Interactive visualization of multi-dimensional data in R using OpenGL

6-Monats-Arbeit im Rahmen der Prüfung für Diplom-Wirtschaftsinformatiker an der Universität Göttingen

vorgelegt am	**09.10.2002**
von	**Daniel Adler**
aus	**Göttingen**

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The visualization of empirical and simulated data in conjunction with function graphs is a common technique for scientists to identify correlations in multivariate data, to compare distributions with known distribution functions and to illustrate state facts.

2D visualizations such as scatterplots, distribution and density function plots, histograms and pair plots are useful graphical techniques for interactive data analysis.

3D visualizations provide an additional coordinate axis. Due to the fact that 3D visualizations are projected on 2D screen displays, *interactive viewpoint navigation* is required to get an adequate overview by exploring the 3D world. In contrast, 2D visualizations do not require special navigation facilities.

Three variable visualizations and the usage of modern graphics drawing techniques like transparency give scientists abilities at hand to analyse multivariate data.

The current R graphics capability lacks the ability to interactively visualize data in 3D and has limited support for 3D graphics plots. Interactive visualization requires a certain degree of graphics rendering speed which optimized engines should deliver.

This work documents the development of a R package named "RGL". It contains an interactive visualization device system with an R programming interface. RGL uses the OpenGL library as the rendering backend providing an interface to graphics hardware. Most graphics hardware vendors pro-

vide an OpenGL driver for their hardware systems and even software-only implementations exist.

1.1 Approach

The R system is widely used throughout the statistical community. RGL extends R with a graphics device system that includes new graphical capabilities. By adapting some common programming interface designs from the current R graphics system, users need less time to study the usage of RGL.

This leads to the approach of analysing R graphics capabilities and its programming interface to derive requirements and guidelines for the software design.

The software concept evolves using a scene database oriented approach to design the functionality.

The development has been done using the methodology of object-oriented software development and an object-oriented programming language. The advantage is a solid transition from the software concept to design, and finally to implementation. The object-orientation provides techniques that structure software systems logically using modularization, abstraction, data encapsulation and polymorphism, which leads to a reduction of complexity. The methodology of *Software Patterns* has been applied to design a solid architecture.

1.2 Further reading

Chapter 2 gives an overview of the R environment and a brief introduction into key features of the R language. A short overview of the graphics capabilities and programming interface is presented. Futhermore, the extension mechanism of R is discussed.

Chapter 3 gives an introduction into the field of interactive visualization. Real-time rendering will be outlined together with an introduction to OpenGL.

Chapter 4 describes the functionality of the device, the graphics rendering model and explains the application programming interface using a top-down

approach.

In Chapter 5 the development process is presented. The chapter starts with an introduction to the object-oriented methodology including the C++ language, UML notations and *Software Patterns*. The R extension mechanism via shared libraries and a discussion about windowing systems give the outline for the architecture design. A detailed bottom-up description on a module- and class-level with details on method implementations round up the C++ implementation. The RGL API implemention in R completes the software system documentation.

Chapter 6 gives some examples to illustrate features of RGL package.

Chapter 7 gives a summary and a future outlook of further development plans.

Chapter 2

R

R is an open-source implementation of the *S* and its successor *Splus*. It is a statistical computation environment with an interpreted programming language, an interactive command line interface, a graphics plotting device system, a documentation system and a package mechanism. Add-on packages extend the system and a network supports the distribution. It has been ported to three major platforms (Microsoft Windows, Apple Macintosh and to the X11 windowing system running across a variety of operating systems, including Linux and BSD derivates).

2.1 Language

The *R language* is a dialect of *S*, which was designed in the 1980s and has been in widespread use in the statistical community. This section focuses on some remarkable properties of the language. Detailed information about S is given in [3]. The R Language Reference is distributed with the R software and is available online. [18]

The R language is an interpreted language used with an interactive command line interface. R provides different data types to store data in memory.

Vectors represent variable number of data elements of a specific storage data type. Supported storage data types are *logical*, *integer*, *double*, *complex* and *character*.

Lists are generic vectors containing elements of varying data types. They

can be attributed with *names* used for named indexing.

The *function* data type is the elementary execution unit that extends the R language with new functionality. Functions are defined using an argument list for input data and a body of R statements implementing the functionality probably providing an object as return value. The arguments can have default values. Functions are called using the object name with arguments that are given either in order or can be given as *tagged arguments*. That is, the data to be passed to the function is labeled using the form `label=value`. When combing default values and using tagged arguments, the number of arguments and its position in the list can vary.

The "..." (dot-dot-dot) data type is a *List* with unknown elements. It is commonly used in function argument list definitions to dispatch arguments to different specialized functions. Functions can pass the "..." object to subfunctions where it gets actually evaluated. When providing a standardized list of arguments for a group of public functions, the "..." object can be used to dispatch the arguments to an internal function that actually evaluates the arguments. When changing the interface, only the internal function must be modified.

R supports object-orientation using generic functions and specializing functions for a special class. Generic functions dispatch the control flow to specialized functions depending on the class of an object.

A *recycling rule* is used by operators that requires its operands to be of the same length. In case that the operands differ in length, they are recycled to the length of the longest.

2.2 Graphical subsystem

The graphical subsystem consists of a device system, device drivers for different plotting devices and an API [1].

[1] Application Programming Interface

2.2.1 Graphics plots

The R graphics plotting facility provides a variety of graphs for data analysis. They can be divided into low-level and high-level functions for plotting. A generic interface for setting graphical parameters is supported across the majority of commands using the `...` data type.

Many plotting functions are generic functions with a default function implementation which are suffixed with '`.default`'.

The `plot()` function is a generic plot function which produces different graphs depending on the class of the given data.

Multivariate data is plotted using `pairs()` providing two specialized pair plots. `qqnorm()`, `qqline()` and `qqplot()` are distribution-comparison plots. `hist()` produces histograms and provides two specialized histogram plots. `dotchart()` constructs a dotchart of the data. `image()`, `contour()` and `persp()` are used to plot multivariate data using three variables.

The low-level plotting commands give full control over the complete graphics tasks and can be used to implement new types of graphs or to extend the graph. Display limits on a per axis base must be defined at the beginning of a plot and are fixed throughout the session. If the graphics plot hits the display limits, it gets clipped.

Interaction commands can be used to identify (`identify()`) or locate (`locate()`) datasets in an existing plot using the pointing device of the windowing system.

`plot.new()` and `frame()` complete the current plot, if there is one and start a new plot.

The `par()` command is used to set or query graphical paramaters. A group of parameters is defined as read-only and can not be modified. The other parameters can be set by `par()` explicitly or can be passed to `plot()`, `points()`, `lines()`, `axis()`, `title()`, `text()` and `mtext()`.

2.2.2 Devices

Graphics devices in the sense of R are graphics plotting devices. The R devices can be divided into interactive and batch mode devices. Interactive

devices immediately display plots, while batch mode devices output the plot to a file. A variety of device drivers are implemented. All supported windowing systems (currently Win32, Macintosh, X11) provide an interactive device driver that is capable of displaying the output using the graphics facilities of the windowing system. Batch mode devices are provided for the following output file formats: postscript, pdf, pictex, png, jpeg, bmp, xfig and bitmap. R devices can be configured to contain several plots layouted on the display. This is useful for comparing multiple plots.

A sample plot session would be carried out like this:

1. Open a device
2. Optionally setup device parameters (e.g. multiple plots)
3. Start a new plot by setting up axis limits
4. Post a sequence of graphics plotting instructions
5. Optionally, start a new plot
6. Close device

R has an interesting behaviour for interactive plots. In case of a new plotting command, it automatically opens an interactive device if none is already opened.

2.3 Documentation

R is delivered with comprehensive documentation about the language[18], writing extensions[19] and data import/export[17]. Functions, data sets and general features are documented in the *Rd documentation format*, which is part of the R distribution. The Rd format is a subset of LaTeX, representing a meta-format for a variety of documentation formats. Transformation tools are provided that generate HTML, LaTeX, pdf, Windows HTML Help Format and R manual pages. The R manual pages are viewed by a browser component of the R system. The Rd format provides the writing of examples when documenting functions. Additionally the `example(`*`topic`*`)` command can be issued, which will execute example code in the manual page indexed by *`topic`*.

2.4 Extensions

R extensions are delivered as *R packages*. A package must provide a description. Optionally, it can contain R functions, datasets, documentation, shared libraries and additional files. The description provides informations about version, authors, dependencies to other packages and copyright issues. The documentation is written in the Rd meta-documentation format. Shared libraries are used to extend R with foreign code. R is able to interface to C or Fortran code in a shared library. A package optionally contains the source code to build the libraries or pre-compiled binaries for a specific platform. A utility tool suite written in *Perl* helps automating the process of building, verification and installation of packages. The packaging standard helps implementators to deliver functionality, datasets and documentation in a portable way.

2.5 Distribution

Packages are distributed in the *Comprehensive R Archive Network* (CRAN) that is accessable through the internet. Package authors upload package releases. R provides an automated download and install mechanism to install packages from the CRAN.

Chapter 3

Interactive visualization

Interactive visualization is common in the fields of computer aided design, scientific visualization and computer games. A computer graphics system is the core providing computer-generated images.

Angel[2] describes five major components that can be found in a computer graphics system: Processor, Memory, Frame buffer, Output devices and Input devices. At present, almost all graphics systems are raster based. A picture is produced as an array - the raster - of picture elements, or pixels, within the graphics system.

3.1 Geometry-based Computer graphics

Geometry-based computer graphics use three-dimensional primitives such as points, lines, triangles, quadrilaterals and polygons as basic building blocks to contruct complex objects. The primitives are described using vertices in a local coordinate space. The object is transformed using translation, rotation and scaling to map it into the eye coordinate space, where it actually gets rendered. Attributes such as color, material properties, normal vectors, edge flags and texture coordinates are associated with the vertex. This section focuses on geometry-based computer graphics used for high-performance rendering. For detailed information on 3D computer graphics, including ray-tracing and ray-casting, see [21].

3.2 Homogeneous coordinates and transformations

In computer graphics, geometry transformation, homogeneous coordinates and 4×4 matrices became generally accepted as they provide several advantages. Homogeneous coordinates are described using four component vectors $(x, y, z, w)^T$. The w parameter decides whether the homogeneous coordinate is a vertex point with $w > 0$ (default is $w = 1$) or a normal vector with $w = 0$. A 4×4 matrix is used to transform a homogeneous coordinate into another space. Rotation, scaling, shearing, translation and even orthogonal and perspective projections can be expressed by such matrices.

An example, that shows a translation transformation which must treat vertex points and normal vectors differently due to the w component is given below. A homogeneous coordinate $\vec{v}$ is transformed using a translation transformation given by the transformation matrix $\mathbf{T}(t_x, t_y, t_z)$ translating the space t_x steps in x-direction, t_y steps in y-direction and t_z steps in z-direction.

$$\mathbf{T}(t_x, t_y, t_z)\vec{v} = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} x + t_x w \\ y + t_y w \\ z + t_z w \\ w \end{pmatrix}$$

If $\vec{v}$ is a normal vector, the transformation is unaffected due to $w = 0$. If $\vec{v}$ is a vertex point with $w = 1$, the transformation translates the point appropriately. Translation is a special case of transformations, where normals and points are treated differently. Rotation, scaling and shearing transformations affect normal vectors the same way vertex points are affected.

Often, objects are transformed using a combination of basic transformations. The basic transformation matrices can be concatenated to get a complex 4×4 transformation matrix that conserves the desired transformation steps. Vertex points and normal vectors are transformed using one matrix multiplication.

When primitives are drawn on the screen, they are projected using a viewing volume. Primitives that are partially inside the viewing volume must be clipped, while primitives completely outside can be dropped from further processing as they are not visible. When primitives are inside the viewing volume they are simply passed by. Normalized projection transformations,

that use the w coordinate to perform testing and clipping in an efficient way, exist. After normalized projection transformation the space is transformed into normalized device coordinates using a perspective division step, that divides the x,y and z components by its w component. To test if a vertex actually lies in the viewing volume, the normalized x, y and z components must lie in the range of [-w;w].

Detailed informations about homogeneous transformations are given in [13] and [2] .

3.3 Lighting

"Lighting is the term that is used to designate the interaction between material and light sources, as well as their interaction with the geometry of the object to be rendered."[13]

3.3.1 Shading

"Shading is the process of performing lighting computations and determining pixels' colors from them. There are three main types of shading: flat, Gouraud, and Phong. These correspond to computing the light per polygon, per vertex and per pixel. In flat shading, a color is computed for a triangle and the triangle is filled with that color. In Gouraud shading, the lighting at each vertex of a triangle is determined, and these lighting samples are interpolated over the surface of the triangle. In Phong shading, the shading normals stored at the vertices are used to interpolate the shading normal at each pixel in the triangle. This normal is then used to compute the lighting's effect on that pixel."[13]

"Most graphics hardware implement Gouraud shading because of its speed and much improved quality."[13]

The lighting equation used in rendering systems is not given here. Moreover, this section focuses on the material properties as they are used for appearance definitions.

3.3.2 Material

In real-time systems a material consists of a number of material parameters, namely ambient, diffuse, specular, shininess, and emissive. The color of a surface with a material is determined by these parameters, the parameters of the light sources that illuminate the surface, and a lighting model.[13]

- The *Diffuse* component specifies the color component, that, when hit by the light ray, is scattered in all directions.
- The *Specular* component lets surfaces appear shiny. Most of the light that is reflected is scatttered in a narrow range of angles (*shininess* paramater) close to the angle of reflect.
- The *Ambient* component specifies the color component, that is present due to ambient lighting.
- The *Emission* component specifies the color component that is emitted.

Material components interact with its counter-part given as light intensity components from all light sources, except the emission component which is directly applied to the surface, even if no light source is enabled. Color plate 1 depicts the effects of ambient and specular material on a sphere.

