtotop()
rgl.viewpoint(0, 20)

olddir <- setwd(tempdir())
for (i in 1:45) {
  rgl.viewpoint(i, 20)
  filename <- paste("pic", formatC(i, digits = 1, flag = "0"), ".png", sep = "")
  rgl.snapshot(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)
rgl.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 defining the grid.

Usage

```r
glf.surface(x, z, y, coords = 1:3, ...,
            normal_x = NULL, normal_y = NULL, normal_z = NULL,
            texture_s = NULL, texture_t = NULL)
```

Arguments

- **x**: values corresponding to rows of y, or matrix of x coordinates
- **y**: matrix of height values
- **z**: values corresponding to columns of y, or matrix of z coordinates
- **coords**: See details
- **...**: Material and texture properties. See `rgl.material` for details.
- **normal_x**, **normal_y**, **normal_z**: matrices of the same dimension as y giving the coordinates of normals at each grid point
- **texture_s**, **texture_t**: matrices of the same dimension as z giving the coordinates within the current texture of each grid point

Details

Adds a surface mesh to the current scene. The surface is defined by the matrix of height values in y, with rows corresponding to the values in x and columns corresponding to the values in z.

The coords parameter can be used to change the geometric interpretation of x, y, and z. The first entry of coords indicates which coordinate (1 = X, 2 = Y, 3 = Z) corresponds to the x parameter. Similarly the second entry corresponds to the y parameter, and the third entry to the z parameter. In this way surfaces may be defined over any coordinate plane.

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.
rgl.surface always draws the surface with the ‘front’ upwards (i.e. towards higher y values). This can be used to render the top and bottom differently; see rgl.material and the example below.

If the x or z argument is a matrix, then it must be of the same dimension as y, and the values in the matrix will be used for the corresponding coordinates. This is used to plot shapes such as cylinders where y is not a function of x and z.

NA values in the height matrix are not drawn.

Value

The object ID of the displayed surface is returned invisibly.

See Also

rgl.material, surface3d, terrain3d. See persp3d for a higher level interface.

Examples

```r
# volcano example taken from "persp"
#
data(volcano)
y <- 2 * volcano # Exaggerate the relief
x <- 10 * (1:nrow(y)) # 10 meter spacing (S to N)
z <- 10 * (1:ncol(y)) # 10 meter spacing (E to W)
ylim <- range(y)

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

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

rgl.open()
rgl.surface(x, z, y, color = col, back = "lines")
```

rgl.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 rgl 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.user2window()

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 = rgl.cur(), 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
rglIds

See Also

select3d

Examples

```r
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)
}
```

rglIds

\textit{rgl id values}

Description

All objects in an \texttt{rgl} scene have a numerical id. These ids are normally stored in vectors of class \texttt{c("rglIds", "numeric")}, which will also have class "rglHighlevel" or "rglLowlevel" depending on whether a high level function like \texttt{plot3d} or \texttt{persp3d}, or a low level function created the objects.

Usage

```r
rglId(ids = integer())
lowlevel(ids = integer())
highlevel(ids = integer())
## S3 method for class 'rglId'
print(x,
      rglwidget = getOption("rgl.printRglwidget", FALSE),
      ...
)
```

Arguments

- \texttt{ids} A vector of object ids.
- \texttt{x} An "rglId" object to print.
- \texttt{rglwidget} Whether to create and print an rgl widget. If false, nothing is printed.
- \texttt{...} Other arguments which will be passed to \texttt{rglwidget} if it is used.
Details

These functions and classes are intended to allow rgl scenes to be automatically displayed in R Markdown documents. However, this is not fully in place yet, so explicit rglwidget() calls are still recommended.

Note that all objects in the current scene will be printed by default, not just the ids in x. (The 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

```r
x <- matrix(rnorm(30), ncol = 3, dimnames = list(NULL, c("x", "y", "z")))
p <- plot3d(x, type = "s")
str(p)
if (interactive())
  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

```r
rglMouse(sceneId,
  choices = c("trackball", "selecting",
               "xAxis", "yAxis", "zAxis",
               "polar", "zoom", "fov",
               "none"),
  labels = choices,
  button = 1,
  dev = rgl.cur(),
  subscene = currentSubscene3d(dev),
  default = par3d("mouseMode", dev = dev, subscene = subscene)[button],
  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

Examples

if (interactive()) {
  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" 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

**Examples**

```r
save <- options(rgl.useNULL = 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, the 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())
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

if (requireNamespace("orientlib")) {
  persp3d(volcano, col = "green")
  if (requireNamespace("lattice"))
    lattice::wireframe(volcano, screen = rglToLattice())
  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**

```r
glwidget(x = scene3d(), width = figWidth(), height = figHeight(),
         controllers = NULL, snapshot = FALSE,
         elementId = NULL, reuse = !interactive(),
         webGLOptions = list(preserveDrawingBuffer = TRUE),
         shared = NULL, ...)
```

**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` Control of snapshot of scene. See `writeWebGL` for details.
- `elementId` The id to use on the HTML `div` component that will hold the scene.
- `reuse` A logical variable indicating whether rgl objects from earlier scenes should be referenced. See the 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.
- `...` 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 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 `shinyDemo()` for an example.

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`.

If the `reuse` argument is `FALSE` (the default in interactive use), earlier information will be cleared before drawing the new scene. If `TRUE`, earlier data will be re-used in the current scene, so it may be smaller and faster to load. In both cases information from the current scene (added to earlier information if `reuse=TRUE`) will be saved for possible use in a future scene. If `reuse=NA`, the saved information will neither be used nor updated.

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.
The `webGlOptions` argument is a list which will be passed when the WebGL context is created. See the WebGL 1.0 specification on [https://www.khronos.org/registry/webgl/specs](https://www.khronos.org/registry/webgl/specs) 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.

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 `reuse = TRUE`, a record will be kept of objects in the scene and they need not be included in the HTML generated for later scenes. This is normally useful only in `rmarkdown` documents which can have many `rgl` scenes; if the widget is displayed in RStudio, only one scene will be shown.

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.

Appearance

The appearance of the display is set by the stylesheet in `system.file("htmlwidgets/lib/rglClass/rgl.css")`. The widget is of class `rgl WebGL`, with id set according to `elementId`. (As of this writing, no special settings are given for class `rgl WebGL`, 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.

Examples

