DSC 2003 Working Papers
(Draft Versions)
http://www.ci.tuwien.ac.at/Conferences/DSC-2003/

A Framework for an R to OpenGL Interface for Interactive 3D graphics

Daniel Adler and Oleg Nenadić *

Abstract

We describe a framework for providing interactive 3D graphics in R. The key component is the interface between R and OpenGL, which provides a set of commands for specifying objects and operations that enable three-dimensional graphical visualization. An alpha version of the software, the core of which is written in ANSI-compliant C++, is available for testing and feedback (Win32 and X11).

An important goal in the design has been to facilitate portability to different operating systems. An object-oriented approach is used throughout. A simple and intuitive user interface for navigating in 3D using a pointer device is provided by moving the viewer around the surface of a sphere that surrounds the relevant space; the view is focussed on the centre of the sphere.

The focus of the current implementation is to manipulate 3D "primitives" (for example points, lines, triangles, quads, spheres, text, etc.) which constitute the building blocks for more complex 3D objects (such as histograms, perspective plots, scatterplot, axes, etc.). A number of attractive OpenGL features, such as multiple lighting, fog, texture-mapping, alpha-blending (transparency) and side-dependent rendering are accessible directly from R via functions that control shapes and appearance. Further functions control device-and-scene management (opening closing windows, clearing selected objects), environment setup (setting lights, bounding boxes, viewpoints) and export (making and exporting snapshots). The calling sequences are similar to those used in the existing and familiar R graphics functions, such as `persp`.

*Institut für Statistik und Ökonometrie, University of Göttingen

1 Introduction

RGL[1] is an add-on library that extends the R system[2] with a 3D visualization device system with interactive viewpoint navigation. It is implemented in C++ using OpenGL[4] as the realtime rendering backend.

A major goal of the project was to develop an appropriate abstraction to the underlying operating system services (such as the windowing system and OpenGL), so that porting to major R platforms with slightly different base services is possible. The software can be delivered as an add-on package with no modification to the original R source tree.

The RGL device system runs as an autonomous subsystem in the R run-time process as the R device interface has been designed for plotting devices and lacks interface requirements for 3D visualization.

The render engine and supported data objects are optimized for large dataset geometry and appearance visualization using several speed-up techniques such as display lists, fake visualization tricks and eye-candy special effects.

The library has been ported to the Win32 and X11 platforms so far, with support for MinGW, GCC and Visual C++ (win32 only) compiler systems.

2 Visualization Model

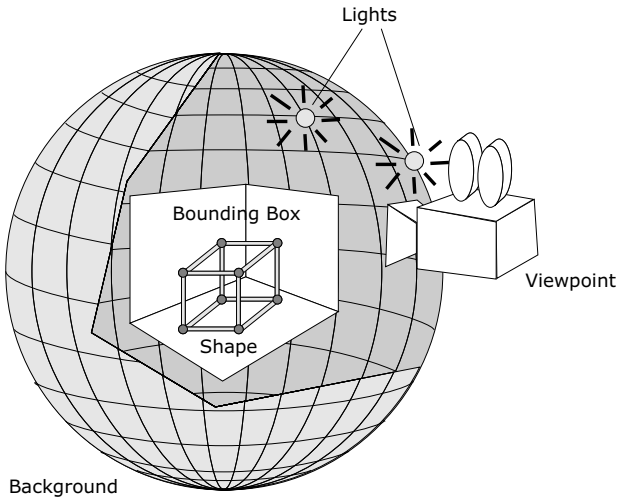

Figure 1: A typical scene

The visualization model uses five base object types, that describe a complete

[1] Accidently, our project name collides with an R package written by Duncan Murdoch[3]. Allthough both projects deal with the same problem, our approach differs substantially in the architectual part with the most evident difference beeing the cross-platform portability.

scene (see figure 1).

