Scene synchronization in close coupled world representations using SCIVE

Marc Erich Latoschik, Christian Froehlich, Alexander Wendler

Abstract— **This paper introduces SCIVE, a Simulation Core for Intelligent Virtual Environments. SCIVE provides a Knowledge Representation Layer (KRL) as a central organizing structure. Based on a semantic net, it ties together the data representations of the various simulation modules, e.g., for graphics, physics, audio, haptics or Artificial Intelligence (AI) representations. SCIVE's open architecture allows a seamless integration and modification of these modules. Their data synchronization is widely customizable to support extensibility and maintainability. Synchronization can be controlled through filters which in turn can be instantiated and parametrized by any of the modules, e.g., the AI component can be used to change an object's behavior to be controlled by the physics instead of the interaction- or a keyframe-module. This bidirectional inter-module access is mapped by, and routed through, the KRL which semantically reflects all objects or entities the simulation comprises. Hence, SCIVE allows extensive application design and customization from low-level core logic, module configuration and flow control, to the simulated scene, all on a high-level unified representation layer while it supports well known development paradigms commonly found in Virtual Reality applications.**

Index Terms—**Application Framework, Intelligent Virtual Environment,, Simulation Core, Ubiquitous Computing, Virtual Reality**

I. Introduction

Developing Virtual Reality (VR) applications or real-time simulations in general can be a complex task. Besides the integration of a variety of hardware devices for input and output, the desired simulation features often demand an extensive combination of special purpose software modules, e.g., for the simulation of graphics, sounds, collisions, physics or haptics. Furthermore, advanced requirements for *believable* worlds or *intelligent* environments frequently demand the integration of Artificial Intelligence methods to either support intelligent behavior of the application itself, e.g., in the area of smart graphics, ubiquitous computing and multimodal interaction, or to provide autonomous entities---like agents--with cognitive capabilities and access to a semantically described environment.

There is already a multitude of tools and frameworks for developing VR applications. In the next section, we will briefly illustrate representative examples to shape out useful and required as well as missing features which motivated us to develop SCIVE. Its specific aspects will then be explained in the following sections.

II. Related Work

Dating back to Sutherland's early work on an HMD with hidden line graphics [1], many real-time VR applications center around the graphical representation since visual perception is considered a primary sense to be stimulated for immersive Virtual Environments. Real-time 3D computer graphics encompasses work on rendering algorithms as well as on graphics interfaces---from direct rendering APIs like GKS [2] OpenGL or shader-based approaches to high-level graphics data structures with advanced capabilities: Scene graph tools like Open Inventor [3], OpenGL Performer, Open Scene Graph, OpenSG [4] or X3D [5] follow an hierarchical scene structure which additionally provides performance optimizations, e.g., for picking, culling or state sorting. Extension mechanisms, field route data propagation networks as well as a scripting layer support the design of new customized nodes and interconnected application graphs using rapid prototyping mechanisms.

Several purpose-built VR development tools adopt these concepts and additionally provide VR specific key features: First, input/output device customizability and embedding [6] is mandatory, see, e.g., AVANGO [7], Lightning [8], VR Juggler [9] or commercially available ones like the CAVELib™ or the WorldToolKit® Second, network distribution features are commonly integrated, e.g., in AVANGO [7], MASSIVE 3 [10], DIVE [11] or Net Juggler. They either allow distributed rendering on cluster architectures, hence again output device support, or to develop shared virtual environments, e.g., for collaborative work. Third, application programmers often require an entity centered access to world states or world logic which is often realized using event mechanisms as---illustrated exemplarily---in Lightning [8].

Manuscript Received 20.10.2006

M. E. Latoschik is with the AI and VR Group of the Faculty of Technology, Bielefeld University, 33594 Bielefeld Germany (e-mail:marcl@techfak.uni-bielefeld.de).

C. Froehlich is with the AI and VR Group of the Faculty of Technology, Bielefeld University, 33594 Bielefeld (corresponding author, phone: +49 521 106 2918, e-mail:cfroehli@techfak.uni-bielefeld.de).

