Max–Planck–Institut für biologische Kybernetik
Max Planck Institute for Biological Cybernetics

Technical Report No. 152

veLua - an interface between the Virtual Environments Library (veLib) and the programming language Lua

Library Version 1.6.0

Gerald Franz

August 2006

[1]Department Bülthoff, Max Planck Institute for Biological Cybernetics, Spemannstr. 38, 72076 Tübingen, Germany.
E-mail: gerald.franz@tuebingen.mpg.de

This report is available in PDF–format via anonymous ftp at ftp://ftp.kyb.tuebingen.mpg.de/pub/mpi-memos/pdf/TR-152.pdf. The complete series of Technical Reports is documented at: http://www.kyb.tuebingen.mpg.de/bu/techr/

veLua - an interface between the Virtual Environments Library (veLib) and the programming language Lua

Library Version 1.6.0

Gerald Franz

Abstract. Abstract The Virtual Environments Library (veLib) is an extensible framework for the development of distributed realtime high-performance virtual reality applications. This paper describes the recently added facultative interface to the programming language Lua, which allows users to develop complete simulations without digging into the complexities of the C++ programming language underlying the veLib.

Keywords: Virtual reality, distributed systems, veLib, Lua.

Contents

1 Introduction

1.1 Background

Today's software projects typically consist of several thousands lines of code which makes the learning and understanding of the code and its application a considerable effort. In order to respond to this, a common countermeasure is the introduction of several layers of abstraction which expose differently complex interfaces for different purposes. This structure enables users to selectively learn a software step by step and significantly reduces the time necessary to create first productive applications. Furthermore, this layered structure accounts for different levels of expertise between users and developers and is able to provide both groups with an appropriate level of flexibility and power.

In the computer game industries, the different skill levels and fields of responsibilities of programmers and game designers have led to a model of a complete separation between low level functionalities (often called the game engine), which are shared between multiple games and coded in the programming languages C or C++, and the particular game logic content, which is implemented using a specialized language. Besides custom solutions for individual projects, in particular the programming language Lua has found widespread application for this purpose (e.g., World of Warcraft, Far Cry, Half-Life 2).

1.2 Objective

This document describes the Lua interface of the veLib, which is implemented in the files veLua.h and veLua.cpp. This interface greatly simplifies the creation of veLib applications and often leads to a more intuitive and logical code structure. In fact, veLua enables users to write sophisticated applications without requiring a C++ compiler or a complex development environment. Often all necessary simulation logic can be implemented by adding a few lines of Lua code to the veLib initialization files.

The intention of this document is to provide a fairly complete yet condensed reference manual of veLua, therefore it will be neither a programming guide to the Lua language itself nor a complete introduction to the veLib. Nevertheless, readers that have a basic knowledge of any remotely C-like programming language such as C++, Pascal, Java, Perl, Python, or even BASIC and also roughly understand the central entities of the veLib (see following paragraph) should quickly become able to write basic veLua applications based on the provided documentation.

After a quick overview on the veLib and Lua, the following Section 2 describes the few veLib-specific additions to the Lua language. Afterwards, the specific extensions to the veLib's XML based initialization files are documented which enable Lua scripts to read input data from devices and to control simulation objects. Finally, the veLua C++ application programmers' interface is briefly described. This section mainly addresses experienced veLib programmers and users and is not necessary for the mere application of veLua.

veLua status. veLua has been developed with low priority during the first half of 2006 and has finally reached a state of sufficient maturity for a first public release in August 2006. A test application scenario (see Figure 1) has proven its stability and general usefulness for realworld applications, especially as alternative for users that are not too familiar with advanced concepts of the C++ programming language. One can expect that future developments are mainly extensions of the currently minimal scope. Due to the relative novelty of this interface, however, some changes introducing minor incompatibilities of future versions cannot be completely ruled out.

1.3 Central entities of the veLib

The Virtual Environments Library (veLib) is an extensible framework for the development of distributed realtime virtual reality applications. Among the fairly large collection of C++ classes making up the veLib, two types of entities play a central role, devices and simulation objects.

