IPT & EGVE Workshop (2005)
R. Blach, E. Kjems (Editors)

Towards a general concept for distributed visualisation of simulations in Virtual Reality environments

J.Metze[1], B. Neidhold[1], and M. Wacker[2]

[1]Lehrstuhl für Computergraphik und Visualisierung, TU Dresden, Germany
[2]Computergraphik, HTW Dresden, Germany

Abstract

We present a concept for Virtual Reality-Systems that allows easy design and implementation of applications in virtual environments as well as fast integration of new hardware and software components into an existing system of this type. Starting from an abstract module based approach for distributed systems we provide a guidance for specifying and designing components for VR-Applications with focus on the communication, administration and visualisation. Finally we have implemented a proof-of-concept for a cab simulator to demonstrate the effectiveness of our approach.

Categories and Subject Descriptors (according to ACM CCS): I.3.2 [Graphics Systems] Distributed/network graphics, C.2.4 [Distributed Systems] Distributed applications, I.3.7 [Three-Dimensional Graphics and Realism] Virtual reality

1. Introduction

In the last years computer graphics have entered our physical world by augmenting or even replacing reality applying photorealistic visualization and rendering on high resolution display technologies. Moreover, interaction technologies are added to render certain tasks of our life more efficient, repeatable, and understandable. The goal in the field of Virtual Reality (VR) is to immerse the user into a computer-generated synthetic world of his own making by visual, auditory, and tactile spatial presence. Since VR-Systems haven proven their usefulness providing sufficient robustness, functionality, and flexibility to find acceptance and to support its seamless integration in our real world, there has been a great interest and boast of new technologies as well as applications in many fields like medicine, automotive construction, or rapid prototyping to name just a few. The big advantage of VR-Systems is the ability to iterate certain tasks using computer models without huge costs or risks for humans. Additionally, it is used as a new tool for communication between wide spread users. Hence, one of the hottest topics in VR research is the idea of *distributed* VR. Additionally this solves another problem: Since models and the underlying systems for simulation, interaction, and visualisation become more and more complex, the simulated world runs not only on one, but on several computer systems, where the computers are connected over network.

However the use of VR is successful when it brings something new to the whole experience while maintaining the viability and usefulness of a product. Often VR-Systems were designed for specific tasks (or field of tasks) applying a special set of hardware and software. New hard- and software was difficult to integrate. To this end a structured design approach is critically important for the development of virtual and augmented environments in real-world applications. We are driven by this philosophy presenting a modular framework to be used on any kind of existing and future VR-System. Hence the same toolbox can be used on different systems or applications in order to compare or combine them to one large system. New components are easily integrated to have the latest technology available. Nowadays fast technology development is especially notable in the field of visualisation. Therefore we introduce a flexible shading system that transfers some parts of the visual rendering code into the object description (3D data). This allows us to integrate new visual effects on the fly without recompiling the code.

© The Eurographics Association 2005.

delivered by
EUROGRAPHICS DIGITAL LIBRARY
www.eg.org diglib.eg.org

2. Related Work

Since the birth of the notion *Virtual Reality* there has been an enormous activity in this field. Commercial industry as well as research institutes have established a huge pool of hard- and software to be applied in VR-Systems. Often the problem with commercial development in this field is the fact that the software is not open source making it difficult to incorporate new hard- and software modules. In the meantime all big companies have discovered the additional benefits using VR and have created their own custom software to visualize and manipulate large scale systems. One of the best-known and most successful standard for distributed VR has been DIS (Distributed Interactive Simulation protocol) with its predecessor SIMNET, which was well optimized for military simulations [PBL*94].

The Distributed Interactive Virtual Environment (DIVE, [CO93a] [CO93b] [CO96] [FGS99]) started in 1991, is an Internet-based multi-user VR-System where participants navigate in 3D space and see, meet and interact with other users and applications. It is based on a peer-to-peer approach with no centralized server, organized as a memory shared over a network where a set of processes interact by making concurrent accesses to the memory. Consistency and concurrency control of common data (objects) is achieved by active replication and reliable multicast protocols. A user sees a world through a rendering application called visualizer. Another general design for VR originated in 1996 is the Studierstube Augmented Reality Project [SFH*00]. Many applications show the wide range of usability where the developer can fall back upon a great pool of software tools like the Studierstube Render Array and the OpenTracker - An Open Software Framework for VR Input [RS01].

CAVERN [LJD97] (the CAVE Research Network) is built on top of the CAVE architecture and offers a distributed collaborative environment. Its toolkit is centred around the IRB, a network and database part, therefore decoupling it from the visualisation used. They aim to use the connected resources of many facilities to improve the design, training and education in virtual reality environments.

The VR Juggler [Bie00] approach – an open source virtual reality application development framework is very similar to the concept presented here. It provides to the developers a toolbox of application programming interfaces that abstract all interfaces including the display surfaces, object tracking, selection and navigation, graphic rendering engines, and graphical user interfaces. Also the FreeVR-project [Bil] is an open-source virtual reality interface/integration library. It has been designed to work with a wide variety of input and output hardware, with many device interfaces already implemented. One of the design goals was to be easily run in existing VR facilities, as well as newly established VR-Systems.

DIVERSE - An Open Source Virtual Reality Toolkit [KASK02] is a cross-platform, open source API for developing virtual reality applications that can run almost anywhere. The goal of DIVERSE is to enable developers to quickly build applications that will run on the desktop as well as various immersive systems with the ability to interact with many other APIs and toolkits like OpenGL, Open Scene Graph, SGI Open GL Performer, and Coin. The OpenMASK [Fre] (Open Modular Animation and Simulation Kit) software platform for the development of modular applications in the field of virtual reality. It can be used to describe the behavior or motion control of a virtual object as well as input devices control like haptic interfaces. For the visualization, Performer (Sgi) or the OpenSG framework (Fraunhofer Institute) may be used.

Avango [Tra99], a Fraunhofer Institute project uses OpenGL and VRML to visualize a distributed virtual reality environment. Its unique selling points are its scripting ability and the shared scene graph.

Covise [WSWL02] [WRR], coming from a SGI and supercomputer background is a highly modularized toolkit for a collaborative working environment with its main focus on computation fluid dynamics and finite elements methods.