```r
save <- getOption("rgl.useNULL")
options(rgl.useNULL=TRUE)
example("plot3d", "rgl")
widget <- rglwidget()
if (interactive())
  widget

# Save it to a file. This requires pandoc
filename <- tempfile(fileext = ".html")
htmlwidgets::saveWidget(rglwidget(), filename)
browseURL(filename)
```
scene

**scene management**

---

**Description**

Clear shapes, lights, bbox

**Usage**

```r
clear3d( type = c("shapes", "bboxdeco", "material"), defaults, subscene = 0 )
rgl.clear( type = "shapes", subscene = 0 )
pop3d( ... )
rgl.pop( type = "shapes", id = 0 )
rgl.ids( type = "shapes", subscene = NA )
```

**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

- **...**
  - generic arguments passed through to RGL-specific (or other) functions

**Details**

RGL holds several lists of objects in each scene. There are lists for shapes, lights, bounding box decorations, subscenes, etc. `clear3d` and `rgl.clear` clear the specified stack, or restore the defaults for the bounding box (not visible) or viewpoint. With `id = 0` `rgl.pop` 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.

`rgl.clear` and `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`. 
rgl.ids returns a dataframe containing the IDs in the currently active subscene by default, or a specified subscene, or if subscene = \( \emptyset \), in the whole rgl window, along with an indicator of their type.

Note that clearing the light stack leaves the scene in darkness; it should normally be followed by a call to \texttt{rgl.light} or \texttt{light3d}.

See Also

\texttt{rgl}, \texttt{rgl.bbox}, \texttt{rgl.light}, \texttt{open3d} to open a new window.

Examples

```r
x <- rnorm(100)
y <- rnorm(100)
z <- rnorm(100)
p <- plot3d(x, y, z, type = 's')
grl.ids()
lines3d(x, y, z)
grl.ids()
if (interactive() && !rgl.useNULL()) {
  readline("Hit enter to change spheres")
grl.pop(id = p["data"])
spheres3d(x, y, z, col = "red", radius = 1/5)
boxed()
}
```


description

This function saves a large part of the RGL state associated with the current window to a variable.

Usage

```r
scene3d()
## S3 method for class 'rglscene'
plot3d(x, add = FALSE, ...)
## S3 method for class 'rglobj'
plot3d(x, ...)
## S3 method for class 'rglscene'
print(x, ...)
## S3 method for class 'rglobj'
print(x, ...)
```
Arguments

x An object of class "rglscene"
add Whether to open a new window, or add to the existing one.
... Additional parameters, currently ignored.

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, writePLY, writeOBJ and writeSTL write the scene to a file in formats readable by other software.

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.
obstacles A list containing the RGL lights, background and objects in the scene.

The objects in the objects component are of class "rglobjct". 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 "rglobjct"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.
sceneChange

Author(s)

Duncan Murdoch

See Also

writeWebGL, 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(),
            delete = NULL, add = NULL, replace = NULL,
            material = FALSE, rootSubscene = FALSE,
            delfromSubscenes = NULL, skipRedraw = FALSE)

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.
sceneChange

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.

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

```r
## 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

```r
rgl.select3d(button = c("left", "middle", "right"),
              dev = rgl.cur(), subscene = currentSubscene3d(dev))
select3d(...)```

Arguments

- `button` Which button to use for selection.
- `dev`, `subscene` The rgl device and subscene to work with
- `...` Button argument to pass to `rgl.select3d`

Details

This function 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.

If the scene is later moved or rotated, the selected region will remain the same, no longer corresponding to a rectangle on the screen.

Value

Returns 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

```r
# Allow the user to select some points, and then redraw them
# in a different color

if (interactive()) {
  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)
    rgl.pop()
    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

```r
selectpoints3d(objects = rgl.ids()$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.

Author(s)

Duncan Murdoch

See Also

`select3d` to return a selection function.

Examples

```r
xyz <- cbind(rnorm(20), rnorm(20), rnorm(20))
ids <- plot3d(xyz)

if (interactive()) {
  # 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) {
```
**setUserShaders**

```
spheres3d(xyz[ids[, "index"], , drop = FALSE], color = "blue",
    alpha = 0.3, radius = 0.2)
```

```
})
```

---

**setUserShaders**  
*Set user-defined shaders for rgl objects.*

---

**Description**

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.

**Usage**

```
setUserShaders(ids, vertexShader = NULL, fragmentShader = NULL,
    attributes = NULL, uniforms = NULL, scene = scene3d())
```

**Arguments**

- **ids**  
  Which objects should receive the shaders.

- **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”.

- **scene**  
  A scene3d object to modify.

**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 code automatically generates shaders for each object. This function allows them to be written by hand, for testing new features, hand optimization, etc. Currently it is not easy to get copies of the default shaders; they need to be obtained from a Javascript debugger while displaying the scene.

**Value**

A modified version of the scene.
Author(s)
Duncan Murdoch

See Also
rglwidget for display of the scene in WebGL.

Examples

```r
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 ambient0;
uniform vec3 specular0; // light*material
uniform vec3 diffuse0;
uniform vec3 lightDir0;
uniform bool viewpoint0;
uniform bool finite0;
```

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, 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

```r
shapelist3d(icosahedron3d(), x = rnorm(10), y = rnorm(10), z = rnorm(10), col = 1:5, size = 0.3)
```
Description

These functions allow an *rgl* scene to be embedded in a Shiny app.

Usage

```r
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
shinyGetPar3d

**Examples**

```r
## 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(rgl.close())
  plot3d(rnorm(n), rnorm(n), rnorm(n))
  rglwidget()
}, outputArgs = list(width = "auto", height = "300px"))
```

```r
## 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**

