



velib.kyb.mpg.de

Getting started

Gerald Franz & Michael Weyel

Version 1.4.0, January 20, 2006

Contents

1	What is the veLib?	2
2	About this document	2
3	Getting the veLib	2
4	Installation	4
4.1	Hardware requirements	4
4.2	Software requirements	4
4.3	Customizing veConfig.h	4
4.4	Compiling the library	5
4.5	Problems and solutions	5
4.6	Getting further information	6
5	Tutorial	6
5.1	A minimal stand-alone program	7
5.2	A minimal XML initialization file	10
5.3	A minimal scalable program	12
5.4	Advanced veLib demos - short description	14

1 What is the veLib?

The Virtual Environments Library (veLib) is an extensible framework for the development of distributed real-time high-performance virtual reality applications. It was designed to offer a convenient and unified interface for all sorts of different input and output devices and to hide the hassle of different system architectures and communication methods from the user under a simple generic abstraction layer. Its design goals are, roughly ordered from most to least important, scalability, extensibility, flexibility, stability, simplicity, ease of use, performance, completeness, fanciness.

More specifically, it contains ready-made interfaces to all sorts of normal input devices (keyboard, mouse, joysticks, game pads, etc.), a basic 3D graphics engine, a nice spatial audio framework, a network communication layer, basic simulation logic such as collision detection and motion models, and various auxiliary functions such as overlay functionality, portable file input/output, and timers. The veLib is a platform independent C++ library, currently it is actively maintained for Linux and MS Windows.

And last but not least, the veLib is free (GPL) software. You can use it for no charge in your own projects and adapt to your particular needs. For licensing details please refer to the licensing documents shipped with the veLib as part of the online documentation or visit the veLib web site (<http://velib.kyb.mpg.de/docu>).

2 About this document

This document guides you through the first steps when using the veLib. More specifically, it helps you in getting the veLib running and provides an introductory tutorial on its basic programming concepts. The code demonstrated in this document complies with the veLib 1.0 syntax conventions.

The document is targeted primarily at beginners that intend to use the veLib for developing (virtual reality) applications. It is presumed that readers already have basic skills in C++ and object-oriented programming, and know how to use the development tools of their system platform. Furthermore, a very basic knowledge on OpenGL and the syntax of XML is generally helpful.

3 Getting the veLib

Download as zip archive. The most convenient way to get the veLib is to download it from the official veLib homepage as complete zip archive: <http://www.kyb.mpg.de/prjs/facilities/velib>.

The website has a link to the download section in the left side menu. Before downloading, one has to request a personal password. This registration process is required by German laws, all personal data except of the eMail address are optional and will be handled according to the regulations by law concerning the protection of data privacy. The personal password will then be sent by eMail together with further instructions. It does not expire, hence, for later downloads (e.g., updates) the password can be reused.

After downloading, the veLib package has to be unzipped using platform-specific tools (e.g., unzip under Linux, winZip under Windows).

Check out from the CVS-tree (account needed, currently only granted to core veLib developers). Another way to get the veLib is directly via CVS. This method may be used for downloading the most current

development version of the veLib that may offer additional features but may not be as well tested as the official package. Thus, this method mainly addresses the experienced programmer. In addition, also the latest stable version is available via CVS. This method may be advantageous if a previous version of the veLib is already installed, so, only the differences have to be transmitted.

CVS (Concurrent Version System) is an (open source) source code management system. For more information on CVS, one may see at <http://cvsbook.red-bean.com/cvsbook.html>. The newest veLib version can always be found on the MPI's CVS-Server though it is not guaranteed to be a stable one. You can either get a CVS-account with write permissions for the CVS-server, but those accounts are only granted to core veLib-developers. Everyone else please use the anonymous checkout. Both methods are described below for Linux and Windows OS.