A main goal of the veLib is the separation of program logic from the peculiarities of controlling and interacting with the underlying hardware or low-level programming interfaces. This objective is achieved

Figure 1: Screenshot of the veLua example application.

by the concept of an abstract *device* interface. A large number of physical device operations (e.g., communicating with joysticks, keyboards, mice, sound cards, motion platforms) or virtual device operations (network connections, 3D graphics libraries, logfiles) consist of a specific initialization at the beginning and a more or less steady data flow afterwards transmitting changes of the device state (i.e. its input) to the application and/or transmitting application changes (i.e. output) to the devices. The veLib accounts for this by providing a family of device classes that all share a similar initialization mechanism based on specific XML files and a common set of input/output methods for the runtime communication. In fact, recent veLib applications instantiate devices solely depending upon the information provided in the initialization files. This allows for a great flexibility such that applications can be ported between different setups solely by editing a few lines of XML code.

The second basic entity of the veLib are *simulation objects* which are a generic abstraction of the scene or simulation content. In the underlying C++ code, simulation objects are implemented as various types of ve::dataContainers. At this basic level, simulation objects are a data structure consisting of various axes, flags, and ids which describe the entire state of an object at a given time (e.g., its position, velocity, visibility) and relate the object to specific resources. Certain devices make use of these resources (e.g., geometry models, sound files, textures) in order to provide a perceptual (e.g., visual, auditory) representation for the user. Simulation objects have also special axis and flag containers to query and store the input state of an associated input device. Furthermore, the data containers underlying simulation objects are the basic entities of the veLib network protocol and therefore allow simulations to be distributed over multiple computers.

1.4 The Lua programming language

According to its official website `http://www.lua.org`, “Lua is a powerful light-weight programming language designed for extending applications”. Lua combines a simple procedural syntax similar to Pascal with powerful and flexible data description constructs based on associative arrays which allow for object-oriented designs. Lua is dynamically typed, interpreted from bytecode, and has automatic memory management with garbage collection. This renders it ideal for configuration, scripting, and at the same time convenient for the user and easily learnable.

Further arguments that militate for Lua are its small and portable code, high speed, mature state yet still constant improvement, liberal license (MIT style, see Appendix B), its large user base, and the complete and very instructive documentation. For further information about this nice and modern language, please refer to its website, particularly to the reference manual (Ierusalimschy, de Figueiredo, & Celes, 2006) and the excellent introductory book by the main author (Ierusalimschy, 2006).

2 Lua API

The original release (August 2006) of veLua provides the complete functionality of Lua 5.1 including the Lua standard libraries. Therefore, Lua scripts can make use of all mathematical functions, string manipulation, and file input/output facilities of Lua as documented in the Lua manual (Ierusalimschy et al., 2006) and user guide (Ierusalimschy, 2006). In addition, a minimal set of functions has been defined that mainly allow Lua scripts to access application variables and to control veLib simulation objects (cf. Section 1.3).

veLua scripts normally are fairly brief and linear programs. Unlike conventional veLib applications written in C++, they rarely have a main loop. This is because normally the entire veLua script is executed each frame and variables keep their values beyond the individual calls. The following example script (a simple frame counter) clarifies this and shows a useful approach how to do initializations:

```
if frames==nil then -- nil means undefined variable, first call
  frames = 1
else -- entered each frame after initialization
  frames = frames + 1
end
```

2.1 Global constants

veLua defines a few global constants or symbolic names which increase the readability of often recurring code such as the selection and manipulation of simulation objects' axis containers and individual axes. In order to make the syntax as simple as possible, these constants have been added to the global namespace, so take care not to use these names for user-defined variables. In order to reduce this risk, they are all written entirely in capital letters, which is a useful and common convention for the discrimination between constants and variables.

X = 0, Y = 1, Z = 2, H = 3, P = 4, R = 5 symbolic names for standard six-degrees-of-freedom axes.

INPUT = 0, POS = 1, VEL = 2, ACC = 3 symbolic names for the veLib simulation object default axis containers.

INPUT typically refers to to six float input states between -1.0 and 1.0 (e.g., the position of the individual axes of an associated joystick)

POS represents the spatial position and orientation of a simulation object. Note that veLua uses degrees for rotation angles as the veLib, whereas standard Lua uses radians.

VEL represents the velocity of an object. Velocities are always encoded relative to the local coordinate system of an object. Therefore, axis X means left-right, Y forward-backward, Z upward-downward, H rotation around Z axis, P rotation speed around X axis, R rotation speed around Y axis.

ACC represents the current acceleration of an object with respect to its local coordinate system.

INPUT = 0, FLAGS = 1 symbolic names for the veLib simulation object default flag containers.