```r
shinyGetPar3d(parameters, session, subscene = currentSubscene3d(rgl.cur()), tag = "")
shinySetPar3d(..., session, subscene = currentSubscene3d(rgl.cur()))
```

**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.

**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 `shinyGetPar3d` 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](https://shiny.rstudio.com/articles/communicating-with-js.html) describes the underlying mechanisms used by these two functions.

**Examples**

```r
if (interactive()) {
  save <- options(rgl.useNULL = TRUE)

  xyz <- matrix(rnorm(300), ncol = 3)

  app = shiny::shinyApp(
    ui = shiny::bootstrapPage(
      shiny::actionButton("redraw", "Redraw"),
      rglwidgetOutput("rglPlot")
    ),
    server = function(input, output, session) {
      # This waits until the user to click on the "redraw"
      # button, then sends a request for the current userMatrix
      shiny::observeEvent(input$redraw, {
        shinyGetPar3d("userMatrix", session)
      })
      # This draws the plot whenever input$par3d changes,
      # i.e. whenever a response to the request above is
      # received.
      output$rglPlot <- renderRglwidget({
```
if (length(rgl.dev.list())) rgl.close()
col <- sample(colors(), 1)
plot3d(xyz, col = col, type = "s", main = col)
par3d(userMatrix = input$par3d$userMatrix)
rglwidget()
)
)
shiny::runApp(app)
options(save)
}

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, 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) 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

```r
example(plot3d, ask = FALSE)
show2d({
  par(mar=c(0,0,0,0))
  plot(x, y, col = rainbow(1000), axes=FALSE)
})
```

### spheres3d

*add sphere set shape*

#### Description

Adds a sphere set shape node to the scene

#### Usage

```r
spheres3d(x = NULL, y = NULL, z = NULL, radius = 1, ...)
rgl.spheres(x = NULL, y = NULL, z = NULL, radius, ...)
```

#### 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
- `...`: Material properties. See `rgl.material` 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-coordinate, 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.

#### Value

A shape ID of the spheres object is returned.
spin3d

See Also

rgl.material, aspect3d for setting non-isometric scales

Examples

```r
open3d()
spheres3d(rnorm(10), rnorm(10), rnorm(10), radius = runif(10), color = rainbow(10))
```

---

**Description**

This creates a function to use with play3d to spin an rgl scene at a fixed rate.

**Usage**

```r
spin3d(axis = c(0, 0, 1), rpm = 5,
       dev = rgl.cur(), 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

```r
# Spin one object
open3d()
plot3d(ch3d(col = "lightblue", alpha = 0.5))
if (!rgl.useNULL())
  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
  rgl.pop(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())
  play3d(f, duration = 2)
```

---

**sprites**  
*add sprite set shape*

### Description

Adds a sprite set shape node to the scene.

### Usage

```r
sprites3d(x, y = NULL, z = NULL, radius = 1, shapes = NULL, userMatrix,
  ...)  
particles3d(x, y = NULL, z = NULL, radius = 1, ...)  
rgl.sprites(x, y = NULL, z = NULL, radius = 1, shapes = NULL, userMatrix,
  fixedSize = FALSE, ...)
```

### 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 sphere radius
shapes  NULL for a simple square, or a vector of identifiers of shapes in the scene
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?
...  material properties when shape == 0; 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 gasses. The centre of each square will be at the coordinates given by x, y, z.

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 rgl.ids). These objects will be removed from the scene and duplicated as a sprite image in a constant orientation, as specified by userMatrix. The origin 0, 0, 0 will be plotted at the coordinates given by x, y, z.

The userMatrix argument is ignored for shapes = NULL. For shapes, sprites3d defaults the matrix to r3dDefaults$userMatrix while rgl.sprites defaults it to an identity transformation.

If any coordinate is NA, the sprite is not plotted.

The id values of the shapes are retrieved using rgl.attrib(id, "ids"); 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.

See Also

rgl.material

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"))
subdivision3d  
generic subdivision surface method

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, ... )
divide.mesh3d(mesh, vb = mesh$vb, ib = mesh$ib, it = mesh$it )
normalize.mesh3d(mesh)
deform.mesh3d(mesh, vb = mesh$vb, ib = mesh$ib, it = mesh$it )

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
it indices for triangular faces
ib indices for quad faces
vb matrix of vertices: 4xn 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 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.)
subscene3d

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 = rgl.cur())
useSubscene3d(subscene)
addToSubscene3d(ids, subscene = currentSubscene3d())
delFromSubscene3d(ids, 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.

ewviewport
    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.
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, len = 50)*pi/180, 50, 50, byrow = TRUE)
long <- matrix(seq(-180, 180, len = 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)
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
surface3d

Examples

example(plot3d)
subsceneInfo()

---

surface3d

*add height-field surface shape*

---

**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 defining the grid.

**Usage**

```r
surface3d(x, y, z, ..., normal_x = NULL, normal_y = NULL, normal_z = NULL)
terrain3d(x, y, z, ..., normal_x = NULL, normal_y = NULL, normal_z = NULL)
```

**Arguments**

- `x`: values corresponding to rows of `z`, or matrix of `x` coordinates
- `y`: values corresponding to the columns of `z`, or matrix of `y` coordinates
- `z`: matrix of heights
- `...`: Material and texture properties. See `rgl.material` for details.
- `normal_x`, `normal_y`, `normal_z`: matrices of the same dimension as `z` giving the coordinates of normals at each grid point

**Details**

Adds a surface mesh to the current scene. The surface is defined by the matrix of height values in `z`, with rows corresponding to the values in `x` and columns corresponding to the values in `y`. This is the same parametrization as used in `persp`.

If the `x` or `y` argument is a matrix, then it must be of the same dimension as `z`, and the values in the matrix will be used for the corresponding coordinates. This is used to plot shapes such as cylinders where `z` is not a function of `x` and `y`.

If the normals are not supplied, they will be calculated automatically based on neighbouring points. `surface3d` always draws the surface with the ‘front’ upwards (i.e. towards higher `z` values). This can be used to render the top and bottom differently; see `rgl.material` and the example below.

For more flexibility in defining the surface, use `rgl.surface`.

`surface3d` and `terrain3d` are synonyms.