On Linux:

1. Get an account and password for our CVS-Server (contact Michael Weyel michael.weyl@tuebingen.mpg.de for that) OR use the anonymous checkout (see 4.)!
2. If you have an account: login on the CVS-Server (supposed that your user name is "verner"):

```
cvs -d :pserver:verner@cvs.tuebingen.mpg.de:/veLib login
```
3. Check out the source code:
 - (a) For the development version:

```
cvs -d :pserver:verner@cvs.tuebingen.mpg.de:/veLib co -d veLib private
```
 - (b) For the latest stable version:
please download the latest zip archive to get the most recent stable version

On Windows:

1. Get CVS for Windows (the following descriptions suppose that you use WinCVS, which can be downloaded for free at <http://sourceforge.net/projects/cvsgui/>)
2. Get an account and password for our CVS-Server (contact Michael Weyel michael.weyl@tuebingen.mpg.de for that) OR use the anonymous checkout!
3. In WinCVS, goto Admin->Preferences and enter `:pserver:verner@cvs.tuebingen.mpg.de:/veLib` in the field "Enter CVS root" (supposed that your user name is "verner")
4. Choose "passwd" file on CVS server for authentication
5. Click on the "Ports" tab, mark "For 'pserver' (password) port:" and enter 2401 for that port
6. Goto Create->Checkout module..., enter "private" as the module name for the developer version. Choose a local directory, where the sources should be checked out to and click OK. If you logged in correctly, the veLib source is now copied to the directory you specified.

4 Installation

4.1 Hardware requirements

The veLib has no specific requirements regarding hardware. Any reasonably current PC system (2001+) will do provided that you get the required software running (see below). Of course, for getting a decent performance out of the veLib, a modern system having good 3D graphics and sound accelerator cards is advantageous. Furthermore, a joystick or similar input device is required in some demos and is generally a good intuitive interaction device for 3D simulations.

4.2 Software requirements

Most basically, the veLib needs a running operating system. Here you can either use a recent Linux, or Windows (when using Visual C++.net only Windows 2000 and XP, when using MinGW also Windows 98 and Me).

Additionally, the veLib is heavily based on two basic libraries that currently are a prerequisite for all veLib programs:

- OpenGL, a platform-independent high performance graphics API, <http://www.opengl.org>.
- SDL, the Simple DirectMedia Layer, v1.2, <http://www.libsdl.org>. SDL provides a portable high performance hardware interface. It is internally used for handling keyboard, mouse, and joystick input, window handling, and multi threading.

Furthermore, the veLib itself makes internally use of a few helper libraries. These library bindings are optional (see Section 4.3), but strongly recommended in order to make use of the complete functionality:

- OpenAL (<http://www.openal.org>), a platform-independent spatial audio library.
- libJPEG (<http://www.ijg.org/>), a graphics library for image and texture handling.
- libpng (<http://www.libpng.org/pub/png/>) and zlib (<http://www.gzip.org/zlib/>), a graphics and a compression library for image and texture handling.

Note that on Linux systems most of these libraries normally are already installed. After downloading and compiling the additional packages it is necessary to either install them in your system's standard paths, or to put or link the headers and compiled libraries into the external/include resp. external/lib subdirectories of the veLib root directory.

4.3 Customizing veConfig.h

You do not have to provide all the auxiliary libs if you do not need a certain functionality (e.g. if you don't want to load any jpeg-pictures, using the libJPEG is unnecessary). But the veLib needs to know which libs to use and which to avoid. You may need to make some adjustments to `src/include/veConfig.h`. This file is quite self explanatory and well documented, just have a look into it before compiling the veLib. Furthermore, the linker settings of the compiler have to be adjusted correspondingly. Under Linux and MinGW, these adjustments are made in the file `src/makeinclude` in the section `"# libraries to link with"`.

4.4 Compiling the library

Linux. The use of the gcc compiler version 3.2 or higher is recommended. Compiling the lib may also work with earlier versions, but are not officially supported. Make sure that all different external libraries are compiled with the same compiler that you use for building the veLib.

If all preparatory steps are done correctly, the veLib is compiled by just switching in a shell to the top level veLib directory and running “make”.

MS Windows. Currently, there are two ways to compile the veLib under Windows. One can either use the free MinGW compiler (<http://www.mingw.org>) and follow the Linux preparation and compilation instructions, or use Microsoft’s Visual C++ compiler v7.1+. Here you can load the project file and press the build button for the lib-project, hopefully the veLib will be compiled in a few minutes. Currently, all the sources in the demo and server/client directories should also compile without problems.

4.5 Problems and solutions

veLib starts compiling but stops after a while with an error message. It may happen from time to time (especially if you try to compile the developer version), that not all veLib related sources compile properly. If you get any error during compilation, please first check how far your compiler got. Most of the time, it won’t have been the veLib itself that generated the error, but one of the provided example applications. In most cases, you can ignore that, the veLib will work with your application. If the error was caused by the veLib itself, please check again if all your settings are correct and if you have all the necessary libraries installed. If that does not help, please post your problem to the veLib mailing list (see Section 4.6).