INPUT provides the state of the up to 128 buttons of an associated input device. If the simulation object is associated to a joystick, the standard buttons are normally just numerated starting by 0. If the simulation

object is associated to a keyboard, the individual keys are mostly identified by their ASCII value. In case of doubt, please refer to the documentation of veTypes.h.

FLAGS provides storage for 128 user-definable binary states. All of them are freely available and might be used for example to exchange information between simulation objects and code written in C++.

ALIGN_LEFT = 0, ALIGN_CENTER = 1, ALIGN_RIGHT = 2, ALIGN_BOTTOM = 3, ALIGN_TOP = 4 symbolic names for the alignment of overlay text (see Section 2.4).

2.2 Basic veLib functionality

The functions accessing general veLib functionality are collected in the Lua table construct called *ve*, analogous to the veLib's standard C++ namespace.

ve.getVar(name) generic variable read access.

Returns the value of a C++ variable exposed via the provided name or nil in case that no correspondingly named variable has been exposed (cf. Section 4.2).

ve.setVar(name, value) generic variable write access.

Sets the value of a C++ variable exposed via the provided name (cf. Section 4.2). In case that the variable has been exposed as read-only, this function call has no effect.

ve.getGlobal(name) a function reading a global value that is accessible by all veLua scripts.

veLua applications typically consist of a collection of several scripts controlling single simulation objects or aspects. While its interface favors a modular design of widely independent and specialized scripts, applications nevertheless often require a mechanism to exchange data between individual scripts or to adapt their behavior according to some global state. In veLua this is implemented via a shared hash table which is equally accessible by all scripts.

The ve.getGlobal(name) function allows accessing the global hash table and returns the value associated to the passed string name. If no suitably named entry in the global hash can be found, a nil value is returned.

ve.setGlobal(name, value) a function writing a global value that is accessible by all veLua scripts.

veLua applications typically consist of a collection of several scripts controlling single simulation objects or aspects. While its interface favors a modular design of widely independent and specialized scripts, applications nevertheless often require a mechanism to exchange data between individual scripts or to adapt their behavior according to some global state. In veLua this is implemented via a shared hash table which is equally accessible by all scripts.

The ve.setGlobal(name, value) function allows accessing the global hash table and sets the value associated to the passed string name. If a hash entry having the same name key already exists, the old value is simply overwritten. Note that the current implementation of this function stores information either numerically or as string, and does not provide a mechansim to directly store and exchange complex constructs such as tables. Due to this restriction, the global hash may also be used to communicate to the underlying C++ applicationin later versions of veLua.

ve.exit() terminates program execution.

ve.now() returns current simulation time in seconds.

ve.deltaT() returns time difference to last frame in seconds.

2.3 Simulation object interface

A central part of a typical simulation is a scene consisting of various entities or simulation objects (cf. Section 1.3). A main feature of veLua is to provide a convenient interface for implementing the dynamic behavior of these objects. Individual scripts can be bound to control the behavior of individual simulation objects. This structure normally results in small, self-contained, and therefore clear pieces of code. In addition, the same functions allow accessing the properties of further objects which allows specific events to be triggered based on the state of others (e.g., opening of a door based on the proximity of the observer). All these functions belong to the table *obj*. For more information on the actual application please refer to the example scripts below (Section 2.6), the annotated examples in the XML description (Section 3), and to the example application provided with the the veLua source code.

obj.new(name, resource[, script][, inputdevice]) adds a new object to the simulation and returns its name.

name - the first argument must be a string providing a unique name for the object

resource - defines the name of the resource which will be associated to the object

script - optional argument defining the name of a script to be used to control the behavior of the object

inputdevice - optional argument defining the name of the associated device which provides input to the object.

obj.delete(name) removes an object from the simulation. The object is identified by its name.

obj.setAxis([name,] type, axis, value) sets the value of an axis of an object.

name - optional first string argument defines the name of the object to be manipulated. If no name is provided, the default object bound to this script is taken.

type - numerical argument defining the type of axis container to be manipulated. A simulation object contains four sets of axes which are identified by a numeric constant (INPUT, POS, VEL, ACC, see Section 2.1).

axis - numerical argument defining the individual axis to be manipulated. Normally, the symbolic names X, Y, Z, H, P, or R are used (cf. Section 2.1).

value - numerical argument providing the new value of the axis.