3.4 Real-time Rendering

"Real-time rendering is concerned with making images rapidly on the computer. It is the most highly interactive area of computer graphics. An image appears on the screen, the viewer acts or reacts, and this feedback affects what is generated next. This cycle of reaction and rendering happens at a rapid enough rate that the viewer does not see individual images but rather becomes immersed in a dynamic process."[13]

"The core of a real-time rendering system is a *graphics rendering pipeline*. The main function of the pipeline is to generate, or render, a two-dimensional image, given a virtual camera, three-dimensional objects, light sources, lighting models, textures, and more."[13] The pipeline renders graphics primitives that are send down the pipeline in successive order.

The architecture of a graphics rendering pipeline can be divided into three conceptional stages. Some stages become more and more implemented in hardware, thus providing more powerful rendering speed as the hardware evolves. This makes it difficult to describe the underlying algorithms in detail. This section will give an overview of common operations in a logical order. Hardware and software implementations may vary the order and provide additional features.

- *Application stage*: This stage is implemented in software. The application has full control of the geometry- and rasterizer stage. This stage decides what actually gets through the pipeline and how it is processed.
- *Geometry stage*: Incoming geometry primitives are transformed to screen coordinates. Lighting calculations are performed on primitive polygons.
- *Rasterizer stage*: The screen-mapped 2D primitives are drawn to the framebuffer.

3.4.1 Application stage

The application stage contains the application logic. Some applications require a flexible way of rendering graphics. A database of geometry objects, appearance, viewpoint, light sources and additional effects provide a general way of processing scene descriptions. This stage uses the database and a rendering strategy to setup pipeline settings, transformations and send the geometry through the pipeline.

3.4.2 Geometry stage

The primitives described by vertex coordinates and attributes are processed in this stage. Table 3.1 gives an overview of typical data. For each vertex the following operations are applied.

1. *Model & View Transformation*: The vertices and normals (if lighting calculation is enabled) are transformed from local coordinates into world coordinates and from world coordinates to eye coordinates using a model & view transformation matrix.

vertex coordinate	$(x, y, z, w)^T$ with $w > 0$
color (no lighting)	red,green,blue,alpha
normal vector (lighting)	$(x, y, z, 0)^T$
ambient color (lighting)	red,green,blue
diffuse color (lighting)	red,green,blue,alpha
specular color (lighting)	red,green,blue
emission color (lighting)	red,green,blue
specular shininess (lighting)	value
2D texture coordinate	$(s, t)^T$

Table 3.1: Vertex attributes

position	$(x, y, z, w)^T$ with $w \begin{cases} = 0 & \text{distance light} \\ > 0 & \text{point light} \end{cases}$
ambient light	red,green,blue
diffuse light	red,green,blue
specular light	red,green,blue

Table 3.2: Light attributes

2. *Lighting*: If enabled, the lighting calculation takes place using the vertex attributes and light source attributes (see table 3.2) to calculate the vertex color.

3. *Projection Transformation*: The transformed vertices are transformed a second time using a normalized orthogonal or perspective projection transformation, which is described by the viewing volume of the synthetical camera.

4. *Clipping / Perspective Division*: Clipping ensures that only those primitives whose vertices are completely contained in the viewing volume, are rendered. If the primitive crosses the volume, it gets clipped. As mentioned in section 3.2, this process can be optimized using perspective division and normalized device coordinates in conjunction with homogeneous coordinates.

5. *Viewport Transformation*: The 2D normalized vertices are mapped to window- or screen-coordinates.

6. *Culling*: A hidden-surface removal test can reject 2D primitive faces from further processing, whether they should be culled when facing back face or front face. Vertices of a face are defined in a certain orientation (counter-clockwise or clockwise). The test determines the orientation of the 2D primitive, whether the initial orientation changes.

3.4.3 Rasterization stage

In this stage the primitives actually get written as pixel values into the framebuffer. Depth value and optionally texture coordinates are associated with the vertices of the 2D clipped primitives. The framebuffer usually consists of multiple buffers to implement optimized operations. The list below describes commonly found buffers:

- *Color buffers*: The buffer holds the color values representing the pixels. A front and back buffer is common for double-buffering. The rendered image is written to the back buffer and when finished, copied as one big memory block to the front buffer. This technique is used for reducing display flickers.

- *Depth buffer*: The depth buffer keeps track of the pixel's depth values.
- *Accumulation buffer*: The accumulation buffer accumulates images generated by multiple rendering passes to create image composition effects such as motion blur, depth-of-field and anti-aliased high-quality image outputs due to super sampling.
- *Stencil buffer*: The stencil buffer is used for masking operations.
- *Auxiliary buffers*: Some implementations provide auxiliary buffers that can store rendered output to be used for later input.

Primitives are converted to a set of possibly affected pixel locations, or raster fragments. A fragment contains color information (including the alpha channel) with optional depth and texture values.

The list below briefly describes common operations that take place before finally getting a fragment as a pixel on the color buffer.

1. *Shading*: A shading algorithm can be used to color fragments between vertices (used for lines and polygons) by interpolating the vertex colors.
2. *Texturing*: The fragments color value is modified using a texture value. Texture values are mapped using interpolation between vertex texture coordinates.
3. *Fog*: The Fog feature mixes a fog color with the fragment color depending on the depth value of the fragment.
4. *Depth-test*: A depth test can be used to prevent fragments to overwrite objects that are closer to the camera. The depth values of primitives written to the framebuffer are tracked using the depth buffer. The fragments depth value is tested with the depth value of the depth buffer. This technique greatly simplifies rendering as the order of objects is not important and object intersection is handled on a fragment level. The depth values between vertices are interpolated, when using the depth buffer test.
5. *Blending*: Blending is commonly used for transparency effects. The fragments color value is mixed with the value already written to the

color buffer. Alpha blending uses the color's alpha component to control the blending.

"For high-performace graphics, it is critical that the rasterizer stage be implemented in hardware."[13]

3.5 OpenGL

Graphics hardware has evolved over the last years. The geometry and rasterization stage moved more and more into dedicated graphics hardware, thus accelerating the real-time rendering process dramatically. By using a graphics hardware abstraction, software that will exploit todays hardware-accelerated graphics and is open for future improvements of hardware implementations, can be written.

"OpenGL is a software interface to graphics hardware. This interface consists of about 250 distinct commands that are used to specify the objects and operations needed for producing interactive three-dimensional applications."[22]

This section discusses some important corner stones, that have been used for implementation.

3.5.1 State machine

OpenGL is a state machine [22], that is set to various states which remain in effect until the next change. Various operations can be toggled, parameters and transformations can be changed.

3.5.2 Client/Server architecture

Graphics hardware mostly provides its own graphics processor with dedicated memory. Some computer graphics systems are even distributed over a network. OpenGL does take this implication into account and defines a client/server architecture. The client is the application (application stage) that uses the OpenGL API. The server is the graphics hardware. The client uses the API to send state changes, geometry and appearance information, which are processed on the server side.

3.5.3 Display lists

Display lists are compiled sequences of OpenGL commands stored on the server-side. The client constructs the display list, and calls it each time it would normally issue the commands. This reduces the data transfer dramatically and server implementations can optimize the display list for execution.

3.5.4 Vertex arrays

Vertex arrays and additional attribute arrays are supported since OpenGL version 1.1 . The vertex coordinates and attributes are referenced by the client and dereferenced using OpenGL vertex array dereferencing commands. The former method is a function call per attribute and coordinate to define one vertex of a primitive. As many objects share vertices, this method is not optimized as shared vertices are transformed multiple times.

3.5.5 Windowing system interface

OpenGL does not provide any interface for initializing an OpenGL capable window or to get input events. It defines the interface to the OpenGL graphics rendering machine. The windowing systems are responsible for providing a way to initialize a so called *OpenGL context* for a given window. All major platforms provide extensions to support OpenGL: Win32 provides the *wgl* extension, X11 provides the *GLX* extension and Macintosh provides the *agl* extension.

3.6 Interaction

Common input devices used in interaction are pointing and keyboard devices. A pointing device has two degrees of freedom (x- and y movement) and one button at minimum.

Chapter 4

Functional Design

This chapter explains in detail the functionality on an abstract level. A short overview of the programming interface is given in advance.

4.1 Goals

As every project starts with *the idea* and *the goals*, RGL does not make an exception. This section will outline the major corner stones that have been choosen for the development of an interactive visualization device system:

- A workspace of multiple device instances provides a gentle working environment for comparing visualizations analogous to multiple views and plotting devices in R.
- Interactive navigation facilities should be designed intuitive. Conventional input devices such as the pointing device[1] should be supported.
- The viewpoint should keep the focus on the visualized data model, while exploring the data.
- The description of a scene should be performed by an API in R, providing complex visualizations through combination of basic primitive building blocks. The interface should be suitable for interactive command-line editing.

[1] also known as the *mouse*

- An undo function would support the user to develope complex visualizations in a *trial-and-error* process.
- The data space should be visualized appropriately. Dimensional extends given as a bounding box with axis tick-marks surrounding the data would provide users helpful informations about the dimensional ranges of data.
- Recorded image stills and animations are useful for presentation purposes. The possibility to store a snapshot on mass storage devices is sufficient to support the creation of image stills and animations.

4.2 Device

The device is the topmost abstraction for the task of interactive visualization, analogue to R devices that abstract the task of plotting. It contains a graphical user-interface component that displays the rendering output, receives user-input from the pointing device and provides a service to save a snapshot on mass storage devices. Multiple devices can be opened simultaneously. A management unit is required to keep track of all devices. API calls concerning a specific device are dispatched to the currently active device through the management unit.

4.3 Model

Models are abstractions of the real world. To visualize data, the user must deal with an abstract model of real-time rendering to describe what should be rendered and how it should appear. The RGL visualization model is depict in figure 4.1. It contains five object types that define a scene:

- The *Viewpoint* describes the location, orientation and camera properties that will be used as the virtual eye of the user exploring the data space.
- *Shapes* are a collection of basic building blocks to visualize data in three-dimensional space. Seven different shape types are supported providing different geometry objects.

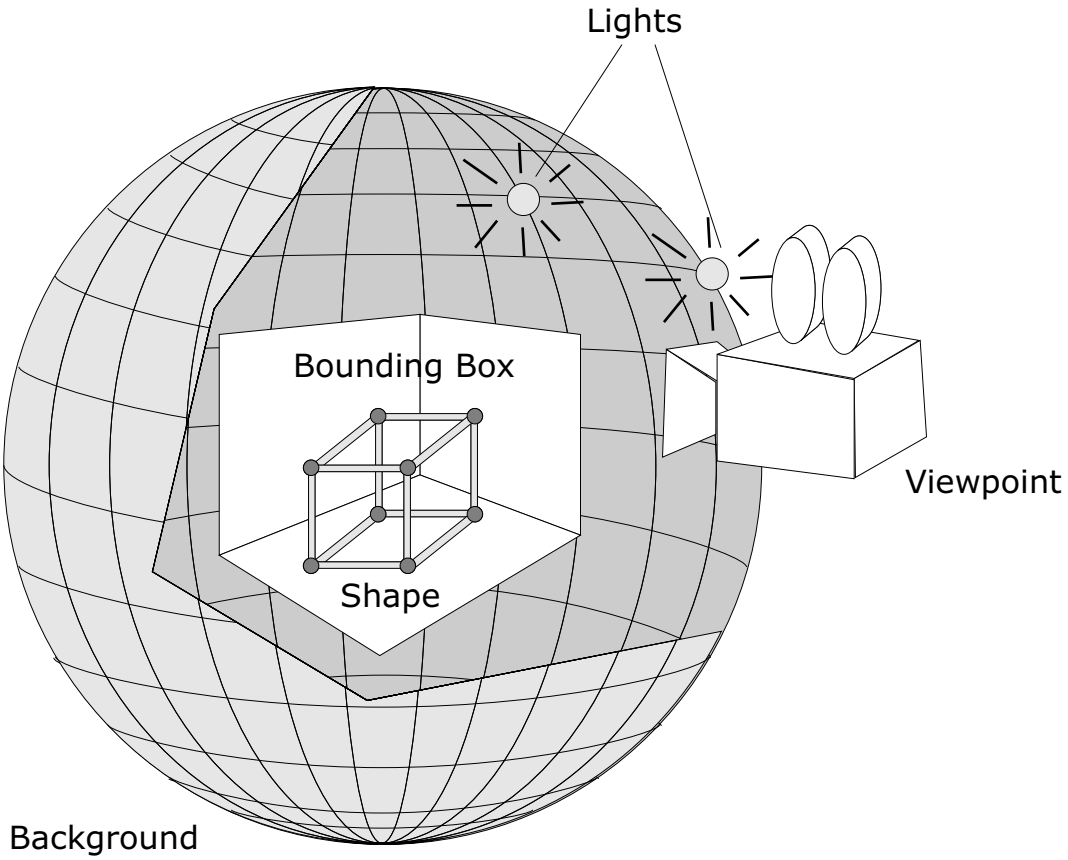

Figure 4.1: Scene objects

- The *Light* object represents a light source. The total of light objects describe the lighting conditions.
- The *Bounding Box* object encloses the data space described by the total of all *Shapes*. It measures the extends of the Shapes currently visualized and displays tick-marks with labels on the three axis dimensions.
- The *Background* object describes environmental properties such as the background rendering style and the fog conditions.

4.3.1 Coordinate system

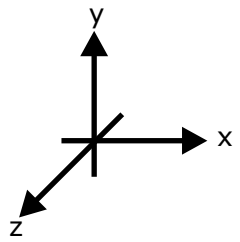

Figure 4.2: Coordinate system

When defining geometry in a model, consistent coordinate system conventions must be defined first. Figure 4.2 depicts the coordinate system, where x and y axis points to the right and top and the z axis points to the viewer.