- *Shapes* are basic geometry informations and constitute basic building blocks. Currently eight basic shape types are supported: Points, lines, triangles, quadrangles, texts, spheres, surface and 3d sprites (screen-aligned quadrangles). Several attractive appearance properties can be applied to the geometry such as solid colouring, lighting material, texture-mapping, alpha blending, gouraud shading and side-dependant fill mode.
- The *Viewpoint* surrounds the overall geometry is focused on the center. Interactive navigation is provided using the pointing device which controls the viewing direction (using polar coordinates), zoom and field-of-view.
- Directional *Light* sources define the lighting conditions.
- The *Bounding Box* tracks the range of the geometry and optionally displays axis labeling.
- The *Background* provides either solid colour or an environmental sphere enclosing the scene.

3 Programming Interface

The RGL API comprises 21 R commands. Care has been taken to be semantically compatible with the R plotting facilities where appropriate.

- *Device Management Functions* provide support for multiple devices. Devices are automatically opened, in case one is required and none is already opened.
- *Scene Management Functions* provide support for undo operations.
- *Shape Functions* consitute the basic building blocks that can be combined to build complex visualization scenarios.
- *Environment Setup Functions* provide control to background and bounding box decorations, axis labeling, lighting conditions and viewpoint.
- *Export Functions* allow for making still picture exports.
- One *Apperance Function* is used internally by API functions to setup appearance properties in a generic way.

Most functions call a counter-part C++ function in the shared library. A detailed overview of the API is appended at the end of this paper.

4 Architecture

To allow a seamless integration into the R system, the software design requires a high degree of abstraction, so further to achieve the overall design goal of cross platform portability. Due to the fact that R lacks a portable interface to the windowing system and OpenGL, our architecture provides these services.

4.1 Modules

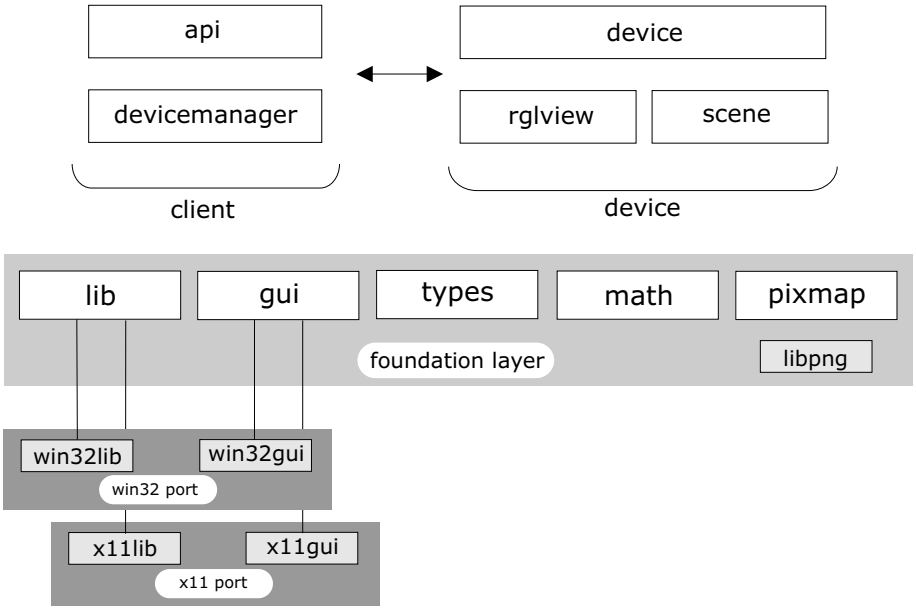

Figure 2: Architecture

Figure 2 gives a brief overview of the software modules involved in the architecture. The foundation layer represents the platform abstraction which consists of five core services.

4.2 Foundation Layer

The *lib* module provides library initialization and destruction entry-points where a platform specific integration strategy into the R system is implemented. The *gui* module provides a portable object-oriented framework of components to build graphical user-interfaces with support for OpenGL contexts and fonts. Back-end modules represent the peers to the actual windowing system. Matrix/Vector arithmetic classes with support for homogenous transformations, a common technique in realtime rendering, are implemented in the *math* module. Common fundamental datatypes and structures can be found in the *types* module. Datatype abstraction for pixmap import and export is provided in the *pixmap* module. A png pixmap datatype handler is currently implemented.