A. Wendler was with the AI and VR Group of the Faculty of Technology, Bielefeld University.

As pointed out in [12] a module based approach has several advantages. But finding an abstract layer of module interconnection schemes is still an open research topic. Here FlowVR [12] favors a noncentrally managed approach concerning module synchronization. The messages being passed between the modules are processed by filters and synchronizer-objects which resolve nonlocal constraints. In contrast to this, one central synchronization scheme supported by SCIVE bases on a delayed rule based conflict management and resolution step.

A.Discussion

Believable virtual worlds require more then just graphics and input/output handling. Multiple simulation modules are required (see,e.g. [13]) to render animations, sounds, physics, or haptics (to name just a few) or to include intelligent behavior [14], e.g., for novel multi-modal interaction techniques or the animation of autonomous agents in the worlds. Such modules are either included in the VR development tools on a case by case base, or they are integrated a priori into holistic architectures as found in many 3D game engines like the Doom 3 Engine, the Unreal Engine 3, the Source Engine, the C4 Engine or the CryENGINE™.

Both, the case-by-case as well as the holistic architecture, have their drawbacks when it comes to application customization and longer-term reusability, persistence and portability. The first one requires a deep understanding about the internal algorithms and data structures of the utilized tool. It requires extensive low-level implementation efforts to customize or exchange a specific module, e.g., if a certain software library is no longer available or if it is not available on a given operating system. The holistic approach often doesn't even allow an extensive modification. Either the provided features meet an application's requirements or not. If source access is granted, extending such a tool rises the same problems as the first case, if not, the tool will render inappropriate for the task. The following sections will introduce aspects of a simulation core architecture for intelligent environments which follows a modular approach. It allows a fine grained control over simulation module data exchange and synchronization while it minimizes module dependencies.

III. Concepts and Architecture

SCIVE's general architecture provides the base mechanisms to build intelligent real-time capable applications. Its design incorporates concepts to latch simulation modules based on concepts and techniques which proved to work satisfactory regarding prior related work.

The architecture allows module interconnection schemes ranging from simple **loose coupling** to a tight **close coupling**. In the first case, a given module is infrequently contributing to an overall world state, e.g., the module's simulation results might only access down to one attribute of one specific entity only once in a while with respect to the main simulation rate. In the second case, a module might access the complete world state, every entity and every attribute for every main simulation step performed.

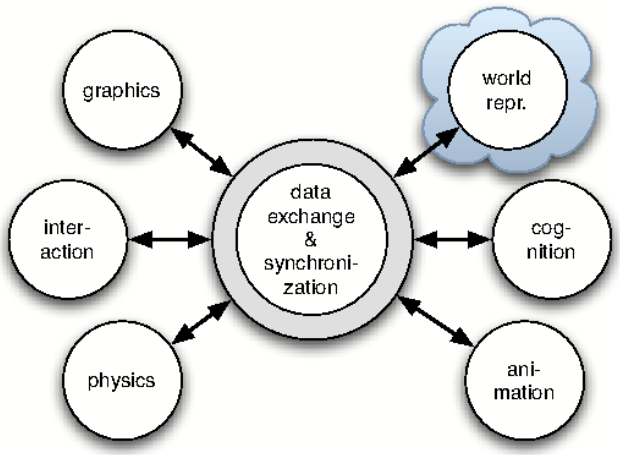

Figure 1. Conceptual interconnection scheme: Every required module should be able to exchange data with every other module ranging from single entity attributes to complete world states. This requires facilities for data exchange and control flow.

Modules are conceptually coupled by a **temporal synchronization facility** (see figure 1) which serves two purposes: First, it bootstraps the system by loading the required world representations for the modules to assure an initial coherent state in all modules. Second, it triggers the modules' local simulation loops asynchronously and controls the following data collection, **conflict resolution**, and data propagation steps.

This provides an implicit performance boost on parallel architectures (using multiple cores, CPUs, or hosts). The synchronization overhead is accepted in favor of a clean design. It separates the necessary low-level application logic in the conflict resolution step which is the requirement for the following parametrization: So-called **filters** implement a specific logic between attributes and hence control module access to the world state, e.g., they determine the values of attributes. A **data exchange facility** [15]---on the other hand---translates these values between different representations of conceptually same attributes in the diverse modules.

