RGL: A R-library for 3D visualization with OpenGL

Daniel Adler, Oleg Nenadić, Walter Zucchini

Institut für Statistik und Ökonometrie, University of Göttingen [1]

Abstract

RGL is a library of functions that offers three-dimensional, real-time visualization functionality to the R programming environment. It ameliorates a shortcoming in the current version of R (and most other statistical software packages), namely the inability to allow the user to conveniently generate interactive 3D graphics.
Since 3D objects need to be projected on a 2D display, special navigation capabilities are needed to provide insight into 3D relationships. Features such as lighting, alpha blending, texture mapping and fog effects are used to enhance the illusion of three-dimensionality. Additional desirable features for interactive data analysis in 3D are the ability to rotate objects, and to zoom in/out so as to examine details of an object, or alternatively, to view it from a distance.
The goal of the project described here was to provide a "3D engine" with an API (Application Programming Interface) designed for R. It is implemented as a portable shared library (written in C++) that uses OpenGL. The syntax of the RGL commands has been based on that of the related and familiar standard R commands, thus ensuring that users familiar with the latter can quickly learn the usage of RGL.
This paper outlines the capabilities of the of the RGL library and illustrates them using some typical statistical applications.

Keywords: R, OpenGL, graphical techniques, interactive visualization, real-time rendering, 3D graphics.

1 Introduction

The R-language (Ihaka and Gentleman 1996) is a convenient and powerful tool for statistical data analysis. The CRAN (Comprehensive R Archive Network, *http://cran.r-project.org*) provides a facility for online distribution and automatic installation of R as well as custom packages. Being an open project with many contributors CRAN continuously receives new contributed libraries which are written in a standard and well-documented format. A shortcoming of current version of R is the lack of sophisticated methods for 3D visualization. The main goal of the project outlined here is to provide an interface for R which acts as a "3D engine".

[1] Corresponding author: Oleg Nenadić, Platz der Göttinger Sieben 5, 37073 Göttingen, Germany. email: onenadi@uni-goettingen.de

Graphical visualization is an integral part of statistical modelling and data analysis. Two–dimensional plots such as scatterplots, histograms and kernel smoothers are used routinely for visualizing and analyzing data. The extension to three dimensions for visualization, though seemingly trivial, generates a number theoretical and practical challenges. Special capabilities are needed to create the illusion of three dimensionality when projecting 3D objects on a 2D display. Appearance features such as lighting and alpha-blending need to be created; the user should be able to navigate around the space and to zoom in or out.

A consequence of the quantum leap that graphics hardware has taken in recent years in terms of 3D–capabilities is that 3D applications are no longer restricted to powerful CAD–workstations; they can be carried out on current entry–level computers. Nevertheless for real-time rendering the computations need to be carried out efficiently to avoid bottlenecks. RGL makes use of OpenGL as well as an important feature R, namely the facility to call foreign code via shared libraries.

This paper describes the RGL library and gives some examples of what it currently offers in terms of 3D real-time visualization. The remainder of the paper is arranged as follows. Section 2 outlines the RGL function set. These provide a number of "low–level" operations that serve as building blocks for high–level plotting operations. Section 3 gives examples of some applied statistics graphical displays that illustrate the use of the RGL library.

2 The RGL–package

The core of RGL is a shared library that acts as an interface between R and OpenGL. In order to provide convenient access to OpenGL–features, a set of R–functions which act as an API (Application Programming Interface) was written.

2.1 The RGL functions

The RGL API currently comprises 20 functions which can be divided into six categories:

- **Device management functions** control the RGL window device. Similar to the R–functions `win.graph( )`, `dev.cur( )`, `dev.off( )` etc., they open and close devices, control the active device focus or shut down the device system.
- Unlike R graphic-functions, RGL provides the option to remove certain or all objects from the scene with the **scene management functions**.
- The **export function** enables the user to create and store PNG (Portable Network Graphics) snapshots from a specified device. These can be used, for example, to create animations in batch mode.
- The building blocks for 3D–objects are **shape functions** which provide the essential plotting tools. Primitives, such as points, lines, triangles and quads

(planes) as well as higher level objects like text, spheres and surfaces are plotted with these functions.

