Improving the AVANGO VR/AR Framework — Lessons Learned

Kuck, Roland Wind, Jürgen Riege, Kai Bogen, Manfred

Schloss Birlinghoven
Fraunhofer IAIS
53754 Sankt Augustin
Tel.: +49 (0) 2241/14-2366
Fax: +49 (0) 2241/14-2040
E-Mail: roland.kuck@iais.fraunhofer.de

Abstract: Our open source VR/AR framework AVANGO has been used in several of our R&D projects since 1996. We recently started the development of the next generation version called AVANGO NG as the development of external dependencies stalled. We present the design criteria for the vintage system and the new one, what lessons we learned from implementing and using the first version, and how that influenced our design decisions. We cover the internal system design as well as the user visible parts of the framework.

Keywords: VR/AR Frameworks, Dependency Graph, Field Mechanism

1 Introduction

Frameworks offer a high-level interface for application developers by hiding the complexity of the underlying architecture. Our AVANGO® *Virtual Reality/Augmented Reality* (VR/AR) framework provided support for distributed, interactive virtual environments [Tra03]. We started development already in 1996 and it has proven many times to be production-ready and was used in several research and commercial projects. Recently challenges arose with external libraries and components that affected the framework as a whole:

- AVANGO used *OpenGL Performer* [RH94], a performance-oriented scene graph toolkit offered by SGI. While still being sold, its development has stalled. It is also basically only available in a 32-bit version. The 64-bit version uses the IA-64 instruction set, which is not supported by most available 64-bit capable, PC-based workstations.

- *Scheme* was the scripting language used with AVANGO. While being a powerful language and well suited [STF00], it is not widespread. This results in two major consequences: People interested in using AVANGO had not only to learn the concepts of AVANGO itself but also a *functional* language to actually be able to write any application. As few other people use this language it is also difficult to find good advice or help when facing problems. Also only few (external) libraries exist for *Scheme*, requiring

a large amount of implementation for components that are taken for granted in other languages. This is especially problematic when rapid-prototyping an application.

- Even though AVANGO itself was licensed using an *open source* license, its distribution was limited due to the dependance on the commercial library *Performer*. This probably resulted in many people refraining from using AVANGO.

We therefore decided to look for alternatives to AVANGO. We present the results of our review in section 2. As no other framework or library fulfilled our requirements completely, we decided to rework AVANGO. The basic ideas from AVANGO were not changed and an overview is given in section 3. The new system is called AVANGO NG. We replaced the scene graph library with *OpenSceneGraph* [OSG] and the scripting language with *Python*. We also changed the way these libraries were integrated based on the lessons learned from AVANGO. This is described in section 4. We gained a lot of experience from using AVANGO over the years and enhanced the component model and the scripting language binding, described in sections 5 and 6. We draw conclusions from this work and provide future work in section 7.

2 Review of other systems

As already indicated in the previous section we developed a concrete concept of what a framework for VR/AR environments should be able to do. A system should meet the following requirements:

- Distribution on the application side, so that shared VR/AR sessions via the network are possible.
- Rapid prototyping of the application using an appropriate scripting interface.
- A flexible display setup that supports amongst others a variety of different screen configurations, stereo visualizations, and head-tracked viewpoints.
- An easy way for extending the existing framework by own modules.

Besides our own implementation a lot of other VR/AR frameworks exist. We examined a subset, introduced below, based on their popularity and relationship to our ideas.

VR Juggler [BJH+01] is an Open Source virtual platform for VR application development that provides a unified operating environment, abstracts the complexity of VR systems and supports run-time changes in the hard- and software configuration of the environment. VR Juggler has no dependency on a specific rendering engine, therefore it allows but also requires applications to bring their own. VRJuggler does support render distribution via NetJuggler[AGL+02], but it has no direct high-level support for application distribution.

HECTOR [WGB+07] supports the idea of rapidly prototyping applications using a *Python* scripting interface. The framework consists of a micro kernel and a communication module. Many central components like the event propagation system are directly implemented

in *Python*, which makes the system easily extendible. This framework has an interesting concept, but it seems that this project is no longer active.