Verdi (Virtual Environment for Real-time Distributed applications over the Internet) consist of several components providing a multi-users virtual reality server technology using multicast to optimize the sharing of information among distributed users (VRSAT). VRML and Java-based technology (Cortona) allow end users to navigate and to interact in shared 3D virtual worlds. The Avalon [Ava] project extends the X3D/VRML System by the functionality for interaction devices and complex projection devices such as a CAVE or a Heye-Wall. It is component based and is easily controlled by standard protocols like http or soap.

Our approach was written from the ground up. This was done both for minor legal reasons (a GPL license may not be applicable for a closed source industry project), and for finding the narrow path between flexibility and extensibility on the one hand and runtime performance while maintaining a high visual quality using modern graphic hardware on the other hand. Most existing designs suffer from concentrating on the collaborating/network aspects and therefore neglect the simulation and visualisation parts. So the design decisions of the presented concept were driven by the requirements of the visualisation and simulation modules. The goals were first to combine a state of the art visualisation with the flexibility to incorporate new technology and second to decouple visualisation from the simulation module. Due to the many possible applications out of this visualisation-simulation-concept the extension to a generalized VR-System was the next consequent step. However we always paid attention to keep the system small, preventing code and feature bloating. We chose a module based architecture especially to fulfil the needs for a quick integration process (postulated by a prototyping environment) as well as one prerequisite for distributed systems or modules. Each module has therefore to abstract from the underlying plat-

© The Eurographics Association 2005.

form and from other modules to ease the exchangeability. Our system is designed as a module library and not as a class library, so each part can be developed separately or integrated from external origins (even as closed source, when interfaces are adhered to). To maximize the benefits from a flexible concept and add new content as simple as possible, an integration pipeline is supported by an intuitive tool chain, explicitly serving the needs of non-programmers.

We started from a conceptual analysis of a VR-System which is described in chapter 3. Here we define the major goals and requirements for a distributed VR-Framework. In the next section we detail the different modules of our concept from the content creation up to the rendering of the simulated world. Especially our concept for the communication module is described here. Finally we apply our framework to a cab simulator as a proof-of-concept (section 5). We conclude this paper by an outlook to future work and a summary of our concept.

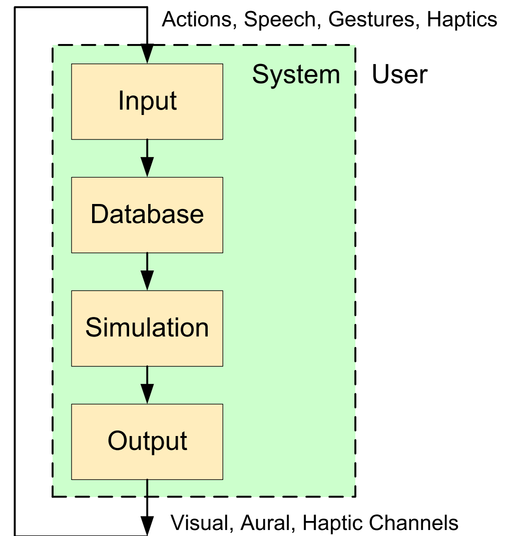


Figure 1: *The general system-user-loop.*

3. The system

Typically VR-Systems – composed of heterogeneous hardware and software parts – are used to setup (new) application scenarios and are characterized by a system-user-loop (figure 1) which consists of the user input (such as actions, speech, gestures, or haptics), the internal system including the steps: input interface between human and machine, the model database, a simulation process to calculate the new state of the scene, and finally an output step as interface between machine and human. To close the loop, a visual, aural, or haptic output is provided to the user. Desirably for new applications or new scenarios it should not be necessary to make changes to the underlying back end. However each new application needs to integrate existing or emerging hard- and software technology from wide spread domains (from new hardware devices and interfaces, hardware drivers over the physical simulation with new algorithms up to databases and research domains such as data mining, usability and ergonomics surveys) into the system-user-loop. In most VR-Systems, the integration of new components is a tedious work. Special modules are implemented for each application without the possibility to integrate them into one large unified system. In our system we provide an interface to incorporate different modules into the VR-System by an intuitive integration process. After the analysis of the new applications' requirements, its stock of hardware and existing content, ill fitted interfaces are adapted and missing data is extracted or created. To speed up the integration process a user-friendly tool pipeline has to be created. In our concept we present a well defined tool chain for content creation (section 4.1) using a general 3D data format, and a style sheet transformation process (section 4.5) to adapt interfaces.

To fit the VR-System to new application requirements each step in the system-user-loop is modelled through one or multiple modules specified by a well defined interface-set up, e.g. different input controllers such as tracking devices, steering wheels, mice etc.. If none of the existing modules matches the needed functionality, either a converter or a new module is created. Usually the modules are instantiated only once, one typical exception to the rule for VR-Systems being the visualisation system (a subsystem of the output step), as some output technologies (CAVE, multi-segmented power wall, Heye-Wall) require more than one point of view. Moreover, since no single PC could process the needed complex system the modules have to be distributed on different PC's. To specify the data and control flow through the system the communication links are laid out (e.g. defining components and channels between the modules, see section 4.5). By distributing the modules to different run-times or computers we gain several advantages over a single program version: If one subsystem fails it will not affect the others either through separate address spaces or explicit network interfaces. Using network technology the modules are not restricted to run on a single machine or operating system which simplifies the integration of external or proprietary software (e.g. common render tools like OpenGL Performer or Open Scene Graph). As to the natural use of threads and processes the usage of upcoming hyper threading and dual core technology will further boost the system's performance. Using this modularisation and distribution technologies we gain a flexible and extensible VR-System.

4. Details

In this section we describe the different steps for building a VR-System and integrating new technology and content in more detail. The first chapter deals with the work flow of

© The Eurographics Association 2005.

content creation and design in order to provide a database for the VR-System. Then, the next chapters focus on the necessary steps in the system design itself as already seen in figure 1. The different modules and the communication relationships are described. Here we lay out the general concept of our approach whereas we go more into detail concerning our application environment and implementational details in the next section.

