Flexible Abstraction Layers for VR Application Development

Gerwin de Haan* Michal Koutek† Frits H. Post‡

Delft University of Technology, The Netherlands

Abstract

The development of domain-specific Virtual Reality applications is often a slow and laborious process. The integration of the domain-specific functionality in an interactive Virtual Environment requires close collaboration between domain expert and VR developer, as well as the integration of domain-specific data and software in a VR application. The software environment needs to support the entire development process and software life cycle, from the early stages of iterative, rapid prototyping to a final end-user application. In this paper, we propose the use of flexible abstraction layers in the form of a dynamic scripting language, which act as the glue between VR system components and external software libraries and applications. First, we discuss the motivation and potential of our approach, after which we overview related approaches. Then, we describe the integration of a Python interpreter in our VR toolkit. The potential of our integration approach is demonstrated by rapid prototyping features, the flexible extension of core functionality and the integration of an external toolkit. We conclude with an overview of implications our approach has for the future development of new framework features and application integration.

Keywords: Virtual Reality, Application Development, Scripting Languages.

Index Terms: I.3.7 [Computing Methodologies]: Computer Graphics—Virtual Reality; D.2.6 [Software]: Software Engineering—Programming Environments

1 Introduction

After decades of experience with Virtual Environments, a large gap still exists between developer and end-user experience. Although the robustness of 'core' VR technology increases, other fundamental issues in VR application development arise from the typical requirements that characterize custom VR applications. These include the continuous involvement of end-users and interactivity while maintaining performance. A highly flexible software architecture is needed to match the nature of the development and maintenance life-cycle of VR applications.

During our research on interaction techniques in Virtual Environments, our VR software framework gradually evolved and extended. Recent application and framework refactoring efforts revealed that productivity is limited by the long development cycles that go into *(re-)designing, (re-)implementing and debugging* new features in our C++ software architecture. This is a limiting factor during our collaboration with domain experts, and influences the direct applicability and acceptance of VR solutions.

In general, a VR application is built on the underlying VR framework, which consists of a set of carefully orchestrated heterogeneous components. Especially in the early, exploring stages of collaboration with domain-experts, many changes continuously evolve the ideas on and the state of the VR application. The provided framework API needs to be flexible enough to enable use and extensibility, while avoiding a bloated, overall up-front design. In addition, there is also the need for expressive end-user modeling and programming. Therefore, we opt for a continuous, rapid software prototyping environment, in which many small features and links between components can be tested and evaluated. A loosely designed set of high level abstractions, reusable components and code snippets allow developers and eventually experienced users to use rapidly create complete applications by extending basic functionality in an ad-hoc fashion.

*e-mail:g.dehaan@tudelft.nl
†e-mail:m.koutek@tudelft.nl
‡e-mail:f.h.post@tudelft.nl

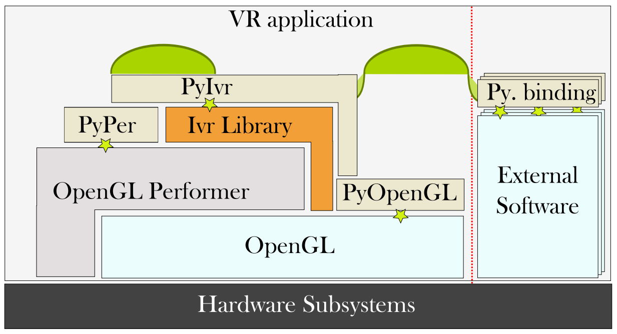


Figure 1: Overview of the iVR software layers (see Section 3.1). The VR application has native access to various components and external software or through Python. Stars indicate Python bindings on the underlying libraries. Arches indicate custom Python layers which provide higher level abstractions.

We propose a VR application development paradigm based on flexible abstraction layers through a single abstraction language. The abstraction layers here are code fragments that abstract, combine and wrap lower level code. The goal of this approach is to facilitate *(1)* continuous, iterative software development, including features such as rapid prototyping, profiling and debugging, *(2)* flexible integration and configuration of heterogeneous VR and external software, *(3)* seamless evolution from early software prototypes to flexible end-user applications and *(4)* ease-of-use, lowering the learning curve and empowering end-users.