- **Environment functions** are used to modify the viewpoint, the background and the bounding box. Also provided is a function for adding light sources to the scenery.

- The **appearance function** `rgl.material(...)` controls the appearance properties of shapes, backgrounds and bounding box objects. The parameters have been generalized to one interface for all object types that support appearance parameters. Parameters, that do not have any influence on particular object types, are ignored.

A complete list of the currently implemented RGL functions is given in Appendix A. Further details on standard graphics options can be obtained using `help(par)` within R. The command `example()` illustrates the use of some RGL functions.

2.2 Shape functions

The *shape functions* are the essential part of RGL; they provide access to plotting primitives. The currently implemented primitives are shown in figure 1.

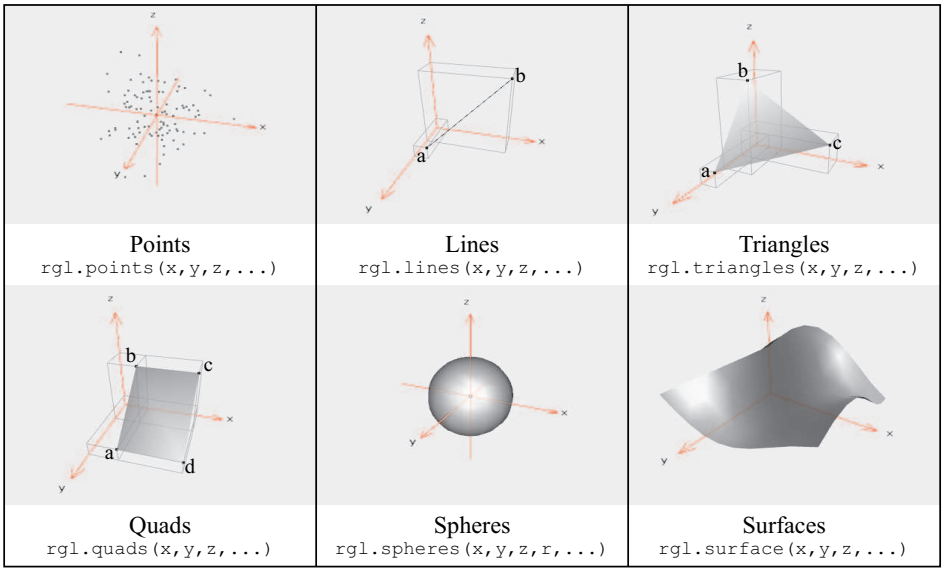

Figure 1: The 3D–primitives of RGL

- **Points** in 3D–space can be drawn with `rgl.points(x,y,z,...)`.

- 3D **Lines** are drawn with `rgl.lines(x,y,z,...)`. In this case two sets of (x, y, z) coordinates need to be specified. The first node of the line (a) is determined by the first elements of vectors `x`, `y` and `z`, while the second node of the line (b) is given by the second elements of those vectors.

- 3D **triangles** are created with the function `rgl.triangles(x,y,z,...)` in a similar way to 3D–lines. The vectors `x`, `y` and `z`, each of length three, specify the coordinates of the three nodes a, b and c of the triangle in 3D–space.

- A further extension are **quads** (planes) which can be drawn with the function `rgl.quads(x,y,z,...)`. In this case `x`, `y` and `z`, each of length four, specify the coordinates of the four nodes (a, b, c and d) of the quad.