All sorts of linker errors turn up when trying to compile an application that is linked to the veLib with Microsoft Visual C++. The veLib is build with MS Visual C++ as Multi-threaded (compiler option /MT) if build in Release mode or as Multi-threaded Debug (compiler option /MTd) if build in Debug mode. If you link your own application to the veLib, make sure that you use the exact same compiler option or you may get all sorts of weird linker errors. You can set this option by right-clicking on the project name in the solution-explorer-window and then selecting “properties” (alternatively go to “Project” in the Menu bar and select “XYZ Properties”, where XYZ stands for your project name). On the Properties page, open the C/C++ folder and go to the “Code-Generation” section. You will find an entry “Runtime library”, which needs to be set to Multi-threaded or to Multi-threaded Debug respectively.

You may also need to set some linker options to build without the corresponding libraries, the errors from the development environment will provide the necessary information. For doing this, again open your projects property page. Open the Linker folder and go to the “Input” section. In the row “Ignore specific library” just enter the name of the lib that you want to have ignored, for example libc.lib;libcd.lib (separated by semi-colons;).

After updating your veLib repository on Linux from version 0.x to version 1.x the veLib itself compiles fine, but the demos bail out with lots of linker errors. Your linker tries to link with an old shared version of the veLib (`src/lib/libve.so.xyz`). The new veLib is a static library (`src/lib/libve.a`). The easiest solution is to remove the old version by typing `rm -f src/lib/libve.so*` in a shell.

On windows, the library and all demos compile flawlessly, but when trying to run a demo (or a veLib based application), the program returns immediately showing a message similar to “*The application failed to start, because sdl.dll was not found. Re-installing the software may fix this problem.*”. The application misses a required shared (dynamically linked) library (in this example SDL). Putting a copy of the requested .dll file into the demo (or your application) directory will fix this.

4.6 Getting further information

The veLib comes with a pretty complete html-help and API reference, situated at docs/html/index.html. The same, or ideally an advanced version of this online help is also found on the veLib website (<http://www.kyb.tuebingen.mpg.de/prjs/facilities/velib/docu>). However, this online help certainly does not cover all aspects. So, if you find any bugs or need help on a certain topic, feel free to contact the main maintainer Michael Weyel (michael.weyel@tuebingen.mpg.de). You may also send him a mail if anything in this document does not become clear or if something that is described here is not working as it should.

Additionally, there is a veLib mailing list, where all news and changes concerning the veLib are announced. You may also post a veLib related problem there and see if any of the other subscribers can give you a solution for it. If you want to become part of the list, please contact Michael Weyel (michael.weyel@tuebingen.mpg.de)

5 Tutorial

Having successfully installed the veLib and its dependencies, it is now time to write a first program!

This tutorial introduces the core classes of the veLib and shows how to structurize programs to make them easily portable on various devices and hardware platforms. This is done by first setting up a small stand-alone program that will be subsequently explained step-by-step and extended to a completely scalable application. This final program does not do much, yet it is a useful core that in similar form underlies most interactive 3D computer simulations from screen savers over ego shooters to full-featured physically-correct professional flight training simulators. In its most basic state it just allows the free navigation through a virtual landscape, the virtual camera is controlled via keyboard and mouse (see Figure 1). Yet it may be generically useful as starting point for own projects. While this tutorial of course covers only a small fraction of the veLib functionality, it provides the indispensable survival veLib programming kit and the prerequisites for understanding the further more advanced demos that cover various aspects in more detail (see Section 5.4). The basic classes introduced in this tutorial are:

- `ve::xmlIni`, the veLib interface for initialization and file input/output
- `ve::device`, the representation of joystick, display, motion platform, audio, network...
- `ve::vec6f` and `ve::flag128`, basic data structures
- `ve::motion` and `ve::collision`, the veLib approximation of a physical model
- `ve::time`, a class for all time related purposes
- `ve::dataContainer`, the communication interface for device states and simulation object states
- `ve::deviceContainer`, the abstraction layer for (sets of) devices



Figure 1: Screenshot of the running tutorial program.