**See Also**

`rgl.material`, `rgl.surface`. See `persp3d` for a higher level interface.
Examples

```r
# 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(z)

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")
```

Description

Adds text to the scene. The text is positioned in 3D space. Text is always oriented towards the camera.

Usage

```r
rgl.texts(x, y = NULL, z = NULL, text, adj = 0.5, pos = NULL, offset = 0.5,
family = par3d("family"), font = par3d("font"),
cex = par3d("cex"), useFreeType = par3d("useFreeType"), ...)
text3d(x, y = NULL, z = NULL, text, adj = 0.5, pos = NULL, offset = 0.5,
usePlotmath = is.language(texts), ...)
text3d3d(x, y = NULL, z = NULL, texts, adj = 0.5, pos = NULL, offset = 0.5,
usePlotmath = is.language(texts), ...)
rglFonts(...)
```

Arguments

- `x, y, z` point coordinates. Any reasonable way of defining the coordinates is acceptable. See the function `xyz.coords` for details.
text3d

text character vector to draw
texts character vector to draw
adj one value specifying the horizontal adjustment, or two, specifying horizontal and vertical adjustment respectively.

pos a position specifier for the text. If specified, this overrides any adj value given. Values of 1, 2, 3 and 4, respectively indicate positions below, to the left of, above and to the right of 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.

family A device-independent font family name, or ""
font A numeric font number from 1 to 5
cex A numeric character expansion value
useFreeType logical. Should FreeType be used to draw text? (See details below.)
usePlotmath logical. Should plotmath3d be used for the text?

... In rgltexts, material properties; see rgl_material for details. In rglfonts, device dependent font definitions for use with FreeType. In the other functions, additional parameters to pass to rgl.texts.

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 instead of rgl.texts. 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.

rglfonts returns the current set of font definitions.

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; with font 5 recoded as family "symbol" font 1.

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 if rgl was built to support FreeType.

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, rgl ships with just 3 font files, for regular versions of the serif, sans and mono families. Additional free font files are available from the Amaya project at http://dev.w3.org/cvsweb/Amaya/fonts/. See the example below for how to specify a full set of fonts.

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.

See Also

r3d, plotmath3d

Examples

code3d()

famnum <- rep(1:4, 8)
family <- c("serif", "sans", "mono", "symbol")[famnum]
font <- rep(rep(1:4, each = 4), 2)
cex <- rep(1:2, each = 16)
text3d(font, cex, famnum, text = paste(family, font), adj = 0.5,
       color = "blue", family = family, font = font, cex = cex)

## 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", 
                "ESSTIX9.TTF", ...)
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

```r
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.

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

```r
if (requireNamespace("plotrix")) {
  # 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))
      rgl.pop(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)
  }
}
```

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 = integer(), hidden = integer(),
  subscenes = NULL,
  label = deparse(substitute(ids)),
  ...)

triangulate

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.
subscenes The subscenes in which to toggle the objects.
label The label to put on the button.
... 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.

Examples

```r
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())
  widget
```

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 \texttt{xyz.coords}.
See Details for how z is handled.
random Whether to use a random or deterministic triangulation.
plot Whether to plot the triangulation; mainly for debugging purposes.
partial If the triangulation fails, should partial results be returned?

Details

Normally \texttt{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 \texttt{NA} values in x or y. Each of these pieces should be a simple, non-self-intersecting polygon, separate from the other pieces. (Though some minor exceptions to this rule may work, none are guaranteed). The nesting of these pieces is determined.

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.

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 \texttt{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, even when a triangulation is possible. Generally using \texttt{random = TRUE} will find a successful triangulation if one exists, but it may occasionally take more than one try.

Author(s)

Duncan Murdoch

References

See the Wikipedia article “polygon triangulation” for a description of the ear-clipping algorithm.

See Also

\texttt{extrude3d} for a solid extrusion of a polygon, \texttt{polygon3d} for a flat display; both use \texttt{triangulate}.
Examples

theta <- seq(0, 2*pi, len = 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")

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
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, 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", ...
See Details.
objid A single rgl object id.
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.
viewpoint

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

Author(s)
Duncan Murdoch

Examples

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

theta <- seq(0, 6*pi, len=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())
widget
options(saveopts)
```

---

Description
Set up viewpoint

Usage

```r
view3d( theta = 0, phi = 15, ...)
rgl.viewpoint( theta = 0, phi = 15, fov = 60, zoom = 1,
scale = par3d("scale"), interactive = TRUE, userMatrix,
type = c("userviewport", "modelviewport") )
```

Arguments

- `theta, phi`  polar coordinates
- `...`  additional parameters to pass to rgl.viewpoint
- `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

```r
## Not run:
# animated round trip tour for 10 seconds

rgl.open()
shade3d(oh3d(), color = "red")

start <- proc.time()[3]
while ((i <- 360*(proc.time()[3] - start)) < 360) {
  rgl.viewpoint(i, i/4);
}

## End(Not run)
```
Write HTML/JavaScript code to control a WebGL display.

**Description**

These functions write out HTML code to control WebGL displays on the same page.

**Usage**

```javascript
subsetSlider(subsets, labels = names(subsets),
    fullset = Reduce(union, subsets),
    subscenes = currentSubscene3d(), prefixes = "",
    accumulate = FALSE, ...)

subsetSetter(subsets, subscenes = currentSubscene3d(), prefixes = "",
    fullset = Reduce(union, subsets),
    accumulate = FALSE)

clipplaneSlider(a=NULL, b=NULL, c=NULL, d=NULL,
    plane = 1, clipplaneids, prefixes = "",
    labels = signif(values[1,3], ...)

toggleButton(subset, subscenes = currentSubscene3d(), prefixes = "",
    label = deparse(substitute(subset)),
    id = paste0(basename(tempfile("input"))), name = id)
```