- It is possible to construct (approximate) spheres or arbitrary complex objects using small triangles but that can be time-consuming code and (unless special care is taken) computationally inefficient. Thus, although they are not primitives, 3D **spheres** are provided for convenience via the *shape function-set*. A sphere with center (x, y, z) and radius r is plotted with the function `rgl.spheres(x,y,z,r,...)`. If `x`, `y`, `z` and `r` are vectors of length n then n spheres are plotted using a single command.

- It is possible to approximate a 3D **surface** using quads but, again this can be time-consuming and computationally inefficient. For example constructing a surface using quads based on n^2 nodes can be computed using a double loop involving the transfer of 4 nodes of the quads n^2 times. The function `rgl.surface(x,y,z,...)` offers a convenient and efficient way of constructing surfaces that avoids redundantly looping over individual quads. One simply specifies a matrix `z` of "heights" corresponding to the nodes whose coordinates are given in the vectors `x` and `y`.

The above shape functions support additional attributes, such as colors and other appearance features.

2.3 Appearance features

Features, such as alpha blending (transparency), side–dependant rendering and lighting properties can further enhance the illusion of three-dimensionality. Some selected appearance features are illustrated in Figure 2.

- **Lighting** is an important element of 3D graphical displays. Different types of light and reflective properties of objects are supported by OpenGL. Referring to the top left panel of Figure 2:
  (a) *Specular lighting* determines the light on the highlight (spot) of an object,
  (b) *ambient lighting* is the light–type of the surrounding area,
  (c) *diffuse lighting* is the type of light scattered in all directions equally, and
  (d) *shininess* refers to the reflective behavior of 3d–objects (glossy or matt).

- **Alpha blending** controls the transparency properties of 3D–objects. It is set using `alpha=x`, where $x \in [0, 1]$ is the transparency level; Setting $x = 0$ renders the objects fully transparent and $x = 1$ renders them entirely opaque.

- The use of **texture mapping** might not be initially evident. Nonetheless, various possible applications exist, which require (or at least benefit from) this feature. Referring to the top right panel of Figure 2 texture mapping (c) takes a bitmap as input (a) and wraps it over the surface of a 3D–object (b).