A.System configuration

General system setup and module configuration is provided via an XML-format called SCML (Simulation Core Modeling Language). SCML allows the application designer to specify the modules in a declarative way along with some important

parameters. The use of an external declarative format has some advantages. The application designer can build simulations without going deep into the source code of the simulation core. This protects the user and the system equally. Because SCML is based on XML it is quite comfortable to use and the user only needs few---if any---programming experience to design an application, for XML is quite intuitive and easy to learn. The following shows an SCML-example which sets up the three simulation modules:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE simulation SYSTEM "./scive.dtd">
<simulation>
 <components>
   <component type="Graphics">
    <configuration path="OPENSG"/>
    <priority level="normal"/>
    <load-time rank="2"/>
    <frequency clock="1"/>
   </component>
   <component type="Physics">
    <configurationpath="share/xml_data"/>
    <priority level="high" />
    <load-time rank="3" />
    <frequency clock="20" />
   </component>
   <component type="Semantic">
    <configuration path="share/semnet.xml"/>
    <priority level="normal"/>
    <load-time rank="1"/>
    <frequency clock="1"/>
   </component>
 </components>
</simulation>
```

The example SCML fragment illustrates the top-level definition of an application supporting three different modules which are interconnected to produce a coherent simulation of a believable artificial environment. Each module's definition is encapsulated in a `component` environment by specifying the module's specific type. In case of the example, a combined module for the graphical output and the user interaction, a module for the physical simulation of the world and a semantic-module which provides a semantic net for the knowledge representation layer [15] are defined. Additional parameters in the `component` environment configure the modules and their interconnection inside the SCML-definition: The `configuration`-parameter either specifies the path to a specific file used by the module---as specified for the `Semantic`-component in the example fragment---or in case of the `Physics`-component, the path to a directory with model definitions. The other displayed parameters (`load`-time, `frequency` and `priority`) are used for the temporal synchronization, which will be discussed in detail in the following section.

IV. Temporal Synchronization

Since the different simulation modules run asynchronously and each with a different update frequency, they have to be synchronized to ensure a consistent state of the simulated world. The different clock rates are necessary to guarantee accurate computation of the specific simulation-data. For example, the physics engine requires a higher clock rate than the graphics renderer. While the renderer computes one frame the physics simulator may has to compute twenty internal steps to ensure a mathematical accurate result. These differences become even clearer when we take a look at haptics feedback. A module computing haptics feedback needs a minimum update frequency of approximately 1000 hertz to give the user a realistic sensation. Hence inter-module synchronization becomes important to keep module data consistent with each other.

The temporal synchronization within SCIVE is divided into two basic areas. On the one hand the so called macrotemporal area and on the other hand the microtemporal area. While the macrotemporal area covers all the steps which are required at the startup of the system, as well as those steps executed when loading new modules at runtime, the microtemporal area includes those, which are executed for every master simulation step.

A.Macrotemporal Processing

This section deals with the necessary steps for the macrotemporal computing of the simulation. This includes initializing of the modules as well as loading of the simulation data. As mentioned above in Section III-B, the specification of the modules is accomplished via SCML. SCML also includes some parameters important for the temporal synchronization. Such a parameter is the `load-time`-parameter, which has to be specified for every module participating in the simulation. This parameter determines the position of the module in the initialization-order of the simulation. The components are initialized in ascending order according to their `load-time`-parameter.

