RGL in 2007

Duncan Murdoch
University of Western Ontario

Daniel Adler
University of Goettingen

May 11 2007

Abstract

In this paper we describe changes to the RGL package.

1 Introduction

Murdoch (2001) presented a 3D graphics package for R at DSC 2001. This package used the OpenGL library to render shaded polygons, lines and points in dynamic displays. Unfortunately, it was written in Delphi, a Pascal variant which at the time existed only on Microsoft Windows. Independently, Adler (2003) developed a package with the same overall goals, also using OpenGL, but coded more portably in C++. He and his supervisors presented this at the DSC 2003 meeting and at Interface 2003 (Nenadic et al., 2003), and he later won the John M. Chambers Statistical Software Award for 2003 for the work. At the DSC 2003 meeting the present authors met and decided to merge their work into a single package, keeping the name `rgl`, based on Adler's work. Murdoch's package was renamed to `djmrgl`.

RGL is an interface to the OpenGL library. We refer the reader to the references above and the manuals (OpenGL Architecture Review Board et al., 2005) for a discussion of OpenGL. In this paper we will concentrate on the changes to the Adler (2003) package that have occurred since the projects were merged.

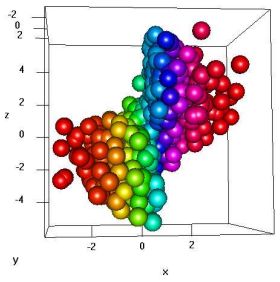

Figure 1: The `plot3d()` function with `type="s"` for spheres.

2 *3d interface

During the fusion of `djmrgl` and `rgl`, the API was extended with a generic 3D API. The functions in this API are named in the pattern `*3d`. The idea behind the new API is to establish a generic 3D API that is stable to the R user, while at the same time, it will be open to developers to implement different 3D devices such as Scene dataformat generators and more implementations or bindings to existings 3D engines. While the legacy `rgl.*` functions are still available, the use of the generic 3D interface is encouraged.

The semantics of the `*3d` interface are similar to those of classic R graphics. Changes made to material properties in `*3d` calls remain local to those calls, and the new `material3d()` function allows individual material defaults to be set (similar to the `par()` function).

Higher level `*3d` functions were also added analogously to the classic plot functions: `plot3d()`, `axes3d()`, etc. See Figure 1. Lower level functions to add objects to plots were also added: `points3d()`, `lines3d()`, and so on.

By default, the `plot3d()` function rescales coordinates so that the bounding box around the scene appears to be a cube. More general control of the aspect ratio is also possible with the `aspect3d()` function.

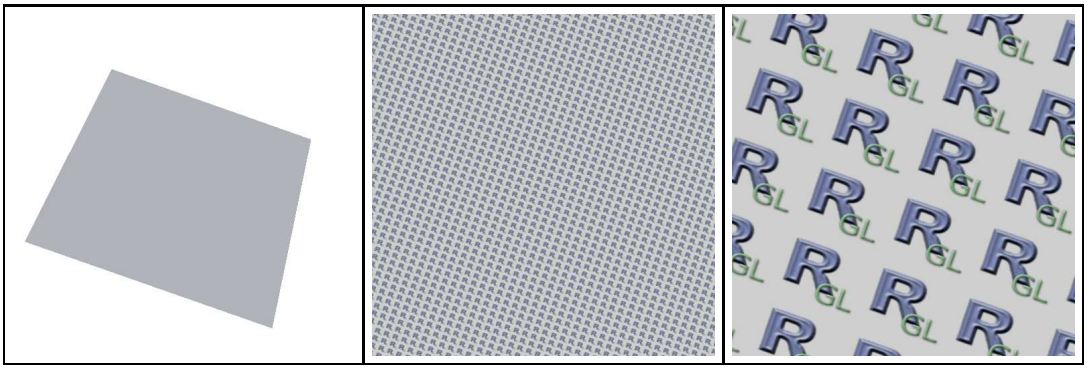

Figure 2: A square with a mipmapped texture, at the original size, and magnified 10 and 100 times.

3 Rendering Improvements

In computer graphics, "textures" are images mapped onto surfaces. The images may appear as though printed there, or may modify other properties of the surface.

Mapping textures on surfaces is difficult. If the surface normals are nearly orthogonal towards the view direction, poor quality texture mapping effects can appear. There are also difficult problems involved in rescaling the images as the rendered size of the image is changed in real time by the user.

Mipmapping is an advanced technique in texture mapping that uses multiple pre-computed scaled down versions of the same texture image in the texture sampling process. These speed up high quality rendering.

In `rgl`, mipmapping can be enabled by the material parameter `texmipmap`. Additionally `texminfilter` and `texmagfilter` material parameters specify the filtering method for rescaling. The best quality for shrinking an image is achieved using `texminfilter="linear.mipmap.linear"`, which uses linear interpolation between pixels within two rescaling levels, and then linear interpolation between them to select the texture value (Figure 2). For expanding the image, the best quality is achieved by `texmagfilter="linear"`, to use linear interpolation between pixels in the original image. Other possibilities are to replace the linear interpolation with the selection of the nearest pixel or magnification level; these are usually faster, but show more rendering artefacts.

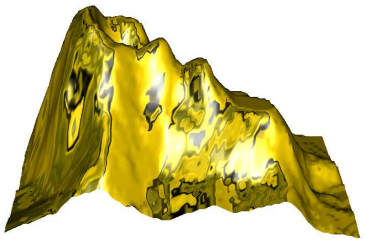

Figure 3: A "solid gold" rendering of the Maunga Whau volcano.