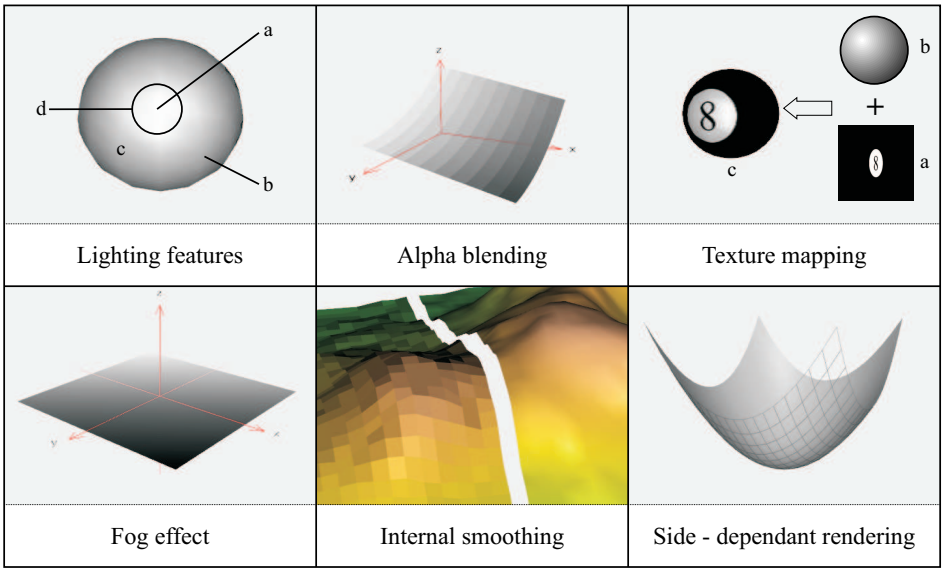

Figure 2: Appearance features

- **Fog effects** can be used to enhance the illusion of depth in a 3D–scene. Objects closer to the viewpoint appear clearer than distant objects. The strength of the fog–effect is a function of the distance to the viewpoint. Linear, exponential and squared exponential strength of effect are supported.
- **Internal smoothing** determines the type of shading applied. Referring to the middle left panel of Figure 2 the flat shading on the left part of the figure results is obtained using `smooth=F` while the right part results from the (default) goraud shading using `smooth=T`.
- **Side dependant rendering** allows the "front" and the "back" side of an object to be drawn differently. Three drawing modes are supported: solid (default), lines and points. The example in Figure 2 was created with the option `back="lines"`, so the front side of the surface is drawn with solid color while the back side of the object is displayed as a grid.

The appearance features outlined above are not essential for 3D rendering but they substantially enhance the illusion of three-dimensionality and thereby simplify typical tasks involved in exploratory data analysis (such as discovering relationships, identifying outliers) and in assessing the fit of models, as illustrated in the examples in Section 3.

2.4 The navigation system

Real interactivity would not be given if the user could not explore the three–dimensional space. The purpose of the navigation system is to provide intuitive access to navigation in 3D. A pointing device (commonly a mouse) which allows for movement in two directions, is used for travelling on a sphere surrounding the scenery (Figure

3 a). Zooming into the scene or away from it is performed by holding the right mouse button pressed and moving the mouse forward or backward.(Figure 3 b).

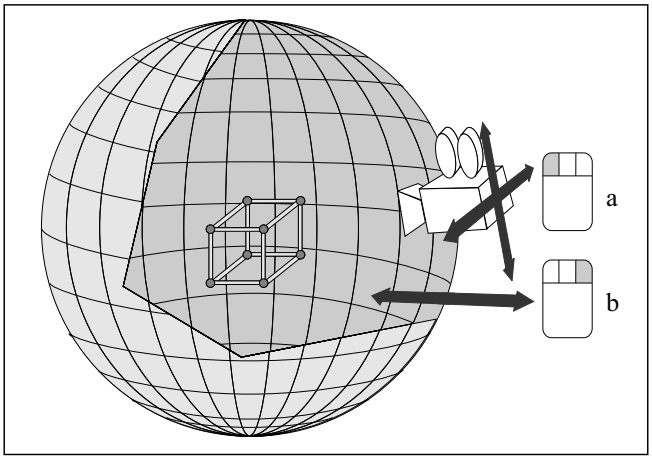

Figure 3: RGL Navigation system

The strength of the perspective distortion or field of view (FOV) is controlled by moving the mouse while holding the middle mouse button pressed. The function `rgl.viewpoint(theta=0,phi=15,fov=60,zoom=0)` enables one to set the navigation status without a pointing device. The position on the sphere is specified by the azimuthal direction, `theta`, and the colatitude, `phi`). The perspective distortion and zoom–level are adjusted with the arguments `fov` and `zoom`, respectively.

3 Examples from applied statistics

The RGL functions described above constitute the basic building blocks for more complex objects. We now give five examples to illustrate how this can be done.

Example 1: 3D-histograms
A 3D histogram can be constructed by repeatedly calling `rgl.quads(x,y,z,...)`. The computations are performed with standard R–commands, while the actual drawing is carried out with RGL.

The histogram shown in Figure 4 c) is composed of bins (Figure 4 b) constructed with 6 quads (the sides in figure 4 a). Thus a convenient method of constructing 3D histograms it is to first prepare a new "primitive", say `bin3d`, that uses quads having the required attributes and then to write a function that simply compiles the bins into a histogram. The input parameters to the latter function are two vectors of coordinates, `x` and `y`, that specify the histogram breaks, and a matrix `z` whose entries specify the heights of the bins.