Example: `obj.setAxis(''observer'', ACC, Z, -9.81)` - sets the acceleration of the object "observer" in Z direction to -9.81 m/s, corresponding to the average gravitational acceleration on the earth.

obj.getAxis([name,] type,axis) returns the value of an axis of an object.

name - optional first string argument defines the name of the object to be manipulated. If no name is provided, the default object bound to this script is taken.

type - numerical argument defining the type of axis container to be read. A simulation object contains four sets of axes which are identified by a numeric constant (INPUT, POS, VEL, ACC, see Section 2.1).

axis - numerical argument defining the individual axis to be read. Normally, the symbolic names X, Y, Z, H, P, or R are used (cf. Section 2.1).

Example: `posX = obj.getAxis(POS, X)` - assigns current X position of the associated simulation object to the variable posX.

obj.getAxes([name,] type) returns all six axes describing the position, velocity, acceleration, or input state of an object as one table.

name - optional first string argument defines the name of the object to be manipulated. If no name is provided, the default object bound to this script is taken.

type - numerical argument defining the type of axis container to be read. A simulation object contains four sets of axes which are identified by a numeric constant (INPUT, POS, VEL, ACC, see Section 2.1).

Example: `vel = obj.getAxes(VEL)` - assigns current velocity of the associated simulation object to the variable vel. After this command, vel is of type table, the individual axis values are stored in the subordinate variables vel.x, vel.y, vel.z, vel.h, vel.p, vel.r. Note the lowercase letters labeling the individual axis variables.

obj.setFlag([name,] type, flag, value) sets a single flag of a simulation object.

name - optional first string argument defines the name of the object to be manipulated. If no name is provided, the default object bound to this script is taken.

type - numerical argument defining the type of flag container to be manipulated. A simulation object contains two flag containers which are identified by a numeric constant (INPUT, FLAGS, see Section 2.1).

flag - argument defining the individual flag to be manipulated. Normally this is an integer value between 0 and 127. alternatively, one letter string constants (e.g., “r”, “e”) can be used, in this case the ASCII value of the letter is used.

value - the new value of the flag. Either false or true.

Example: `obj.setFlag(''lamp'', FLAGS, 0, true)` - Sets the first flag of the simulation object “lamp” to true. Another script may interpret this value as “lamp on” and update the scene accordingly.

obj.getFlag([name,] type, flag) returns the value of a single flag of a simulation object as boolean value.

name - optional first string argument defines the name of the object to be manipulated. If no name is provided, the default object bound to this script is taken.

type - numerical argument defining the type of flag container to be read. A simulation object contains two flag containers which are identified by a numeric constant (INPUT, FLAGS, see Section 2.1).

flag - argument defining the individual flag to be read. Normally this is an integer value between 0 and 127. alternatively, one letter string constants (e.g., “r”, “e”) can be used, in this case the ASCII value of the letter is used.

Example: `if obj.getFlag(INPUT, ''q'')==true then ve.exit(); end` - Reads the input flag having the ASCII value of the letter q (71) and exits the simulation if this flag is 1. If the object controlled by the script is connected to a window input device listening to keyboard events, this means that pressing the q key will terminate the program execution.

2.4 Overlay plane interface

In addition to a 3D scene, visualizations often also need some mechanism to display numeric or text information or images on the screen. Since doing this in 3D is a major hassle, the veLib and veLua provide a minimal set of functions for basic 2D graphics. While the display of rectangles, images, or text on the overlay plane is a non-trivial task using the low-level veLib functions involving the management of many data containers, veLua reduces this to a streamlined interface consisting of three versatile functions which are collected in the table *ovl*.

Please note that overlay objects are rendered in the same sequences as the functions have been called, so, if you want to display a text on top of a rectangle, you have to call ovl.rect() before ovl.text().

ovl.text(x,y, alignHor, alignVer[, text, ...]) draws textual or numerical information on the screen.
In order to change an already written text, write a different text at the same position and alignment. In order to remove a text, just call this function without any argument apart from the position and alignment values.

x - horizontal position in overlay coordinate units.

y - vertical position in overlay coordinate units.

alignHor - horizontal alignment of the text relative to the given x—y position. Valid arguments are the predefined constants ALIGN_LEFT, ALIGN_RIGHT, and ALIGN_CENTER.

alignVer - vertical alignment of the text relative to the given x—y position. Valid arguments are the predefined constants ALIGN_BOTTOM, ALIGN_TOP, and ALIGN_CENTER.

text - an arbitrary sequence of strings or numbers to be displayed.