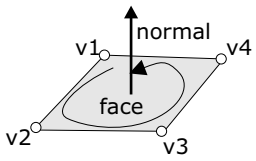

Figure 4.3: Face orientation

The orientation of faces is important for culling as mentioned in section 3.4.2. Different drawing styles such as plotting the vertices, drawing the edges or filling the entire face can be associated. Front faces are defined in a counter-clock wise order as illustrated in figure 4.3. The normal vector

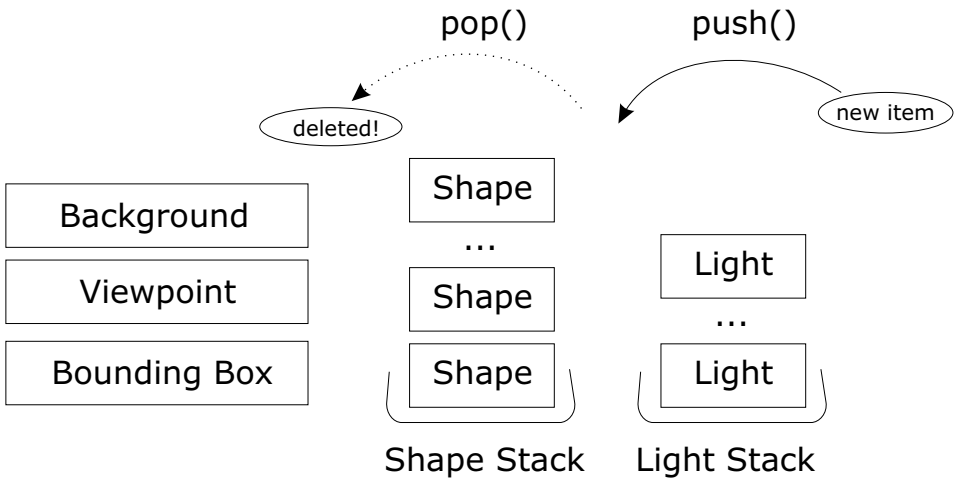

Figure 4.4: Logical database model

is orthogonal to the surface and points out of the front side. It is used for lighting calculation.

4.3.2 Scene database

The *Scene database* stores the model and provides an editing interface. Figure 4.4 depicts a logical model of the database. The scene database, or the *Scene*, holds three single object slots to store the *Background*, the *Viewpoint* and the *Bounding Box*. Two stacks store multiple *Shapes* and *Lights*.

An Object is added by *pushing* it on to the *Scene*, where it either replaces an object in a slot or is added on a stack, depending on the object type.

Any changes in the database result in a refreshment of the display reflecting the new state of the *Scene*.

An *undo* operation can be performed for *Shapes* and *Lights* that *pops* the topmost, or last added, object from the stack.

The *Bounding Box* is optionally and can be *pop*ed as well. A *Background* and *Viewpoint* can be replaced, but have to exist.

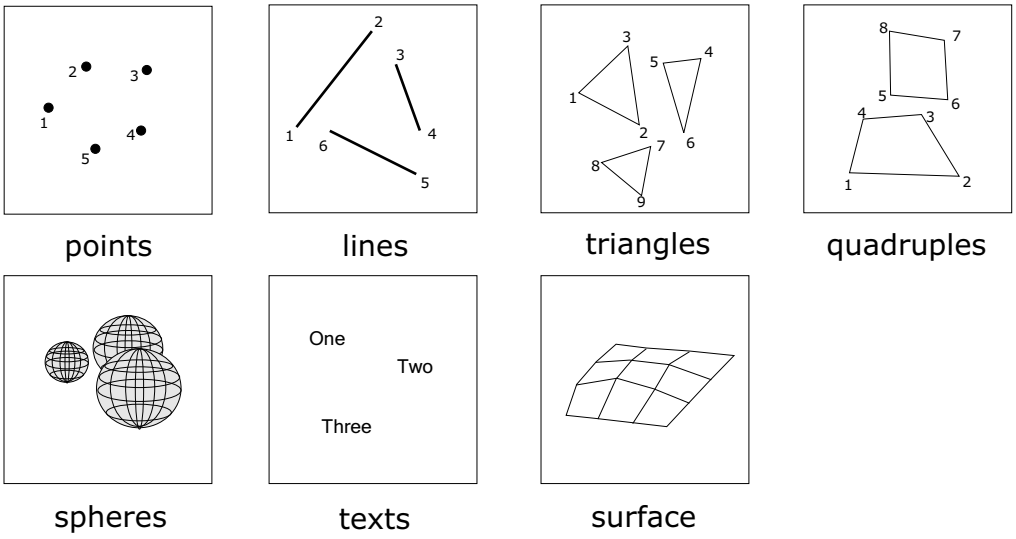

Figure 4.5: Shape objects

4.3.3 Shape

Shapes describe geometrical objects. Seven different shape types are supported, which are depict in figure 4.5. The objects are defined by vertex coordinates probably given by data to be visualized, analogue to standard R graphical data analysis. Normal vectors will be automatically generated for lighting calculations.

Points, *Lines*, *Triangles*, *Quadriangles* and *Texts* are primitive shapes while *Spheres* and *Surface* are high-level shape types.

The *Texts* shape draws text labels using a bitmap font with horizontal text alignment.

The *Spheres* shape describe a set of sphere meshes with a user-definable radius per sphere. The *Surface* shape is adapted from R graphics plotting function `persp()`, that generates a heightfield mesh using x-grid and z-grid vectors and an y-heights matrix.

Parameter	*Type*	*Details*
`theta`, `phi`	polar coordinates	position θ $(-\infty^o; \infty^o)$ and ϕ $[-90^o; 90^o]$
`fov`	angle	field-of-view angle $[0^o; 180^o]$
`zoom`	scalar	zoom factor $[0; 1]$

Table 4.1: *Viewpoint* parameters

4.3.4 Viewpoint

The *Viewpoint* navigation has been carefully designed to achieve interactive navigation while keeping the focus on the data.

Navigation through pointing devices require a constrained navigation paradigm, because common pointing devices provide two degrees of freedom at a time, while in three-dimensional space complex navigation movements exist.

A viewpoint navigation model, where the *Viewpoint* is moving on a sphere orbit that encloses all shapes, has been developed. The position is given in polar coordinates using two angles, so that the pointing device is suitable for controling the position. The camera is oriented towards the center of the sphere, ensuring that the data is focused. Table 4.1 lists the parameters of the *Viewpoint*.

The polar coordinates are given using two parameters, `theta` and `phi`, which describe the azimuth angle θ and colatitude ϕ respectively. The ϕ angle is locked between $[-90^0; 90^0]$ to force users to think in polar coordinates.

The `fov` parameter describes the field-of-view angle of the lens aperture, while `zoom` is a scaling factor that results in zooming into the center of the projection plane.

Button	*X-axis drag*	*Y-axis drag*
left	θ angle	ϕ angle
middle	-	field-of-view angle
right	-	zoom factor

Table 4.2: Dragging actions for interactive viewpoint navigation

Position, field-of-view angle and zoom can be controled interactively using the pointing device. The pointer must be located in the device window. A

drag operation is launched by pressing a specific button (table 4.2). Moving (dragging) the pointer across the device window while holding the button pressed will modify viewpoint parameters. An immediate refreshment of the display reflects the changes. By moving the pointer while the button is held pressed, parameters of the *Viewpoint* are modified. When the button is released the drag operation completes.

4.3.5 Light

Parameter	*Type*	*Details*
`theta`, `phi`	polar coordinates	position
`viewpoint.rel`	logical	relative position to viewpoint
`ambient`	color	ambient light intensity
`diffuse`	color	diffuse light intensity
`specular`	color	specular light intensity

Table 4.3: *Light* parameters

The *Light* object represents distance light sources that are positioned using polar coordinates analogue to the *Viewpoints*. Parameters are listed in table 4.3. The parameters `ambient`, `diffuse` and `specular` specify the intensity that is emitted from the light source for the particular material components. The `viewpoint.rel` logical specifies if the position is relative to the *Viewpoint*.

4.3.6 Background

Parameter	*Type*	*Details*
`sphere`	logical	spherical background mesh
`fogtype`	enumeration	`none`, `linear`, `exp` or `exp2`

Table 4.4: *Background* parameters

The *Background* object is responsible for the decoration of the environment including the background layer and atmospherical fog effect. It simply fills

the background with a solid color or, if `sphere` is set `TRUE`, renders an environmental sphere geometry, that is viewed from the inside. Color plate 5 and 6 depict a scene a with a sphere geometry. Color plate 5 displays a wire frame of the sphere geometry, while color plate 6 uses a texture. Further, an atmospherical fog effect can be enabled. Color plate 2 shows a scene with a *Background*, where a linear fog effect is enabled. A fog factor given by a function $f(z)$ (with z = the distance of a vertex from the *Viewpoint*) is used to mix the original vertex color with the background color. The fog factor function can be specified by `fogtype`, where `linear` is a linear function, `exp` is an exponential function and `exp2` is a squared exponential function. The fog effect is disabled using `fogtype = "none"`.

4.3.7 Bounding box

Parameter	*Type*	*Details*
`xat`,`yat`,`zat`	numeric vector	user-defined tick-marks
`xlab`,`ylab`,`zlab`	character vector	user-defined labels
`xunit`,`yunit`,`zunit`	numeric value	tick-mark unit length
`xlen`,`ylen`,`zlen`	numeric value	number of tick-marks
`marklen`	numeric value	tick-mark length
`marklen.rel`	logical	relative tick-mark length

Table 4.5: *Bounding Box* parameters

The *Bounding Box* decorates the data space described by the total extends of all *Shapes* using an axis-aligned bounding box geometry where its inside is displayed. As *Shapes* can be added or removed, the *Bounding Box* will automatically reflect the new data space. Tick marks can be given to mark discrete values on the axis dimensions. Table 4.5 lists the parameters to setup tick-marks and labels. Three different axis tick-mark labeling methods are available:

- The `at`-method is used for user-defined values.
- The `unit`-method automatically sets up tick-marks in unit steps.
- The `len`-method automatically sets up the specified number of tick-marks which are equally spaced across the dimensional extents.

Each axis dimension can be applied with an individual axis scaling method.

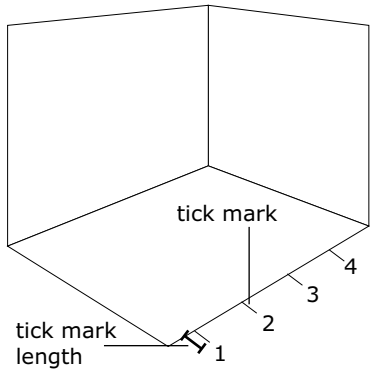

Figure 4.6: *Bounding Box* tick-marks

The tick-mark length as depict in figure 4.6 is defined by `marklen` and `marklen.rel` and specifies the length of the tick-mark line coming out of the bounding box. If `marklen.rel` is `TRUE`, the tick-mark length is calculated using the formula

$$\frac{1}{\texttt{marklen}} \times \text{bounding sphere radius enclosing the bounding box}$$

This provides a quite constant length of the tick-mark, that is independent of the actual data space extends. Otherwise, the length is taken from `marklen`.

4.3.8 Appearance

Appearance information is encapsulated in an object, that is implicitly contained in the *Shape*, *Background* and *Bounding Box* objects. Table 4.6 lists all parameters. The appearance definition has been generalized using one interface for all objects that require appearance parameters. Some parameters do not have an influence and are ignored by several object types.

Geometry is drawn using the color values, that are given by the `color` vector. The *Texts* shape uses one color per label. The *Spheres* shape uses one color per sphere. All other shape types use one color per vertex. The *Background* uses the first color for solid clearing and fog effect, and the second color for the environmental sphere color. The *Bounding Box* uses the first color for

Parameter	*Type*	*Details*
`color`	multiple colors	color or diffuse component
`lit`	logical	affected by lighting calculation
`specular`	color	specular component
`ambient`	color	ambient component
`emission`	color	emissive component
`shininess`	numeric value	specular exponent ($[0.0; 128.0]$)
`alpha`	numeric value	alpha blending if $alpha < 1$
`texture`	character string	path to image file
`size`	numeric value	size of points and lines
`front`, `back`	enum	face side rendered as points, lines, filled or culled
`smooth`	logical	flat or smooth shading
`fog`	logical	affected by fog calculation

Table 4.6: Appearance parameters

the bounding box geometry and the secondary color for axis tick-marks and labels. The color vector is recycled in case the number of colors does not match the required size.

The `lit` logical specifies if lighting calculation should be performed on the associated geometry. Lighting calculation uses the `color` vector as the diffuse component, `specular` , `ambient`, `emission` and `shininess` to describe the material.

Alpha blending is enabled for `alpha` < 1.0.

2D Texture mapping is supported by *Spheres*, *Background* and *Surface*. A pathname of an image picture is given by `texture`. *Spheres* and the environmental sphere geometry of the *Background* uses a spherical texture mapping while *Surface* uses a xz-plane texture mapping.

The `size` parameter affects primitives that are rendered as points or lines specifying the point and line pixel sizes.

`front` and `back` specifies the face rendering style for both face sides:

- `point`: Vertex points are drawn
- `line`: Edges are drawn

- `fill`: Face is drawn
- `cull`: Face get culled, thus is not drawn

Points and *Lines* shapes are not affected by the face rendering style.

The `smooth` logical specifies the shader algorithm. If set `true`, a gouraud shader is used interpolating colors across vertices using bilinear interpolation. Otherwise flat shading occurs using the first vertex color of a face to fill the entire area of the primitive. Color plate 2 shows the effect of `smooth`.

The `fog` logical, specifies if fog calculation should be performed.

4.4 API

This section will explain the *Application Programming Interface* of the RGL package.