4.1. Content creation

Every VR-Simulation needs threedimensional content as a basis to interact in and to display. Usually these 3D-scenes have an immense complexity (e.g. large scale outdoor or city scenes) and to this end we provide the concept of splitting the modelling process into several steps or abstraction layers. With this *hierarchical editing*, each layer can be edited independently (see figure 2) in the modelling pipeline. To simplify the data transport between the single layers and to avoid conversion problems we designed a flexible and extensible XML-Data format that is used throughout the modelling pipeline (see also section 5). For some applications (e.g. visualisation of medical data) the hierarchy may be not applicable, in this case our general XML-Data format allows the artist to model the whole scene as a single object directly in the polygon modeller.

Now we detail the work flow of the content creation, modelled by different layers in the hierarchical system: In the basic layer single atomic objects (e.g. tree, house, or car) in the VR-Environment are modelled by the *object editor* and stored in an object database for later (re)use. To incorporate highly specialised tools for polygonal modelling, with which artists are well trained to work, export plugins to standard modelling software have to be provided. In the next abstraction layer an artist designs special local *tiles* that represents rectangular areas of the VR-System combining an underlying terrain topology with objects from the object database (e.g. 50m x 50m tile with crossroads, trees, and houses. Special care is taken of designing streets and fitting them to the landscape. The diverse tiles are also stored in a tile database for later (re)use. The last layer of the hierarchical geometry design pipeline is the *scenario editor* where several tiles from the tile database can be placed together intuitively and stored as a whole scene. A screen shot of such an editor is shown in figure 3. Note that automatic border matching and a gap filling algorithm has to be provided. Using this pipeline large landscapes can be modelled at different level of details ranging from microscopic up to global scale.

Until now we described the modelling process of complex but static 3D scenes. All dynamic or simulation driven objects in the system are defined directly in the *workspace editor*. A simple car for example is dynamically added to the scene by assigning a car object out of the object database to a physical car simulation module. While simulating, the position of the car and its wheels (modelled as sub-objects)

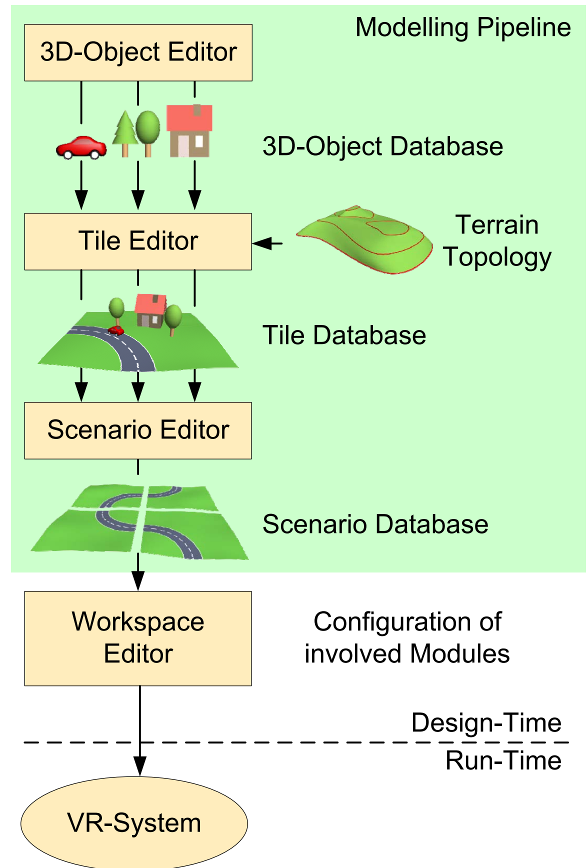


Figure 2: *Data flow for content creation.*

is controlled by the simulation module and displayed by the visualisation. All dynamic object connections and configuration parameters of involved modules are stored together and can be loaded directly by the VR-System.

4.2. Modules

The term *modules* in the context of VR-Systems can be related to one of three parts following the input-processing-output paradigm. Input modules log or measure data from external software or hardware devices. As there is a great variety of possible input devices a generalized input component eases the prototyping of new input devices. By distinguishing analogue from digital sources we can easily map buttons and axes to commands or normalized range data. The acquired data is sent to the processing part (simulation) that uses all of the incoming data to compute a new system state that is transferred to the output modules. Hereby the type of simulation can vary from machine simulation over crowd simulation to weather simulation. The output modules interpret this data to present the new system state to the user. By

© The Eurographics Association 2005.

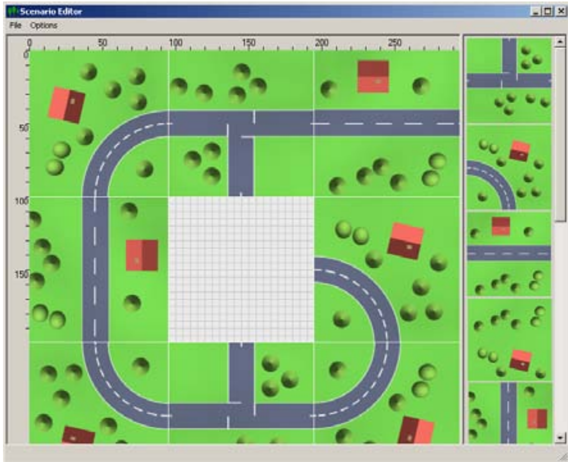


Figure 3: *Screen shot of the Scenario Editor.*

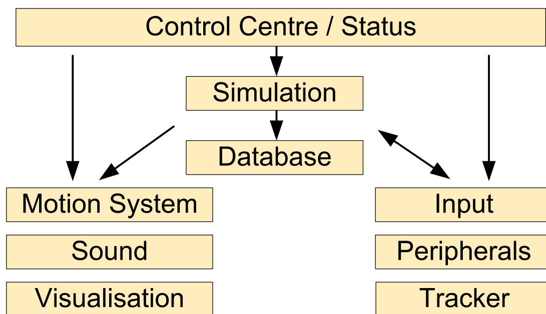


Figure 4: *Modules in the VR-Environment.*

designing them as thin as possible (e.g. move all time consuming computations to other modules), we can channel the available processing power to the rendering system, resulting in higher frame rates and increased visual quality. Each module is defined over its domain application and can have more than one instance, e.g. to allow cloning of output channels (such as CAVE walls). A central data module acts as a server preserving a consistent state across the system. The state of all modules is controlled through a defined interface by a special control centre (see section 4.4) that additionally has customized scripting abilities. Finally, as the modules are distributed over a wide range of machines, they have to be as platform independent and as free of interdependencies as possible.

