Towards a Scalable High Performance Application Platform for Immersive Virtual Environments

Matthias Bues, Roland Blach, Simon Stegmaier, Ulrich Häfner, Hilko Hoffmann,
Frank Haselberger

Competence Center Virtual Environments
Fraunhofer Institute for Industrial Engineering (IAO)
Nobelstr. 12, D-70569 Stuttgart, Germany
phone: +49-711-970-2232, fax: +49-711-970-2213
Matthias.Bues@iao.fhg.de
http://vr.iao.fhg.de

Abstract. Software for the development and generation of virtual environments used to run on specialized and expensive hardware. This situation has dramatically changed in the last two years. This paper describes a scalable high performance application platform for immersive environments using commodity hardware. Scalability will be obtained with a coarse cluster of specialized nodes, with emphasis on distributed real-time rendering. As data synchronization method a distributed shared memory approach is used. Dedicated to this hardware setup, a modular component oriented design is presented.

1 Introduction

Virtual Reality (VR), in our understanding, is an interface technology which allows direct multimodal interaction with dynamic and responsive computer generated or so-called virtual environments. Immersive virtual environments (VE's) should operate in real-time and consider the spatial superposition of user and data-environment, that is, the response time and update rate of the system is high enough that it generates an experience of continuity and a perceptible 3D-environment. This requires not only 3D-based real-time computing and render systems but also real-time spatial registration of the user and his behavior.

We see the main purpose of VR-technology in the enhancement of human computer interaction. Especially in problem domains of high complexity the use of immersive virtual environments enables more direct perception and manipulation. Obvious application domains are complex evaluation or planning tasks like architecture or design, medical training, fluid dynamics in engineering, assembly planning, etc..

One of the major obstacles for broader research and development was definitely the usage of specialized and expensive hardware which does not scale adequately. Also network bandwidth restricted the distribution of real-time systems.

Nevertheless there have been various approaches to distributed virtual environments. Common usage were multi-user environments like the DIS/HLA [Kuhl99] based military simulations which focus on large scale multi-user applications. Similar systems are described in [Singhal99] which gives a good overview of the state of the art. Another important domain was the coupling of super-computer-based simulations with immersive visualization systems. But it was not very common to use these distributed systems for single user purposes as an substitute for expensive high end graphic real-time systems.

This has changed dramatically in the last two years. The availability of an open source operating system on commodity hardware platforms with sufficient graphic power has started a serious approach to commodity hardware based VE research and development. On the one hand it promises more fine grained and cost effective scalability as with specialized workstations. On the other hand new bandwidth problems induced by the multi-purpose design of the hardware architecture require new creative solutions for real-time systems. This suggests a design of virtual environment system software closer dedicated to a commodity hardware based computer cluster.

Our target application domain is a single/few user system in a local distributed environment for the sake of pure performance scalability. We do not intend to create large scale networked virtual environments.

2 Conceptual Overview

The choice of the granularity of system components seems to be a crucial factor for the usability on the application development level, as we have experienced with the Lightning VR System [Blach98].

Our approach tries to bundle functionalities to coherent units of hard- and software. These units gain a certain autonomy and independency which makes the whole system more robust and manageable. To spend a complete hardware system for even simple but computationally intense tasks seems to be affordable because of the comparative cheapness of commodity hardware. To achieve this level of independence of the system components an asynchronous approach to communication is necessary.

For many applications, the most important scalability aspect is the scalability of graphics output, ranging from variable channel count of the display environment to scalability of overall graphics throughput, and parallelization of special rendering tasks. A distributed approach combining multiple graphics subsystems on multiple hosts is pursued to overcome the limitations of the PC architecture.

Other important abilities are rapid configuration, run-time access and prototyping of applications where the introduction of an interpreted scripting language has proven useful. Therefore two levels of the application developer interface should be considered:

- A C++ native object extension, to extend the system with a kind of plug-in structure. As long as the shared memory based data pool can be accessed other languages are also conceivable.

- A scripting layer for easy configuration and runtime access.

We will experiment further with a multi-language support. In real world applications obviously only one language will be reasonable. The language of choice must be able to build and maintain larger applications. Our experience with Tcl [Ousterhout93] has shown that it is a very convenient choice for small and medium size applications but seems not to be able to help structuring large scale applications.

3 An Early Prototype: Personal Immersion (Lightning 1.7 distributed)

Commodity hardware like the PC architecture implies some limitations in comparison to classic graphics workstations, namely concerning bus bandwidth, memory bandwidth, and multiprocessor capabilities. A PC-based scalable graphics architecture therefore requires a distributed approach.