The RGL API contains 20 public R functions. Functions are prefixed by "`rgl.`" to prevent name clashing with R graphics commands. Some Functions provide a "`...`" (dot-dot-dot) argument, which is used for appearance parameters that are dispatched to the `rgl.material` function. The functions can be grouped into six categories (see Figure 4.7):

- *Device management functions* open and close devices, control the active device focus, and shutdown the entire system.
- *Scene management functions* provide a service to remove objects from the scene database.
- *Export functions* are used to create snapshot images.
- *Shape functions* are used to add shapes to the scene database.
- *Environment setup functions* provide functions to replace the *Viewpoint*, *Background* and the *Bounding Box*, and a function to add *Lights*.
- One *Appearance* function, `rgl.material()`, actually sets up appearance properties.

Device Management	Scene Management	Export Functions
rgl.open()	rgl.clear()	rgl.snapshot()
rgl.close()	rgl.pop()	
rgl.cur()		
rgl.set()		
rgl.quit()		

Shape Functions	Environment Setup	Appearance
rgl.points()	rgl.viewpoint()	rgl.material()
rgl.lines()	rgl.light()	
rgl.triangles()	rgl.bg()	
rgl.quads()	rgl.bbox()	
rgl.spheres()		
rgl.texts()		
rgl.surface()		

Figure 4.7: Overview of the RGL API

4.4.1 Device management functions

```
rgl.open()
rgl.close()
rgl.set(id)
rgl.cur()                       # returns id
rgl.quit()
```

Multiple devices can be opened simultaneously. One device at a time has the current focus of control from the R command line. Each device is uniquely identified by an id which is of type integer. One can query the id of the current device by using `rgl.cur()` and can change the focus using `rgl.set(`*`id`*`)`, analogue to the R device management functions `dev.cur()` and `dev.set()`.

`rgl.open()` explicitly opens a new device, while `rgl.close()` explicily closes the current device.

To force a complete shutdown of RGL, one can issue `rgl.quit()`.

4.4.2 Scene management functions

```
rgl.clear(type = "shape")
rgl.pop(type = "shape")
```

The functions remove objects from the scene database. The `type` argument is a character string specifying the object type. "`shape`" and "`light`" select one of the two stacks, while "`bbox`" selects the Bounding Box slot. `rgl.clear()` removes all objects of that type, while `rgl.pop` removes the top-most object from the stack. If `type` is "`bbox`", it will remove the object from the slot. If no device is opened, the operation will fail.

4.4.3 Export functions

```
rgl.snapshot(filename, format = "png")
```

Export functions provide the service to save a snapshot to mass storage using a specific pixel-image file format. The snapshot is taken from the current device. If no device is opened, the operation will fail. Supported file formats:

- PNG[2]

4.4.4 Shape functions

Add primitive *Shapes*

```
rgl.points(x, y, z, ...)
rgl.lines(x, y, z, ...)
rgl.triangles(x, y, z, ...)
rgl.quads(x, y, z, ...)
```

`x`, `y` and `z` are data vectors that define vertices in space.

[2]Portable Network Graphics

Add *Text* shape

```
rgl.texts(x, y, z, text, justify="center", ...)
```

The `text` argument is a vector of character strings. `justify` specifying the horizontal text layout. Valid string values are: `"left"`, `"center"` and `"right"`. The `text` vector is recycled, in case more vertices are given than character strings.

Add *Spheres* shape

```
rgl.spheres(x, y, z, radius, ...)
```

`x`, `y` and `z` are vectors that specify the centers of the spheres. The `radius` argument specifies the radius per sphere and recycles the vector to the number of centers given, if necessary.

Add *Surface* shape

```
rgl.surface(x, z, y , ...)
```

`x` and `z` specify a vector of grid unit values for the x and z axis. `y` specifies a matrix of y values with the dimension: x-length $\times$ z-length. A surface height field mesh is generated.

4.4.5 Environment functions

Replace *Viewpoint*

```
rgl.viewpoint(
  theta = 0.0, phi = 15.0,
  fov = 60, zoom = 0,
  interactive = TRUE
)
```

`interactive` is a logical, specifying if interactive navigation using the pointing device should be enabled. Other parameters are described in detail in section 4.3.4.

Add *Light*

```
rgl.light(
  theta = 0.0, phi = 0.0, viewpoint.rel = FALSE,
  ambient = "#FFFFFF", diffuse = "#FFFFFF", specular = "#FFFFFF"
)
```

Adds a *Light* to the database. Parameters are described in detail in section 4.3.5.

Replace *Background*

```
rgl.bg( sphere = FALSE, fogtype = "none", ... )
```

Parameters are described in detail in section 4.3.6.

Replace *Bounding Box*

```
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, ...
)
```

Parameters are described in detail in section 4.3.7.

4.4.6 Appearance

```
rgl.material (
  color = "#666699",    lit = TRUE,
  ambient = "#000000",  specular = "#FFFFFF",
  emission = "#000000", shininess = 50.0,
  alpha = 1.0,          smooth = TRUE,
  texture = NULL,       front = "fill",
  back = "fill",        size = 1.0,
  fog = TRUE )
```

Appearance properties are defined using *Shape* and *Environmental Setup* functions. The color data type uses the form `"#RRGGBB"`. For details on the parameters see section 4.3.8.

Chapter 5

Software Development

This chapter discusses the software development including analysis issues, the software architecture, software design and implementation details.

The chapter starts with an introduction to techniques that have been applied along the software development process. The methodology of object-orientation will be outlined, including software patterns, UML diagram notations and key features of the C++ programming language.

Afterwards, an analysis of architectural issues to implement the system design is given. This forms the basis for the design and implement, starting with an over of the software architecture, which is then described in a bottom-up approach with a varying level of detail, either looking at a module, class or method level.

5.1 Object-orientation

The methodology of object-orientation is used in software development for describing real-world phenomenons on an abstract level, for modeling software architectures and designs, and for the implementation using a programming language that supports object-oriented techniques.

The object-oriented approach, as the name implies, regards every entity as an *object*, that can be characterized by *attributes* and *methods*.

Attributes are of a certain data or object type and store information about the object. Methods can be regarded as functions that are bound to objects

and have access to the attributes.

Objects are abstracted using *classes*. A class defines the attributes and methods that an instance (or object) will have.

Classes are structured hierarchically by using the concept of inheritance. A subclass inherits all attributes and methods of the classes it inherits from in the hierarchy.

Interfaces are defined as abstract classes that declare the methods without implementation definition. Classes that provide a certain interface are inherited from the abstract class and implementing the interface methods.

Four general techniques, that summarize object-orientation, can be named:

- *Classes* do reduce complexity through abstraction. Instead of defining each object of a system, objects are generalized to classes, and these are defined.
- *Data Encapsulation* provides class designers the ability to define an interface for it classes, which the client has to use to obtain data. The internal structure of the object remains hidden from the clients.
- *Inheritance* is a technique for structuring classes hierarchically. Certain concepts in software design can be defined once and reused in subclasses through inheritance. Polymorphism and abstract interface definition can be expressed through inheritance.
- *Polymorphism* is used for generalizing behaviour for a group of objects and classes. Objects can implement an abstract behaviour differently. Clients do not have to know the exact object type when they deal with that object as they call an abstract base type method uppon the unknown object.

5.1.1 Notation using the UML

The *Unified Modeling Language*, or UML, is a graphical notation system to describe object-oriented models. A small set of elements is used throughout this chapter, which is depict in figure 5.1. Classes are either drawn simple using a box, or detailed with the relevant attributes and methods to describe.

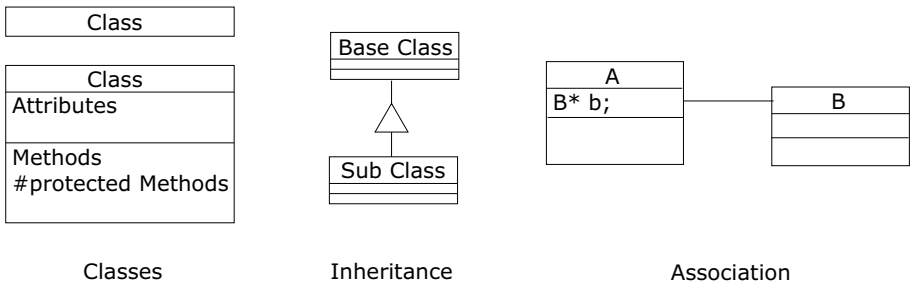

Figure 5.1: UML notations

Protected methods are marked with a "#", and are used where it emphasizes the design. Inheritance is expressed using a triangle on a line, the triangle points up to the base class, where the subclass inherits from. An association between two classes is expressed by a line connecting the two classes. For details about UML, see [16].

5.1.2 Software patterns

Patterns describe problems and show ways to solve them on an abstract level. They can be tracked back into the 1970s where the architect Christopher Alexander[1] introduced this technique to describe strategies for architects to solve problems using pattern descriptions. Software patterns were first presented in the book "Design Patterns - Elements of Reusable Object-Oriented Software" written by "the Gang of Four": Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides [7].

Software patterns can help solving concrete problems or help finding approriate design[7] and architectural structures[6]. They document problems by describing the forces on the topic, give hints, solution approaches and the consequences for the whole project. Software patterns do not provide exact problem-solution descriptions. They describe common circumstances on an appropriate level abstracting the problem into a way, that the solutions can be applyied in several domains.

5.1.3 C++

The C++ language is a successor of the C language and belongs to the family of compiled languages which execute at machine-level speed, in contrast to R, which is an interpreted language. C++ is a hybrid language, providing features for procedural programming (C), generic programming (templates) and object-oriented programming. Some remarkable language features, that have been used to implement the software are briefly described. A detailed description of C/C++ is beyond the scope of this thesis. See [9] and [12] for detailed information on the C language and the Ansi C Standard. See [20] and [8] for detailed information on the C++ language and object-oriented programming in C++.

Modularization

When building software in C++, the software project is split into modules. A module typical consists of three items:

- *Source code* (suffixed with `.c`,`.C`,`.cpp` or `.cxx`): The source code implements the module.
- *Header file* (suffixed with `.h` or `.H`): Public interface, classes, datatypes and function signatures are held in the header file. Other module sources, that require access to the module, include this file.
- *Object file* (suffixed with `.o` or `.obj`): Compiled native machine code and data.

A linker program is issued to bind all object files to an executable or library.

By applying the technique of modularization, the software implementation can be divided into several modules on a first decomposition level.

Object-oriented programming in C++

C++ supports object-oriented programming.

Classes contain data and method fields and can inherit from multiple super classes. Instances, or objects, are created by a *constructor* which initialized

a object to its initial state. A *destructor* is called when the object is about to be deleted. Constructors and destructors are chained across the class hierarchy. Base class constructors are called first. The actual class constructor is executed at last. Destructors are called in reversed order.

Data encapsulation is supported by defining the access to the fields using the keywords `private`, `protected` and `public`. Private fields are accessable exclusively by the class. Protected fields are accessable by the class and derived classes. Public fields are accessable by all entities.

Class can be composed of objects. C++ ensures, that contained objects are constructed and destructed automatically when the composite object is constructed and destructed, respectively.

Polymorphism is used by declare a method *virtual*, so that subclasses can overload the method with a specific implementation.

Abstract classes can be defined using pure virtual methods that do not provide an implementation. They can be derived but not instantiated.

Fundamental C datatypes such as `int` (integer) and `float` (floating-point) provide builtin operator implementations for C operator symbols ("`+`", "`-`", "`/`", "`*`", ...). In C++, custom class operator functions can be defined. It is usefull in cases where complex mathematical datatypes such as vectors and matrices are designed and the symbolic expression of the operators are appropriate. The applicability of user-defined operators is constrained due to the fixed priority between the operators built in the C++ compiler.

Virtual destructors provide an interface for destroying an unknown instance. The client knows the base class where the virtual destructor is defined. The destructor method call is delegated to the specific destructor of the instance class type.

5.2 Analysis

The software design analysis focuses on the device object and its environment. Figure 5.2 shows the device as a black box with its input/output relations to the environment. Input requests calls are coming from the R system due to API calls. Input event messages are sent from windowing system to notify about user actions through the graphical user-interface. The

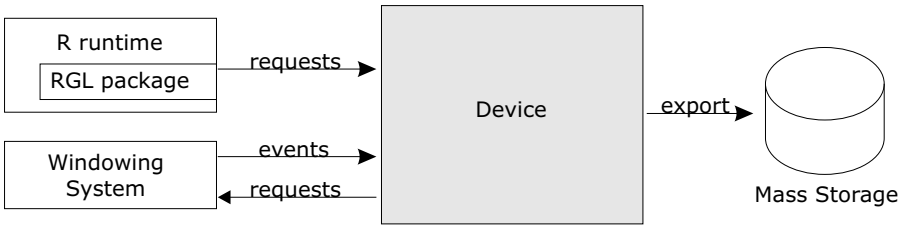

Figure 5.2: Device Input/Output

device produces rendered computer-graphics frames using an OpenGL context obtained from the windowing system each time the database changed. Images can be exported to mass storage.

R provides an extension interface for new functions, as mentioned in section 2.4. It lacks the ability to extend its graphical user-interface with new components. Functions must return control flow, otherwise R would hang up. Therefore, the analysis explores the integration opportunities of the device into R and the windowing system.

5.2.1 Shared library

The concepts of shared libraries and linking are discussed in this section, as it is the way of interfacing R with foreign code using a C or Fortran interface. A shared library represents a container for code and data. In contrast to executable files, a shared library can not run standalone. It is linked to a process at load- or run-time by the system loaders and linkers. *Load-time linkage* is done when the executable was dynamically linked to a shared library. *Run-time linkage* is done dynamically at run-time using the system loader services which provides loading and unloading of a shared libraries and locating of functions and data through symbols. This method is commonly used to support dynamic extensions without rebuilding the core system. R supports dynamic run-time linkage of shared libraries. The R function `library.dynam()` requests the system loader to attach a shared library to the running R process. The function `.C(symbol, parameters ...)` is used to locate and call shared library functions that are dynamically linked to the R process at run-time.