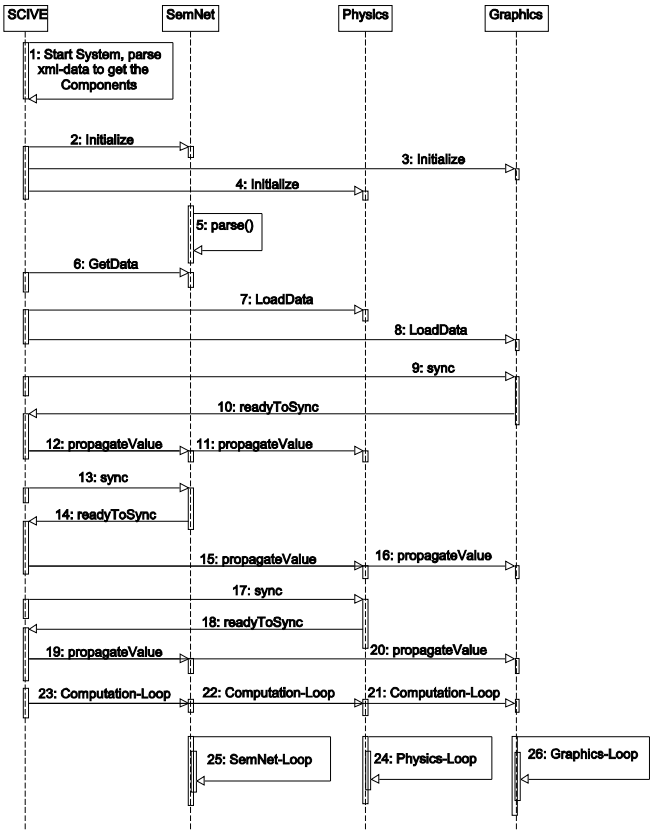

Figure 2. Macrotemporal processing of SCIVE

To give an impression of what steps are executed during the macrotemporal processing of SCIVE, we will present the necessary actions on the basis of the sequence diagram in figure 2. It illustrates the macrotemporal processing of the SCIVE-framework with three simulation-modules (Graphics, Physics and Semantic). The bracketed numbers---**(i)**--- represent the numbers according to the various steps in the sequence diagram.

The first step **(1)** starts-up the system and initializes SCIVE. This process parses the SCML-configuration including the definition of the desired modules and processes this information internally. Once SCIVE has gathered all necessary information from the SCML-file, each component is loaded along with their parameters **(2-4)**. When properly initialized, the Semantic-module parses the database specified in SNIL (SEMANTIC NET INTERCHANGE LANGUAGE) **(5)**. After the database has been parsed, SCIVE saves the data for the different modules in separated maps, and loads them along with the required representations for the data-exchange mechanism inside each specific module. The next steps **(9-20)** trigger the initial data-synchronization for every participating simulation module: SCIVE sends a *sync*-signal to each module which tells them to synchronize their data. After the modules answer with a *ready-to-sync*-signal, SCIVE propagates the data-values to each module registered for them.

This initial data-synchronization ensures that a consistent world state exists between the modules. Once this consistent state is established, SCIVE signals the modules to start their own processing loops **(21-23)**. The processing of the different components is displayed in steps **(24-26)** and will be elucidated in the following section.

B.Microtemporal Processing

Microtemporal processing within SCIVE will be illustrated for an example application which---since SCIVE doesn't predefine any application architectures---here centers around the graphics representation as the central world state.

Hence, the first step in the sequence-diagram displayed in figure 3 synchronizes the graphics-module's data with the other participating components **(1)**. As a result, all simulation modules now initially work on the data which was generated in the prior master simulation step. The sync-signal is answered by a ready-signal **(2)** from the graphics module and SCIVE propagates the data to the other modules **(3-4)**. The next steps consist of the parallel computations of the other simulation modules (in this case the Physics- and the Semantic-module). Steps number **(6-7)** start the processing of these modules inside their own update frequencies. The processing itself is displayed in **(8-9)**. Once the modules finish computation, they synchronize their results with each of the other modules, again through SCIVE`s data-exchange mechanism **(9-12)**.

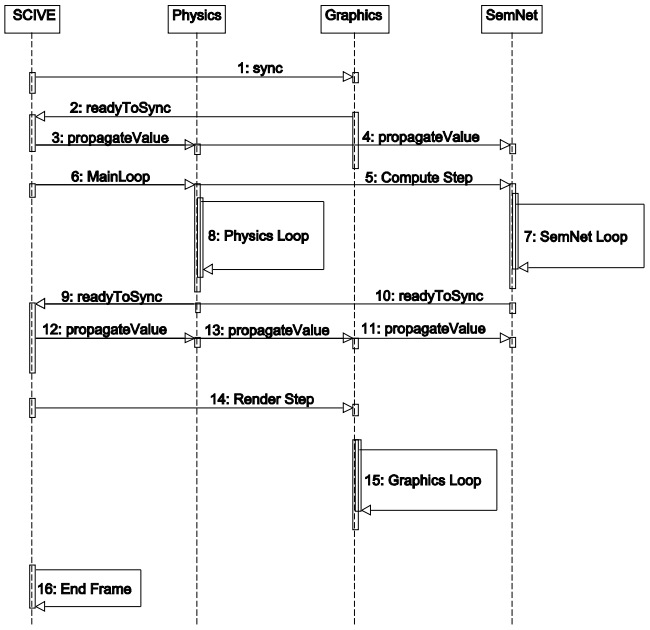

Figure 3. Microtemporal processing of SCIVE

The final step in the microtemporal processing is the

rendering of the computed scene. SCIVE tells the renderer to display the scene **(14)** and the graphics renderer executes its render-loop **(15)**. Occurring conflicts between the different simulation modules are resolved via filters, which will be explained in the following section.

V. Controlling Flow

The concurrent access to the central database by the various simulation modules could lead to an inconsistent world state resulting in wrong or unexpected behavior of entities, if the attribute would not be guarded or controlled in some way.

Here, SCIVE provides a special type of a data propagation graph utilizing a system of connections and filters. For example, in a simple configuration filters can forward the output of the physics module to one entity, a second entity can be controlled by the skeletal animation, and yet another entity can be controlled by the user interaction. In more complex scenarios, the filters can compute new values for controlled entity attributes by combining outputs of two or more modules or filters. SCIVE offers various filters that can change attribute values directly or indirectly, e.g., a developer can choose whether he wants an object to be dragged by directly setting the new position in all modules (including the physics module) or by applying appropriate forces generated by the physics engine. Filters can be instantiated manually or by the various simulation modules which allows automatic flow control by the application logic, e.g., if the multi-modal interpretation module triggers a drag action, a filter is set in place which binds the interaction target entity to the interaction module.