In this paper, we describe our work towards the proposed VR application development paradigm of integrated rapid software prototyping. Here, the focus is on the flexible abstraction layers which connect several aspects of our VR library, applications and external, domain-specific toolkits and libraries. We first describe related work in Section 2. Section 3 and 4 describe our prototype and several examples. We conclude with a short discussion and our view on future VR application development in Section 5.

2 Related Work

Work on flexibility in application development and VR integration can be found in VR-related approaches, as well as in more general and domain-specific learning and problem solving environments. Often, approaches are separated in system-specific facilities and end-user application specific features.

In *VR system frameworks*, run-time flexibility —on a system level— has been acknowledged and applied many times. The Bamboo [17] plug-in framework is an early system providing a micro kernel-based VR architecture in which all system elements are pluggable components that could be added, removed, and replaced at run time. In VR Juggler [3] the possibilities of run-time reconfiguration of the VR system to a higher level than only the system are described. The focus in these approaches is mainly on system configuration flexibility, and less on flexibility of behavioral aspects of specific VR application. In VHD++ [13], also the integration of domain-specific applications in such a framework is discussed, while maintaining extensibility and flexibility. The approaches mentioned above achieve their flexibility through existing and well documented software design patterns for the creation of a solid system framework and protocols.

Dynamic scripting languages enable iterative design techniques, allowing a dynamic approach to application and system design. Many VR systems provide these facilities in their frameworks to achieve development flexibility. As part of this work, we maintain an online overview [1] of VR frameworks and graphics APIs in which scripting support is provided. In our prototype description we discuss scripting language features and their integration in more detail. In all listed approaches, with Visum [7] being a notable exception, a C/C++ based VR kernel and graphics API is used. Visum uses Python and PyOpenGL as the core of the framework. The level of abstractions and scripting integration features differs widely between systems. The common aspect of all approaches is the availability of high level abstraction layers on top of the complete VR framework. In addition, Coloseum3D [1], Panda3D [8] and Avango [15] also use a scripting language as the main integration layer of various system components, providing low-level, flexible access to developers and end-users.

Here, *specification of interactivity* is also main aspect of VR application development. An early example of 3D graphics interaction specification in a native scripting language can be found in the Alice toolkit [5]. An analysis of describing and modeling interaction behavior is given in [4]. Modeling languages are used to avoid programming and validity issues for the end-user, while maintaining flexibility. This can quickly become complex, as describing behavior beyond simple examples requires more elaborate constructs (such as control-flow elements) and one needs to interact with application code, on a VR system-level, but maybe also on a domain-specific level through external software. Hendricks et al [9] propose an interactive, scripting approach to overcome many issues of this duality in modeling language. We want to provide the user with a unified syntax and semantics for describing interactivity with (external) application components.

In many *domain-specific application areas* rapid prototyping facilities were proposed, ranging from high level APIs and Domain Specific Languages to full-blown Problem Solving Environments. On many levels, scripting support is available. The *tinkering* during experimentation and development proves useful [12] and is finding its way into new software design methods such as extreme programming and aspect-oriented design. SuperGlue [10] uses Scheme language constructs for describing a visualization process of domain-specific data. It is presented as an answer to the existing visualization platforms that overemphasized ease-of-use by the use of GUI, which failed to adequately address issues of extensibility. VHD++ [13] uses stub components to which application developers should connect their applications. Here, flexibility in this approach is achieved by a strict communication protocol and design pattern.

3 Prototype Description

The main aspect of our proposed VR application development paradigm is the use of abstraction layers, on which all interactive, programmable aspects of the VR application and individual components are founded. Our current prototype implementation is a first step towards the application of these abstraction layers in a VR application development environment.

3.1 Software Layers

A schematic overview of our *interactive Virtual Reality (iVR)* system is shown in Figure 1. This is a recent rewrite of our *RWB library* [11], a closed, C++, OpenGL Performer based VR toolkit, where its monolithic characteristics were transformed to a microkernel approach, augmented with separate flexible components. The set of components and domain-specific applications, combined with the scripting language layer, now form the basis for creating a VR application.