Textures can also be used via "environment mapping" to render a surface so that it appears to be a perfectly reflective mirror. Typically, a panorama image (e.g. the same image as for the scene background spheres) is chosen as the texture image, but any image with a good mix of light and dark detail works quite well. The image is wrapped on a virtual sphere surrounding the shape. The texture coordinates for the shape's surfaces are then calculated using the viewpoint and surface normals to choose the reflection on the virtual sphere. As this texture mapping process is highly dependent on the view direction, the user gets the illusion of a perfect mirror (Figure 3). When synchronizing the texture of the background sphere and the environmental map of some objects an illusion of perfect mirroring can be achieved. In `rgl` this feature can be enabled by setting the logical material parameter `texenvmap`.

Besides better support for rendering of textures, `rgl` has recently gained more general support for linking them to user scenes. Users can now give texture coordinates corresponding to each vertex of polygons being rendered, in order to show effects such as repeated textures (Figure 2) or transformations from rectangular coordinates to spherical ones (Figure 4). Similarly, users can specify the normals to the surface: this allows the two edges of the map in Figure 4 to be joined without a visible edge.

OpenGL allows solid surfaces to be rendered in any order, and based on their apparent depth in the screen (their "z-order") it will only display those that are closer to the viewer, giving automatic hidden surface removal. However, transparent surfaces must be handled with more care, as they must

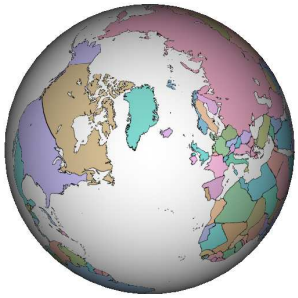

Figure 4: A rendering of the `mapdata` package `worldHires` map on a globe.

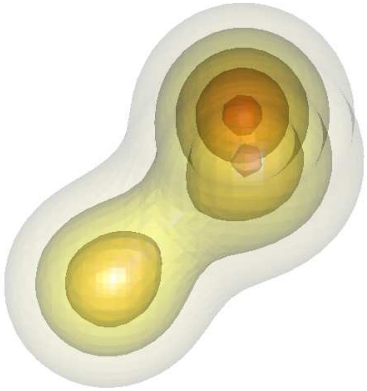

Figure 5: 3D contours produced by the `misc3d` package and rendered in `rgl`.

be blended on top of each other in z-order (i.e. more distant surfaces must be rendered first). `rgl` has much improved transparency handling. It renders transparent objects (each corresponding to a single `rgl.*` call) in the correct z-order relative to each other, and the internal components of each object are rendered in the correct z-order.

Artefacts are still possible when multiple intersecting transparent objects occur in the same scene because `rgl` does not interleave the rendering of the internal components of different objects. If a user wants to render such shapes (e.g. Figure 5), then all of the surfaces should be pre-computed and rendered as one object.

4 Other changes

Besides the major changes listed above, `rgl` has had a number of smaller additions and changes:

- There is now support for working with homogeneous matrices, used in perspective calculations, and to specify rotations, translations and scaling of geometric objects.
- Mouse handling is more flexible, with a number of optional behaviours configurably associated with each mouse button: trackball rotation, one-axis rotation, etc.
- There is basic low level support for item selection using `select3d()`.
- Adler's original implementation of `rgl` maintained objects on a stack, and provided the `rgl.pop()` function to delete the most-recently drawn object. Murdoch's implementation implemented a hierarchical organization of objects, whose contents could be modified from R. The current `rgl` falls somewhere between these: objects are maintained on a stack, but each primitive has a numerical identifier, and `rgl.pop()` can delete by identifier.
- In Windows, `rgl` now displays its windows within the MDI frame if R is running in MDI mode.
- Missing values encoded as `NA` are now handled reasonably well.
- A major effort has been put into making `rgl` as portable as R across platforms. This has been done by using the GNU build tools (`autoconf`, etc.) to help write configure scripts. MacOSX presented a particular challenge: depending whether R is run in the GUI or not, completely different graphics systems are used, and `rgl` needs to link against two different libraries implementing OpenGL. On that platform we currently build two separate shared libraries, and decide at package load time which one to load.
- The `rgl.snapshot` and `rgl.postscript` functions have been improved, so that images can now be saved to disk reasonably reliably.

5 What is still to come

Currently font support in `rgl` is very limited. Text can only be displayed on one font and one size. Work is under way to use FTGL (a cross-platform font support library for OpenGL); anyone interested in helping with this is asked to contact us. We hope that this effort will also allow full Unicode support to be achieved.

We are also planning in the future to allow better control of the layout of `rgl` scenes within windows, and of those windows on screen.

- R3D specification

References

Adler, D. (2003). *Interactive Visualization of Multi-Dimensional Data in R Using OpenGL*. Diplomarbeit, University of Goettingen.

Murdoch, D. J. (2001). RGL: An R interface to OpenGL. In Hornik, K. and Leisch, F., editors, *Proceedings of the 2nd International Workshop on Distributed Statistical Computing, Vienna, March 2001*. ISSN 1609-395X, http://www.ci.tuwien.ac.at/Conferences/DSC-2001/Proceedings.

Nenadic, O., Adler, D., and Zucchini, W. (2003). RGL: A R-library for 3D visualization with OpenGL. In *Computing Science and Statistics, Proceedings of the 35th Symposium on the Interface*. http://www.galaxy.gmu.edu/interface/I03/I2003HTML/AdlerDaniel.html.

OpenGL Architecture Review Board, Shreiner, D., Woo, M., Neider, J., and Davis, T. (2005). *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 2*. Addison-Wesley Professional.