**Arguments**

- **subsets** A list of vectors of object identifiers; the slider or setter will choose among them.
- **labels** Labels to display corresponding to each subset. If NULL, numeric labels will be shown.
- **fullset** Objects in the subscene which are not in fullset will not be touched.
- **subscenes** The subscenes to be controlled.
- **prefixes** The prefixes of the WebGL scenes to be controlled.
- **accumulate** If TRUE, the subsets will accumulate (by union) as the value increases.
- **id** The id of the input control that will be generated.
- **name** The name of the input control that will be generated.
- **a, b, c, d** The parameter values to change. Leave as NULL to hold the parameter constant.
- **plane, clipplaneids** The identifier of the particular clipplane to modify.
- **subset** The subset that the button should toggle.
- **label** The button label.
Details

subsetSlider and clipplaneSlider generate an HTML5 slider control with a text label. To display no text, set the labels to blanks in the ... argument.

subsetSetter is a low-level function to produce the Javascript code used by subsetSlider. This code will not touch objects in the subscene 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.

toggleButton generates a button that toggles the presence of objects within a subscene. If the first entry in subset is in the subscene, all are deleted, otherwise all are added.

clipplaneSlider allows a clipping plane to be moved to a sequence of positions. Any non-NULL entries among a, b, c, d should be vectors; the slider will set those parameters of the selected clipping plane to corresponding values as the slider is moved.

Value

subsetSetter returns a length-one character vector of class "propertySetter".

The other functions use cat to write their output and invisibly return the id of the control that was generated.

Author(s)

Duncan Murdoch

See Also

playwidget and toggleWidget for a newer, preferred method of inserting controls into a scene.

Older functions include writeWebGL. See propertySlider and propertySetter for lower level, more flexible functions.

Examples

```r
subsets <- list(1:3, 1:4, 1:5)
subsetSlider(subsets = subsets)
toggleButton(1:3)
clipplaneSlider(d = seq(0, 1, len=20), clipplaneid = 123)
```

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

```r
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 = NULL)
```

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.

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.

Author(s)
Duncan Murdoch

References

See Also
scene3d saves a copy of a scene to an R variable: writeWebGL, writePLY, writeOBJ and writeSTL write the scene to a file in various other formats.

Examples
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)

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 = NULL)
readOBJ(con, ...)

writeOBJ
writeOBJ

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>con</td>
<td>A connection or filename.</td>
</tr>
<tr>
<td>pointRadius, lineRadius</td>
<td>The radius of points and lines relative to the overall scale of the figure, if they are converted to polyhedra.</td>
</tr>
<tr>
<td>pointShape</td>
<td>A mesh shape to use for points if they are converted. It is scaled by the pointRadius.</td>
</tr>
<tr>
<td>lineSides</td>
<td>Lines are rendered as cylinders with this many sides.</td>
</tr>
<tr>
<td>pointsAsPoints, linesAsLines</td>
<td>Whether to convert points and lines to “point” and “line” records in the OBJ output.</td>
</tr>
<tr>
<td>withNormals</td>
<td>Whether to output vertex normals for smooth shading.</td>
</tr>
<tr>
<td>separateObjects</td>
<td>Whether to mark each RGL object as a separate object in the file.</td>
</tr>
<tr>
<td>withTextures</td>
<td>Whether to output texture coordinates.</td>
</tr>
<tr>
<td>ids</td>
<td>The identifiers (from rgl.ids) of the objects to write. If NULL, try to write everything.</td>
</tr>
<tr>
<td>...</td>
<td>Additional arguments (typically just material) to pass to tmesh3d.</td>
</tr>
</tbody>
</table>

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 = ...). There is no support for files that use multiple textures.

The defaults for pointsAsPoints and linesAsLines have been chosen because Blender (http://www.blender.org) 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
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.

References

The file format was found at http://www.martinreddy.net/gfx/3d/OBJ.spec on November 11, 2012.

See Also

scene3d saves a copy of a scene to an R variable; writeWebGL, 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 this shape
# file of Comet 67P/Churyumov-Gerasimenko, from the ESA:
open3d()
shade3d(readOBJ(url("http://sci.esa.int/science-e/www/object/doc.cfm?fobjectid=54726"), 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

Usage

```r
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 = 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 `rgl.ids`) of the objects to write. If NULL, try to write everything.

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 ([http://www.blender.org](http://www.blender.org)) 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
writeWebGL

References

The file format was found on http://www.mathworks.com on November 10, 2012 at a URL that no longer exists; currently the format is described at https://www.mathworks.com/help/vision/ug/the-ply-format.html.

See Also

scene3d saves a copy of a scene to an R variable; writeWebGL, writeASY, writeOBJ and writeSTL write the scene to a file in various other formats.

Examples

```r
filename <- tempfile(fileext = ".ply")
open3d()
shade3d(icosahedron3d(col = "magenta"))
writePLY(filename)
```

writeWebGL

Write scene to HTML.

Description

Writes the current scene to a collection of files that contain WebGL code to reproduce it in a browser.

Usage