Virtools [VIR] is a widely used commercial development kit for 3D applications that supports many common hardware components and comes with a lot of built-in functionality. Furthermore Virtools comes with an own scripting language, called VSL. The VSL interface allows the system to be scripted at runtime. The VR Library/Publisher module supports client-server distribution where the parameters of a scene are synchronized with the server, but it does not support a peer-to-peer communication of stand-alone VR/AR applications.

Other VR/AR frameworks in use are Lightning [BLRS98] and Avalon [Beh06], which are both similar to AVANGO in the respect that they use fields to represent the state of an object, but do not provide a sophisticated application-level network distribution support.

3 Overview of AVANGO

AVANGO and AVANGO NG use a scene graph consisting of nodes to describe the scene to be rendered. Each node is a so-called *field container* representing object state information as a collection of *fields* similar to *Open Inventor* [Wer93]. Events can be propagated within the AVANGO framework using *field connections*. This data-flow graph is orthogonal to the scene graph. Field container whose fields have changed are evaluated before rendering the next frame. Fields are also used to provide support for two other features:

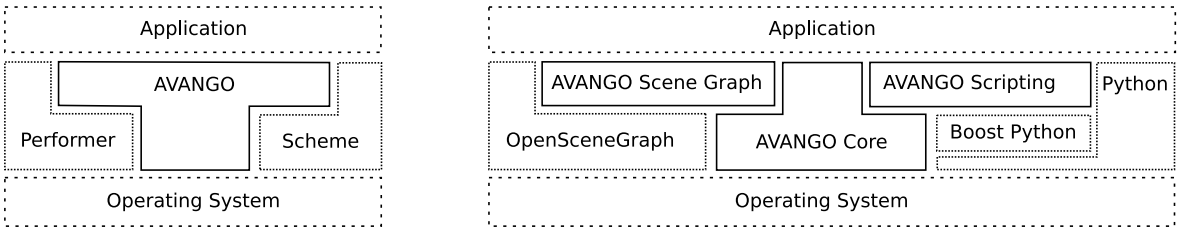
- *Distribution.* The concept of fields and the fact that the state of a field container is completely defined by the value of its fields directly leads to the use of fields for serialization and network distribution of field containers within AVANGO.

- *Component-Based Design.* Fields define the public visible interface of a field container. Connections between fields are set up outside the field container, which leads to a loose coupling between field container instances.

For most nodes and field containers it is sufficient to use the fields as the *interface* to interact with. Furthermore using this kind of interface has the following advantages:

- *Reflection.* Fields are registered within the type system of AVANGO, which allows field containers to be queried for their fields at runtime. This simplifies the implementation of scripting interfaces in AVANGO and allows the access of field container documentation at runtime.

- *Duck Typing.* Fields are only accessed using generic accessor methods provided by the field container base class. This allows for a programming model where only the names and types of fields are important and no inheritance relationship is required. This is called *duck typing* (“If it walks like a duck and quacks like a duck, I would call it a duck.”)



(a) AVANGO (b) AVANGO NG

Figure 1: System overview

Layered on top of these core functionalities AVANGO provided many nodes and field containers, e.g. to control the viewing and display setup. As AVANGO NG uses the same foundation, these concepts and design can be used as is.

4 System Architecture

AVANGO NG shares a lot of *C++* code with AVANGO. We tried to avoid investing time and effort into code that has proven many times to be production-ready. Some dependencies on external libraries were removed or replaced. This resulted in changes in the structure and the way different parts interact. Following a more elaborated look at the overall design of AVANGO NG, we describe some details on specific changes.