A. Conflict resolution core

The filter-based data-flow is provided by SCIVE's conflict resolution component. It is an optional facility but it provides the necessary functionality for the design of complex interconnected applications. It can be completely enabled and disabled on the fly. With disabled conflict resolution, each value change will be immediately applied to the central database and the databases of the modules, overwriting all earlier changes. With enabled conflict resolution, all value changes (triggered as events by SCIVE) will be delayed until the beginning of the final render stage. In this case, the requested state changes will be stored to use them as input for the filter stage. During this stage, filter can 1) forward an event, 2) combine it with other events and the relating values, or 3) completely block it. The order of events is less important for the conflict resolution since it applies filters with respect to the event source and its simulating feature. The last filter in the chain is connected back to the attribute container in the main database. After the evaluation of filters and propagation of changes, all modules must be informed about the rejected changes. This additional step is necessary, because the module that has requested the change, has possibly already changed its internal state and representation.

B. Filters

Connections and filters establish an event propagation graph. They receive events they have registered for. Filters can process computations on the signaled attribute values and finally produce new signals. In addition, instead of returning events, a filter can trigger execution of specific actions in the simulation modules, e.g., to apply some forces in the physics module or to generate a new animation which simulates an agent's reaction to external influences (as eventually triggered by the other modules). A certain required action often can be implemented with different methods and hence filters. For example, dragging of an entity can be implemented by setting the new position as a result from the interaction module which basically follows the user's "drag" hand or by applying some forces by the physics module to the entity. In the basic SCIVE interconnection, the interaction module can just change the position where the actual action is determined by the current filter.

The following filters realize the required functionality for the example application.

1) **Last module** pass through. This is the same as if no conflict resolution is done.
2) **Random module** selection pass through.
3) **Specific module** pass through.
4) **Prioritized module** list selection. Pass the events from a priority sorted list of modules. If queue is empty for a chosen module, take the next lower prioritized module.
5) **Physics module** pass through. Apply forces to the entity for all other events with position changes.
6) **Skeletal animation module** pass through. Generate a dynamic animation and blend it with the current animations for all other events with position changes.

The user can implement additional filters e.g. for calculating an average value of the incoming events. The conflict resolution set-up defined for the example application allows to simulate physical-based animations on the fly and to mix them with motion captured or pre-calculated animation data. This example illustrates SCIVE's powerful extensibility which is utilized at this point to produce believable interactions of skeletal animated characters with the environment in real time. In order to react to physical forces, a physical representation of the character is built up which, on the one hand, influences other physical bodies and, on the other hand, reports the displacement of the character caused by collisions back to SCIVE. In case of a collision, the established filter interconnection decides how to react to the

displacement. The agent could just drag the affected parts back, wobble, or fall down. The reaction can depend on the force and the place of the impact. Figure 4 shows the generated animation as result of a collision between a character and a static object as well as the interaction between the animated and a dynamic object.