We selected Python as the unifying abstraction language layer for multiple reasons. First, this General Purpose Language integrates well with existing C and C++ code. The Python interpreter can be extended or the interpreter can be embedded in the C application. Second, a solid basis is provided by its wide availability on various platforms, a large standard set of tools, and wrappings of many (scientific) software packages. Its simplicity and flexibility allow for a smooth transition from powerful lower-level access to higher-level, more user-friendly constructs. Also, the interactive introspection and self-parsing facilities enable us to extend the language with special purpose (such as domain-specific) sub-languages.

3.2 Wrapping of existing software components

Abstraction layers in the high level scripting language form the basis for interactive control of various systems components. Special conversion code is necessary to take care of the type and value conversions between the C++ and Python environments. This intermediate conversion code layer, the so called wrappers or binding, can be created manually or (semi-) automated by external software. The level of automation is determined by the wrapping method used and the complexity of the C++ constructs.

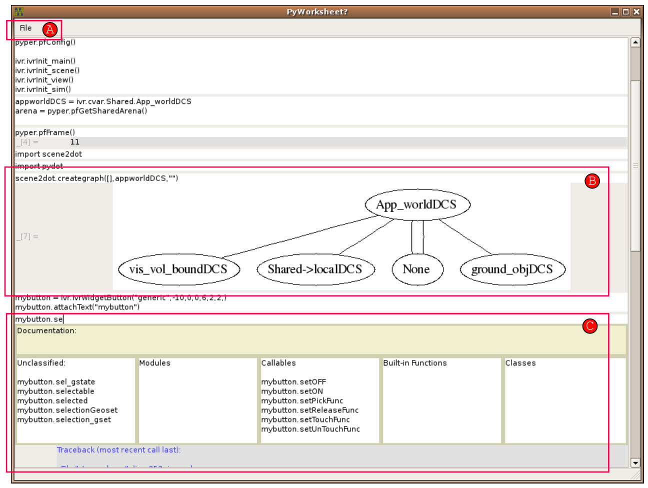
Many Python wrapping generators exist, of which *SWIG*[2] and *Boost.Python*[3] are the most popular. SWIG parses declarations in header files to generate a intermediate Python and library file, which expose the wrappings [2]. Boost.Python uses *template meta programming*, and uses some helper code and the C++ compiler itself to generate a wrapper library. Difficulties can arise when complex data is communicated between two C++ components for which different wrapping generators are used. Other issues in performance and usability also influence the choice of the wrapping solution, but technical details are outside the scope of this document.

Our iVR toolkit intensively uses OpenGL Performer functionality through *PyPer* [16], SWIG generated OpenGL Performer bindings. We use SWIG to wrap our iVR library to avoid cross-wrapper difficulties. In this way, we are safe to transparently mix Performer data types and functions with iVR functions in a Python environment. We use two helpful SWIG provided features to enhance the usability of our wrappings. First, SWIG *Director* classes enable *cross language polymorphism*, allowing for easy subclasssing and extension of existing C++ classes in Python. Second, source code documentation such as comments, parameter and data types are made directly available in the Python interpreter. Using our wrappings, we obtain most functionality directly in Python syntax. The thin and adaptable layering of abstraction levels can provide both flexibility and performance on a low-level, while on a higher level the ease-of-use and expressiveness is maintained.

[1] http://visualisation.tudelft.nl/VRdev

[2] http://www.swig.org/

[3] http://www.boost.org/libs/python/doc/



```
pyper.pfConfig()

ivr.ivrInit_main()
ivr.ivrInit_scene()
ivr.ivrInit_view()
ivr.ivrInit_sim()
appworldDCS = ivr.cvar.Shared.App_worldDCS
arena = pyper.pfGetSharedArena()

pyper.pfFrame()
_[4] =          11
import scene2dot
import pydot
scene2dot.creategraph([],appworldDCS,"")
_[7] =
mybutton = ivr.ivrWidgetButton("generic",-10,0,0,6,2,2,)
mybutton.attachText("mybutton")
mybutton.se
Documentation:

Unclassified:              Modules    Callables                  Built-in Functions    Classes

mybutton.sel_gstate                   mybutton.setOFF
mybutton.selectable                   mybutton.setON
mybutton.selected                     mybutton.setPickFunc
mybutton.selectionGeoset              mybutton.setReleaseFunc
mybutton.selection_gset               mybutton.setTouchFunc
                                      mybutton.setUnTouchFunc
        Traceback (most recent call last):
```