```r
writeWebGL(dir = "webgl", filename = file.path(dir, "index.html"),
           template = system.file(file.path("WebGL", "template.html"), package = "rgl"),
           prefix = ",",
           snapshot = TRUE, commonParts = TRUE, reuse = NULL,
           font = "Arial", width, height)
```

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dir</td>
<td>Where to write the files.</td>
</tr>
<tr>
<td>filename</td>
<td>The filename to use for the main file.</td>
</tr>
<tr>
<td>template</td>
<td>The template web page to which to write the Javascript for the scene.</td>
</tr>
<tr>
<td></td>
<td>See Details below.</td>
</tr>
<tr>
<td>prefix</td>
<td>An optional prefix to use on global identifiers in the scene; use different</td>
</tr>
<tr>
<td></td>
<td>prefixes for different scenes displayed on the same web page. If not blank,</td>
</tr>
<tr>
<td></td>
<td>it should be a legal identifier in Javascript and HTML.</td>
</tr>
<tr>
<td>snapshot</td>
<td>Whether to include a snapshot of the scene, to be displayed in browsers that</td>
</tr>
<tr>
<td></td>
<td>don’t support WebGL, or a specification of the snapshot to use. See details</td>
</tr>
<tr>
<td></td>
<td>below.</td>
</tr>
<tr>
<td>commonParts</td>
<td>Whether to include parts that would be common to several figures on the same</td>
</tr>
<tr>
<td></td>
<td>page. Currently this includes a reference to and copy of the ‘CanvasMatrix.js’</td>
</tr>
<tr>
<td></td>
<td>file in the output.</td>
</tr>
</tbody>
</table>
reuse When writing several figures on the same page, set this to a dataframe containing values to reuse. See the Value section below.

font The font to use for text.

width, height The (optional) width and height in pixels of the image to display. If omitted, the par3d("windowRect") dimensions will be used.

Details

This function writes out a web page containing Javascript that reconstructs the scene in WebGL.

Use the template argument to give the filename of a web page that is to contain the code for the new scene. It should contain a single line containing paste0("%, prefix, "WebGL"), e.g. %WebGL% with the default empty prefix. That line will be replaced by the Javascript and other code necessary to reproduce the current scene. The template may also contain the string "%rglVersion%" which will be replaced with the current rgl version number. If template is NULL, the output will simply be written directly to the main file.

To put more than one scene into a web page, use different values of prefix for each. The prefix will be used in identifiers in both Javascript and HTML, so it is safest to start with a letter and only use alphanumeric characters.

WebGL is a fairly new technology for displaying 3D scenes in browsers. Most current browsers support it to some extent, though it may not be enabled by default; see http://get.webgl.org for details. A major exception currently is Microsoft’s Internet Explorer, though plugins are available. Currently writeWebGL has a number of known limitations, some of which will be gradually eliminated as development progresses:

• The bounding box decorations are fixed; labels do not move as they do within R.
• User-defined mouse controls are not supported.
• Missing values are not handled properly.
• WebGL browsers generally do not support more than 65535 vertices per object. writeWebGL will print a warning if this limit is exceeded, but it is up to the user to break his scene into smaller objects. (And 65535 vertices may not be small enough!)

Value

The filename is returned. If reuse is not NULL, it will have an attribute called "reuse" which contains a dataframe with columns "id" and "prefix" identifying the prefix used for objects drawn in this scene. This dataframe can be used as the reuse argument in subsequent calls to writeWebGL.

Note

This function is deprecated in favour of the rglwidget() function.

Note

If commonParts is TRUE, the output includes a binary copy of the CanvasMatrix Javascript library. This file is necessary for the Javascript code written by writeWebGL to function properly, but only one copy is needed if the output page contains multiple writeWebGL figures.

Its source (including the copyright notice and license for free use) is included in the file named by system.file("htmlwidgets/lib/CanvasMatrix.src.js", package = "rgl").
writeWebGL

Author(s)

Duncan Murdoch.

References

http://www.webgl.org

See Also

scene3d saves a copy of a scene to an R variable; writeASY, writePLY, writeOBJ and writeSTL write the scene to a file in various other formats.

Examples

plot3d(rnorm(100), rnorm(100), rnorm(100), type = "s", col = "red")

# This writes a copy into temporary directory 'webGL', and then displays it
filename <- writeWebGL(dir = file.path(tempdir(), "webGL"),
                          width = 500, reuse = TRUE)

# Display the "reuse" attribute
attr(filename, "reuse")

# Display the scene in a browser
if (interactive())
  browseURL(paste0("file://", filename))