A first prototype of a scalable distributed VE System is the Personal Immersion System. Based on the Linux port of Lightning 1.7, it is capable of driving single- or multi-wall immersive projection environments. It is based on a distributed scene graph/application approach. As a result, the network bandwidth needed for synchronization is kept at a minimum, independent of the amount of actual per-frame scene graph updates. Standard 100MBit Ethernet components are used for cluster interconnection. To keep the synchronization of the buffer swaps of each graphics channel as tight as possible, an additional light-weight synchronization using the handshake lines of a serial port is established. One node in the cluster acts as a master; it acquires tracking and input device data and propagates these data to the other hosts in the cluster. Usually, the master node also runs a full application/rendering task. If the data acquisition overhead is considerable, e.g. from computationally intensive calibration tasks, the master can also be fully dedicated to data acquisition and preprocessing.

In current configurations, passive stereo projection is used; a typical single-wall configuration runs on two nodes, each generating one eye view. Each node is equipped with one graphics card, allowing to exploit the immediate-mode throughput of modern PC graphics accelerators to the maximum extent possible. Fig. 1 shows a block diagram of this configuration.

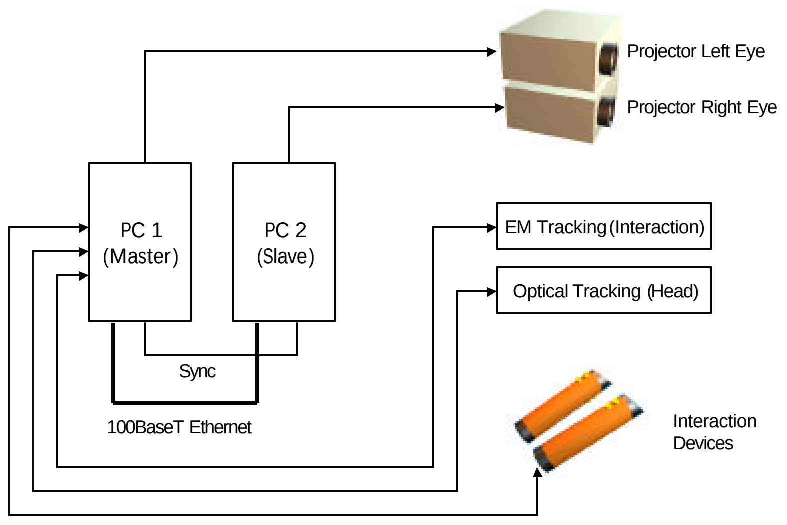

Fig. 1 Personal Immersion configuration

The Personal Immersion System was evaluated in various application environments [IPTW2000], proving its usability being comparable to traditional high-end configurations. Advantage of the approach pursued with Personal Immersion is its compatibility with existing Lightning Applications on high-end platforms. Since the full set of the underlying scene graph API can be used, even applications performing much run-time modification to the scene graph can be ported to the Personal Immersion platform with minimum effort. An application example are immersive surface or volume modeling systems.

Drawbacks are the replication of the computational tasks of the application on each image generator node, which is unnecessary in most cases, and the limited granularity in distribution of image generation and application-side computational tasks.

4 A Scalable Personal Immersion System

4.1 Interactive Core System

The interactive core system is able to model and therefore abstracts all components and the communication behavior of immersive interactive applications in virtual environments. It is in itself a complete system for the definition and description of the interactive behavior of such environments. The specific device or render modules should be as loosely coupled as possible.

The main goal is to achieve a real-time behavior of the application control module with a timing being independent of I/O specific limitations, namely graphics throughput.

Consider the situation that an independent device process is used and that this process is fast enough to register all input events sent by any device subsystem, e.g. a spatial tracking device. A commonly used scene graph API, OpenGL|Performer^{TM}, uses a pipelined multiprocessing model consisting of the stages application, cull and draw, each running in a separate process in the full multiprocessing configuration [Rohlf94]. However, theses processes are synchronized at frame boundaries, resulting in the frame rate of the entire system being limited by the process with the longest frame time. Preprocessing of interaction data and application control is performed in the application stage. In practice, this means that in case of heavy graphics load, a double click event can easily be lost. There are two main directions to tackle this problem. A history buffer for input streams can be introduced, from which all events accumulated during the last frame are processed in the current frame. Major drawbacks of this solution are the immanent overflow problem, and the application process frame time still limiting system frame rate. The second way, which is our approach, is to run the application control loop asynchronously from the renderer main loop. [Grimsdale 91] describes a similar method. The application control loop processes the inputs and control related functions in real-time and possible render modules/devices sample the specific system state at a certain time and processes it as fast as possible. This can probably be applied to other function modules, e.g. collision detection or physical simulation.

Although custom solutions for most load balancing problems as described above can be found, these solutions do not apply for the general case.