5.1 A minimal stand-alone program

We directly jump into the code (Note: the source of these examples is located in the demo subdirectory):

Source code tut01.cpp.

```
// demo/tut01.cpp
#include <veLib.h>

int main( int , char** ) {
    ve::xmlIni ini;
    ini.load("iniTut.xml");
    ve::deviceWindow devWindow(ini);
    ve::deviceGraphicsGL devVideo(ini);
    ve::vec6f position(0.0f, -110.0f, 1.6f, 310.0f, 0.0f, 0.0f);
    ve::vec6f velocity, acceleration, inputAxes;
    ve::flag128 inputButtons;
    ve::collisionSurface collisionModel(ini);
    collisionModel.addObject(0, position, 1.6f, 1.0f);
    ve::motionSimple motionModel(ini, 0, &collisionModel);
    ve::chrono timer;
    while(!inputButtons[ve::BUTTON_2]&&!inputButtons[ve::KEY_ESCAPE]) {
        timer.update();
        devWindow.getInput(inputAxes, inputButtons);
        motionModel.updateObject(0, inputAxes, position, velocity,
                                acceleration, timer.deltaT());
        devVideo.setOutput(position, inputButtons);
        devVideo.update(timer.deltaT());
    }
}
```

```

        devWindow.update(timer.deltaT());
        timer.sleep(0.01);
    }
    return 0;
}

```

Step by step. Now we will discuss the previous example in more detail:

```
#include<veLib.h>
```

The veLib classes are accessible via various header files. For convenience purposes, the header veLib.h includes the complete public interface in one command and also includes common C++ standard headers.

A typical simulation consists of at least an initialization part and a main loop. In the first part all necessary simulation objects are created:

```

ve::xmlIni ini;
ini.load("iniTut.xml");

```

A central concept of the veLib is the usage of initialization files. These files are written in XML. For parsing XML files, the veLib provides two classes, `ve::xml` and `ve::xmlIni`. While `ve::xml` provides the low level language structure and parsing functionality, `ve::xmlIni` defines a convenient high level interface especially for initialization files. Their content will be briefly explained in the next section. The exact meaning and various options of these files is subject of another document ([veLibXml.pdf](#)). The most important message for now is that XML objects and initialization files are required for initializing devices. They can be instantiated and initialized using the statements above.

The first line also shows the veLib namespace, `ve::`. To avoid its explicit usage, one can import it completely by the statement `using namespace ve;`. Yet its advantage of an explicit use of the namespace is the better recognizability of veLib commands.

```

ve::deviceWindow devWindow(ini);
ve::deviceGraphicsGL devVideo(ini);

```

The veLib implements the access of hardware using the unified device concept. Device interfaces are one core service of the library. All devices are derived classes from the parent class `ve::device` which defines a common interface. This means, independent from the actual type, all devices can be addressed using exactly the same syntax. In this case, a window object is opened and a 3D OpenGL visualization object is created. The exact parameters for the initialization are provided in the previously loaded `ve::xmlIni ini` object. The window class does not only provide a draw area for the 3D graphics device, but also allows access to keyboard and mouse events that in all contemporary operating systems are bound to the window focus.

```

ve::vec6f position(0.0f, -110.0f, 1.6f, 310.0f, 0.0f, 0.0f);
ve::vec6f velocity, acceleration, inputAxes;
ve::flag128 inputButtons;

```

A 3D simulation normally deals with objects that are somewhere located in space and have certain properties. In this tutorial a few variables are needed to control and influence the camera position and check for user events. For these purposes the veLib makes heavily use of two basic data structures, `ve::vec6f` coordinate objects and `ve::flag128` state containers.

A `ve::vec6f` object (often called a "sixdof") contains six float values that normally represent the X, Y, Z position as well as the H, P, R (heading, pitch, and roll) orientation coordinates of an object, thus a position

and orientation in space is completely described. The veLib coordinate system is similar to OpenGL's, the only differences are that +Z is the default for the up direction and +Y for forward. The generic class `ve::vec6f` is used as well to store an object's velocity (linear and angular), acceleration, or the state of the axes of an `ve::device`. In the later case, the devices always normalize the axis values to the range of -1.0f to 1.0f. The `ve::vec6f` class defines several operators to set, access, or transform its content, most common is the access operator `[]` which allows to read or change a single ordinate that can be specified via the `ve::X`, `ve::Y`, `ve::Z`, `ve::H`, `ve::P`, and `ve::R` coordinates.