Index

*Topic **dplot**
  ellipse3d, 33
  par3dinterp, 56
  play3d, 70
  spin3d, 141

*Topic **dynamic**
  abclines3d, 6
  addNormals, 7
  aspect3d, 20
  axes3d, 22
  bg3d, 24
  cylinder3d, 30
  grid3d, 37
  light, 41
  matrices, 42
  mesh3d, 45
  par3d, 51
  persp3d, 60
  planes3d, 69
  plot3d, 75
  points3d, 82
  r3d, 89
  rgl-package, 4
  rgl.bbox, 95
  rgl.bringtobottom, 97
  rgl.material, 97
  rgl.open, 101
  rgl.pixels, 102
  rgl.postscript, 103
  rgl.primitive, 105
  rgl.setMouseCallbacks, 107
  rgl.snapshot, 108
  rgl.surface, 110
  rgl.user2window, 114
  scene, 123
  select3d, 128
  shapeList3d, 133
  spheres3d, 140
  sprites, 142

subdivision3d, 144
surface3d, 149
text3d, 150
viewpoint, 159

*Topic **graphics**
  bgplot3d, 26
  extrude3d, 34
  identify3d, 40
  mfrow3d, 48
  observer3d, 50
  persp3d, 60
  persp3d.deldir, 63
  persp3d.function, 64
  persp3d.trisht, 67
  polygon3d, 83
  readSTL, 91
  rgl.attrib, 92
  scene3d, 124
  selectpoints3d, 129
  subscene3d, 145
  subsceneInfo, 148
  triangulate, 155
  turn3d, 157
  writeOBJ, 164
  writePLY, 166
  writeWebGL, 168

*Topic **utilities**
  hook_rgl, 39
  rgl.Sweave, 111
  .check3d, 5
  %>% (import), 41
  %>% 41

abclines3d, 6, 38, 70
addNormals, 7, 14, 15, 44
addToSubscene3d (subscene3d), 145
ageControl, 8, 41, 74
ageSetter, 9, 89
all.equal, 14, 45
alphashape3d::ashape3d, 16, 17

171
arc3d, 11
decorate3d, 61
arrow3d, 12, 13
decorate3d (plot3d), 75
arrows, 13
defform.mesh3d (subdivision3d), 144
as.character, 40
deldir, 63
as.mesh3d, 14, 17, 18, 20
delFromSubscene3d (subscene3d), 145
as.mesh3d.ashape3d, 16
divide.mesh3d (subdivision3d), 144
as.mesh3d.deldir (persp3d.deldir), 63
dodecahedron3d (mesh3d), 45
as.mesh3d.rglId, 14, 17, 45
dot3d, 90, 134
as.mesh3d.tri (persp3d.triSht), 67
dot3d (mesh3d), 45
as.mesh3d.triSht (persp3d.triSht), 67
elemld2Prefix, 32
as.triangles3d, 19
ellipse3d, 33, 140
asEuclidean (matrices), 42
expression, 151
asEuclidean (matrices) (asMesh), 42
extrude3d, 34, 84, 156, 157
asHomogeneous (matrices), 42
figHeight (figWidth), 36
aspect3d, 20, 54, 76, 141
figWidth, 36
clear.mesh3d, 18, 27
gc3d, 49
clearmesh3d, 18, 27
get3dDefaults (par3d), 51
clearSubsceneList (mfrow3d), 48
getWidgetId (asRow), 21
clearPlaneControl, 29
GramSchmidt, 36
clearLplanes3d, 77, 158
grid, 38
clearLplanes3d (planes3d), 69
grid3d, 37
clearLplaneSd (planes3d), 69
highlevel (rglIds), 115
clearLplaneSlider, 89
hook_rgl, 39, 113
clearLplaneSdSlider (webGLcontrols), 161
hook_webgl, 113, 122
clearLcloud, 120
hook_webgl (hook_rgl), 39
col2rgb, 10
icosahedron3d (mesh3d), 45
createWidget, 73, 121
identityMatrix (matrices), 42
createWidget (73), 121
import, 41
cube3d, 4, 90
layout, 48, 49
cube3d (mesh3d), 45
layout3d, 147
cuboctahedron3d (mesh3d), 45
layout3d (mfrow3d), 48
currentSubscene3d (subscene3d), 145
legend, 26
currentSubscene3d (subscene3d) (current3d), 145
light, 41
curve, 66
light3d, 124
cylinder3d, 30
light3d (light), 41
cylinder3d (mesh3d), 45
lines3d, 84, 90, 105
cylinder3d (model3d), 82
lines3d (points3d), 82
locator, 128
lowlevel (rglIds), 115
INDEX

magick, 71
material3d, 16, 23, 25, 47, 52, 76, 125
material3d (rgl.material), 97
matrices, 42, 55
matrixSetter (propertySetter), 86
mergeVertices, 44
mesh3d, 14, 15, 27, 34, 43, 45, 46, 64, 68, 90, 134, 145
mfrow3d, 48, 147
movie3d, 109
movie3d (play3d), 70
mtext, 23
mtext3d, 138
mtext3d (axes3d), 22
newSubscene3d, 49, 148
newSubscene3d (subscene3d), 145
next3d (mfrow3d), 48
normalize.mesh3d (subdivision3d), 144
observer3d, 50, 54
octahedron3d (mesh3d), 45
oh3d (mesh3d), 45
open3d, 4, 5, 77, 90, 101, 113, 124
open3d (par3d), 51
par, 48, 49
par3d, 4, 20, 44, 47, 49, 51, 56, 71, 76, 77, 100, 107, 108, 114, 125, 146, 151, 160
par3dinterp, 56, 58, 72, 87, 88
par3dinterpControl, 58
par3dinterpSetter (propertySetter), 86
par3dsave, 57
particles3d (sprites), 142
pch3d, 59
persp, 61, 120, 149
persp3d, 17, 60, 65, 66, 77, 100, 111, 115, 149
persp3d. deldir, 61, 63, 78
persp3d. formula, 61
persp3d. formula (plot3d.formula), 77
persp3d. function, 61, 64
persp3d. tri (persp3d. triSht), 67
persp3d. triSht, 67
pipe, 22
pipe (import), 41
planes3d, 6, 69, 77, 79, 158
play3d, 57, 70, 141
playwidget, 8, 29, 41, 58, 72, 74, 85, 86, 121, 127, 135, 155, 158, 162
playwidgetOutput (shiny), 135
plot.default, 77
plot3d, 4, 17, 20, 52, 59-61, 75, 77, 79, 115, 125
plot3d.default, 78
plot3d. deldir, 77
plot3d. deldir (persp3d. deldir), 63
plot3d. formula, 77
plot3d. function, 77
plot3d. function (persp3d. function), 64
plot3d. lm, 78
plot3d. rglobject (scene3d), 124
plot3d. rglscene (scene3d), 124
plot3d. tri (persp3d. triSht), 67
plot3d. triSht (persp3d. triSht), 67
plotmath, 81
plotmath3d, 80, 151, 152
plotrix, 153
png, 26, 138
points, 59, 60
points3d, 11, 60, 82, 90, 105
poly, 80
polygon3d, 13, 35, 83, 156
pop3d (scene), 123
pretty, 38, 96
print.rgl1d (rglIds), 115
print.rglobject (scene3d), 124
print.rglscene (scene3d), 124
propertyControl, 29, 58, 73, 74, 85
propertySetter, 86, 162
propertySlider, 10, 161, 162