For detailed information on system loaders, linking and operating system specific issues, see [11].

5.2.2 Windowing system

A windowing system has direct access to input (pointer, keyboard) and output (graphics cards) devices that make up the foundation of the graphical user-interface. It abstracts the direct access logically through windows - rectangle portions on the screen that can be managed. Windows represent the general interface between applications and the user. As the applications and the windowing system are not running in the same process space, they communicate over messaging. The application runs in a main loop, waiting for messages. When a message arrives, it is evaluated and the application reacts appropriately on it. Further, it sends messages to the windowing system requesting drawing and management operations. With this technique multiple applications can share the display and input devices. A typical gui application session is given below:

1. connect the windowing system
2. request the windowing system to create graphical user-interface resources
3. run main loop until quit message
    - wait for messages
    - dispatch message to appropriate handler function
4. disconnect

The dispatch mechanism has not been mentioned so far, though it is the central point of interest for the integration process. When managing multiple graphical user-interface components in an application, an object-oriented design has evolved over the years, where the evaluation of messages is encapsulated into the object,but not in the application. The dispatching mechanism on the application side routes the messages according to screen positions and nested parent-child window relations to a handling procedure.

R has been ported to three windowing systems with slight differences in the way the dispatching of messages is implemented. R does not provide an abstraction of the windowing system, neither an interface to extend R with

new graphical user-interface components so far[1], so it uses common interfaces of the windowing system.

The Win32 API of the Microsoft Windows windowing system uses a registration mechanism of a window class that contains a pointer to a handling function. Windows are created by using a specific class. The dispatch mechanism is provided by the Win32 API and uses the class field to delegate control flow to the handling function. This way, the RGL extension registers a window class and can run in the same GUI thread as R.

The X11 windowing system provides a low-level interface provided by the Xlib library. It provides the interface to wait for an event but the dispatching must be implemented by the application.

Three different implementation strategies must be concerned for porting issues:

- The windowing system provides a dispatch mechanism and registration service: The system runs in a single-threaded model where the communication between API and device is done through direct calling.
- The windowing system does not provide a dispatch mechanism:
  - The windowing system is thread-safe. A separate thread runs a main loop in parallel to the R main loop and processes its events. The communication mechanism between the API and the device manager is implemented using shared memory with a synchronization mechanism.
  - The windowing system is not thread-safe. A separate process is launched managing the devices. The communication is done through IPC[2]. This technique must be concerned, if a multi-threaded model would not work properly, due to components that are not thread-safe.

One can come across the problem by abstracting threads and implement the system using message passing on an abstract level. This abstraction would fit good into the R core, providing other packages a general interface for

[1] this refers to the version 1.5.1 of R

[2] inter-process communication

extending the R graphical user-interface. The current implementation was developed on Win32 and uses a single-threaded model.

For more informations about this topic, and porting issues, see [10] for detailed information on posix and SUN threads. Detailed information about the X11 windowing system is given by [14] and [15].

5.3 Architecture

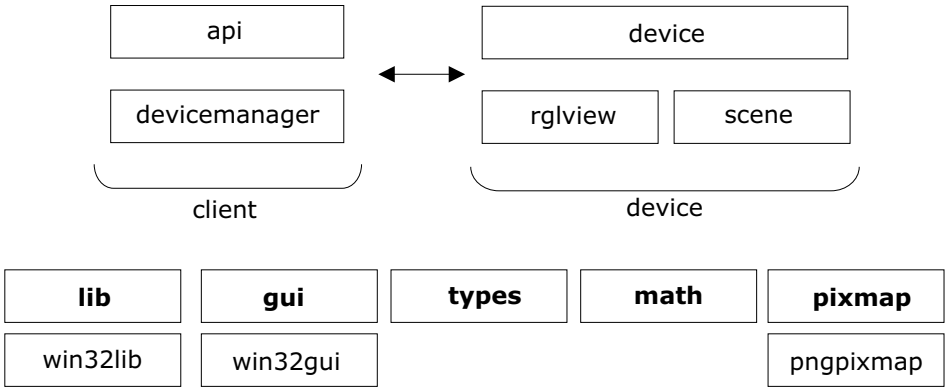

Figure 5.3: Overview of C++ modules

The architecture of the device system is built using a foundation layer. Two domains logically split the device system into a client and a device domain. Figure 5.3 gives an overview of all modules describing the architecture. Some modules require platform-specific implementations, which are separated by prefixing a platform identifier (e.g. "win32").

The next sections describe each domain and its modules, starting with the foundation layer, then focusing the *scene* module. Afterwards, the device domain and client domain are discussed.

5.4 Foundation layer

The foundation layer provides services, classes, utilities and abstracts platform-specific issues such as the windowing systems and the library startup.

The layer consists of five modules:

types	fundamental data types and structures
math	general mathematics and geometry
pixmap	picture image services
gui	abstract windowing toolkit
lib	general library services

5.4.1 *types* module

The types module provides constants, fundamental data types and data structures.

Double-linked lists

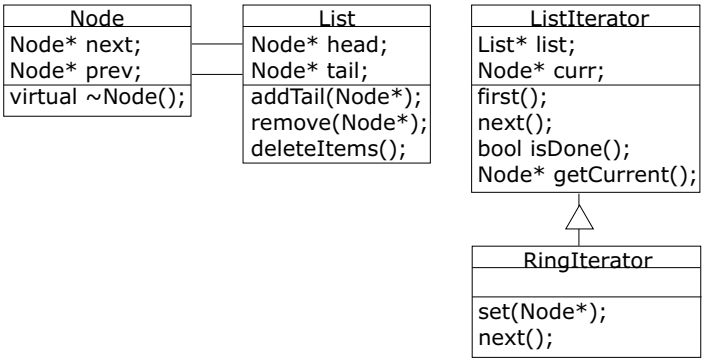

Figure 5.4: Class diagram: Double-linked lists

Double-linked lists are efficient data structures for managing a variable number of items. Figure 5.4 gives an overview of the classes that are provided to support this data structure. The *Node* base class is used in conjunction with the *List* class to implement the storage organization. Items that should be organized in a double-linked list are derived from the *Node* class. The nodes are linked by using two pointers, `prev` pointing to the previous node or `NULL` (head of list) and `next` pointing to the next node or `NULL` (tail of list). The *List* class holds the head and the tail of the list. The *ListIterator* and *RingIterator* classes are provided for iteration in a List. The *ListIterator* implements a sequencial iteration, while the *RingIterator* automatically

recycles. A typical iteration in C++ is given below:

```
List* list;
ListIterator iter(list);

for( iter.first(); !iter.isDone(); iter.next() ) {
  Node* node = iter.getCurrent();
  // do something
}
```

The `List::deleteItems()` method destructs all items in the list by calling the virtual destructor `Node::~Node()`.

5.4.2 *math* module

The *math* module contains classes, types and functions that implement mathematical operations and geometrical computations.

Vertex classes

Vertex
float x,y,z

float getLength();
normalize();
Vertex cross(Vertex);
float operator*(Vertex);
Vertex operator*(float);
Vertex operator+(Vertex);
Vertex operator-(Vertex);
operator +=(Vertex);
rotateX(float degrees);
rotateY(float degrees);

Vertex4
float x,y,z,w
float operator*(Vertex4); Vertex4 operator*(float); Vertex4 operator+(Vertex4);

The *Vertex* and *Vertex4* classes represent a vertex in 3D coordinates and homogenous coordinates respectively. Several operators for vector/vector and vector/scalar arithmetic are provided. Rotation transformations around the X and Y axis are supported.

Matrix4x4 class

Matrix4x4
float data[4*4];
Vertex4 operator* (Vertex4); Matrix4x4 operator* (Matrix4x4); Vertex operator* (Vertex);

The *Matrix4x4* class implements $Matrix4x4 \times Vertex4$, $Matrix4x4 \times Matrix4x4$ and $Matrix4x4 \times Vertex$ multiplication. As OpenGL provides transformation for rendering, this class is entirely used for evaluation of certain conditions in space at the application stage before sending geometry. OpenGL can be queried to get the current transformation, that can be written to an *Matrix4x4* object. Control points and normals using *Vertex4* are then transformed on the application stage. The $Matrix4x4 \times Vertex$ operation is provided using $w = 1$.

Sphere class

Sphere
Vertex center; float radius;

A *Sphere* object describes a sphere in three-dimensional space.

PolarCoord class

PolarCoord
float theta; float phi;

PolarCoord objects describe polar coordinates, which are defined by two angles: θ (`theta`) specifies the azimuthal direction while ϕ (`phi`) specifies the colatitude. Polar coordinates can be used to describe orientations around a center or to locate a point on a reference sphere, as used by the *Viewpoint* described in section 4.3.4. This section will describe the geometry transfor-

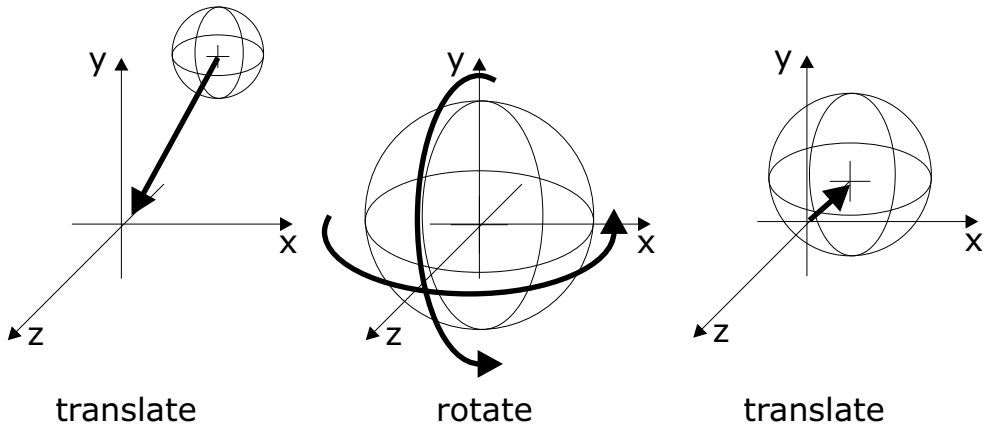

Figure 5.5: Geometry transformation of polar coordinates

mations that are used in conjunction with polar coordinates to implement orientation in polar coordinates. A world sphere is given as the space, that is observed by a viewpoint. Figure 5.5 illustrates the required transformations.

1. Translate center of the sphere to the origin.
2. Rotate θ degrees around the y-axis.
3. Rotate ϕ degrees around the x-axis.
4. Translate -radius units along the z-axis.

The transformation in OpenGL is defined in reversed order and is given as an example below:

```
Sphere s;
PolarCoord polarCoord;

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

glTranslatef( .0f, .0f, -s.radius };

glRotatef( polarCoord.phi,   1.0f, 0.0f, 0.0f );
glRotatef(-polarCoord.theta, 0.0f, 1.0f, 0.0f );
```

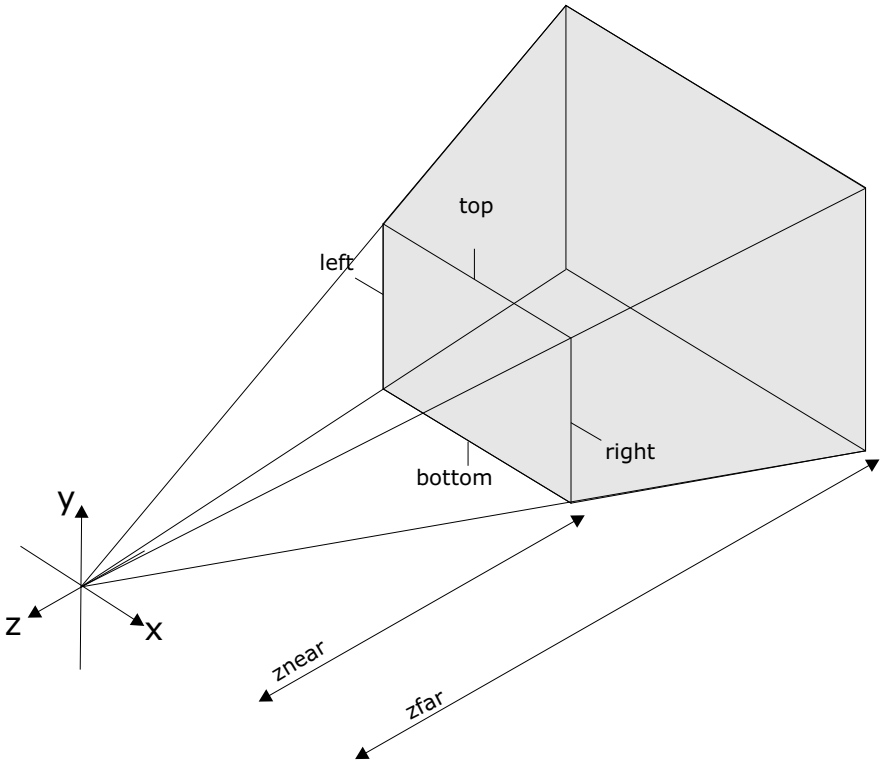

Figure 5.6: Perspective viewing volume frustum

```
glTranslatef( -s.center.x, -s.center.y, -s.center.z );
```

***Frustum* class**

Frustum
float left,right; float bottom,top; float znear,zfar;
enclose(float radius, float fov);

A *Frustum* is a geometrical object that is used for perspective projection. It can be described as a head clipped pyramid where the pyramids top is located at the origin. The bottom lies down the negative z-axis. Two planes parallel to the xy-plane located by `znear` and `zfar` clip the pyramid to a

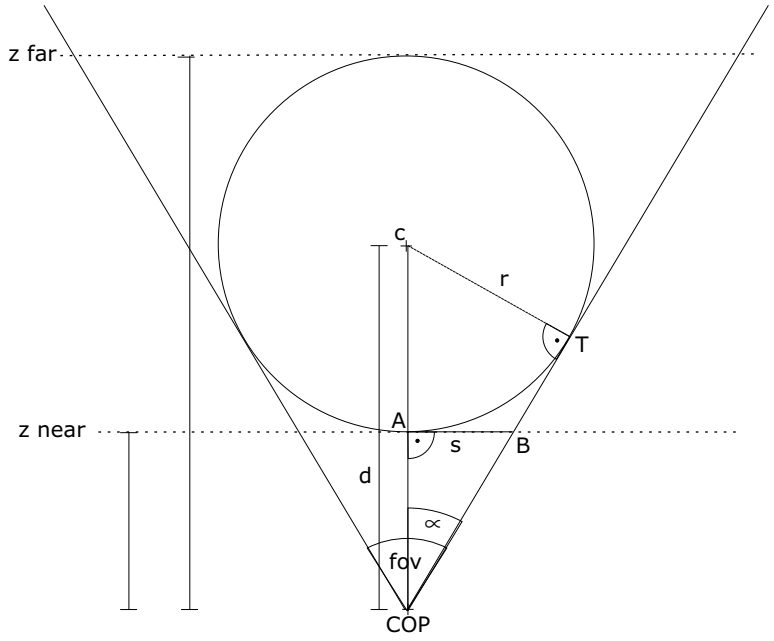

Figure 5.7: Top-view of frustum enclosing a bounding sphere volume

frustum. The left, right, bottom and top parameters describe the planar extends of the frustum at the znear plane. The frustum defines the viewing volume that is used for perspective projection as depict in figure 5.6.