Concerning control and data flow, we extend the data flow model of the VR system Lightning [Blach98]. So-called Application-Objects hold the main functionality. The state of the system is completely contained in so-called fields which are actually attributes of the Application-Objects. These fields can be linked to other fields or procedurally set from elsewhere. The structure of the linked objects and their behavior describe the application. Every object has a central update function which reads incoming data, processes it and updates the object state. This is similar to other data flow oriented systems as e.g. [VRML97]. We introduced container objects which control other objects and have the possibility to run their own autonomous simulation loop via threading. One key container object type has the role of the update manager of objects it owns. There is no central hidden update manager as in the VR system Lightning because we found that in most applications, the sequence of the data propagation is not only manageable by the application developer but wanted. Another important container object type is the renderer which abstracts the various hardware and library specific renderer.

The Object Database Pool resides completely in shared memory. Unrelated processes can easily access this database. This offers the opportunity to modularize the application components on binary level which makes them easier to develop, control and maintain. The communication protocol is string based and timestamps are used for synchronization. Reference counting is the only security mechanism which allows to control the usage of the database objects. The synchronization of object state across host boundaries relies on mirroring the shared data pool via a distributed shared memory approach which can use various networking technologies for transportation, as described in section 4.3.

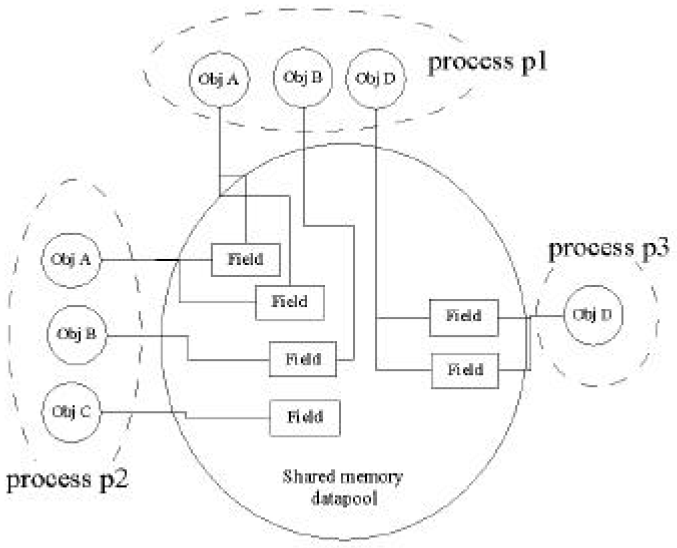

Fig. 2 Shared memory data pool as central repository of system state

The scripting and configuration system consists up to now of four different language bindings Tcl/iTcl, Java, Javascript and Python. Which one will be the language of choice for productive application development has to be proven in the near future. The interpreter is also an object which runs in its own thread where commands can easily be sent from external processes. The usage of various interpreters in the same application is conceivable but does not seem reasonable to us.

4.2 Graphics/IG System

On the graphics/image generation side, a coarse-grained scaling approach similar to the granularity of Personal Immersion is pursued in a first approach. This means one single node and graphics pipe per output channel; of course, multiple channels per node are also possible if the respective node provides appropriate hardware resources. The scene graph is completely replicated on all nodes; scene graph updates are propagated on a per-frame basis. This approach can easily be implemented using the distributed object database pool as described above, running multiple visual renderers on the various nodes, each registering interest in the visual objects in the shared data pool. The multiple visual renderers can be synchronized at frame boundaries to drive stereoscopic and multi-channel displays.

To overcome the limitations of this distribution approach, a more fine-grained distribution of image generation tasks is pursued in parallel, based on tiling a single output channel and recombining the final image. An network protocol independent of a particular Scene Graph API is designed to distribute scene graph updates across the visualization cluster. This allows the rendering hosts to run on various operating system platforms. For combining the final output image, commodity graphics hardware allows two approaches of accessing frame buffer data. First one is reading back the framebuffer through the host interface of the graphics card and conveying it via a high speed network to the host being in charge of output image reassembly. Samanta et al. [Samanta00] have shown the feasibility of such an approach. The second way is to use the video or DVI output of the graphics card, using custom hardware to reassemble the final image. An example of this is described by Humphreys et al. [Humphreys2001]. Such solutions have the huge advantage of yielding the entire system bus and networking bandwidth for other tasks; however, the need for custom hardware contradicts to the commodity hardware paradigm of our approach. This is the main reason for us to prefer the a system bus/networking based architecture.

4.3 Distributed shared memory

A flexible and high-performing distribution and shared memory model is a key building block of our system.

Interprocess-communication is simple and fast as long as the communicating processes have access to the same physical memory. Distributed shared memory

(DSM) implementations aim at providing the same simplicity when communicating between processes on different nodes with no shared physical memory.