propertySlider (propertySetter), 86
qmesh3d, 15, 33
qmesh3d (mesh3d), 45
quads3d, 90
quads3d (points3d), 82
quote, 151
r3d, 4, 47, 89, 102, 145, 151, 152
r3dDefaults, 52, 100, 123, 151
r3dDefaults (par3d), 51
rainbow, 66
readOBJ (writeOBJ), 164
readSTL, 91
registerSceneChange (sceneChange), 126
renderPlaywidget (shiny), 135
renderRglWidget (shiny), 135
rgl, 90, 124
rgl (rgl-package), 4
rgl-package, 4
rgl.abclines, 70
rgl.abclines (abclines3d), 6
rgl.antialias (rgl.open), 101
rgl.attrib, 9, 92, 94, 95, 125, 130
rgl.attrib.count (rgl.attrib.info), 94
rgl.attrib.info, 94, 94
rgl.bbox, 95, 96, 100, 102, 124
rgl.bg, 100, 102
rgl.bg (bg3d), 24
rgl.bringtotop, 71, 97, 109
rgl.clear, 42, 102
rgl.clear (scene), 123
rgl.clipplanes (planes3d), 69
rgl.close (rgl.open), 101
rgl.cur (rgl.open), 101
rgl.dev.list (rgl.open), 101
rgl.getMouseCallbacks
  (rgl.setMouseCallbacks), 107
rgl.getWheelCallback
  (rgl.setMouseCallbacks), 107
rgl.ids, 91, 93, 94, 143, 163, 167
rgl.ids (scene), 123
rgl.init (rgl.open), 101
rgl.light, 100, 102, 124
rgl.light (light), 41
rgl.lines, 102
rgl.lines (rgl.primitive), 105
rgl.linestrips, 82
rgl.linestrips (rgl.primitive), 105
rgl.material, 69, 82, 96, 97, 105, 106, 110, 111, 140, 141, 143, 149, 151
rgl.open, 4, 90, 101, 113
rgl.pixels, 102
rgl.planes, 6
rgl.planes (planes3d), 69
rgl.points, 82, 102
rgl.points (rgl.primitive), 105
rgl.pop, 42, 82, 96, 102, 106
rgl.pop (scene), 123
rgl.postscript, 103, 112
rgl.primitive, 47, 82, 100, 105
rgl.printRglWidget (rglwidget), 120
rgl.projection (rgl.user2window), 114
rgl.quads, 102
rgl.quads (rgl.primitive), 105
rgl.quit (rgl.open), 101
rgl.select, 106
rgl.select3d, 106
rgl.select3d (select3d), 128
rgl.set, 4
rgl.set (rgl.open), 101
rgl.setMouseCallbacks, 53, 107
rgl.setWheelCallback, 54
rgl.setWheelCallback
  (rgl.setMouseCallbacks), 107
rgl.snapshot, 71, 102–104, 108
rgl.spheres, 102, 106
rgl.spheres (spheres3d), 140
rgl.sprites, 102, 106
rgl.sprites (sprites), 142
rgl.surface, 102, 106, 110, 149
rgl.Sweave, 39, 111
rgl.texts, 102, 106
rgl.texts (text3d), 150
rgl.triangles, 70, 102
rgl.triangles (rgl.primitive), 105
rgl.useNULL, 4, 55, 102, 113
rgl.user2window, 114
rgl.viewpoint, 55, 102, 104, 109
rgl.viewpoint (viewpoint), 159
rgl.window2user (rgl.user2window), 114
rglFonts (text3d), 150
rglHighlevel (rglIds), 115
rglId (rglIds), 115
rglIds, 115
rglLowlevel (rglIds), 115
rglMouse, 116
globj-class (scene3d), 124
glscene-class (scene3d), 124
glShared, 86, 118, 121, 122
glToBase (glToLattice), 119
glToLattice, 119
glwidget, 32, 36, 41, 74, 115–117, 120, 132, 135, 155
glwidgetOutput, 122
glwidgetOutput (shiny), 135
rotate3d, 35, 90, 134
rotate3d (matrices), 42
rotationMatrix (matrices), 42
RweaveLatex, 113
saveWidget, 121
scale3d, 134
scale3d (matrices), 42
scaleMatrix (matrices), 42
scene, 123
scene3d, 92, 121, 124, 131, 164, 166, 168, 170
sceneChange, 126
segments3d, 6, 13, 90, 105
segments3d (points3d), 82
select3d, 40, 53, 90, 115, 128, 129, 130
selectpoints3d, 128, 129
setupKnitr (hook_rgl), 39
setUsershaders, 131
shade3d, 13, 84, 90, 134
shade3d (mesh3d), 45
shape3d (mesh3d), 45
shapelist3d, 47, 133
shiny, 135
shiny::session$sendCustomMessage, 127
shinyGetPar3d, 136
shinySetPar3d (shinyGetPar3d), 136
show2d, 138
sliderInput, 29, 73, 158
snapshot3d (rgl.snapshot), 108
spheres3d, 90, 140
spin3d, 71, 72, 141
sprintf, 71
sprites, 142
sprites3d, 59, 90
sprites3d (sprites), 142
subdivision3d, 34, 90, 144
subscene3d, 70, 145
subsceneInfo, 125, 147, 148
subsceneList (mfrow3d), 48
subplotControl, 74
subsetControl (propertyControl), 85
subsetSetter, 86
subsetSetter (webGLcontrols), 161
subsetSlider (webGLcontrols), 161
surface3d, 61, 79, 111, 149
Sweave, 111, 112
Sweave. snapshot (rgl.Sweave), 111
Sys.sleep, 71, 112
tagList, 74, 117
tags, 74
terrain3d, 90, 111
terrain3d (surface3d), 149
tetrahedron3d (mesh3d), 45
text, 81
text3d, 40, 53, 59, 60, 81, 90, 150, 153, 154
texts3d (text3d), 150
thigmophobe, 153
thigmophobe3d, 153
title, 23
title3d (axes3d), 22
tkrgl, 57
tmesh3d, 15, 16, 35, 64, 68, 157, 165
tmesh3d (mesh3d), 45
toggleButton, 155
toggleButton (webGLcontrols), 161
toggleWidget, 74, 121, 154, 162
transform3d, 90
transform3d (matrices), 42
translate3d, 134
translate3d (matrices), 42
translationMatrix (matrices), 42
tri.mesh, 67
triangles3d, 19, 70, 90, 91
triangles3d (points3d), 82
triangulate, 35, 84, 155
turn3d, 35, 157
useSubscene3d, 4
useSubscene3d (subscene3d), 145
vertexControl, 74, 158
vertexSetter, 10
vertexSetter (propertySetter), 86
view3d (viewpoint), 159
viewpoint, 159
webGLcontrols, 161
wire3d, 90, 134
wire3d (mesh3d), 45
wireframe, 120
writeASY, 92, 162, 166, 168, 170
writeOBJ, 92, 125, 126, 164, 168, 170
writePLY, 92, 125, 126, 164, 166, 167, 170
writeSTL, 92, 125, 126, 164, 166, 168, 170
writeSTL (readSTL), 91
writeWebGL, 25, 32, 52, 89, 92, 101, 121, 125,
126, 162, 164, 166, 168, 168
xy.coords, 105, 157
xyz.coords, 6, 13, 14, 31, 40, 41, 59, 69, 76,
78, 81, 82, 84, 105, 114, 128, 134,
140, 142, 150, 153, 156