The `enclose()` method computes the parameter of the frustum, so that it encloses a sphere given by the parameter `radius`. The `fov` (field-of-view angle) parameter decides the opening angle of the view volume and is analogues to the angle of a cameras aperture. Figure 5.7 shows the top view of a frustum enclosing a sphere given by the radius r and using a field-of-view angle fov. The two right triangles are drawn in which are used to derive the formula to solve the six parameters. By applying trigonmetrical laws on the right triangle connected by the points COP, T and C , the distance d is computed as given in equation 5.3. The $znear$ and $zfar$ planes are derived from d. The line length s is computed using the right triangle connected by the points COP, B and A (equation 5.7). As the frustum is symmetrical and equally spaced in x and y direction, s can be used for the parameters `left`, `right`, `bottom` and `top`.

$$\alpha = \frac{fov}{2} \tag{5.1}$$
$$\sin\alpha = \frac{r}{d} \tag{5.2}$$
$$d = \frac{\sin\alpha}{r} \tag{5.3}$$
$$znear = d - r \tag{5.4}$$
$$zfar = znear + 2r \tag{5.5}$$
$$\tan\alpha = \frac{s}{znear} \tag{5.6}$$
$$s = znear\tan\alpha \tag{5.7}$$

AABox class

AABox
Vertex vmin,vmax

invalidate();
bool isValid();
operator+=(AABox);
operator+=(Sphere);
operator+=(Vertex);
Vertex getCenter()

The *AABox* class implements a axis-aligned bounding box volume described by two vertices, `vmin` and `vmax`. "+ =" operators are implemented for three different geometry types (*AABox*, *Sphere* and *Vertex*) that possibly extend the bounding box volume. The `invalidate()` method defines the *AABox* to be invalid, or empty, while the method `isValid()` returns `true` if it is valid, or not empty.

5.4.3 *pixmap* module

Pixmap
int width,height; unsigned char* data;

init(width, height);
load(char* fname);
save(PixmapFormat* format
, char* fname)

PixmapFormat

checkSignature(FILE* file);
load(FILE* file, Pixmap* pixmap);
save(FILE* file, Pixmap* pixmap)

The *pixmap* module provides the service to load and save pixel images. A variety of formats exists to store pixel images either in memory or on mass storage. Pixel images held in memory are optimized for access and are stored in an application-specific format. When pixel images are stored on mass storage, open file formats with detailed header information that use compression on the image data are commonly used. The *Pixmap* class is used as the memory representation of pixel images, providing the image data in an uncompressed format. The format uses three channels per pixel (red,green,blue) with eight bits per channel. The render engine uses the load service to import pixmaps to be used for texture mapping. The save service is used to implement the snapshot mechanism provided by the `rgl.snapshot` API function. Loading and saving of pixmaps is implemented using the abstract interface *PixmapFormat* to support multiple file formats. A PNG pixel format handler is implemented in the module *pngpixmap* using the free libPNG library.

5.4.4 *gui* module

The *gui* module provides an abstract windowing toolkit with support for OpenGL contexts. The toolkit provides basic building blocks to implement new graphical user-interface components in a platform-independent way. The software design is shown in a UML class diagram depict in figure 5.8. The *View* base class is a logical area on a window. The *Window* class derived from the *View* class and possesses a real window. The window implementation is encapsulated through the interface *WindowImpl* which *bridges* the abstract component hierarchy with an implementation hierarchy. This design is adapted from the "Bridge" pattern and is described in detail in [7] page 151. Window Implementation *objects* providing the *WindowImpl* interface

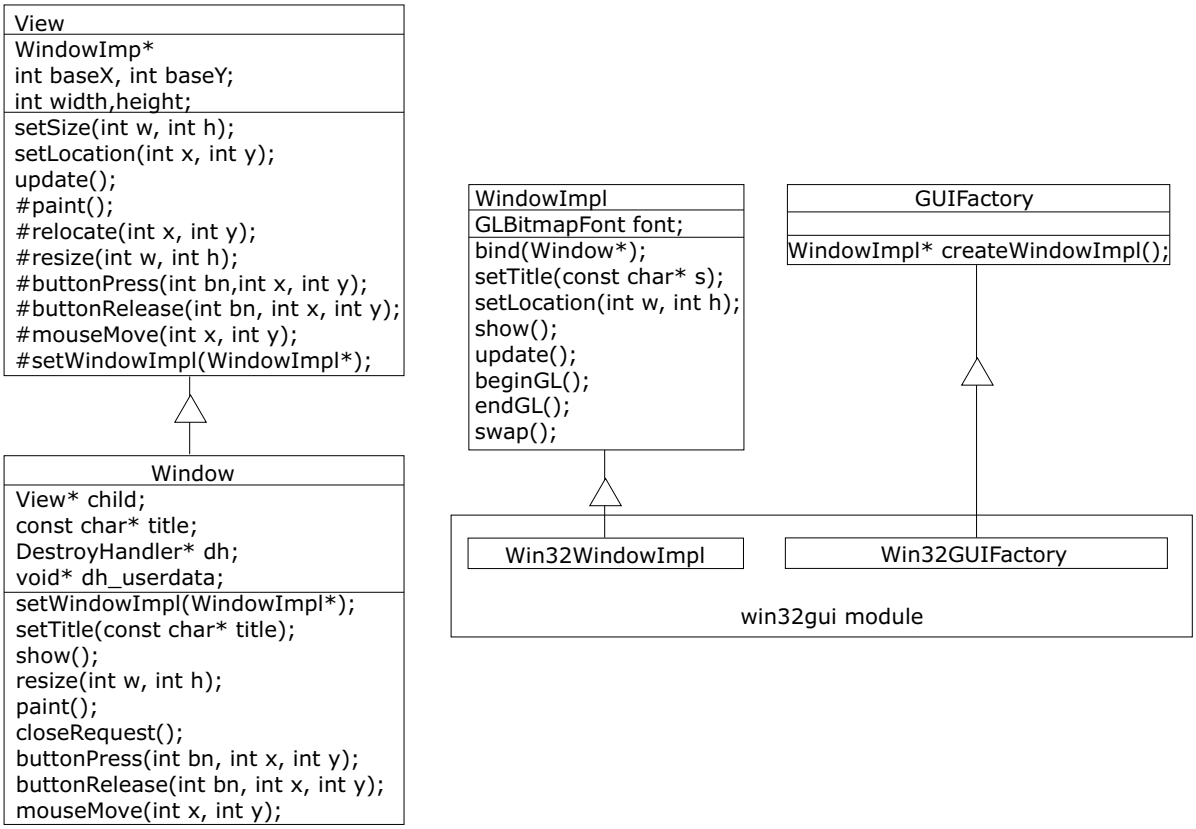

Figure 5.8: *gui* classes

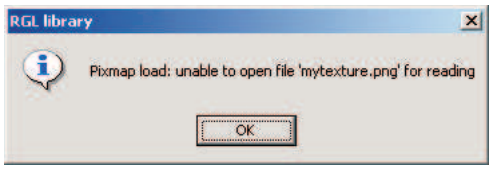

Figure 5.9: *printMessage()* on Win32 platform

are created through the *GUIFactory* interface using `createWindowImpl()`. This mechanism has been adapted from the "Factory Method" pattern (see for details [7], page 107). A platform-specific implementation requires two things:

- *WindowImpl* is derived and the window object of the specific platform is implemented there.
- *GUIFactory* is derived and the creation of the window object is implemented there.

The combination of both patterns in a software design decouples the implementation and abstraction. Extending the toolkit with new components takes place in the abstract hierarchy, while new base services extend the "Bridge" through inheritance of WindowImpl. The gui toolkit is a foundation for future improvements of the RGL gui. See section 7.2.5 for details.

5.4.5 *lib* module

The *lib* module provides an abstract interface for basic operating-system services and is responsible for library initialization and shutdown.

Basic services

```
void printMessage(const char* string);
```

`printMessage()` prints a text message to the user. On the Win32 platform a typical message box appears(Figure 5.9).

Initialization and shutdown

Shared libraries provide an entry and an exit point. The implementation of these points is platform dependent.

On initialization, two things must be initialized:

1. a *GUIFactory* implementation (e.g. *Win32GUIFactory*) is instantiated and a pointer to the *GUIFactory* interface (using casting) is stored in the global `guiFactory`.
2. a *DeviceManager* object is instantiated and a pointer is stored in the global `deviceManager`.

On shutdown, the objects must be destroyed in reversed order:

1. the *DeviceManager* object pointed by `deviceManager` is destructed.
2. the *GUIFactory* implementation is destructed.

5.5 *scene* module

The module contains the *Scene* class that implements the database management and rendering. Furthermore, a class hierarchy of 15 database classes depict in figure 5.10 and several utility classes, including the *Material* class, are provided.

5.5.1 Database

In section 4.3, the logical model has been explained. This section discusses the database implementation details.

The database provides a tiny interface of three generic methods to edit the scene:

```
class Scene { // ...
  public:
    void clear(SceneNode::TypeID selection);
```

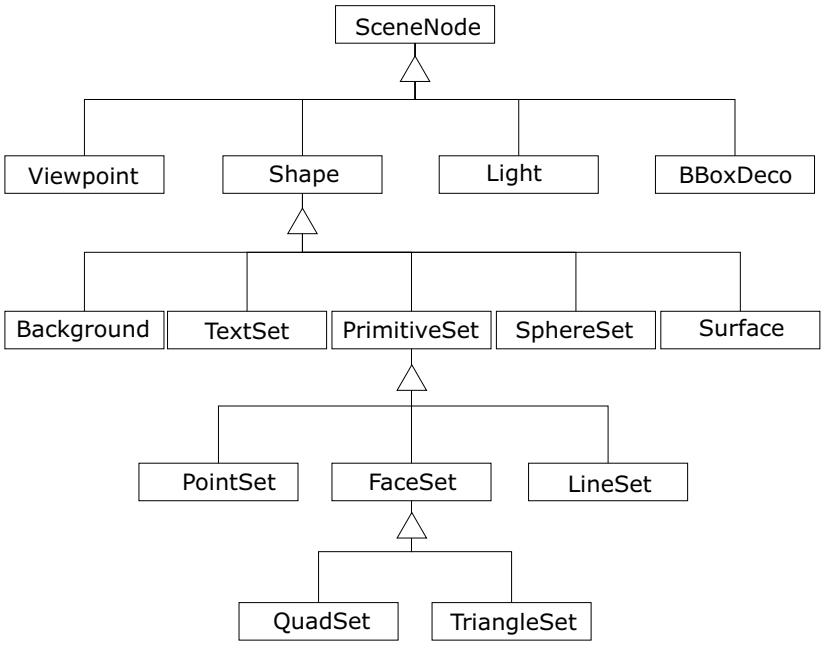

Figure 5.10: Class hierarchy of the scene database

```
    void push (SceneNode* node);
    void pop  (SceneNode::TypeID selection);
// ...
};
```

The interface is designed to be small, so that it can be extended with new object types without changing the scene interface. Therefore, the object type must be identified at run-time.

Run-time type information

SceneNode
TypeID typeID;
SceneNode(TypeID); TypeID getTypeID();

The abstract *SceneNode* base class implements the run-time type information. It is derived from the *Node* class, so that *SceneNode* objects can be

held in a *List* container.

The enumeration datatype *TypeID* is used as the run-time type identifier:

```
enum TypeID { SHAPE, LIGHT, BBOXDECO, VIEWPOINT, BACKGROUND };
```

Run-time type information requires a consistent usage across the class hierarchy. All derived classes are forced to provide run-time type information. This is guranteed through the constructor:

```
// protected constructor:

SceneNode::SceneNode(const SceneNode::TypeID inTypeID)
: typeID(inTypeID) { }
```

Derived classes are forced to pass a `TypeID` parameter when calling the *SceneNode* constructor. As there is no default constructor, an error will occur at compile-time if a derived class does not call the *SceneNode* constructor in a proper way.