4.3. Visualisation

Raised requirements with regard to visual quality often make it necessary for existing applications to tweak or expand the visualisation system, whereas new applications incorporate new technology to become state-of-the-art (per pixel lighting effects, HDR, soft skinning). Because applications (existing as well as new ones) should resort to a module stock, they should use the same visualisation system. The required extensibility could be achieved by using different versions of the visualisation source code so that new technology does not break existing one. But bug fixing and maintaining multiple versions is inefficient in terms of time and possible version conflicts. Moreover changes to the visualisation code require a compilation and are therefore time consuming and error prone. To solve this problem, modifications to the visualisation system should be applied to the data instead to the code obtaining a flexible and extensible system. Programmable graphic hardware and in particular high level shader languages (GLSL, HLSL) present a way to implement unique graphical effects and transfer the material and lighting system from the core system to the data creation process supported by tools such as RenderMonkey [ATI]. A visualisation object is now defined by a 3D model that encapsulates the material parameters and by an instruction code that defines the shader to draw the model. Changes to either the model or the shader code can be propagated directly to the running application, so not even an application restart is required. Hence, newer technologies like parallax mapping or a more complex system like precomputed radiance transfer (PRT [SKS02]) can be used by the application without making changes to the underlying visualisation system. Using a shader system of this type, we face two remaining problems: multi pass rendering and application control. A vertex or fragment shader defines only a single computation pass. To combine them we need a way to specify the shader to use for the corresponding pass. Additionally a shader has only very limited local control. A vertex shader can control several vertex attributes such as position, normal or colour, a pixel shader is limited to a pixel's colour and depth value. There is no way to change the object's or application's behaviour. Examples are loading a texture that is specific to a certain kind of graphical effect (e.g. a gradient texture for a projective shadow mapping), setting application wide settings (e.g. alpha blending) or texture stage parameters (e.g. texture addressing modes). All this requires the application to gain knowledge about the shader being used and to choose a different code path. But this hinders the extensibility and flexibility. A first yet platform dependent step (and therefore not flexible enough for a generalized solution) is offered by Microsoft's DirectX. Though it is superseded by another concept in the presented system its core ideas are still valid. DirectX extends the shader language with a descriptive effect language (now known as DirectX standard annotation syntax, DXSAS). It enables the effect to use multi pass rendering and has limited application control in a way that it gives access to the graphics API. CGFX proposes a solution that supports DirectX as well as OpenGL but does not allow any sort of control flow or scripting and is therefore not a viable solution to some problems.

So we propose an extended approach that keeps the name *effect* (see figure 5) but makes use of a scripting language

© The Eurographics Association 2005.

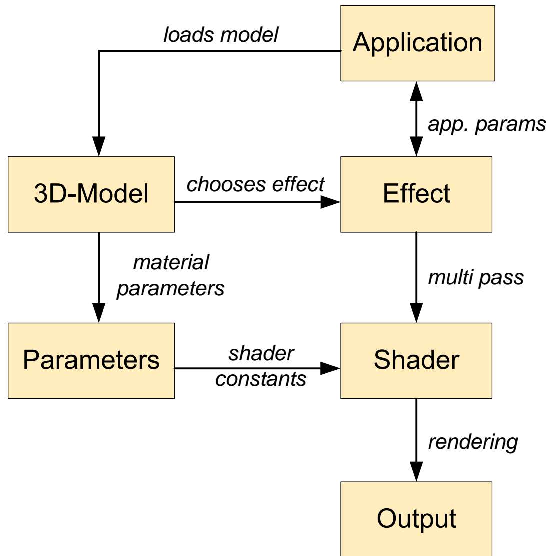


Figure 5: *Shading system control flow.*

to implement the desired features. The application calls the script with a defined interface to support multi pass rendering (Initialize(), BeginPass(), EndPass() etc.). The script itself uses a generated interface that wraps the visualisation system to change application or object specific settings. As the script is only invoked on a per object base, the speed hit is negligible. The step from a descriptive effect language to a procedural one allows advanced features like effect animation, shader level of detail, or user defined quality settings. Most of these are supported by DXAS or CGFX, but the choice of technique or animation control remains in the application code. A scripting system allows the user to change this behaviour on a much finer scale without making any changes to the application code itself, therefore fortifying flexibility.

4.3.1. Projection

The final sub module of each visualisation chain consists in displaying the produced images on some output device. This could range from a head mounted display up to a tiled multi projection display or a CAVE. This chapter deals with the problems especially arising with projection methods: To maximize the immersion in VR-Systems a field of view larger than 180 degrees, desirably 360 degrees is to be covered. To this end several displaying units are used which (back-)project the generated scene by possibly overlapping images on planar or curved surfaces. Hence, just projecting the produced images directly without further correction would produce spatial discontinuities resulting from different projection angles to the wall and different constructions of the projectors which noticeably hinder the immersion in the VR-System. The analysis of the projection methods yields separate problems resulting in different distortion types. The first consist in the installation of the projectors. Usually they are mounted on the floor or to the ceiling, therefore the projection wall is not orthogonal to the optical axis. The second problem comes from the projection onto non-planar walls. Especially in simulators the images are displayed onto cylinder-, dome-, or sphere-shaped walls. To sum up all the mentioned projection types result in a non-uniform pixel density of the image perceived as distortion by the user which influences considerably the impression of the image.