A `ve::flag128` state container is most basically a struct of 4 unsigned int32 variables that contains 128 bool values. It is most often used to store the state of input device buttons. That means, for example, that for each key of a keyboard (normally 101-104 keys) one bit is reserved that can contain its current press state. In `src/include/veTypes.h` a complete set of constants is defined to give all keys and buttons explicit human-readable names (for an example see the while loop later on). Also for this class a similar set of access methods is defined, for more information please refer the veLib API reference manual.

```
ve::collisionSurface collisionModel(ini);
collisionModel.addObject(0, position, 1.6f, 1.0f);
ve::motionSimple motionModel(ini, 0, &collisionModel);
```

Besides access to input/output devices and basic data types the veLib also provides a few classes that are useful to implement some physical logic in simulations. Typical recurrent tasks are collision detection and the transformation of user input into object movements. For these purposes the veLib offers basic ready-made motion and collision classes that are instantiated here. The `ve::collisionSurface` models just keeps an object on a surface geometry that is defined in the ini file. The exact parameters have to be defined for each object separately in the `addObject()` method. Its first parameter is just an id to identify the object later on. The further parameters are the object's position that may be altered by the collision, its preferred altitude over ground and maximum step height. The third command finally initializes a motion model object and binds the collision model to it.

```
ve::chrono timer;
```

Any realtime computer simulations normally needs high performance timing and timer functionality. For this purpose, the veLib offers the class `ve::time`. `ve::time` objects basically provide timer functionality, that means they provide methods that return the exact number and fraction of seconds that have passed since the timer construction as double value. The actual accuracy of timers differ from platform to platform, but should at least have millisecond precision.

Having set up the timer, all objects required in this simulation are available. The main loop can be started:

```
while(!inputButtons[ve::BUTTON_2]&&!inputButtons[ve::KEY_ESCAPE]) {
```

The while statement initializes the main loop of this tutorial program. Each frame of the simulation the complete loop is processed. Two alternative abort criteria are defined: As soon as the `inputButtons` container contains the value `TRUE` at the positions `ve::BUTTON_2` or `ve::KEY_ESCAPE`, the loop is not further executed. The two constants are identifiers for key and button states that are defined in `src/include/veTypes.h`.

```
timer.update();
```

veLib timers behave a little bit differently from timers of other libs. Instead of single `timestamp()` function, there are two complementary methods for requesting the current time. The `update()` method actually requests the current system time and updates the internal time variable of the timer object correspondingly. It is important to know that this value will stay constant until the next call of `update()` irrespectively from the actually passed time. The current steady internal state of the timer can be read without updating using the `now()` access method. While this interface may seem at first glance a little bit quirky, it is in fact very

powerful. It conveniently allows to keep a state of virtual contemporaneity over a defined part of a simulation. In this case, `update()` is only called once per frame, thus all commands in one loop cycle virtually happen at the same simulation time.

```
devWindow.getInput(inputAxes, inputButtons);
```

This command reads the current state of the input device into the passed data structures.

```
motionModel.updateObject(0, inputAxes, position, velocity,  
                          acceleration, timer.deltaT());
```

In this line the position and speed variables are updated by the motion model according to the current input state. The first parameter is the object id previously defined integer the `collisionModel.addObject()` call that will be implicitly called from the motion model. Note that the intended motion does not depend on the state of the input axes alone, but also by the time passed since the last frame that is accessible via the `timer.deltaT()` method. Note that the usage of the motion and collision classes is completely optional. One can try this out, for example, by directly coupling input and output by simply copying the `inputAxes` `ve::vec6f` into the position `ve::vec6f`.

```
devVideo.setOutput(position, inputButtons);
```

Here the updated position is passed to the visualization. In this case where no other parameters are specified, the device interprets the passed coordinate as its observer position, leading to a movement of the camera position.

```
devVideo.update(timer.deltaT());  
devWindow.update(timer.deltaT());
```

Finally, both output devices are updated, that means that the draw commands are actually executed by the visualization device and the window buffers are swapped. Again, the time passed since the last frame is passed via the `timer.deltaT()` method to make the simulation speed independent from the actual frame rate of the devices.

```
timer.sleep(0.01);
```

Beside the base functionality similar to a ticking clock, timer objects also allow access to further time related functionality. For example, they also implement platform independent `sleep()` methods. A sleep basically means that the current thread of an application is interrupted, and control is given back to the operating system. Since realtime simulations depend on various services provided by the operation system, it is generally a good idea not to block the CPU completely by the simulation process. Hence, short regular interruptions of the program flow (here for one-hundredth of a second) may actually help to increase the overall performance of an application.