4.3 Device system

The client infrastructure and device is implemented on top of this layer.

The *api* and *device manager* module constitute the client, while the *device* provides the interface and composes the device semantics using the graphical user-interface component implemented in *rglview*. The core 3D engine is implemented in the *scene* module.

4.4 Data model

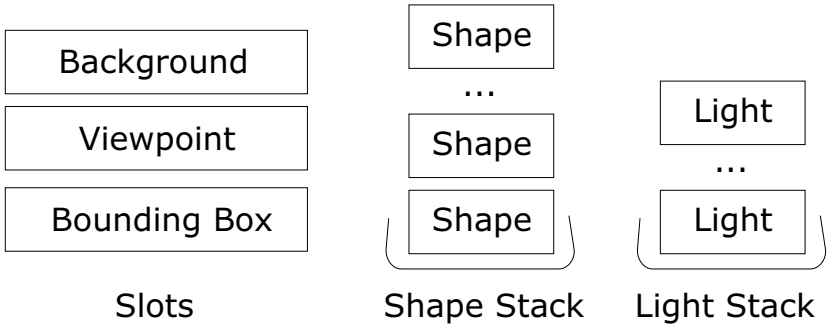

Figure 3: logical data model

The scene description is stored in a composite object model, which gets evaluated by the render engine at a highly frequent rate. Figure 3 shows a logical data model. Multiple shapes and lights are managed simulatenously, using a stack semantic. Three additional object slots are managed which hold one object at a time. Slot objects are replaced, while stack objects can be popped or optionally cleared at once.

4.5 Minimalistic GUI toolkit

The GUI abstraction model has been designed using software patterns. Software patterns support decision making in software design and might prevent from common pitfalls. The abstraction model can be characterized by two common patterns, namely the *Bridge* and the *Factory* pattern taken from [1] (see figure 4). We separate the windowing system abstraction and implementation using two class hierarchies. The abstraction hierarchy represents the base for user-interface component development. The implementation interface hierarchy provides core base services required to get the user-interface working (e.g. a specific windowing resource and further specializations). A concrete *factory* object encapsulates concrete implementation instances of that platform, so that the abstraction uses the factory interface to initiate an implementation object. Both hierarchies are *bridged*, so that windowing events are injected into the abstract objects and core service requests (e.g. drawing) are provided by implementation objects. The bridge gets built by passing an abstract factory interface at early startup to the abstract gui classes which constructs their implementation counter-part at run-time.

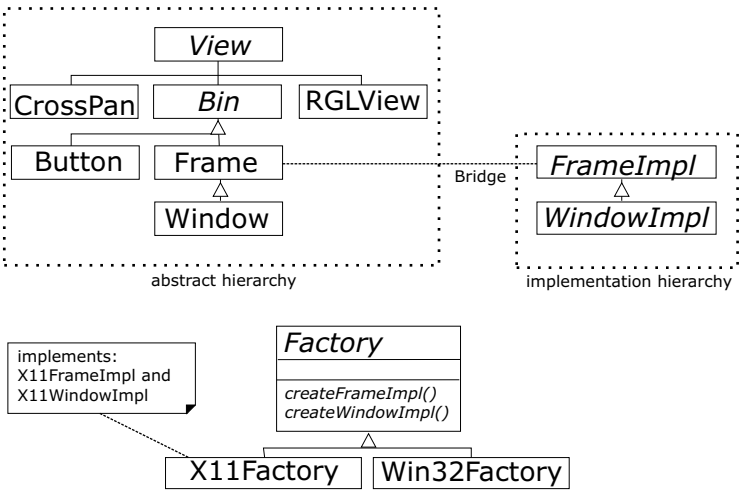

Figure 4: gui abstraction model

Platform ports provide their own factory interface implementation (e.g. X11Factory) which create concrete implementation objects(e.g. X11WindowImpl) providing implementation interfaces (e.g. WindowImpl).

Software built on that architecture deals with abstract gui classes, which themself use abstract interface classes such as Factory and WindowImpl.