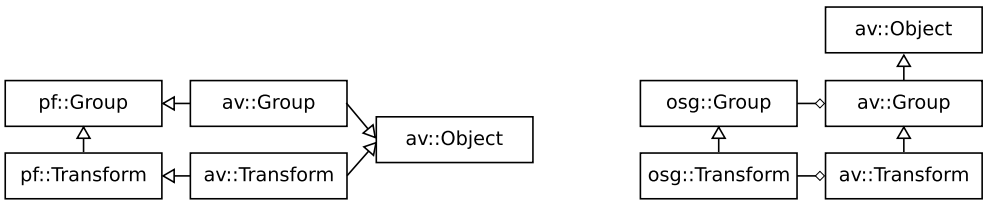
4.1 Refactoring AVANGO to AVANGO NG

In AVANGO the integration of the scene graph was very tight and it was not possible to compile AVANGO without the scene graph library. As we were replacing this library anyway we decided to decouple the *core* of AVANGO NG, i.e. the field and field container mechanism, from any other component. All other modules are then laid on top of this core library and we tried to minimize the inter-module dependencies. This is shown in figure 1.

We literally extracted the core module from AVANGO and added unit-tests. This allowed us to refactor parts of the core to support the enhanced features described in section 5.

In AVANGO the scripting binding was invasive: The classes were modified to directly support the used *Scheme* interpreter. The script bindings in AVANGO NG do not rely on internal knowledge of the classes but only use their external interface. We use the *Boost Python* library [BP], which also allows us to extend or change the interface of classes exported to *Python* if required.

There is also no dependency of external modules to the script binding. The type system of AVANGO NG is used to make new field containers available to the application. We minimized the need for methods that should be called on the field containers in *Python*. Where still needed a submodule would simply use the *Boost Python* library directly.



(a) AVANGO (b) AVANGO NG

Figure 2: Scene-graph integration in both frameworks

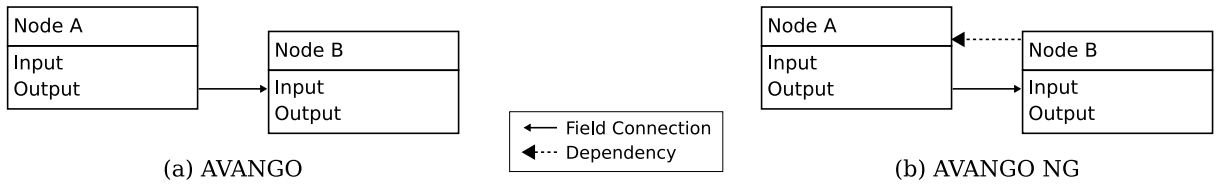
4.2 Encapsulating the Scenegraph

AVANGO used the *adapter* pattern [GHJV93] to integrate *Performer* into the AVANGO type system. More specifically it used a so-called *class* adapter, where an adapter class derives from both the desired interface and the class providing the implementation. This is shown in figure 2 (a).

This variant of the adapter pattern was chosen to be able to build a (*Performer*) scene graph consisting of either *Performer* or *Avango* nodes (see section 4.1.4 in [Tra03]). To build such a scene graph the *Performer* nodes had to be manipulated directly, which was not possible with *Scheme* code. Rather an AVANGO scene graph was most often built from the accessible AVANGO nodes. The only exception were complex model loaders, that built subgraphs to represent the model in AVANGO. To manipulate the model, some of the *Performer* nodes where replaced with the AVANGO adapter. However, the major drawback of the use of class adapters was, that the AVANGO nodes did not form an inheritance hierarchy themselves. This resulted in code duplication and also made it impossible to use more generic interfaces, e.g. to pass a transformation node where a group node was required.

Therefore AVANGO NG uses an *object* adapter. The scene graph node is held by reference only and the AVANGO NG nodes is derived from the most appropriate base class, e.g. a transformation node is derived from a group node. This is shown in figure 2 (b). It is possible to pass an already existing scene graph node to the AVANGO NG adapter, so that in the case of the complex model loader, we simply instantiate new adapters for interesting scene graph nodes instead of replacing them.

Reference counting is now a bit more involved as two objects exist, the AVANGO NG adapter and the *OpenSceneGraph* node. If both would hold each other with reference-counted pointers these objects would never be deleted. Therefore the AVANGO NG adapter does not hold a reference to the *OpenSceneGraph* node. Any request for a reference to the AVANGO NG object itself is deferred to the *OpenSceneGraph* node and the AVANGO NG node registers itself as an observer [GHJV93] with the *OpenSceneGraph* node. When the *OpenSceneGraph* node is destroyed, the AVANGO NG node destroys itself.