Figure 2: Interactive Notebook metaphor during VR development. The worksheet shows code editing, loading/saving operations (A), integrated graphics (B), and available documentation and command completed parameters (C). The interactive graphics or controls in (B) are generated by customizable Python helper code.

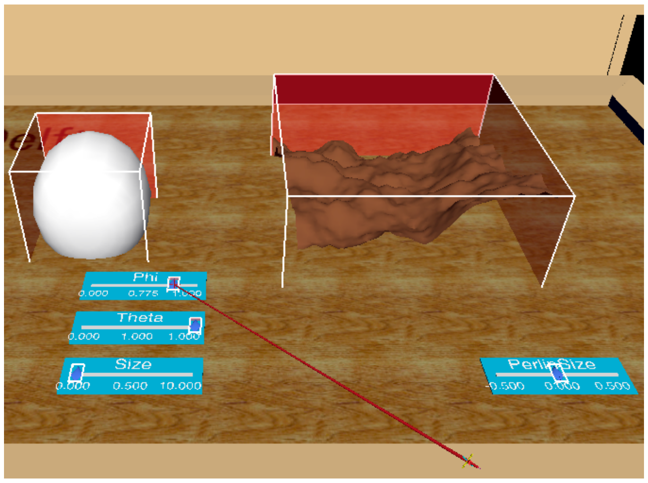


Figure 3: Demonstration of bi-directional integration of VTK in the VR environment. Graphical data from two VTK pipelines is shown in the VR application. Callbacks from widgets directly control parameters of the two VTK pipelines. The Python glue facilitates expressive commands that combine VTK and VR statements

3.3 Control Beyond Wrapping

The structure and functionality of the generated wrappings do not always match the requirements for interactive, run-time development. First, a greater error robustness is required in both the wrappings and the underlying software. Validity assertions and good error handling, including documented error reports is necessary for the interactive development. Second, the directly mapped language constructs may not be on the right abstraction level for the task at hand, for both the developer or end-user. The advanced programming styles and thin abstraction layers in Python snippets enable the use of more flexible software construction. Third, the control flow from different software components must be combined. For example, we need to combine the Python interpreter control, the OpenGL Performer render loop, and even a physics engine. In our method we opted for *extending* Python, which makes the Python interpreter the overall controller of the VR application. This provides us with many features to inspect system state and manipulate the control flow. Finally, performance can be problematic if many operations need to be performed in the Python interpreter. Normally, as much of the code will be glue code and initial set-up code, the performance bottleneck of the VR application will not be in the Python handling. For improved performance one can resort to native C++ components and make use of multi threading and multi processing.

4 Prototype Results

This section demonstrates the important features of the flexible abstraction layers through sample applications. A VR application can be completely written in a script, which is executed by a standard Python interpreter. The iVR functionality is directly available in the running Python interpreter after importing its wrapping modules.

4.1 Run-time Prototyping

The Python interactive interpreter allows interactive control in created scripts. It facilitates a run-time prototyping environment, because program functionality can be inspected, added and changed. We use *IPython*, an enhanced interactive interpreter to enhance the usability and interactivity of this process. IPython provides many extra development features, including object and code inspection, command history, integrated debugging and saving of interactive prototyping sessions. Both the regular and IPython interactive interpreter are *line-based* concepts, where entered sets of lines are interpreted and executed immediately. The use of small, saved testing scripts leads again to a differentiation between saved code and interactive code. We use experimental software developed in our group for the construction of Python-based VR code using the Notebook metaphor. This metaphor provides a unified worksheet for the code, interleaved with interactive graphics (see Figure 2). It provides a smooth transition from interactive prototyping to working code snippets. Graphical representations or interactive widgets are created by the use of small helper snippets, which generate the content based on special object types.