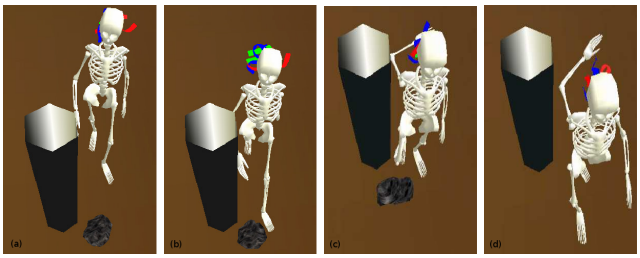

Figure 4. The animated agent filter in action. The motion captured animated agent collides with a relatively heavy (the column) and a lighter (the stone) obstacle in its predetermined path. The collision signals the activation of an animation blending filter which controls the colliding body parts for the specific time interval whereas the collision objects stay under physics control.

C.Filter application

SCIVE's supports filter application via rules which assign filters to certain scene parts. These parts can have different granularities from one attribute of one entity to all attributes in the scene. Prioritization of rules ensures that attributes already connected by a given rule will not be reconnected by lower prioritized rules. For example, a scene with a skeletal animated agent can be described by this two rules:

1) Assign "all properties of object A" to "SkeletalAnimationFilter"
2) Assign "all properties in scene" to "PhysicsOnlyFilter"

The second rule doesn't influence the object A (the agent), because the first rule has a higher priority.

The created filter graph can be changed on the fly by inserting and removing rules. To drag an entity X, the following new rule has to be installed just before the other two:

3) Assign "position of object X" to "InteractionFilter"

This results in the disconnection of the position attribute of entity X from the "PhysicsOnlyFilter" and its connection to the "InteractionFilter". At the interaction end, this rule is simply removed to establish the initial application logic.

This prioritized application of rules, which describe what the user wants to happen in the scene, provides a convenient way to create and assign the---possibly---multitude of required filters. Each rule takes just few lines of code or script that---at the end---defines complex filter graphs which will now automatically be created and removed. This lets application designer focus on application semantics rather then to care about the proper connection of filters and attributes.

VI. Conclusion

This paper has illustrated the basic concepts and mechanisms of SCIVE, a real-time simulation development system for intelligent applications. SCIVE's architecture follows a modular approach which considers long term project requirements as well as application design aspects. It provides a Knowledge Representation Layer which initially serves as an interconnecting representation for the various involved simulation modules' representations. In addition, the KRL provides semantic based access to the scene's entities as well as a unified access to scene semantics. SCIVE defines a simulation application as a set of simultaneously working---freely exchangeable and customizable---modules for the diverse simulation aspects. Data exchange between the independent world representations as well as synchronization between the independent simulation loops can conveniently be configured on a high level which includes the definition of general application logic down to per-attribute changes via the filter and rules concept. SCIVE's general approaches are motivated by, and currently applied to several applications in the area of multi-modal communication in virtual environments and AI supported virtual prototyping. Its capability to integrate physics, animation, AI, etc. for building intelligent agents has just been demonstrated following the example and is currently utilized to design a large scale continuously running virtual world for agent interactions.

Besides SCIVE's basic application oriented extension mechanisms which will incrementally add more modules, filters, and rules, we are currently exploring two technical optimizations: First, to map the inherent parallel interconnection scheme to node---and hence network---distribution using a multi-agent architecture (Distribution is currently provided by two connected graphics and VR modules: OpenSG and AVANGO). Second, to optimize the temporal synchronization and conflict resolution component to signal modules in case of unnecessary computations. SCIVE's AI base formalism offers various extension possibilities which we just began to explore by modeling ontology bindings for multimodal interaction, knowledge supported virtual construction, and game-engine oriented world state. In combination with, and as a control instance for the conflict management, it could provide the basis for a high-level semantic description even of complex simulations.