(a) AVANGO (b) AVANGO NG

Figure 3: Connection and dependency graph

5 Component Model

If a class has a well-defined responsibility, it is easier to implement and easier to reuse [SA05]. The same holds for components in AVANGO, but the implementation of the component model made this difficult: Extra information was hardwired into the base classes for special situations and many components were bigger than necessary because part of their implementation logic had to run in a specific order that was not guaranteed anymore if theses component would have been split up. Both aspects are addressed by adding *external* fields and a well-defined evaluation order to AVANGO NG.

5.1 External fields

A common operation on the scene graph is a *traversal*. While many traversals perform node-specific actions, e.g. using the *visitor* pattern [GHJV93], we sometimes need additional information to perform a given task. A good example is the movement of objects selected using a pick-ray: Once the picked object is found using intersection tests, we need the corresponding transformation node to move the object. We therefore traverse the parents of this object to find the first transformation node that is *tagged* as draggable. Consider a car with four wheels, each replicated using the same geometry and four transformation nodes. If we do not want the wheels to be moved individually we only tag their parent transformation. In AVANGO this tagging was performed by setting the *Dragger* field to an appropriate value. Every AVANGO node contained this *Dragger* field. It was not used by the node itself. We call fields like this *external* fields. Many more existed in AVANGO for different purposes. Adding more would have required changing the core classes.

In AVANGO NG fields can be added at run-time. These fields are specific to the instance where they are added and thus ideally suited to be used as *external fields*. There is no difference to *normal* fields and thus all field mechanisms, e.g. field connections, can be used.

5.2 Evaluation Order

AVANGO always first evaluated *sensors*, special field containers that were used e.g. for input devices. It then evaluated all normal field containers in no predictable order and finally all *actuators* e.g. the render traversal. Consider a simple connection graph as shown in figure 3 (a). Here it is possible that node B is evaluated before node A resulting in incorrect field values in node B. The only possible way to ensure proper evaluation is to combine node

A and node B in a new field container. In AVANGO this has led to a large number of classes with complex interfaces, where many effectively perform similar tasks. In AVANGO NG we therefore define an order in which all nodes are evaluated. As a consequence, sensors and actuators are no longer special classes but normal field containers.

A well-designed field container does the following things in its *evaluate()* method:

1. *Read values from local field values.* We call these fields *input* fields. No external data should be accessed to keep the class isolated from its environment and thus increase reuse.

2. *Calculate new values derived from the read values.*

3. *Write these values to local fields.* We call these fields *output* fields and again no fields of other field containers should be written to.

All value propagation from one field container to another should happen via field connections, as this allows the application to control the connection of field containers and thus increases the flexibility in using the field containers.

If field containers are designed like this, we can use the field connections to derive dependencies. An example is shown in figure 3 (b). If the *Output* field of node A is connected to the *Input* field of node B, then B is dependent on A, i.e. A must be evaluated before B. It is sometimes necessary to have field connections that do not define a dependency relationship called *weak field connections*. Examples are given in section 5.3 and 6.1.

The correct evaluation order is realized by having each field container call an *evaluateDependency()* method. This method first recursively calls the same method on all dependent field containers and then calls *evaluate()* on itself. This method can also be called from within the *evaluate()* method itself, if the field containers know about other dependencies that are not expressed by the field connections.

This mechanism is fail-safe. The *evaluateDependency()* method checks for loops, i.e. if it is called twice for the same objects. In this case it simply returns instead of calling itself again. It also logs a warning message, so that the application developer is informed about the cycle in the dependency graph. This is mostly important for legacy code that has not yet been updated. Even though cycles are broken by this scheme they should still be avoided as they have undefined and unpredictable behavior.