4.2 Internal Extensions

The iVR toolkit, with its abstraction layers and a set of standard Python snippets, provides an accessible, high level VR application skeleton from which development can start immediately. The user can construct his application by using and extending a set of standard widgets, graphical objects and interaction techniques. The close integration of Python and the original C++ code allows us to, gradually, transform existing code toward Python oriented programming methods. These extensions can range from simple widgets, such as a simple valuator with custom behavior, to complete custom interaction handling mechanisms.

Our recent Python-based iVR developments illustrate the flexibility. We developed an event mechanism and state machine components for interaction and behavior modeling. We use *state charts*, a technique for modeling the system state with concurrent, hierarchical state machines. We combine run-time code inspection and interactive state chart manipulation, providing a valuable insight during the interaction development cycles. While the VR application is running, state charts and transitions can be extended and have their graphical representations generated. These new facilities again form new abstraction layers that are easily extended. The use of other external libraries and toolkits allows to further enhance the development process. For example, one can think of GUI-based interaction modeler on top of the current state chart abstraction layer.

4.3 External Software Integration

When integrating external software in VR application constructs, the use of the abstraction layer shifts many difficulties to the wrapping generation process. Once wrappings are available, or if software is already in the abstraction language, constructions from the various components can be mixed with less effort. The limitations of this mixing are dependent on the data size and data format compatibility between the various components.

External software of special interest are domain specific applications and libraries and general-purpose functionality that can be useful for analysis during development. In our current prototype, we integrated *matplotlib*[4], a Python package for mathematical plotting, *Graphviz* [6], graph construction software, and the *Visualisation ToolKit(VTK)* [14], a data visualization package. The use of Graphviz is demonstrated in previous Figure 2, where graphs are integrated in the Notebook environment. This integration with the VR application is currently *uni-directional*. This means that results of Graphviz functions on data structures such as scene graph and state machine hierarchy, are shown in the Notebook environment only, and not directly back in the VE.

For VTK integration in our VR software we wrap the *vtkActorToPF* library to do performance critical conversions from VTK to Performer data, and provide a mixed iVR-VTK abstraction layer. The end-user can mix VTK code directly with iVR constructions in Python scripts, while the abstractions layers perform the underlying communication between the external libraries. VTK generated graphical objects are created by using VTK commands directly in the VR script or by importing an external VTK example file. Figure 3 illustrates this *bi-directional* approach, where resulting objects are first-class objects in the VR application and can be directly integrated with VR interaction and behavior.

The construction of real, domain-specific VR applications builds upon the basic application skeleton using the integration and extension techniques described above. As the entire work flow is an extensive and iterative process, a running VR prototype can be constructed and maintained throughout the cycles of development. As wrappings and abstraction layers for the core functionality of the external software packages are introduced, they can gradually be connected with the VR components. During this process, interactive prototyping experiments with domain-experts and VR developers can lead to insights on new application requirements, for example the need for specialized visualization and interaction techniques.

5 Conclusions and Future Work

The introduction of multiple abstraction layers in existing VR software tool chains using dynamic languages such as Python provides flexible development styles for VR application development. The ease of programming and the multitude of abstraction layers allow both developers and end users to use expressive programming commands at a suitable level of comprehension. A powerful functionality included in Python and the availability of wrappings for external software packages ease the process of application integration. We described the features and benefits of the abstraction layers through the gradual introduction of a Python layer in our existing, C++ based VR toolkit. The interactive scripting environments give run-time access to the running application, providing an interactive prototyping environment. Development efforts are shifted towards the creation of wrappings and interactive development environments. For interactive use, greater error robustness is required in both the wrappings and the underlying software, as well as useful debugging information.

We are working towards a rapid VR prototyping paradigm that provides a solid base for various development styles. The transformation towards interactive control through unified abstraction layers catalyzes the re-design of previous software mechanisms and a change in development philosophy. We envision an integrated development and run-time environment providing interactive control using higher level, visual programming and debugging tools. We expect the abstraction layering and integration of external tools to be key aspects in achieving this goal.