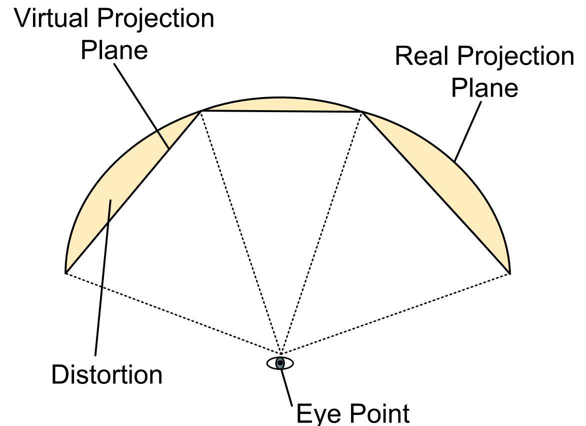
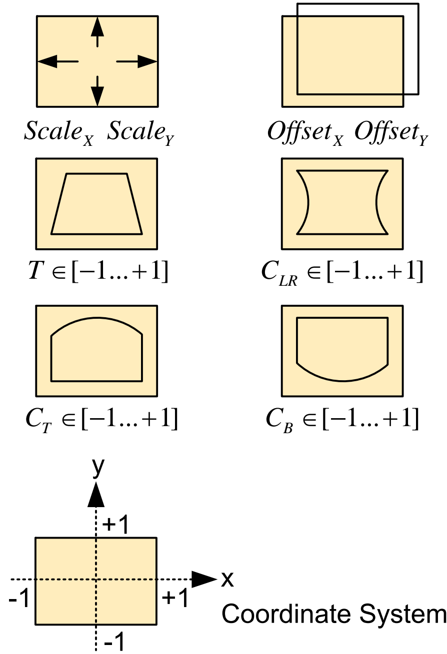


Figure 6: *Distortion due to installation of projectors and projection onto curved surfaces.*

Typically the first problem is solved by a special projection lens or calibration tools of the projector. This of course could be omitted in the case of back projection where the projector can be mounted onto the optical axis of the wall but is very rarely used for large VR-Systems like a CAVE or a simulator. Also the second problem may be tackled by special lenses. However for each projection plane configuration we would need such special cost-intensive hardware.

To preserve the flexibility of our approach and to adapt quickly to different situations and applications we propose a software based calibration algorithm in order to render the displayed images in a VR-System as close to reality as possible. The goal is to maximally reduce the distortion resulting from the projection set-up applying an inverse transformation. Since by a software approach concerning the pixel density of the projected images we are limited by our projectors, we can only reduce the quality of the projection which means a reduction in detail. Hence we trade of quality of the displayed image against a consistent image projection to the user to reduce the spatial discontinuities and maximize the immersion. To this end, two changes in the rendering pipeline are possible. First, before the projection step to the image plane, we could transform the (three dimensional) geometry of scene by an additional transformation which acts on vertex scale of the modelled world. In this

© The Eurographics Association 2005.



$$
\begin{aligned}
x &= Offset_X + Scale_X \left(x + \frac{x}{2} \left(C_{LR} \frac{y^2}{4} + T \cdot y \right) \right) \\
y &= Offset_Y + Scale_Y \left(y + \frac{x^2}{8} \left(C_T + y \cdot C_T + C_B - y \cdot C_B \right) \right)
\end{aligned}
$$

Figure 7: *Parametric pixel transformation function for software distortion.*

case large triangles could cause major problems making further global or view dependant subdivision steps necessary. Of course this leads to additional computational costs. The second, in our eyes, better approach, is pixel based where the image is transformed after the projection step in the rendering pipeline by rendering the image on a curved mesh. This step is calculated easily by modern graphics hardware and can be adapted to upcoming graphics cards. We propose to use a formula (figure 7) that describes the transformation of each pixel (x, y) in the rendered image. To modify the image distortion we use six parameters for scale, offset, trapezoid and 3 types of cushion. With these parameters all distortion situations on cylinder-, dome-, or sphere-shaped walls can be handled well. For further details of our implementation we refer the reader to section 5. We prefer this static parameter driven calibration approach to other previously published automatic calibration systems [CCF*00] [BEK05] because we do not need a visualisation onto irregular or dynamic projection planes. Another disadvantage of such systems is that they need additional hardware such as cameras and much computing power for image recognition.

4.4. Control Centre

As described in section 4.2 the control centre is a special module that is responsible for global control of all involved modules in the VR-System. The main tasks are state control, error handling and configuration control of each individual module as well as global workspace management (see figure 8). Such an module is necessary because all involved modules are distributed over the local network and need to work together in a defined way. To simplify programming and usage of the system we define a global state machine that must be implemented for all modules (see figure 9). In the control centre all modules are listed and any state change can be triggered for every module through the user interface.

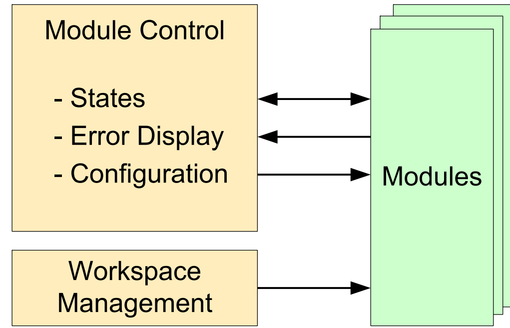


Figure 8: *Tasks of the control centre.*

Initializing the system, all modules are started into the *stopped* state automatically on boot time. This is necessary because all communication between the control centre and the other modules is done through a network channel that requires a running application. In this *stopped* state every module switches to a standby position consuming a minimum of CPU power and waiting for other commands from the control centre. All modules involved in the user interaction (e.g. visualisation, sound, input-controller) should indicate this state to the user. The visualisation for example could show a special screen. While the simulation is running all modules are in the *started* state, doing their assigned job. Especially when you want to stop the simulation for a short time e.g. for changing some parameters or analyse the actual output it is useful to add an additional *paused* state that mixes the behaviour of the other two states: For every module we define individually whether the module should continue its work or behave like in the *stopped* state (e.g. visualisation should go on working but a force feedback device should stop acting).

A global error handling for the modules is also done by the control centre. Through a dedicated network channel each module is able to send an error level together with a human readable message that is displayed in the control centre's user interface. With special error level definitions (*debug*, *info*, *warning*, *error* and *critical*) we are able to provide

© The Eurographics Association 2005.