5.3 Feedback propagation

Even though the field connection graph should not contain cycles, often some form of feedback propagation is required. Consider a simple connection graph consisting of an accumulator node and a ground-following node as an example. The accumulator takes the last transformation matrix and a new relative transformation matrix as input and concatenates the transformations. The relative transformation matrix can e.g. be generated from a joystick. The ground-following node takes the new transformation as input and outputs a corrected

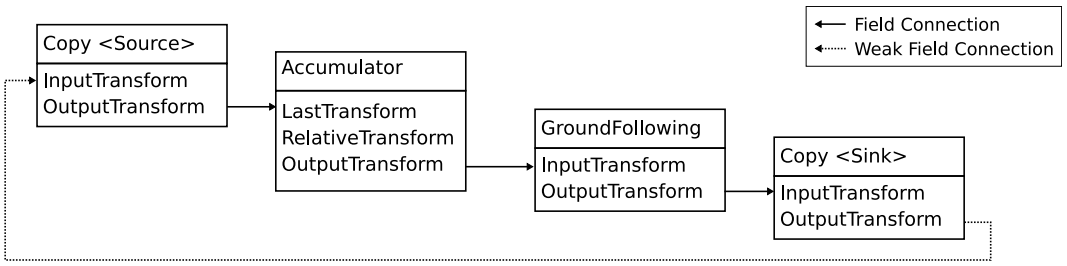


Figure 4: A mover using feedback propagation

transformation that is positioned on the ground e.g. using collision detection techniques. To avoid too large jumps, the position will only be changed by a maximum amount.

The accumulator can not store the last transformation matrix itself, as the ground-following node changes this last transformation. Simply connecting the output transformation of the ground-following node to the last transformation field of the accumulator is not well-defined as it would introduce a cycle.

To solve this problem we use a feedback propagator node. It is a simple node containing input and output fields, which are matrix fields in this case. When evaluated this node simply copies the input field value to the output field. We create two instances and use one as the source and one as the sink of the transformation using normal field connections. We also create a weak field connection from the output field of the sink to the input field of the source. This is shown in figure 4. Weak field connections are drawn dashed.

It should be clear that the dependency graph does not contain any cycles. To understand that this has the desired effect assume that *evaluateDependency()* was called on the ground-following node. This is deferred to the accumulator and to the source. The source copies the value from its input to its output in its *evaluate()* method. This value is the value from the last frame and thus the last transformation as desired. The accumulator then concatenates the transformations. In the ground-following node this value is corrected. As it writes its output value this value is copied to the input of the sink. This leads to an evaluation of the sink. As all dependencies of the sink were already evaluated it is called directly and copies it input value to the output. This again triggers a copy of this value to the source, but as the source was already evaluated the value propagation stops here till the next frame. Starting the evaluation with any other field container leads to identical results.

5.4 Namespaces for Components

While refactoring the *C++* source code, we also introduced namespaces to modules as it is a recommended practice [SA05]. We did not change the component model in a similar way to offer some kind of namespace support, but introduced a naming convention that type names are written in the form *Module::SubModule::Type*, like e.g. *av::osg::Sphere*.

We mirror this type structure in *Python* using the native module support. The *av* namespace can be found in the *avango* module, the *av::osg* namespace in the *avango.osg* module.

```
class Increment(avango.script.Script):
    Input = avango.SFInt()
    Output = avango.SFInt()

    def evaluate(self):
        self.Output.value = self.Input.value + 1
```

Figure 5: A field container implemented in *Python*

Field containers implemented using *Python* (see section 6) will register with a type name that contains the translated prefix of the module in which they are defined.

6 Scripting

In AVANGO it was not possible to implement a field container in the scripting language. Event handlers were written using *Trigger* field containers, that would call a callback function when evaluated. This callback handler would then access fields of other field containers violating the design guides described in section 5.2 that the evaluation order relies on.

Another special field container was provided called *Script*. By accessing fields of this container new fields could be created and callbacks registered. While this provided the same basic capabilities to create a field container as in *C++*, it had multiple drawbacks:

- While normal field containers can be instanciated any number of times, each *Script* that defines new functionality is actually an instance itself. To create multiple instances new *Script* nodes would have to be instantiated and the values of the special fields would have to be copied.
- As *Scheme* offers no direct support for objects this becomes especially problematic when the field containers require access to some internal state.