Twenty-five lines of code, and a first full-featured 3D simulation is done. Almost... Besides the program logic, a complete simulation always consists of some content that is displayed or manipulated. The `veLib` encourages the division of content and logic by using the initialization files for content definition. They will be subject of the following section.

5.2 A minimal XML initialization file

Source code `iniTut.xml`.

```
<?xml version="1.0"?>  
<XperiML version="1.0">  
  <deviceWindow id="0" deviceContainer="input">
```

```

<string id="winTitle"> veLib tutorial </string>
<float id="mouseRelative"> 0 </float>
<float id="mouseNeutral"> 0.1 </float>
<bool id="mouseVisible"> 0 </bool>
<bool id="fullScreen"> 0 </bool>
<int id="zBufferBits"> 16 </int>
<int id="winSizeX"> 800 </int>
<int id="winSizeY"> 600 </int>
</deviceWindow>

<deviceGraphics id="0" class="deviceGraphicsGL">
  <float id="nearClipping"> 0.25 </float>
  <float id="farClipping"> 1000 </float>
  <float id="frustumLeft"> -1.0 </float>
  <float id="frustumRight"> 1.0 </float>
  <float id="frustumBottom">-.75 </float>
  <float id="frustumTop"> .75 </float>
  <float id="frustumScale"> 1.0 </float>
  <bool id="backFaceCulling"> 1 </bool>
</deviceGraphics>

<motion id="0" class="motionSimple">
  <float id="translSpeedMax"> 10 </float>
  <float id="translAccFactor"> 2 </float>
  <float id="translDecFactor"> 5 </float>
  <float id="rotSpeedMax"> 90 </float>
  <float id="rotAccFactor"> 120 </float>
  <bool id="invertAxisH"> 1 </bool>
</motion>

<resources basePath="">
  <resource mime="model/vrml" id="1" name="ground"
url="enviro1.wrl"/>
  <resource mime="model/vrml" id="2" name="background"
url="hemisphere01.wrl"/>
  <resource mime="model/vrml" id="7" name="temple"
url="stonehenge.wrl"/>
</resources>

<scene>
  <object id="1" shape="1" surface="1" pos="0 0 0"/>
  <object id="2" shape="7" pos="35 -91 0 -90 0 0"/>
  <light id="0" enabled="1" position="-0.5 -1.0 1.0 0.0"
ambient="0.3 0.3 0.3 1.0" diffuse="0.7 0.7 0.7 1.0"
specular="1.0 1.0 1.0 1.0"/>
  <background shape="2" pos="0 0 -2.5"/>

```

```
</scene>
</XperiML>
```

When looking at an initialization file, one sees immediately that their content is structured in several sections. The first three sections initialize various internal parameters of the devices and the motion model (see [veLibXml.pdf](#)). In this tutorial, the `<resources>` and `<scene>` sections that define the scene content are briefly explained.

The `<resources>` section defines the content that is in principle available to the simulation. Each resource is normally specified by its mime type (i.e., a standardized way to describe file contents, initially defined for eMailing, see, e.g., <http://www.mindspring.com/~mgrand/mime.html>) and a URL defining its (relative) location in the file system. When a device gets initialized, it parses the passed `ve::xmlIni` object for recognized resources and loads them into memory, so that they are readily available as soon as they are used to instantiate a simulation object. Resources are identified via their id, which can be any arbitrary integer number larger than zero.

The `<scene>` section defines the simulation objects that are to be loaded at startup. Typically it is used to define a static scenery that will not be changed during the simulation. Similar to the resources section, each device parses its ini file for recognized keywords. In this example two objects are defined that are recognized by the graphics device `ve::deviceGraphicsGL` by the keyword attribute `shape`. The argument following the `shape` attribute defines the resource id to be used to instantiate the object. Also scene objects are identified in the simulation by an id for which the same rules apply as for resource ids. The same numbers can be used for resource and object ids, since they address different memory areas. Further attributes either address additional devices, or specify the object further. In this example, the `pos` attribute defines the position and orientation of the objects in 3D space. Further sub-statements of `<scene>` specify general parameters such as lighting that may be interpreted by some devices as well.