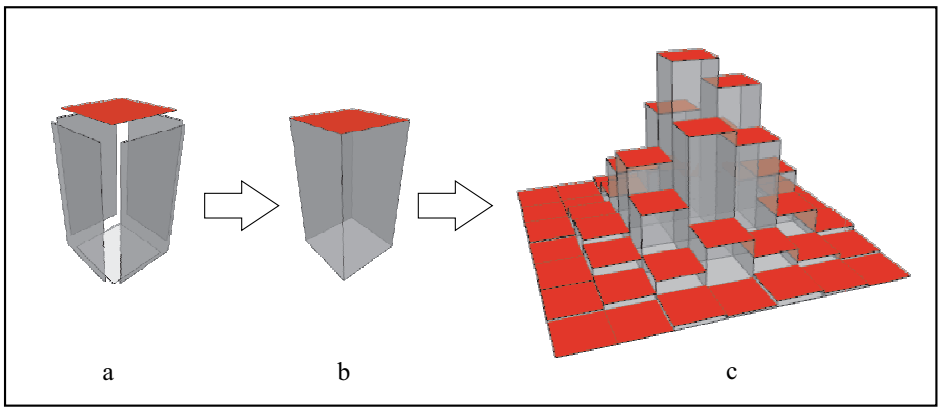

Figure 4: 3D–histogram

Example 2: Two–dimensional densities
Figure 5 shows two bivariate density functions together with spheres that represent the observations. A kernel estimate of the density is displayed as a transparent surface; a fitted bivariate normal distribution is shown as a wireframe. Side-dependent rendering (see Figure 2) was used to display the two densities. Details of the fit of the bivariate normal distribution can be examined by navigating in the space and comparing that distribution with the non-parametric kernel estimate.

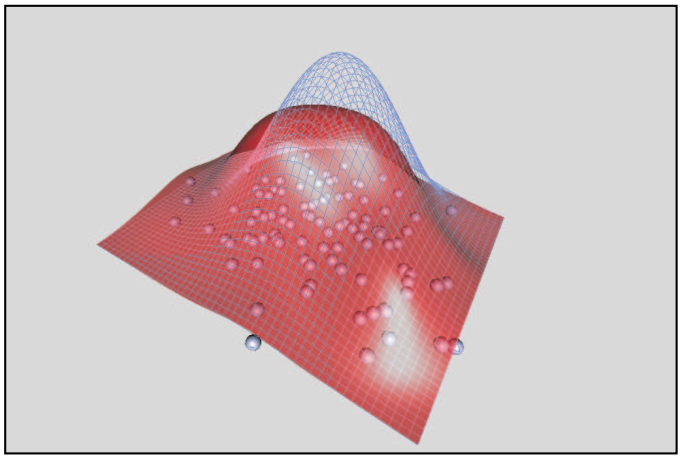

Figure 5: Visualizing and comparing bivariate densities.

Example 3: Three–dimensional densities
Appropriate use of transparency makes it possible to represent three-dimensional probability density functions reasonably convincingly. Figure 6 shows the density function of a 3D normal distribution. The value of the density (the 4th dimension) is indicated by the transparency of particles placed on a fine regular 3D–grid. The illusion of higher dimensionality is enhanced when one navigates around the space and makes use of zooming.

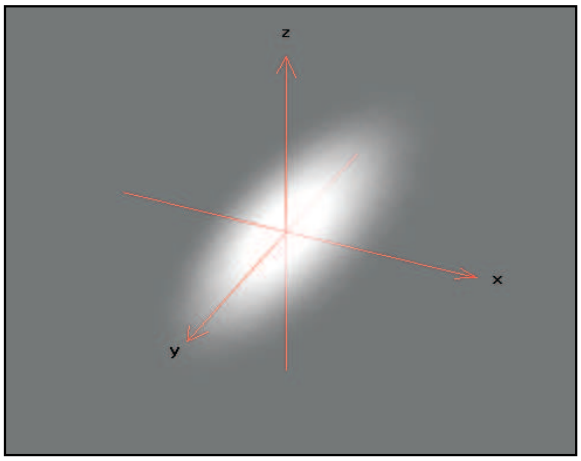

Figure 6: Three–dimensional normal pdf.