AVANGO NG make full use of the object support in *Python*. Figure 5 shows a simple example of a field container implemented in *Python*. Note that every instruction contains needed information and that almost no other syntactic structure is required.

Another important aspect is that once defined, a class implemented in *Python* does not behave different from a class implemented in *C++*. This is especially important in the rapid-prototyping context. One is able to prototype a field container in *Python* and later on can re-implement the field container in *C++* if it is required for performance reasons. Even the instantiation procedure is identical for both.

6.1 Container

As AVANGO NG encourages the use of small components connected via field connections, an application can consist of a large number of components. Maintaining the connection of these classes during the development cycle can be difficult. *Containers* help to reduce this

```
class IncrementByTwo(avango.script.Container):
    Input = avango.SFInt()
    Output = avango.SFInt()

    def __init__(self):
        self.init_super(avango.script.Container)
        node1 = self.register_internal_node(Increment())
        node2 = self.register_internal_node(Increment())
        node2.Input.connect_from(node1.Output)
        node1.Input.weak_connect_from(self.Input)
        self.Output.weak_connect_from(node2.Output)
```

Figure 6: A container implemented in *Python*

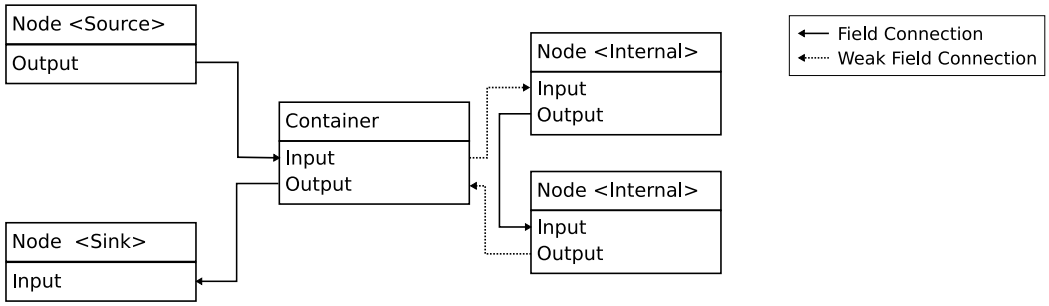


Figure 7: Container with internal subgraph

problem by allowing subgraphs to be encapsulated. A container is itself a field container and its fields are connected to the subgraph it contains. The application then only has to deal with a small number of containers.

We can define containers using the new features of AVANGO NG described in section 5.2. They are defined similar to the field containers using a simple helper class in *Python*. Figure 6 shows an example. In figure 7 the resulting data-flow graph is shown. Weak field connections are used to separate the subgraph from the normal graph. If these connection were normal field connections a cycle would result. We also need to deactivate the automatic evaluation of the internal field containers.

When the container is evaluated, it declares another dependency to the nodes in the internal subgraph using *evaluateDependency()*. As all dependent field containers are evaluated before the container is evaluated, this results in the subgraph being evaluated after these nodes. Finally all other field containers that depend on the container are evaluated. Therefore the correct evaluation order of the complete dependency graph is maintained.

7 Conclusion and Future Work

For the development of AVANGO NG, we used a demand-driven approach where only the features that were needed by actual projects were implemented. This helped us to concen-

trate on the core feature set of AVANGO NG. The code reuse of the old AVANGO was very high, which resulted in a relatively short development time for AVANGO NG.

The move from AVANGO to AVANGO NG was not just an incremental software update but a major redesign. During this process, several things have been changed at once. This together with the demand-driven approach made it difficult to predict the outcome and the amount of work. Fortunately, it turned out to be less work than expected. Also as seen in section 4.1, in AVANGO NG all modules only depend on the core module. Once this module was implemented, work on the other modules could be parallized.

All projects of our research group recently switched from AVANGO to AVANGO NG. Clear trends in the use of new features emerged:

- A lot more field containers are written in the scripting language and many new features are implemented using a collection of smaller components. One reason for this is of course that we are paying special attention to these design decisions and are opting for the use of components.