5.3 A minimal scalable program

Source code `tut02.cpp`.

```
// demo/tut02.cpp
#include <veLib.h>

int main( int , char** ) {
    ve::xmlIni ini;
    ini.load("iniTut.xml");
    ve::deviceContainer devices(ini);
    ve::dataContainer observer;
    observer.position().set(0.0f, -110.0f, 1.6f, 310.0f, 0.0f, 0.0f);
    ve::collisionSurface collisionModel(ini);
    collisionModel.addObject(observer, 1.6f, 1.0f);
    ve::motionSimple motionModel(ini, 0, &collisionModel);
    ve::chrono timer;
    while(!observer.buttons()[ve::BUTTON_2]
        &&!observer.buttons()[ve::KEY_ESCAPE]) {
        timer.update();
    }
}
```

```

    devices.getInput(observer, ve::DC_MASK_INPUT);
    motionModel.updateObject(observer, timer.deltaT());
    devices.setOutput(observer, ve::CMD_OBJECT_SET);
    devices.update(timer.deltaT());
    timer.sleep(0.01);
}
return 0;
}

```

Step by step. Compared to the previous example, this fortunately looks quite familiar, and, surprisingly, the second program is even shorter! We will now have a closer look at the differences.

```
ve::deviceContainer devices(ini);
```

The first decisive difference is the replacement of the window and graphics device by a single `ve::deviceContainer` object. The `ve::deviceContainer` is not a direct representation of a physical device or library interface, but a virtual placeholder for any standard `ve::device` descendant or even a collection of them. It has the same standard interface as any other `ve::device`. The actual instantiation is defined by the passed `ve::xmlIni` object, for each device definition found there a device object will be created. So, any later changes of the input/output devices do not need adjustments of the source code but only an exchange of the initialization file, and several initialization files describing different hardware combinations can be used parallelly.

```
ve::dataContainer observer;
observer.position().set(0.0f, -110.0f, 1.6f, 310.0f, 0.0f, 0.0f);
```

The second difference is the utilization of the `ve::dataContainer` class instead of using the basic `ve::vec6f` and `ve::flag128` classes explicitly. A `ve::dataContainer` object is essentially a standardized collection of 4 `ve::vec6f` objects, 2 `ve::flag128` objects, and a few further ids and user data variables. The container is designed to completely describe the state of typical simulation entities by just one object. All central `veLib` classes (e.g., devices, motion model, collision) offer interface methods accepting `ve::dataContainer` objects instead of numerous single parameters. The `ve::dataContainer` class itself offers various access methods to read or change its content in a similar way as changing corresponding basic components directly. The second code line shows such an access to one of the `ve::vec6f` sixdofs. By using the `position()` access method, all `ve::vec6f` methods are available to manipulate or read the position data. Further access methods will be demonstrated below.

```
collisionModel.addObject(observer, 1.6f, 1.0f);
```

Instead of using and tracking an isolated id as in the previous example, now simply the complete observer object is registered at the collision model. The internal object id of the `ve::dataContainer` is accessible via the `objectId()` methods if necessary.

```
while(!observer.buttons()[ve::BUTTON_2]
      &&!observer.buttons()[ve::KEY_ESCAPE])
```

The control statement of the main loop now checks two flags of one `ve::flag128` member of the `ve::dataContainer` by using the `buttons()` method.

```
devices.getInput(observer, ve::DC_MASK_INPUT);
```

The user input is read into the suitable `ve::vec6f` and `ve::flag128` data structures of the `ve::dataContainer`. To guarantee that only these data fields are overwritten, the optional write mask `ve::DC_MASK_INPUT` is defined as second parameter.

```
motionModel.updateObject(observer,timer.deltaT());
```

The motion model reads the user input and velocity information stored in the `ve::dataContainer` and updates all affected data fields of the container accordingly.

```
devices.setOutput(observer, ve::CMD_OBJECT_SET);
```

Subsequently, the observer object containing the updated camera position is passed to the `ve::deviceContainer` that distributes it to all affiliated devices. The optional second parameter specifies the command to be executed. In this example `ve::CMD_OBJECT_SET` brings the corresponding internal representations of the devices in line with the data stored in the observer object.

```
devices.update(timer.deltaT());
```

Finally, all devices within the `ve::deviceContainer` are updated, in this example the scene gets redrawn and the buffers are swapped.

So, the data flow of a typical scalable `veLib` application can be graphically illustrated as in Figure 2.