Example 4: Representing animal populations
To assess the properties of animal abundance estimators under various conditions it is useful to test the estimators on generated data (see, e.g. Borchers, Buckland and Zucchini, 2002). In particular the behavior of estimators can depend, among other things, on the distribution of animals in the survey region, on the size and composition of the groups (herds) and on their exposure (how easily they can be detected or captured. Figures 7 a) and b) display the specified population density that was used to generate a population comprising several groups that are displayed as spheres. The population density is displayed as a topographic surface – regions with high density are displayed as mountains and those with low density as valleys. Regions having zero density are shown as "rivers". The characteristics of a group, namely its size, type and exposure are indicated by the sphere's radius, color and transparency level, respectively.

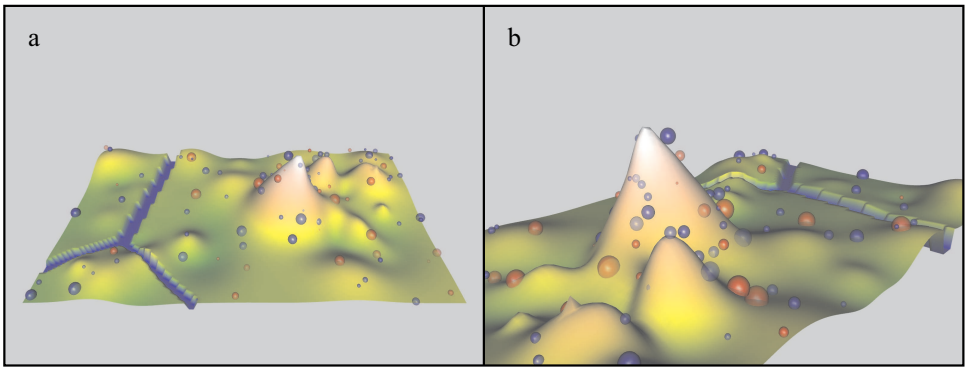

Figure 7: Simulated animal abundance.

Example 5: An application in hydrology
The data for the following 3D–visualization is taken from the South African Rainfall Atlas (Nenadić, Kratz and Zucchini, 2001). Figure 8 shows a topographic map of Southern Africa (South Africa, Lesotho and Swaziland). The mean annual rainfall in the region is displayed in form of clouds. The thickness and (lack of) transparency

of the cloud above a site is used to indicate the magnitude of the mean rainfall at the site.

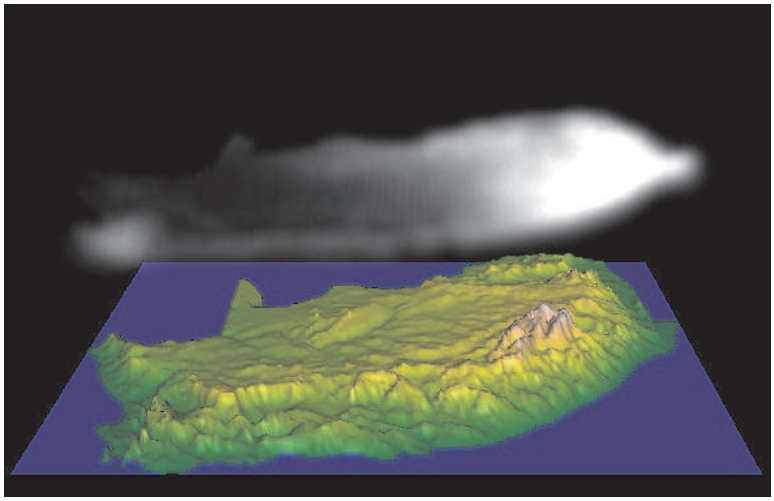

Figure 8: Topographic map of Southern Africa with the mean annual rainfall represented by clouds.