Example: Write the position of the currently bound simulation object into the lower left corner of the screen (given that a veLib standard overlay coordinate system ranging from -1.0 to 1.0 is defined both for x and y direction):

```
pos=obj.getAxes(POS)
ovl.text(-1,-1,ALIGN_LEFT,ALIGN_BOTTOM, pos.x,' ',pos.y,' ', pos.z,'
',pos.h,' ',pos.p,' ',pos.r)
```

ovl.rect(x,y, w,h [, r,g,b,a]) draws a rectangle on the screen.
In order to change the color of an already displayed rectangle, the function can be called again with the same coordinates and the new color. In order to remove a rectangle, just call this function without any argument apart from the position and size values.

x - horizontal position in overlay coordinate units.

y - vertical position in overlay coordinate units.

w - width in overlay coordinate units.

h - width in overlay coordinate units.

r,g,b,a - optional arguments defining the color of the rectangle in the RGB color coordinates ranging from 0.0 to 1.0. The fourth alpha component is optional, if no argument is given, a complete opacity of 1.0 is assumed.

Example: Draw a black rectangle covering the entire screen:

```
ovl.rect(-1,-1, 2,2, 0,0,0)
```

Switching the same rectangle to semi-transparent:

```
ovl.rect(-1,-1, 2,2, 0,0,0, 0.5)
```

ovl.image(x,y, width,height[, filename]) draws an image on the screen.
In order to remove a currently displayed image, call this function with only position and size values and no filename.

x - horizontal position in overlay coordinate units.

y - vertical position in overlay coordinate units.

w - width in overlay coordinate units.

h - width in overlay coordinate units.

filename - optional argument defining the filename and path to the image file. All image formats known to the veLib can be chosen (preferably PNG or JPEG). If no filename is provided, an already displayed image having the same coordinates is removed from the screen.

Example: Display the image “flower.png” in the center of the screen:

```
ovl.image(-.5,-.5, 1,1, ``flower.png'')
```

Removing the same image from the screen again:

```
ovl.image(-.5,-.5, 1,1)
```

2.5 Mathematical functions

The standard Lua math library widely corresponds to the standard C math library and also uses radians for angles. Therefore, veLua defines further convenience functions, which mainly are basic mathematical functions that expect angles in degrees - as OpenGL and the veLib - instead of radians. Note that Lua itself also offers two handy conversion functions, math.deg(value) and math.rad(value), but the provided triangular functions are slightly optimized and therefore preferable.

math.dsin(value) returns the sine of the provided degree value

math.dcos(value) returns the cosine of the provided degree value

math.dtan(value) returns the tangens of the provided degree value

math.dasin(value) returns the arcus sine of the provided value in degrees

math.dacos(value) returns the arcus cosine of the provided value in degrees

math.datan(value) returns the arcus tangens of the provided value in degrees

math.clamp(value, minValue, maxValue) returns minValue if value $<$minValue, maxValue if value $>$maxValue, or otherwise value.

2.6 Example scripts

A minimal motion model. The following code reads the state of some axes of the bound input device and directly translates it into the velocity of the bound simulation object:

```
inputX=obj.getAxis(INPUT,X) -- read the state of the input device's primary
axis
inputY=obj.getAxis(INPUT,Y) -- read the state of the input device's
secondary axis
--use the device's secondary input axis to control the forward speed:
obj.setAxis(VEL, Y, inputY*5.0)
--use the device's primary input axis to control the heading speed:
obj.setAxis(VEL, H, inputX*15.0)
```

Triggering an event / proximity sensor. The following script initiates the motion of the bound object dependent of the position of object "observer":

```
if state==nil then -- initialization, executed only once...
  state=0 -- ...because state gets a value here
  posX=obj.getAxis(POS,X)
  posY=obj.getAxis(POS,Y)
end

-- normal state, check proximity of object "observer":
if state==0 then
  observerPos=obj.getAxes("observer",POS)
  -- calculate observer distance:
  dist=math.sqrt( (posX-observerPos.x)^2 + (posY-observerPos.y)^2)
  if dist<2.5 then -- trigger event
    state=1
    obj.setAxis(ACC,Z,5.0) -- initiate an upward acceleration of 5 m/sec
  end
end
```

3 XML API

veLua adds a few extensions to veLib XML based initialization files. Their standard syntax is described in Franz and Weyel (2005, pp. 19-40). This document, however, slowly becomes dated and needs a revision. Therefore, it is recommended to check the veLib website `http://velib.kyb.mpg.de` for updated information.