Database structure

Scene
List shapes; List lights; int nlights; Viewpoint* viewpoint; BBoxDeco* bboxDeco; Background* background; AABox boundingVolume;
push(SceneNode*) pop(TypeID) clear(TypeID) render(RenderContext*)

The shape and light stacks are implemented using the *List* container class. The database keeps track of the number of lights using `nlights` as a counter,

as there is a limitation of eight lights maximum due to OpenGL[3]. *Viewpoint*, *Bounding Box* (*BBoxDeco* class) and *Background* are held in single object slots. They can be replaced by new objects. At minimum, the database must hold a *Viewpoint* and a *Background* object.

As database objects contain a *TypeID*, they can be identified dynamically at run-time. The client constructs the object and *pushes* it to the database. The `push()` method checks the type information of the incoming object and inserts the object in a proper way, adding *Shape* and *Light* objects to the stacks, and replacing *Viewpoint*, *Background* and *BBoxDeco* objects. `pop()` and `clear()` are used to delete objects from the database. The `selection` parameter selects the object type where the operation should take place. It should be `SHAPE`, `LIGHT` or `BBOXDECO`. `VIEWPOINT` and `BACKGROUND` are ignored as *Viewpoint* and *Background* objects can be replaced, but not removed. While `clean()` deletes all objects of a given type, `pop()` removes the top-most object on the stack. As one *BBoxDeco* object is held at a time, `clear()` and `pop()` delete the *BBoxDeco* object when `selection` is set to `BBOXDECO`.

The total bounding volume of all shapes is stored in the `boundingVolume` *AABox* object. Shapes contain a *AABox*, that is added to the `boundingVolume` to capture the new extends. When shapes are removed, the shape stack is traversed to recalculate the total extends.

5.5.2 Rendering

The rendering process is implemented in the *Scene* method `render()`:

```
void Scene::render(RenderContext* renderContext);
```

The method is called by the graphical user-interface component, when painting its surface. An OpenGL context must have been obtained and activated before method entry. The `renderContext` object is passed and contains informations about the frame to be rendered.

[3]OpenGL gurantees a minimum of eight lights, implementation are free to support more lights

RenderContext
Scene* scene; RectSize size; Viewpoint* viewpoint; GLBitmapFont* font;

The *RectSize* `size` object contains the size of the *Window*, where the rendering is rasterized. The *GLBitmapFont* object encapsulates an OpenGL bitmap font given by the graphical user-interface. The `viewpoint` field is explicitly given, as future improvements are planed to support multiple viewpoints.

The render method traverses the database in a specific order and calls methods to setup the OpenGL state machine, clear buffers, send geometry to the pipeline and call Display Lists. The `renderContext` pointer is passed by throughout the traversal. Different shape geometries are supported by using polymorphism.

The order of operations is described below:

1. Viewport transformation is setup according to the window
2. Depth buffer, and optionally the color buffer determined by the background, are cleared
3. Lighting model is set up, light nodes are called for setup.
4. Background is called for rendering
5. Viewpoint is called for transformation setup
6. Bounding box, if present, is called for rendering
7. Solid shape nodes, that is shapes with alpha-blending disabled, are called for rendering
8. Translucent shapes, are called for rendering

Appearance

Appearance properties for geometry are implemented in the *Material* class. It contains the attributes given in table 4.6 and is described in section 4.3.8.

The interface for OpenGL appearance setup is explained next.

```
class Material {
// ...
void beginUse(RenderContext* renderContext);
void endUse(RenderContext* renderContext);
void useColor(int index);
void colorPerVertex(bool enable, int numVertices=0);
// ...
};
```

A material is activated before the geometry is send, and deactivated afterwards. The `beginUse()` method sets up OpenGL state variables according to the material attributes. The `endUse()` method restores the OpenGL state machine. Some geometry uses a color per vertex, which is implemented using OpenGL vertex arrays. Others, such as the *SphereSet* class use one color per sphere. The `useColor()` method is used between geometry objects to switch to the next color. The `colorPerVertex()` method toggles color support for vertex arrays and recycles colors according to `numVertices`.

Shapes

Shape
Material material; AABox boundingBox; int dlistId;
virtual render(RenderContext*); virtual draw(RenderContext*) = 0;

The abstract *Shape* class provides a generic interface for rendering shapes. It contains a *Material* object and uses an OpenGL Display List for optimized appearance and geometry data transmission. The virtual `render()` method implements the usage of Display List. In case, the `render()` method is called for the first time, a Display List is initialized, the `draw()` method is called, the Display List is stored and executed. Next time, when the `render()` method gets called, the Display List is executed only. The `boundingBox` *AABox* stores the extends of the geometry.

Implementation of new *Shapes*

This section discusses in detail, what is required to implement new shape types:

1. *Constructor* implementation:

   - The *Shape* constructor is called:

     ```
     Shape(Material& in_material,
       SceneNode::TypeID in_typeID=SceneNode::TypeID::SHAPE);
     ```

     Afterwards, the shape contains a copy of `in_material` in the `material` field using the *Material* copy-constructor.
   - If several features are not available, such as texture mapping, or even lighting calculation, the features should be disabled in the `material` object.
   - Geometry data, passed as constructor arguments, is copied or generated into geometry storage classes *VertexArray*, *NormalArray* and *TexCoordArray* that support OpenGL vertex arrays, provided in the *scene* module. The `boundingVolume` contained in the *Shape* is modified when new vertices are inserted to match the extends of the shape.

     An introspection of the material object gives several hints, what geometry should be generated:

     – If lighting is enabled, normals should be generated for faces.
     – If texture mapping is enabled, 2D texture coordinates should be generated.

2. `draw()` method implementation:

   The draw method is used for sending static geometry and appearance information. A typical procedure is listed below:

   (a) Material is activated by calling `material.beginUse()`
   (b) Send geometry by using OpenGL commands and the geometry storage classes. Switch current color inbetween, where appropriate by calling `material.useColor()`.

   (c) Material is deactived by calling `material.endUse()`

3. `render()` method can be overloaded, if required:

   If the default behaviour is not acceptable, the virtual `render()` method can be overloaded. This has been done in the *Background* class.

The seven shape classes *Points*, *Lines*, *Triangles*, *Quads*, *Texts*, *Spheres* and *Surface* differ in the way the geometry is defined. A description is omitted, as the general concept has been descriped in detail.

Background

The *Background* class is used for Color Buffer clearing, background geometry rendering and fog parameter setup.

The Color Buffer clearing is implemented using the method:

```
GLbitfield Background::setupClear(RenderContext* renderContext);
```

If the background sphere geometry rendering is enabled and the material is set up to fill the complete Color Buffer, buffer clearing can be ommited. The method `setupClear()` is called by the `Scene::render()` method which returns a bitfield flag that contains either `GL_COLOR_BUFFER_BIT` (selecting the clear buffer to clear) or `0` (do not clear). The actual clearing operation is implemented in the `Scene::render()`, where the Depth Buffer is cleared in parallel, as OpenGL suports multiple buffer clearing at once.

The `render()` method overloads the `Shape::render()` method:

1. set up viewpoint orientation transformation
2. render geometry using `Shape::render()` which calls `Background::draw()` once, when compiling the displaylist and further calls the display list.
3. set up fog

Bounding Box

The *BBoxDeco* implements the bounding box as described in detail in section 4.3.7. It has not been derived from Shape, as it does not use Display Lists due to its varying geometry. The drawing of the axis tick-marks depends on the viewpoint location.

The inside visible faces (back faces) of the axis-aligned bounding box geometry are drawn only. Axis tick-marks appear on the edge contours of the visible faces. Further, only the nearest contour for a dimension is taken.

The algorithm to solve this problem is given next:

1. The bounding box geometry is transformed in the application stage.
2. Visible faces of the bounding box geometry are drawn. An 8 vertices $\times$ 8 vertex adjacent matrix is used to mark the edges that are involved when a face is drawn.
3. The contour edges are given by the adjacent matrix, that is, edges that are connected between two vertices only once.
4. A static data structure, containing fixed edge lists to be used for labeling the three axis dimensions, selects the contour edges for a specific axis labeling and and the nearest to the viewpoint of the selection is taken.

An example is given, illustrated in figure 5.11

The picture to the left shows the mesh structure of the bounding box including the vertex indices. As this picture is drawn in the standard coordinate system described in section 4.3.1, the possible candidates for labeling the three axis are given below:

Axis	*possible edges*
x	$\{1,2\},\{3,4\},\{5,6\},\{7,8\}$
y	$\{1,3\},\{2,4\},\{5,7\},\{6,8\}$
z	$\{1,5\},\{2,6\},\{3,7\},\{4,8\}$

The picture to the right illustrates a sample scene to be seen from the viewpoint.

Visible faces: $\{2,1,5,6\},\{1,3,7,5\},\{1,2,4,3\}$

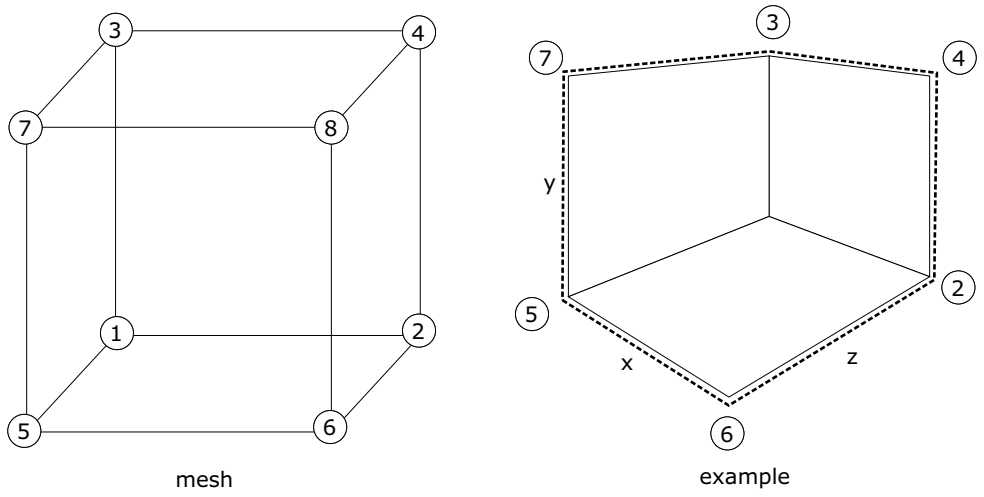

Figure 5.11: Bounding box mesh structure (left) and example with contours and axis (right)

Now, an 8×8 adjacent matrix is build. For each visible face four entries in the matrix are made, using the row of the vertex index where the edge starts and the column of the vertex index where the edge ends.

	$v1$	$v2$	$v3$	$v4$	$v5$	$v6$	$v7$	$v8$
$v1$	0	1	1	0	1	0	0	0
$v2$	1	0	0	1	0	0	0	0
$v3$	1	0	0	0	0	0	1	0
$v4$	0	0	1	0	0	0	0	0
$v5$	1	0	0	0	0	1	0	0
$v6$	0	1	0	0	0	0	0	0
$v7$	0	0	0	0	1	0	0	0
$v8$	0	0	0	0	0	0	0	0

The contours, depict as an itched line in the figure 5.11 is evaluated by searching vertex pairs, that have an edge without a counter-part in the opposite direction.

Contour edges: $\{2,4\}, \{6,2\}, \{4,3\}, \{3,7\}, \{5,6\}, \{7,5\}$

For each axis, the valid contour edges are selected, and the nearest edge to the viewpoint is taken:

X axis	$\{4,3\},\{5,6\} \rightarrow \{5,6\}$
Y axis	$\{2,4\},\{7,5\} \rightarrow \{7,5\}$
Z axis	$\{6,2\},\{3,7\} \rightarrow \{6,2\}$

Translucent rendering

When rendering translucent shapes, the depth buffer is set read-only. Otherwise, translucent shapes that lie in front of others and rendered first would prevent translucent shapes in the back to be rendered afterwards. As solid shapes have been written with depth buffer in read-write mode, translucent shapes will not be rendered uppon solid shapes that are in front of them. With this technique, no surfaces must be sorted and can be rendered in any order. There is a draw-back with this mechanism. The alpha-blended shapes are rendered in a FIFO[4] order as they appear on the stack. But to render translucent images correctly, the order must depend on the distance to the viewpoint, rendering the polygons from far to near. Special treatment of translucent primitives is planed for the future.

Viewpoint class

Viewpoint
PolarCoord position; bool interactive; float fov;

setPosition();
setFOV();
bool isInteractive();
setupTransformation();
setupOrientation(Sphere sphere);

The *Viewpoint* class implements the polar coordinate navigation described in section 4.3.4. The implementation uses the transformation of polar coordinates that has been described in detail in section 5.4.2. It provides two methods for polar coordinate transformation:

- `setupTransformation()` computes a reference sphere from the total

[4]First-In First-Out

bounding volume of all spheres given by scene, sets up the viewing volumn as described in section 5.4.2 and transforms the model and view according to polar coordinates.

- `setupOrientation()` implements the polar coordinate rotation only.

It also provides an interface to change position, field-of-view, zoom and interactivity, which is used by the graphical user-interface component for interactive navigation.

5.6 Device

The device domain consists of three modules:

device	device implementation
rglview	graphical user-interface component
scene	database and rendering engine

The *scene* module has been discussed in section 5.5. The *rglview* module is described next, followed by the *device* module.

5.6.1 *rglview* module

The module *rglview* contains the *RGLView* class.

RGLView class

RGLView
Scene* scene; Viewpoint* viewpoint; RenderContext rctx

update();
snapshot();
#resize();
#paint();
#buttonPress();
#buttonRelease();
#mouseMove();

The class *RGLView* is derived from the *View* class provided by the *gui* module. It implements the graphical user-interface and is responsible for three tasks:

- Launch a rendering job by calling `Scene::render()` and display it in the window using double-buffering.
- Handling input-events coming from the pointing device and modifying the *Viewpoint* object as described in section 4.3.4.
- Exporting a snapshot to mass storage using the *pixmap* module.