References

[1] I. E. Sutherland, "A head-mounted three-dimensional display," in *Proceeding* of the Fall Joint Computer Conference. AFIPS Conference *Proceedings.*, ser. AFIPS, vol. 33, Arlington, VA, 1968, pp. 757–764.

[2] ISO/IEC, JTC 1/SC 24, "Graphical Kernel System for three dimensions (GKS-3D)," ISO/IEC, Tech. Rep. 7942-1:1994, 1994.

[3] P. S. Strauss and R. Carey, "An object-oriented 3D graphics toolkit," in *Computer Graphics*, ser. SIGGRAPH Proceedings, vol. 26, no. 2, 1992, pp. 341–349.

[4] D. Reiners, G. Voß, and J. Behr, "OpenSG: Basic Concepts," www.opensg.org/OpenSGPLUS/symposium/- Papers2002/Reiners Basics.pdf, february 2002.

[5] ISO/IEC, JTC 1/SC 24, "X3d abstract," ISO/IEC, Tech. Rep. 19775-1:2004, 2004.

[6] S. M. Preddy and R. E. Nance, "Key requirements for cave simulations: key requirements for cave simulations," in *WSC '02: Proceedings of the 34th conference on Winter simulation*. Winter Simulation Conference, 2002, pp. 127–135.

[7] H. Tramberend, "A distributed virtual reality framework," in *IEEE Virtual Reality Conference*, 1999, pp. 14–21.

[8] R. Blach, J. Landauer, A. Rsch, and A. Simon, "A Highly Flexible Virtual Reality System," in *Future Generation Computer Systems Special Issue on Virtual Environments*. Elsevier Amsterdam, 1998. [Online]. Available: citeseer.nj.nec.com/396677.html

[9] A. D. Bierbaum, C. Just, P. Hartling, K. Meinert, A. Baker, and C. Cruz-Neira, "VR Juggler: A Virtual Platform for Virtual Reality Application Development virtual platform for virtual reality application development," in *IEEE Virtual Reality 2001 conference proceedings*. Yokohama, Japan: IEEE Press, 2001, pp. 89–96

[10] C. Greenhalgh, J. Purbrick, and D. Snowdon, "Inside massive-3: flexible support for data consistency and world structuring," in *Proceedings of the third international conference on Collaborative virtual environments*.ACM Press, 2000, pp. 119–127.

[11] O. Hagsand, "Interactive MultiUser VEs in the DIVE system," *IEEE Multimedia Magazine*, vol. 3, no. 1, 1996.

[12] J. Allard, V. Gouranton, L. Lecointre, S. Limet, E. Melin, B. Raffin, and S. Robert, "Flowvr: a middleware for large scale virtual reality applications," in *Proceedings of Euro-par 2004*, Pisa, Italia, August 2004.

[13] A. Kapolka, D. McGregor, and M. Capps, "A unified component framework for dynamically extensible virtual environments," in *Fourth ACM International Conference on Collaborative Virtual Environments*, 2002.

[14] M. Luck and R. Aylett, "Applying Artificial Intelligence to Virtual Reality: Intelligent Virtual Environments," *Applied Artificial Intelligence*, vol. 14, no. 1, pp. 3–32, 2000.

[15] G. Heumer, M. Schilling, and M. E. Latoschik, "Automatic data exchange and synchronization for knowledge-based intelligent virtual environments." in *Proceedings of the IEEE VR2005*, Bonn, Germany, 2005, pp. 43–50.

Marc Erich Latoschik was born in 1968 in Herford, Germany. He studied mathematics and later computer science at Paderborn, New York, and the University of Bielefeld, where he finished his german diploma (MS) in computer science in 1996. In 2001, he received his PhD for his work on multimodal interaction for Virtual Reality.
After several accompanying years in the computer business, he joined Bielefeld's AI group in 1996 where he later became head of the AI & VR Lab and a lecturer at the Faculty of Technology. His research focuses on Intelligent Virtual Environments (IVEs). Specifically aimed at novel human-computer interaction methods, he is interdisciplinary devoted to Artificial Intelligence, real-time 3D computer graphics and simulation, language processing and Cognitive Sciences.
Dr. Latoschik is one of the initial founders and elected member of the steering committee of the german SIG (GI-Fachgruppe) on AR&VR. He is an active reviewer and program committee member for several relevant conferences, e.g., the IEEE Virtual Reality conference.

Christian Froehlich was born in 1981 in Buende, Germany. He studied computer science at the Bielefeld University, where he achieved his german diploma (MS) in computer science in March 2006.
Shortly after finishing his diploma, he joined Bielefeld University's AI group, where he is currently working on his PhD. His current field of research lies in designing simulation systems for Intelligent Virtual Environment (IVEs), with special interests in Artificial IntelligenceI, Virtual Reality and Computer Graphics.