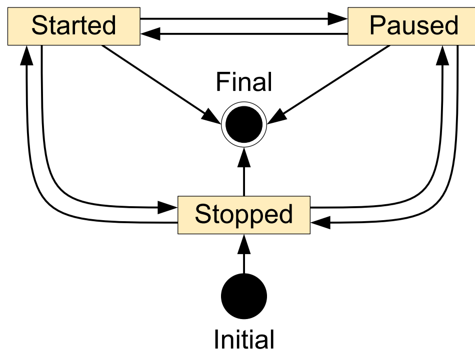


Figure 9: *General module state machine.*

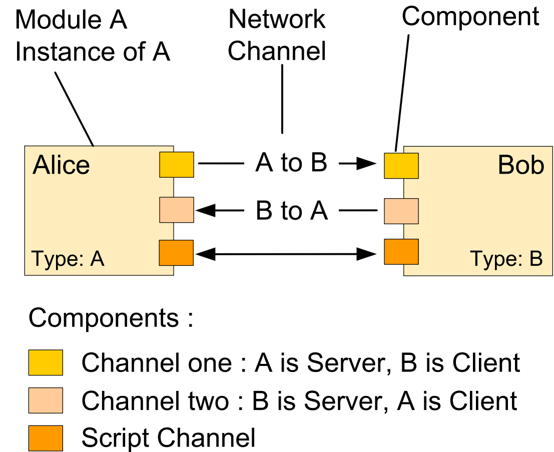


Figure 10: *Abstract communication scheme.*

both error handling and logging to the whole system. In seldom cases modules can exit because of unexpected critical errors that cannot be handled by the control centre. As the network system is no longer running, in this case the modules must be restarted by the user manually (e.g. with a remote control system [ATT]).

Another feature is the runtime configuration of the modules in the control centre by accessing the script interface of every module over a network channel. In detail this function is realized by sending script code when using special user interface element. For example a button called "resolution 800x600" sends the command "vis.sizex=800; vis.sizey=600;" to the visualisation to change their actual resolution. Beside state handling another important task of the control centre is to define the *workspace* in the actual session (see also section 4.1). The workspace includes initial configuration information for all involved modules. When the workspace is changed all modules are notified and reset to the *stopped* state in the new workspace environment.

4.5. Communication

As the modules need to transfer data between each other in order to fulfil the application's requirements some sort of communication infrastructure has to be established. To model the multiple communication links between the different modules some terminology has to be defined (see figure 10).

A module (e.g. input, database or visualisation) consists of one or more components. Each component either offers a service to other modules or uses another modules service. The communication between two components is defined as a channel, the end points are called server and client. As some modules can have more than one instance (e.g. the visualisation with multiple views) the channel either uses a one-to-one (client-server) or a one-to-many (broadcast) connection. In either case the component defines an interface to send method calls (like RPC [Sun88]) over a channel. A low-level network interface packs this structured data into byte arrays and sends them via sockets to the corresponding partner [Met]. The access to the Internet protocol is encapsulated in an exchangeable layer to allow the use of different network technologies [Myr] [ATM]. To define the mapping between a message's signature and its byte representation an IDL [ISO99] like language is used. This specification in XML (see figure 12) is transformed with an XSLT script (see figure 11) into a dynamic library code base. For each channel in the communication system one XML file is specified. The use of XSLT allows a simple integration of different implementation languages (C#, Java etc.) and operating systems by exchanging the code base and or network library. Therefore the usage is transparent to the developer of each module.

The components' architecture allows a quick adaptation to changed or newly defined communication interfaces, since only a code generation and a library rebuild is necessary. Most modules have a clearly defined domain interface but no knowledge about other modules that may use this interface. Using the component notion we bind a subset of the two module's domain interfaces together. During prototyping it would be preferable to have access to the complete functionality without the need to define a XML specification for the whole module. A built-in script component solves this in an elegant way, as each module publishes its interface to the scripting language (supported by an automatic wrapper generator [Bea] [Cel]). The script component gives access to the scripting interpreter on the target machine. So script code is transferred to the communication partner, executed locally and the return value is sent back. As this brings possible security problems (minor ones as the VR-System is a closed system) and a speed hit (source code is encoded less efficient) these type of communication should be limited to prototyping or out-of-band messages. An example for the usage of the script interface is given in section 4.4.

© The Eurographics Association 2005.

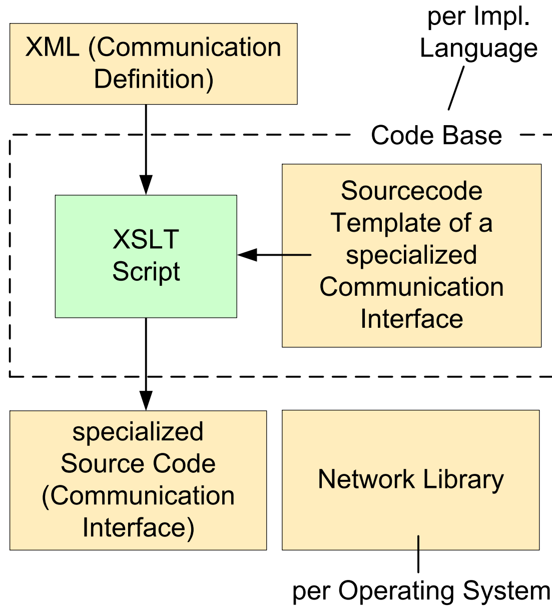


Figure 11: *Dynamic component generator.*

5. Implementation

All concepts described above have been applied to an implementation for the interactive automotive simulator hardware at the TU Dresden (see figure 13). This hardware consists of a six degree-of-freedom motion platform with a closed dome at the top. Inside the dome three projectors generate a 180 degree visualisation in front of a fully functioning replaceable cab that contains all input devices for the user (see figure 14). In the background there is the operator centre containing all computers and fast Ethernet network hardware used by the simulator. Through the module metaphor (see section 4.2) each module normally runs on a different computer to gain flexibility. The computers all over our system are up-to-date standard PCs (Intel or AMD) working with Windows or Linux, depending on the module.

```
<messagetable>
  <!-- computes X*Y over network -->
  <message name="ComputeProduct">
    <timeout>1500</timeout>
    <id>11</id> <!-- unique function id-->
    <returns>
      <type>double</type>
    </returns>
    <parameter>
      <param>
        <name>X</name>
        <type>double</type>
      </param>
      <param>
        <name>Y</name>
        <type>double</type>
      </param>
    </parameter>
  </message>
  <message><!-- ... --></message>
</messagetable>
```

Figure 12: *Example of a simple channel description XML file.*

Figure 13: *Outside the interactive automotive simulator.*

Figure 14: *Inside the interactive automotive simulator.*

As the graphical data exchange format of all tools in the modelling pipeline (described in section 4.1) we designed an own XML based scene graph format called XML3D [NBK*]. Compared to other related formats (VRML or X3D [Web]) the benefits of XML3D are very simple-to-use triangle geometry and shading functions, a powerful hierarchical structuring and grouping system and

© The Eurographics Association 2005.

the possibility to easily store additional data in a way that the file remains readable by the visualisation (e.g. add collision information for the physical simulation). As 3D object converter into XML3D we implemented plugins for Discreet's 3ds max and Alias' Maya to benefit from these state of the art polygonal modelling tools. The other content aggregation tools in the modelling pipeline (Tile-Editor and Szenario-Editor) are implemented in C# because of the powerful and easy-to-use UI classes in Microsoft's .NET Framework.