5.6.2 *device* module

The module *device* contains the *Device* class.

Device class

Device
Window* window; RGLView* rglview; Scene* scene;

push(SceneNode*);
pop(TypeID selection);
clear(TypeID selection);
export(int format, const char*fname);

The *Device* class provides the public interface of the logical device. It initializes a device appropriately, creating a *Scene* instance, *RGLView* instance and a *Window* instance. A *Scene* object pointer is passed to the `RGLView` object. The *RGLView* object is put into the *Window* object.

The `push()`, `pop()` and `clear()` methods are delegated to the *Scene* object. An additional `update()` method call on the *RGLView* object makes the changes in the database visible instantly. The `export()` method is delegated to the *RGLView* object.

5.7 Client

The client domain consists of two modules:

api	C interface of the API
devicemanager	device management logic

5.7.1 *api* module

The *api* module implements the C++ backend functions of the API. They are prefixed with "`rgl_`". API functions are directly implemented in R.

The api module contains two global variables:

- `DeviceManager* deviceManager`: A pointer to a global object, that is used to obtain a handle to the currently active device.
- `Material material`: The `material` object contains the appearance information, that is last stored due to the API function `rgl.material()`.

The implementation backend is briefly described:

- *Shape Functions* and *Environment Setup* functions construct the appropriate database object using the `material` object for appearance properties. The object is pushed to the device obtained by the `deviceManager` object.
- *Device Management* function calls are delegated to the `deviceManager` object.
- *Scene Management* and *Export* function calls are delegated to the currently active *Device* object, obtained by the `deviceManager` object.
- The *Appearance* function `rgl_material()` sets up the `material` object according to the appearance properties.

5.7.2 *devicemanager* module

The *devicemanager* module contains the *Device* class.

***Device* class**

DeviceManager
Device* current; List devices;

Device* openDevice();
Device* getCurrentDevice();
Device* getAnyDevice();
setCurrent(int id);
int getCurrent();

The *Device* class holds all opened devices in the *List* `devices`. `setCurrent` and `getCurrent` implement the `rgl.set()` and `rgl.cur()` API functions that set and query the current device focus.

The `current` field points to the currently active *Device* object, which is accessed by `getCurrentDevice()` or `getAnyDevice()`:

- `getCurrentDevice()` returns a pointer to the current device, or if none exists, return `NULL`.
- `getAnyDevice()` returns a pointer to the current device, or if none exists, it opens a new device. The behaviour has been adapted from R, which has been described in section 2.2.2.

`openDevice()` explicitly creates a new *Device* instance and sets it as the active one.

5.8 API implementation

The API implementation at the top-level is implemented in R. As the shared libary has been described in detail, the chapter completes with a description of the API implementation written in the R language.

R storage mode	*C type*
logical	int *
integer	int *
double	double *
complex	Rcomplex *
character	char **

Table 5.1: Data type mapping between R and C

5.8.1 Interface and data passing between R and C

R functions call shared library C++ functions[5] using the `.C(name, ...)` command. Arguments are passed using a data type mapping scheme given in table 5.1. As R objects are stored in vectors, the C++ parameters are pointers to data vectors. The data vectors are valid until the C++ function returns. If they must be stored for later use, they are copied into allocated memory.

Some interface design conventions have been used, that are listed below:

- Values of the same data type are packed together into one vector.
- Data with varying length (e.g. geometry data) are added to the tail of a fixed length vector when possible, otherwise passed as separate vectors. The length is stored in a cell of an integer-type vector.
- Values are returned using named arguments. In C, a return value is written in a cell that has been given as a pointer to it and is a tagged value in R.

5.8.2 R functions

The R source code is divided into five source code units:

[5] C++ allows static functions to be declared as C functions, the implemention uses C++ statements. Therefore, they are called C++ functions.

zzz.R	package entry and exit code
_internal.R	internal functions
device.R	device management functions and export
scene.R	scene manipulation, plotting and environment setup
material.R	`rgl.material`

Most API functions are implemented by calling a counter-part function in shared library:

- *Appearance* function: The `rgl.material()` function calls `rgl_material()`, that sets up the `material` object.
- *Shape* functions and *Environment Setup* functions: The "..." argument is passed to `rgl.material()` Afterwards, the actual counter-part function gets called, which constructs the desired C++ database object using the `material` object for appearance parameters and passes it to the device.
- *Device Management*, *Export* and *Scene Management* functions: These functions call their counter-part C++ functions.

Internal functions are provided for data value boundary checks, enumeration datatype conversion between R and C++ and data vector recycling.

Chapter 6

Examples

This section demonstrates the capabilities of the current RGL version with some examples.

6.1 Launching RGL

The RGL package will be distributed through CRAN and a Win32 version is attached to this document.[1]

RGL is launched by entering the following command:

```
> library(rgl)
```

If no errors occur, the RGL device system is loaded and ready to receive requests. The following line will open a device window:

```
> rgl.open()
```

[1]Before making RGL publicly available, thorough testing is required.

6.2 Statistical data analysis

6.2.1 Estimating animal abundance

Color plates 7 and 8 show a visualization model of animal abundance estimation [4] from different viewpoints. The sample density is displayed in topological terms, regions with higher density are displayed with higher altitude and vice versa. Regions with a density of zero are displayed as rivers.

This landscape-like visualization is carried out using a *Surface* shape. Futhermore, *Spheres* are used for displaying the sampled population. In this case, the populations are characterized by:

Parameter	*Characteristics*
`radius`	the size of the group
`color`	gender: *red* = female, *blue* = male
`alpha`	exposure of the group

6.2.2 Kernel smoothing

This example uses the *sm.library*, that implements nonparametric smoothing methods described in [5]. Two randomly generated samples from a standard normal distribution are created.

```
n<-100; ngrid<-100
x<-rnorm(n); z<-rnorm(n)
```

The density surface is estimated with kernel smoothing:

```
smobj<-sm.density(cbind(x,z), display="none", ngrid=ngrid)
sm.y <-smobj$estimate
```

As the samples come from a bivariate normal distribution, a parameteric density surface is generated using the ranges of the samples.

```
xgrid <- seq(min(x),max(x),len=ngrid)
zgrid <- seq(min(z),max(z),len=ngrid)
bi.y <- dnorm(xgrid)%*%t(dnorm(zgrid))
```

Now, the samples are visualized as spheres, the estimated non-parametric density surface and the density surface of a bivariate this sample using `rgl.spheres()`:

```
yscale<-20
rgl.clear(); rgl.bg(color="#887777")
rgl.spheres(x,rep(0,n),z,radius=0.1,color="#CCCCFF")
rgl.surface(xgrid,zgrid,sm.y*yscale,color="#FF2222",alpha=0.5)
rgl.surface(xgrid,zgrid,bi.y*yscale,color="#CCCCFF",front="lines")
```

A screen shot of this visualization is shown in Color plate 4.

6.2.3 Real-time animations

A round flight animation can be created by issueing following commands:

```
example(rgl.surface)
for(i in 1:360) {
  rgl.viewpoint(i, i*(60/360), interactive=F)
}
```

6.2.4 Image generation for animation

The example from the above is slightly modified, as after each frame a snapshot is taken and saved to mass storage.

```
for(i in 1:360) {
  rgl.viewpoint(i, i*(60/360), interactive=F)
  rgl.snapshot( paste("d:/tmp/file",i,".png") );
}
```

Chapter 7

Summary and Outlook

7.1 Summary

This section gives a summary of the work presented in the preceding chapters. The major goal of this project was to implement a real-time rendering package for R, that exploits the current situation on the graphics hardware market, where accelerated graphics hardware cards are available for desktop systems at a low price, that would have cost a fortune, years ago. OpenGL has been choosen as the rendering platform, as it is portable across all R platforms, and its scalability for accelerated graphics hardware is currently well supported by hardware vendors.

What has been evolved actually, is a visualization device system, designed for scientists and students used to R graphics plotting facilities. The adaptation approach has shown, that it is possible to use three-dimensional visualizations in a quite same way as using plotting devices, if some core components are well designed. Automatic adjustment of the viewing volume and the bounding box allow the user to concentrate on the data visualization. Appearance features such as lighting, alpha-blending, texture-mapping and fog give an added value to visualizations and possibly motivate users in long-term work sessions.

The software design has been build with long-term goals in mind and profits from its solid architecture for cross-platform portability and complexity reduction due to modularization and object-oriented design.

The following text is quoted from the `README` textfile contained in the R 1.5.1 source-code distribution:

"GOALS [...] Longer-term goals include to explore new ideas: e.g. virtual objects and component-based programming, and expanding the scope of existing ones like formula-based interfaces. Further, we wish to get a handle on a general approach to graphical user interfaces (preferably with cross-platform portability), and to develop better 3-D and dynamic graphics."

Hopefully, this package will have an impact on the further development of R. The integration of plotting and visualization devices with a common interface is desirably.

7.2 Outlook

As of the time of writing, RGL is in the version 0.60a. It is released under the GNU Public License Version 2. The author will maintain the software in the future, fixing bugs and extending the feature list. Several improvements and ideas did not get it into this first release due to time-constraints. The following sections will briefly list several improvements that are planed:

7.2.1 Ports

A port to X11 and Macintosh is planed (in that order).

7.2.2 Rendering improvements

- Optimizing the output quality (optional), using the accumulation buffer for effects such as depth-of-field, motion-blur and anti-aliasing.
- Render alpha-blended faces in a z-distance sorted order.
- Visualize distance lights using lens flares effect.
- Support for vector fonts.

7.2.3 Functionality improvements

- Interactive 3D selection and identification techniques analogue to `identify()` and `locate()` in the R graphical plotting facilities.
- Dynamic simulation visualizations e.g. time-series data and scripting to implement kinetics
- Import and export of scenes using a subset of VRML97 and X3D[1] formats.

7.2.4 Scene improvements

- *PixmapSet* shape: This shape type uses pixmaps as base primitives. Atmospherical clouds effects using alpha-channel pixmaps are considered.
- *Bounding box*: Axis labels and separate axis scalings.

7.2.5 GUI improvements

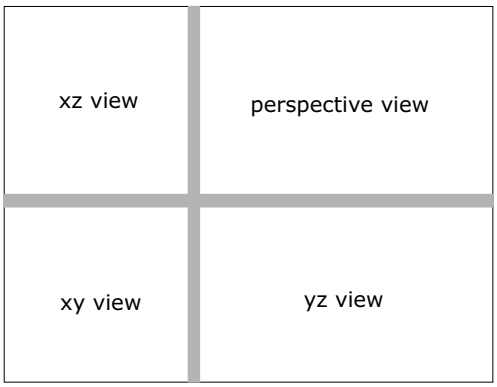

Figure 7.1: Graphical user-interface component *FourView*

A graphical user-interface component is planed, named *FourView* and depict in figure 7.2.5. It contains four child views, each rendering the data model

[1] more information of X3D, see http://www.web3d.org (2002-10-09)

using different projections. The view in the top right corner contains the *RGLView* component. The other three views display an orthogonal projection and will be suited with an interactive traverse facility in space using the pointer device, so that particular regions can be focused. The cross is used to adjust the size of the child views by dragging it across in the component area.

7.2.6 R programing interface improvements

- The visualization of data could be generalized through a generic function, analogue to the `plot()` function in R.

This list of planed improvements is far from complete. Hopefully, the public testing will yield many suggestions, which are gratefully acknowledged and most likely considered for the future versions.

Bibliography

[1] Christopher Alexander. *The Timeless Way of Building*. Oxford University Press, 1979.

[2] Edward Angel. *Interactive computer graphics : a top-down approach with OpenGL*. Addison-Wesley, New Mexico, 1997.

[3] Richard A. Becker, John M. Chambers, and Allan R. Wilks. *The New S Language*. Chapman and Hall, New York, 1988.

[4] Buckland S.T. Borchers, D.L. and W. Zucchini. *Estimating Animal Abundance: Closed Populations*. Springer, Berlin, 2002.

[5] A.W. Bowman and A. Azzalini. *Applied Smoothing Techniques for Data Analysis: the Kernel Approach with S-Plus Illustrations*. Oxford University Press, Oxford, 1997.

[6] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-orientierte Software-Architektur*. Addison-Wesley, Germany, 1998.

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

[8] Nicolai Josuttis. *Objektorientiertes Programmieren in C++*. Addison-Wesley, Germany, 1994.

[9] Brian W. Kerninghan and Dennis M. Richie. *The C Programming Language*. Prentice Hall, United States of America.

[10] Steve Kleiman, Devang Shah, and Bart Smaalders. *Programming with Threads*. SunSoft Press, United States of America, 1996.

[11] John R. Levine. *Linkers and Loaders*. Academic Press, United States of America, 2000.

[12] Martin Lowes and Augustin Paulik. *Programmieren in C: Ansi Standard*. Teubner Stuttgart, Germany, 1995.

[13] Thomas Moeller and Eric Haines. *Real-Time Rendering*. A K Peters, Ltd., United States of America, 1999.

[14] Adrian Nye. *Xlib Programming Manual for Version 11*. O'Reilly and Associates, United States of America, 1988.

[15] Adrian Nye. *X Protocol Reference Manual for X11 Version 4, Release 6*. O'Reilly and Associates, United States of America, 1989.

[16] Bernd Oestereich. *Objekt-orientierte Software-entwicklung*. R. Oldenbourg Verlag, Germany, 1998.

[17] R project. *R Data Import/Export*. http://www.r-project.org.

[18] R project. *R Language Definition*. http://www.r-project.org.

[19] R project. *Writing R Extensions*. http://www.r-project.org.

[20] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1997.

[21] Alan Watt. *3D Computer Grafik*. Pearson Education Limited, 2000.

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