4.6 Render engine

The render engine and data model has been implemented using a class hierarchy depict in figure 5. Rendering is performed using polymorphisms. Appearance information is implemented in the *Material* class which is aggregated by the *Shape* and *BBoxDeco* classes. A central class, named *Scene* class manages the data model and implements the overall rendering strategy.

5 Conclusion

The RGL library has been ported to the X11 and Win32 platform so far. The presented architecture and software design provides a good starting point for further development. As this project is in early stage, many improvements are planned and will be outlined briefly.

The core rendering engine requires several improvements. Transparent materials are not handled correctly due to the fact, that we render faces in no order using the depth buffer (provided by the OpenGL interface). A correct display would require the faces to be sorted and rendered in distance order. A hybrid rendering architecture with support for DirectX (win32 only) and a C++ template-based

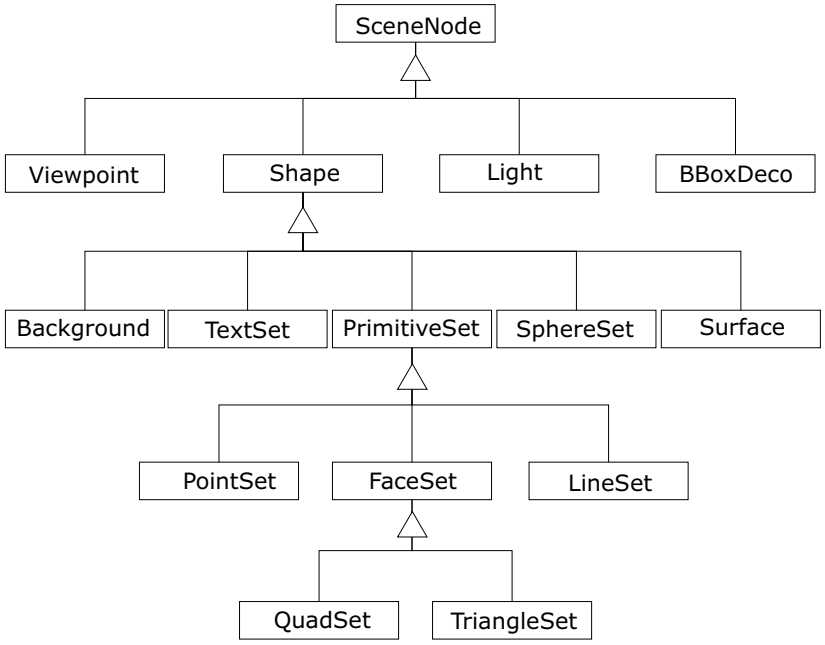

Figure 5: Class Hierarchy

software shader is in discussion.

The portable GUI abstraction is in a rudimentary state, providing the essentials for opening a Window, creating an OpenGL context and dispatching events. We plan to implement a tiled view navigation interface with four separate views (using front, side, top and perspective projection).

Support for dynamic graphics and animation is planned. The 3D web standard VRML97 and its successor X3D are very promising for an open and extendable browser-based rendering architecture.

References

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, United States of America, 1994.

[2] Ross Ihaka and Robert Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996.

[3] Duncan Murdoch. *RGL: An R Interface to OpenGL*. DSC 2001, Vienna, 2001.

[4] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide*. Addison-Wesley, United States of America, 1999.

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,...)`	Add points at (x, y, z).
`rgl.lines(x,y,z,...)`	Add lines with nodes (x_i, y_i, z_i), $i = 1, 2$.
`rgl.triangles(x,y,z,...)`	Add triangles with nodes (x_i, y_i, z_i), $i = 1, 2, 3$.
`rgl.quads(x,y,z,...)`	Add quads with nodes (x_i, y_i, z_i), $i = 1, 2, 3, 4$.
`rgl.spheres(x,y,z,r,...)`	Add spheres with center (x, y, z) and radius r.
`rgl.texts(x,y,z,text,...)`	Add texts at (x, y, z).
`rgl.sprites(x,y,z,r,...)`	Add 3D sprites at (x, y, z) and half-size r.
`rgl.surface(x,y,z,...)`	Add 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 21 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.)*