- The evaluation order implementation requires that no cycles exist in the dependency graph and we were first not sure whether this was attainable in real-life situations. In some more complex situations cycles were indeed created, but all of them could easily be detected and removed by simple design changes.

While all the functionality of AVANGO has been ported to AVANGO NG, some future work can explore further potential for optimization. The dependency graph is currently only used to determine the evaluation order. Some unneeded calculation might be performed, if calculated output values of a field container are not required to render the next frame. A simple algorithm that avoids these calculations is the *Push-Pull* model. To integrate this algorithm into AVANGO NG the dirty flag of a container has to be *pushed* to all containers that depend on it. During evaluation only those field container would be evaluated, whose values are actually needed by the renderer, i.e. when being *pulled*.

Some field containers would need to be updated to make full use of this algorithm. A selector node, that routes one of many input values to its output, would only *pull* values from the select input. These nodes therefore have to control their dependencies themselves and cannot rely on the deduction of dependencies by the system.

Our current experience with AVANGO NG is too limited to estimate if such an optimization has a larger effect on the overall system performance. We will therefore continue to evaluate application design and analyze for potential benefits.

AVANGO NG will be downloadable in the near future from: `http://www.avango.de`

Acknowledgements

We would like to thank the following persons for their contributions to AVANGO NG: Armin Dressler, Thorsten Holtkämper, Christian Nowke, John Plate, Sascha Scholz and Gerold Wesche.

References

[AGL+02] Allard, J., Gouranton, V., Lecointre, L., Melin, E., Raffin, B.: *Net Juggler and SoftGenLock: Running VR Juggler with Active Stereo and Multiple Displays on a Commodity Component Cluster.* Proceedings of the IEEE Virtual Reality, 273–280, 2002.

[Beh06] Behr, J.: *Avalon: Ein skalierbares Rahmensystem für dynamische Mixed-Reality Anwendungen.* Ph.D. thesis, TU Darmstadt, 2006.

[BJH+01] Bierbaum, A., Just, C., Hartling, P., Meinert, K., Baker, A., Cruz-Neira, C.: *VR Juggler: A Virtual Platform for Virtual Reality Application Development.* Proceedings of the IEEE Virtual Reality, 89–96, 2001.

[BLRS98] Blach, R., Landauer, J., Rösch, A., Simon, A.: *A highly flexible virtual reality system.* Future Generation Computer Systems, **14**(3–4), 167–178, 1998.

[BP] *Boost Python.* `http://www.boost.org`.

[GHJV93] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns: Abstraction and Reuse of Object-Oriented Design.* Addison Wesley Longman Publishing Co., Inc., 1993.

[OSG] *Openscenegraph.* `http://www.openscenegraph.org`.

[RH94] Rohlf, J., Helman, J.: *IRIS performer: a high performance multiprocessing toolkit for real-time 3d graphics.* Proceedings of the SIGGRAPH '94, 381–394, 1994.

[SA05] Sutter, H., Alexandrescu, A.: *C++ Coding Standards.* Addison Wesley Longman Publishing Co., Inc., 2005.

[STF00] Springer, J., Tramberend, H., Fröhlich, B.: *On Scripting in Distributed Virtual Environments.* 4. Immersive Projection Technology Workshop (IPT), 2000.

[Tra03] Tramberend, H.: *Avocado: A distributed Virtual Environment Framework.* Ph.D. thesis, Universität Bielefeld, 2003.

[VIR] *Virtools.* `http://www.virtools.com`.

[Wer93] Wernecke, J.: *The Inventor Mentor: Programming Object-Oriented 3d Graphics with Open Inventor, Release 2.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.

[WGB+07] Wetzstein, G., Göllner, M., Beck, S., Weiszig, F., Derkau, S., Springer, J. P., Fröhlich, B.: *HECTOR - scripting-based VR system design.* SIGGRAPH '07: ACM SIGGRAPH 2007 posters, 143, 2007.