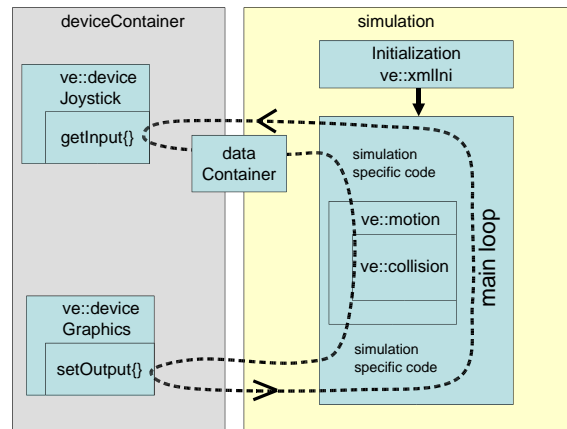


Figure 2: Diagram illustrating the basic structure and data flow of normal `veLib` applications.

Why is this program scalable? Well, there are basically two reasons for that. First, the `ve::dataContainer` combines data structures to completely describe a typical simulation object in one handy C++ object. This makes it much easier to keep simulation code concise that consists of several objects (e.g., multiple observers). More important, the `ve::deviceContainer` completely decouples experiment logic from the instantiation and presentation. A change of the simulation setup (e.g., from desktop during development to distributed VR for the final product) becomes very easy by just adjusting the initialization file. One can drive this even further by using `<include>` statements in the initialization file to separate device descriptions completely from the resources and scene description (see [veLibXml.pdf](#) for further information).

5.4 Advanced `veLib` demos - short description

This section provides a brief overview on all demos that are shipped with the `veLib`. They are located in the demo subdirectory. Note that in contrast to the minimal tutorials presented above, the source code of these demos is heavily annotated, so, they should be widely self-explanatory, and therefore are presented only very briefly in this section.

demo01.helloWorld This is the most simple demo of a veLib based stand-alone application. The program opens a window, loads, and renders a well-known message. The gaze direction can be controlled via the mouse. Press escape or middle mouse button to exit.

demo02.OpenGL This is a small demo application that demonstrates how to use the veLib for low level OpenGL coding. The veLib handles input events and provides a rendering context. The demo renders a spinning colored triangle. Do not bother about various `ve::xml` warnings, here default settings are used, so no XML initialization file is loaded.

demo03.motion This is a simple demo of a veLib based stand-alone application. The program opens a window, loads, and renders a simple scene. The observer position can be controlled via mouse and keyboard. It uses a basic motion model as well as collision to keep the observer on the ground. Press escape or button 2 to exit.

demo04.animation This is the extended version of demo03. The program opens a window, loads, and renders a scene. An animated object is added that makes a lot of noise. As an additional bonus, the current frame rate is calculated to demonstrate the superior timing qualities of the veLib. The observer position can be controlled via a joystick. Use button 2 to exit.

demo05.dataContainer This demo switches demo04 to the `ve::dataContainer`. The functionality is the same as in demo04: The observer position can be controlled via mouse and keyboard.

demo06.deviceContainer This demo extends demo05 by using the `ve::deviceContainer` instead of single input/output devices. This allows to change input/output devices without recompiling the demo by solely adjusting the ini file (`iniDemo06.xml`). The functionality is the same as in demo04: The observer position can be controlled via mouse and keyboard.

demo07.dynamicScene This demo extends demo06 by showing a scene change triggered by the simulation. The observer position can be controlled via mouse and keyboard.

demo08.multiUser This demo extends demo 06 by introducing a second human-controlled observer. Both are represented by a device- and data container. You can adjust the actual device settings in the user specific initialization files (`iniDemo08_1.xml` for user 1, `iniDemo08_2.xml` for user 2). Use button 2 to exit.

demo09.multiPipe This demo does virtually the same as demo06 but uses a different ini file (`iniDemo09.xml`). Therefore, the visualization is distributed on 3 pipes. You may adjust the ini file to distribute the simulation on different computers. Do not run this demo directly, but use `demo09_run.sh`, resp. `demo09_run.bat` to initialize the necessary visualization clients in advance.

demo10.audio3d A test application for spatial audio. Do not forget to unzip `demo10Audio3d_resources.zip` before running it in order to make the demo work.

demo11.xml A small demo introducing the functionality of the `ve::xmlIni` class and veLib initialization files. Besides the initialization of variable values, the usage of include files is demonstrated.

demo12.plugins This demo shows how to load a plugin, this is mainly done via the xml file, so please have a look at `iniDemo12.xml`. Also, please see the `(VEROOT)/plugins` directory for examples on how to create your own plugin files.