Besides the graphical input data, that is loaded by the *database* module during the initialisation process, all other input to the actual VR-Simulation is generated through the *cab* module. This module encapsulates the CAN bus protocol [CAN] used by almost every cab in the automotive industry to trigger the input and control devices (e.g. steering or accelerating). As an optional replacement for the cab module we developed a configurable DirectInput module that connects standard PC peripherals such as keyboard, mouse and joystick to the simulation.

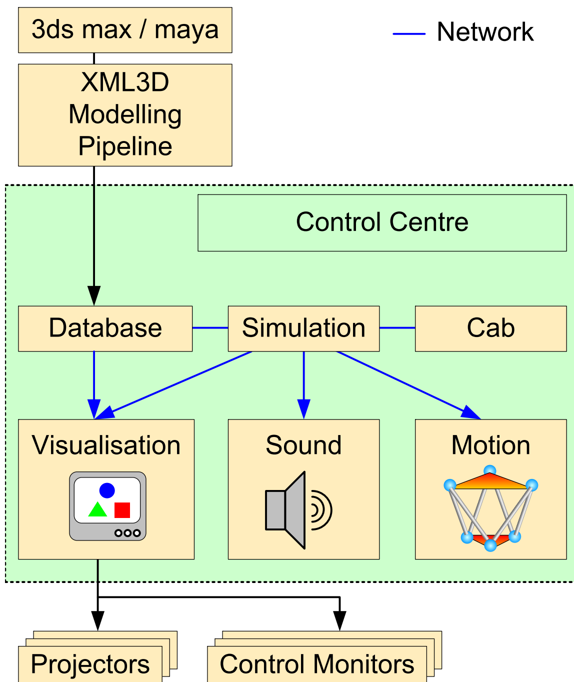


Figure 15: *Developed modules and their communication links.*

The heart of our VR-System (see figure 15), the *simulation* module, provides a real time calculation for the dynamic objects of the simulated machine with the help of a common equation solver (e.g. 4 wheels and 1 body for a simple car). The equations that describe the object-object and object-terrain interaction are obtained from a semi-automatic simplification of high fidelity physical models [KP04]. The *simulation* module generates output data for the three modules *visualisation*, *sound*, and *motion*. Our visualisation (figure 16) generally described in section 4.3 is implemented in C++ on top of an abstraction layer that translates all function calls into either the OpenGL or DirectX API. By this we can profit from the highly optimized DirectX drivers on the Windows Platform as well as from the possibility to run our visualisation on Linux also. This broadens the potential application base, which shows that platform independence in the design of each module is an important issue. Moreover our application runs as a desktop application with multiple monitors, projectors, or even a stereoscopic system (powerwall, CAVE). System state synchronisation issues typical in multi-node environments are solved by broadcasting the simulation data. Hence, using fast graphic hardware we can guarantee synchronized render output. A second advantage is that we can switch the underlying render core to evaluate new technologies that have not been released in both APIs at the same time. By using the described shader system we have decoupled the shader code from the application, a feature required by an automatic shader generation/LOD algorithm, making use of per pixel technologies like normal, specular, or shadow mapping. The pixel based software distortion algorithm we introduced in section 4.3.1 cannot be efficiently implemented on modern graphic hardware in a straight forward (say per pixel) way. So we apply the algorithm to a regular mesh of 50x50 squares in a vertex shader and apply the previously rendered visualisation as a texture to that mesh. To handle the small overlapping regions in multi projection environments we use an additional blending texture that defines the visibility of each pixel in the final rendering. It is applied for each colour channel in a pixel shader. The *sound* and *motion* modules are implemented straight forward. The first one mixes some recorded engine noise samples depending on the engine load, the second one is controlled by a set of parameters (velocities and accelerations) that are transmitted into the device driver of the motion platform.

By now the whole simulator system is used almost all of the time for research and development with the goal to have a VR-environment that matches the real world in the focused aspects as good as possible. At this point we refer the reader to the video (provided as additional material) "Interactive Automotive Simulator" showing the described VR-System in action.

6. Conclusions and Future Work

We demonstrated a new approach for designing and implementing a general interactive VR-System with special focus on extensibility. We identified abstract module types starting from the general system-user-loop and presented abstract methods for communication, system control, and visualisation. Additionally a detailed specification and description of these modules is given. Moreover, a hierarchical data pipeline for content creation is presented that integrates seamlessly into the prior defined system. To demonstrate the

© The Eurographics Association 2005.

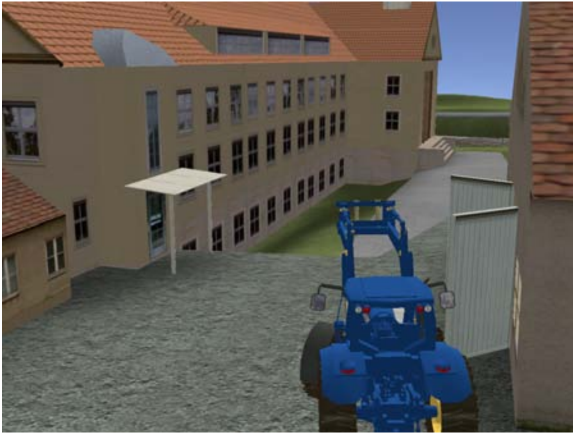


Figure 16: *Screen shot of the visualisation.*

practicability and flexibility of our approach, we outlined the implementation in the field of a interactive automotive simulator.