Generally, veLua introduces a new `<scripts />` section into ini files and adds a few specific attributes to other statements.

3.1 `<scripts />`

The scripts statement serves as central container for all inline scripts, i.e. scripts that are directly defined within the ini file. It has to be placed as direct substatement of the toplevel statement.

Attributes none.

Substatements an arbitrary number of `<script />` statements.

3.2 `<script />`

A statement providing a single script. Either the script itself is defined directly as content of the statement, or indirectly via the attribute url. The same script can be attached to multiple objects, each instance gets its own set of private variables and runs independently from the others.

Attributes

- name: defines a unique name for the script which is used to bind the script to simulation objects.
- mime: defines the script language. This attribute is mainly for future extensions of the veLib which may add bindings for further scripting languages (e.g., Python, JavaScript, CengiScript). In veLua, all scripts should be of mime type “application/x-lua” .
- url: (optional). Instead of defining the script inline in the content of this statement, scripts may be also defined in separate files. Then the url attribute is used to define the filename and optionally relative path to the script.

Substatements none.

Example This example shows the code of an inline-defined minimal main script which just terminates program execution as soon as the Escape key (ASCII code 27) has been pressed.

```
<script name="main" mime="application/x-lua">
  if obj.getFlag(INPUT,27) then ve.exit(); end
</script>
```

3.3 `<deviceXYZ />`

A typical veLib ini file contains initialization parameter for various devices. In order to make use of user input provided via these devices, devices need a unique name attribute.

Example The following window device gets the name “mainWindow”:

```
<deviceWindow id="0" name="mainWindow">
  ...
</deviceWindow>
```

3.4 `<resource />`

veLib scenes are built of simulation objects which require resources to be rendered. In order to generate new simulation objects dynamically within scripts which use specific resources, resources need a unique name attribute.

Example The following VRML model resource gets the name “shark”:

```
<resource mime="model/vrml" id="5" name="shark" url="island/shark.wrl"/>
```

3.5 `<scene />`

veLib simulations normally contain a single scene consisting of a number of static or dynamic simulation objects which already exists from the very start of the application. While normally scripts are bound specifically to individual objects, most application will also require a master or main script which controls the overall data flow (e.g., initialization, switch of program stages, cleanup, shutdown). These main scripts can be bound to the scene by adding a “script” attribute providing the name of the script. Alternatively, a filename of a script can also be provided directly. Main scripts are executed each frame before the scripts bound to simulation objects. They can read input from the device container’s main input device.

Example The following statement defines the script “main” as main script:

```
<scene id="0" script="main">
   (object definitions)
</scene>
```

The following statement directly reads the script file “main.lua” (placed in the same directory as the application) as main script:

```
<scene id="0" script="main.lua">
   (object definitions)
</scene>
```

3.6 `<object />`

veLib simulations normally contain a scene consisting of a number of static or dynamic simulation objects. It is often convenient to control individual objects by individual scripts. In order to bind scripts to individual objects and respond to input of a particular device, additional attributes are required. Furthermore, in order to read or manipulate the state of an object from another script which is not specifically bound to this object, a unique name attribute is needed. A script bound to a simulation object is executed once per frame.

Attributes

- name: defines a unique name for this object.
- script: binds a script identified by its name to this particular simulation object.
- input: defines the input device which can be read by the script. The argument must be identical to the name attribute defined for the device.

Example The following statement binds the script file “camera.lua” to control the observer (i.e., the egocentric position of the camera in graphics devices or the listener in 3D audio devices). The input is read from the device named “mainWindow” which might provide access to the keyboard and mouse. Note that the observer object normally does not have shape (which would be widely invisible from the egocentric viewpoint), but needs the id=”0” attribute, which is the default id of observers in the veLib.

```
<object id="0" name="observer" script="camera.lua" input="mainWindow"/>
```

The following statement binds the script “coconut” to a simulation object having the same name. Note that devices, resources, scripts, or simulation objects may have the same name, but within one class any name should be unique. In this example the input is read from the device named “joystick0”.

```
<object id="7" name="coconut" shape="9" script="coconut" input="joystick0"/>
```

4 C++ API

While veLua allows a wide range of simulations and programs to be written without touching C++ at all, there are nevertheless sometimes good reasons to combine both programming approaches, in order benefit from a combination of maximum flexibility and ease of use.