[4]http://matplotlib.sourceforge.net

Acknowledgements

Part of this research has been funded by the Dutch BSIK/BRICKS project. We thank SARA for providing PyPer, and Jorik Blaas for the interactive notebook prototype.

References

[1] A. Backman. Colosseum: 3d-authoring framework for virtual environments. In E. Kjems and R. Blach, editors, *Proceedings of the 9th IPT and 11th Eurographics VE Workshop (EGVE) '05*, 2005.

[2] D. M. Beazley. Automated scientific software scripting with SWIG. *Future Gener. Comput. Syst.*, 19(5):599–609, 2003.

[3] A. Bierbaum, C. Just, P. Hartling, K. Meinert, A. Baker, and C. Cruz-Neira. VR juggler: a virtual platform for virtual reality applicationdevelopment. In *Virtual Reality, 2001. Proceedings. IEEE*, pages 89–96, Yokohama, Japan, 2001.

[4] T. Burrows and D. England. Yable - yet another behaviour language. In *Web3D '05: Proceedings of the tenth international conference on 3D Web technology*, pages 65–73, New York, NY, USA, 2005. ACM Press.

[5] M. J. Conway and R. Pausch. Alice: easy to learn interactive 3d graphics. *SIGGRAPH Comput. Graph.*, 31(3):58–59, 1997.

[6] J. Ellson, E. Gansner, E. Koutsofios, S. North, and G. Woodhull. Graphviz and dynagraph – static and dynamic graph drawing tools. In M. Junger and P. Mutzel, editors, *Graph Drawing Software*, pages 127–148. Springer-Verlag, 2003.

[7] D. Finkenzeller, M. Baas, S. Thring, S. Yigit, and A. Schmitt. Visum: A vr system for the interactive and dynamics simulation of mechatronic systems. In *Proc. Virtual Concept 2003*, Nov 2003.

[8] M. Goslin and M. R. Mine. The Panda3D graphics engine. *Computer*, 37(10):112–114, 2004.

[9] Z. Hendricks, G. Marsden, and E. Blake. A meta-authoring tool for specifying interactions in virtual reality environments. In *AFRIGRAPH '03: Proceedings of the 2nd international conference on Computer graphics, virtual Reality, visualisation and interaction in Africa*, pages 171–180, New York, NY, USA, 2003. ACM Press.

[10] J. Hultquist and E. Raible. Superglue: a programming environment for scientific visualization. In *Visualization, 1992. Visualization '92, Proceedings., IEEE Conference on*, pages 243–250, Boston, MA, USA, 1992.

[11] M. Koutek. *Scientific Visualization in Virtual Reality: Interaction Techniques and Application Development*. PhD thesis, Delft University of Technology, 2003.

[12] J. Ousterhout. Scripting: higher level programming for the 21st century. *Computer*, 31(3):23–30, 1998.

[13] M. Ponder, G. Papagiannakis, T. Molet, N. Magnenat-Thalmann, and D. Thalmann. VHD++ development framework: towards extendible, component based VR/AR simulation engine featuring advanced virtual character technologies. *2003. Proceedings Computer Graphics International*, pages 96–104, 2003.

[14] W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit*. Kitware, Inc., third edition, 2004.

[15] J. Springer, H. Tramberend, and B. Fröhlich. On scripting in distributed virtual environments. In *Proceedings of the 4th IPT Workshop*, June 2000.

[16] B. Stolk, F. Abdoelrahman, A. Koning, P. Wielinga, J.-M. Neefs, A. Stubbs, A. de Bondt, P. Leemans, and P. van der Spek. Mining the human genome using virtual reality. In *EGPGV '02: Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization*, pages 17–21, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[17] K. Watsen and M. Zyda. Bamboo - a portable system for dynamically extensible, real-time, networked, virtual environments. In *VRAIS '98: Proceedings of the Virtual Reality Annual International Symposium*, page 252, Washington, DC, USA, 1998. IEEE Computer Society.