In order to provide support for a wider field of possible applications more hardware and software components have to be integrated into the general system in the future. Concerning hardware we want to integrate support for VR typical peripherals (tracking systems, data gloves, speech and gesture recognition etc.). We also scheduled the integration of new software technologies for data management with a database to simplify collection and reduction of data obtained in ergonomic studies. Another issue is the integration of a common physics engine for simulating rigid body objects. Finally a generalised sound module is planned that fulfils the requirements of a VR-System and supports EAX or Dolby Surround. Each extension of the system should be implemented as an separate module in order to profit from the easy to use extensibility capabilities of our approach.

References

[ATI] ATI TECHNOLOGIES: Rendermonkey™toolsuite. www.ati.com/developer/rendermonkey.

[ATM] ATM FORUM: Asynchronous transfer mode (ATM): A high speed backbone network technology. www.atmforum.com.

[ATT] ATT LABORATORIES IN CAMBRIDGE: VNC (virtual network computing) remote control protocol. www.realvnc.com.

[Ava] AVALON DEVELOPMENT GROUP: Avalon–an open X3D/VRML environment. www.zgdv.de/avalon/.

[Bea] BEAZLEY D. M.: Swig : An Easy to Use Tool For Integrating Scripting Languages with C and C++. Fourth Annual USENIX Tcl/Tk Workshop, 1996.

[BEK05] BIMBER O., EMMERLING A., KLEMMER T.: Embedded Entertainment with Smart Projectors. *IEEE Computer 38*, 1 (2005), 48–55.

[Bie00] BIERBAUM A. D.: *VR Juggler: A virtual platform for virtual reality application development*. Master's thesis, Iowa State University, 2000.

[Bil] BILL SHERMAN: FreeVR–an open-source virtual reality interface/integration library. www.freevr.org/.

[CAN] CAN IN AUTOMATION: CAN–Controller Area Network. www.can-cia.org.

[CCF*00] CHEN Y., CLARK D. W., FINKELSTEIN A., HOUSEL T. C., LI K.: Automatic alignment of high-resolution multi-projector display using an un-calibrated camera. In *VIS '00: Proceedings of the conference on Visualization '00* (2000), IEEE Computer Society Press, pp. 125–130.

[Cel] CELES W.: tolua–accessing C/C++ code from Lua. www.tecgraf.puc-rio.br/~celes/tolua.

[CO93a] C.CARLSSON, O.HAGSAND: DIVE A multi-user virtual reality system. In *Virtual Reality Annual International Symposium* (1993), IEEE, pp. 394–400.

[CO93b] C.CARLSSON, O.HAGSAND: DIVE A multi-user virtual reality system. *Computers and Graphics* (1993).

[CO96] C.CARLSSON, O.HAGSAND: Interactive multiuser VEs in the DIVE system. *Multimedia 3* (1996), 30–39.

[FGS99] FRECON E., GREENHALGH C., STENIUS M.: The DiveBone – an application-level network architecture for Internet-based CVEs. In *VRST '99: Proceedings of the ACM symposium on Virtual reality software and technology* (New York, NY, USA, 1999), ACM Press, pp. 58–65.

[Fre] FREDERIC DEVILLERS: OpenMASK–Empowering Virtual Reality Technology. www.openmask.org/.

[ISO99] ISO/IEC: Information technology – open distributed processing – interface definition language. www.iso.org, 1999.

[KASK02] KELSO J., ARSENAULT L., SATTERFIELD S., KRIZ R.: Diverse: A framework for building extensible and reconfigurable device independent virtual environments, 2002.

[KP04] KUNZE G., PENNDORF T.: Modulare Softwarearchitektur für die interaktive Simulation von Maschinen- und Fahrzeugsystemen in virtuellen Umgebungen (SARTURIS). *"Software Engineering 2006": BMBF, 2004* (2004).

[LJD97] LEIGH J., JOHNSON A., DEFANTI T.: CAVERN: A Distributed Architecture for Supporting Scalable Persistence and Interoperability in Collaborative Virtual Environments. *Virtual Reality: Research, Development and Applications 2.2*, December (1997), 217–237.

© The Eurographics Association 2005.

[Met] METZE J.: Entwicklung eines Visualisierungssystems für Virtual-Reality Simulationen. Grosser Beleg, TU Dresden, 2005.

[Myr] MYRICOM, INC.: High-performance packet-communication and switching technology for workstation clusters. www.myri.com.

[NBK*] NEIDHOLD B., BÜRGER R., KÖRNER D., FRANZKE O., RICHTER J., METZE J.: XML3D – an extensible 3d data format. www.inf.tu-dresden.de/~bn4/xml3d/wiki.

[PBL*94] PRATT D., BARHAM P., LOCKE J., ZYDA M., EASTMAN B., MOORE T., BIGGERS K., DOUGLASS R., JACOBSEN S., HOLLICK M., GRANIERI J., KO H., BADLER N.: Insertion of an articulated human into a networked virtual environment, 1994.

[RS01] REITMAYR G., SCHMALSTIEG D.: Opentracker-an open software architecture for reconfigurable tracking based on XML. In *VR* (2001), pp. 285–286.

[SFH*00] SCHMALSTIEG D., FUHRMANN A., HESINA G., ARI Z., ENCARNACAO L., GERVAUTZ M., PURGATHOFER W.: The Studierstube Augmented Reality Project, 2000.

[SKS02] SLOAN P.-P., KAUTZ J., SNYDER J.: Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2002), ACM Press, pp. 527–536.

[Sun88] SUN MICROSYSTEMS: RFC 1050 - RPC: Remote Procedure Call Protocol specification. www.faqs.org/rfcs/rfc1050.html, 1988.

[Tra99] TRAMBEREND H.: Avocado – a distributed virtual environment framework, 1999.

[Web] WEB3D CONSORTIUM: Open Standards XML-enabled 3D file format. www.web3d.org.

[WRR] WÖSSNER U., RANTZAU D., RAINER D.: Interactive simulation steering in VR and handling of large datasets.

[WSWL02] WÖSSNER U., SCHULZE J. P., WALZ S. P., LANG U.: Evaluation of a collaborative volume rendering application in a distributed virtual environment. In *EGVE '02: Proceedings of the workshop on Virtual environments 2002* (Aire-la-Ville, Switzerland, Switzerland, 2002), Eurographics Association, pp. 113–ff.

© The Eurographics Association 2005.