4 Summary and outlook

The main goal of the project described in this paper was to provide R users with an additional set of graphical tools to create interactive three–dimensional graphical displays, namely a flexible and convenient generalized interface. The RGL library provides the building blocks to facilitate three–dimensional, real-time visualization in R.

Although RGL is still in development stage it already offers numerous capabilities that extend the current R graphics package. Some elements, such as mesh–primitives or even NURBS have not been implemented. However, with the basic building blocks that RGL provides plus a little creativity, users can already construct their own types of objects and functions and display these interactively in three-dimensions.

In this paper we have focused on the core elements of the package and have given some typical statistical applications to illustrate the capabilities of the current implementation of the library. Details relating to software–architecture have not been given here; a report on that aspect of the project is in preparation.

Apart from enhancing portability (an issue that we have not discussed in this paper) future plans include providing support for dynamic graphics and X3D/VRML support. Feedback and suggestions for improvements and extensions are very welcome and will certainly be considered for future development.

5 References

Borchers, D.L., Buckland, S.T. and Zucchini, W. (2002), *Estimating Animal Abundance: closed populations*. Springer–Verlag, London.

Ihaka, R. and Gentleman, R. (1996), R: A Language for Data Analysis and Graphics, *Journal of Computational and Graphical Statistics*, 5(3), 299–314.

Nenadić, O., Kratz, G. and Zucchini, W. (2002), The Development of a Web–based Rainfall Atlas for Southern Africa, Short Communication, *Compstat 2002*, Berlin.

A Appendix: The RGL–functions

function	*description*
DEVICE MANAGEMENT:	
`rgl.open()`	Opens a new device.
`rgl.close()`	Closes the current device.
`rgl.cur()`	Returns the number of the active device.
`rgl.set(which)`	Sets a device as active.
`rgl.quit()`	Shuts down the subsystem and detaches RGL.
SCENE MANAGEMENT:	
`rgl.clear(type="shapes")`	Clears the scene from the stack of specified type ("shapes" or "lights").
`rgl.pop(type="shapes")`	Removes the last added node from stack.
EXPORT FUNCTIONS:	
`rgl.snapshot(file)`	Saves a screenshot of the current scene in PNG–format.
SHAPE FUNCTIONS:	
`rgl.points(x,y,z,...)`	Draws a point at x, y and z.
`rgl.lines(x,y,z,...)`	Draws lines with nodes (x_i, y_i, z_i), $i = 1, 2$.
`rgl.triangles(x,y,z,...)`	Draws triangles with nodes (x_i, y_i, z_i), $i = 1, 2, 3$.
`rgl.quads(x,y,z,...)`	Draws quads with nodes (x_i, y_i, z_i), $i = 1, 2, 3, 4$.
`rgl.spheres(x,y,z,r,...)`	Draws spheres with center (x, y, z) and radius r.
`rgl.texts(x,y,z,text,...)`	Adds text to the scene.
`rgl.surface(x,y,z,...)`	Adds a surface defined by two grid mark vectors x and y and a surface height matrix z.
ENVIRONMENT SETUP:	
`rgl.viewpoint(theta,phi, fov,zoom,interactive)`	Sets the viewpoint (`theta`, `phi`) in polar coordinates with a field–of–view angle `fov` and a zoom factor `zoom`. The logical flag `interactive` specifies whether or not navigation is allowed.
`rgl.light(theta,phi,...)`	Adds a light source to the scene.
`rgl.bg(...)`	Sets the background.
`rgl.bbox(...)`	Sets the bounding box.
APPEARANCE FUNCTIONS:	
`rgl.material(...)`	Generalized interface for appearance parameters (cf. Section 2.3).

Table 1: *The 20 RGL functions which constitute the API, grouped by category. The usual graphics parameters are permitted as arguments to functions which have "..." in their calling sequence. (For details see* `par()` *in the R base library.)*