4.1 Class appLua

From the C++ side, the main class of veLua “appLua” behaves very similarly to its parent class ve::deviceContainer and manages all device initializations behind the scenes according to the initialization file. As extension to the original deviceContainer behavior, appLua also provides dynamic scene management capabilities (accessible via Lua script references in the ini file) and some convenience methods which reduce the indispensable code to a very few lines:

```
#include "veLua.h"
int main ( int argc, char** argv ) {
  appLua ve(argc,argv);
  while(ve.update(0.01)==0); // the 0.01 means a sleep of 10 ms
  return 0;
}
```

A further handy feature of the appLua class is that some convenient interfaces created for Lua (e.g., the overlay interface) has a direct correspondence in C++. So, even if you do not intend to use Lua itself for a particular project, it might be nevertheless worthwhile to build the application based on appLua. This is generally a good idea, because the feature to add a script to an already compiled application and to specifically query object states during runtime opens up a great tool for logical debugging.

The C++ application may additionally interact with the appLua object in the same way as with the normal device container using the normal getInput() / setOutput() methods. The data containers’ flags and axes values may be used to communicate with scripts.

4.2 Generic variable exposure

Furthermore, scriptLua objects may be used to control individual variables in the application that are not associated to a simulation object or data container:

```
float myFloat=0.0f;
scriptLua scr;
scr.load("myScript.lua"); // load a script from file
// make myFloat accessible via the name "floatVar"
```

```
// by the lua functions setVar()/getVar():
scr.expose(myFloat,"floatVar");
// somewhere in the main loop:
scr.update(); // runs script once and updates exposed variable
```

5 Limitations

veLua as a recent and small-scale project is subject to several limitations. A good share of them have been taken deliberately in order to come up with an interface as simple as possible:

- Devices and resources cannot be instantiated during runtime from Lua scripts.
- Many convenient classes such as the linear algebra / vector mathematics functionality and the utilities are not exposed.
- Large numerical operations are considerably faster in C. Yet in comparison to other bytecode-based languages such as Java or Python, Lua is normally much faster, so, one should not bother too much about this.

Furthermore, some functionality would definitely be nice-to-have, but has not yet been implemented:

- Currently, an interface to collision models is missing. This is a limitation for many simulations. In order to fix this, a refactoring of the underlying veLib code would be helpful which on the long run would also facilitate the addition of a real physics library.
- veLua misses introspection. Currently it is not possible to parse through the objects, devices, or resources from the Lua side.
- For programming applications mixing Lua and C++ code, it would be helpful to have direct C++ correspondences to all Lua functions. Of course, total equivalence is not possible, because the Lua language allows for more flexibility than C and its descendants.

Generally, if for a particular project on of these or other limitations turn out to be a major drawback of veLua, the generic variable exposure and data exchange interfaces (Section 4.2) always allow for hybrid designs that make use of the combined advantages of both Lua and C++, which is in perfect accordance with veLua's basic design idea.

A Acknowledgements

The development of the veLib and veLua was supported by the Max Planck Institute of Biological Cybernetics, Department Cognitive and Computational Psychophysics of Professor Heinrich H. Bülthoff. The author of this document thankfully acknowledges the contributions of all the brilliant programmers to make the veLib an impressively capable yet still lightweight software tool, recently mainly Michael Weyel and Cengiz Terzibas.

Furthermore, the author wants to express his thankfulness to Dr. Bernhard Riecke for proofreading and improving this document. Last but not least, he kindly volunteered to provide the second application case of veLua, which as always will be more complicated as initially expected, but will definitely help to further improve this programming interface...

B Lua 5.0 license

C veLua license (zlib style)

References

Franz, G., & Weyel, M. (2005). *velib reference manual: Library version 1.2.0* (Tech. Rep.). Tübingen, Germany: Max Planck Institute for Biological Cybernetics.

Ierusalimschy, R. (2006). *Programming in lua* (2nd ed.). Rio de Janeiro, BR: Lua.org. (First edition (Lua 5.0) available online at http://www.lua.org/pil/)

Ierusalimschy, R., de Figueiredo, L. H., & Celes, W. (2006). *Lua 5.1 reference manual.* part of all Lua source distributions, available online at http://www.lua.org/manual/. Lua.org.