Page-based DSM implementations extend the capabilities of the well known shared memory mechanisms available in most of today's operating systems to inter-node communication. That means a DSM segment is as usually first created with an API call. Every statement of the programming language can then be used to manipulate the segment's data. The modifications made by a process become visible to other processes, which may - in contrast to ordinary shared memory - even reside on a different node.

Most page-based DSM implementations use a pull scheme: A process works on the local data as long as possible, requesting data from remote nodes only when needed. Major drawback of this technique is that if multiple processes are interested in the newly created data of another process, the data has to be transferred repeatedly to all these nodes; no multicasting takes place. This algorithm provides the same strict consistency as ordinary shared memory. But this strict consistency isn't needed. The so-called "release consistency" is sufficient and allows multicasting. Release consistency has already been implemented in the shared variable DSM-system *Munin* [Bennet90].

Communication using shared memory requires synchronization. This means that prior to any modification on a shared data object, a lock is set on this object. This lock is released after modification. Propagation of the modifications is postponed until the release operation is completed. This allows to accumulate the modifications made between a pair of synchronization operations and multicast them together in one large chunk via multicasting to all nodes sharing the respective object.

This approach may perform a little bit worse compared to common implementations of DSM that delay the transmission until the data is really needed when communication only takes place between two nodes but should perform much better when applied to an environment like ours, where interest of multiple nodes to a particular data object is a common case.

By combining usual UNIX shared memory and the techniques described it is possible to achieve latencies nearly as low as and throughput nearly as high as those using the UNIX shared memory API system calls. The implementation therefore may well eliminate the need for using this API as an application programmer. This is – indeed - the very goal of our efforts.

For inter-node communication, both latency and throughput will suffer considerably when using off-the-shelf networking hardware. Nevertheless we intend to support Ethernet networks to provide a low-cost communication layer for applications with modest requirements. For applications with high bandwidth and tight synchronization requirements, we need something "with a little more kick". We have chosen SCI, the

Scalable Coherent Interface, for this purpose. SCI not only provides extremely low latencies of a few microseconds and a throughput of more than 100 MBytes/,) but also has the advantage of implementing DSM directly in hardware. Using SCI, the main task in implementing the DSM is therefore reduced to the design of an abstraction layer and the implementation of a DSM mechanism for low-cost networking hardware.

4.4 Implementation details

The current core system prototype runs on x86-based Linux Systems. For special purposes (e.g. image generation or audio) the usage of other hardware-software combinations are likely. The basic node configuration is

- AMD Athlon at 1 GHz
- VIA KT133 chipset.
- NVidia GeForce2 or Quadro gfx card
- Dolphin DC330 SCI card, Dolphin Interconnect

The current implementation of the visual renderer uses OpenGL|Performer^{TM} from SGI.

5 Conclusion and future work

Our approach has lead to a scalable, simple and lightweight system for interactive applications in immersive virtual environments. The core system is small and has the freedom for maximal extensibility. Nearly all components can easily be exchanged because of the quasi-autonomous approach of the communication.

First prototypes are already implemented and the further modularization has proven already very helpful for the development process, especially control user interfaces, which are not part of the immersive simulation loop.

The next steps are the evaluation of the adequate scripting language and extensive testing and performance evaluation, especially in real world application contexts. Here we hope to gain insight how to distribute and modularize application components on the cluster. Do we need specialized nodes and which are these? How complex can such systems and applications grow to stay manageable?

All these topics will be questions for the next stage of the virtual environment research - the personal immersion stage

6 References

[Bennet90] Bennet, J.K, Carter, J.B., and Zwaenepoel, W., Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In: Proc. Second ACM Symp. on Principles and Practice of Parallel Programming, ACM 1990, pp. 168-176

[Blach98] Blach, R.; Landauer J., M.; Simon, A.; Rösch, A.: A Flexible prototyping Tool for 3D Real-Time User-Interaction. In: Virtual Environments 98, Proceedings of the Eurographics Workshop, 1998 pp. 195-203

[Grimsdale 91] Grimsdale, C. (1991). dVS-Distributed Virtual Environment System. In Proceedings of Computer Graphics, Computer Animation, Virtual Reality, Visualisation, (pp. 163-170): Blenheim Online.

[Humphreys01] Humphreys, G., et al.: WireGL – a Scalable Graphics System for Clusters. Submitted to SIGGRAPH ’01.

[Kuhl99] Frederick Kuhl, Richard Weatherly, and Judith Dahmann, Creating Computer Simulation Systems - An Introduction to the High Level Architecture, Prentice Hall, Upper Saddle River, NJ. 1999.

[Ousterhout93] Ousterhout, J., Tcl and the Tk Toolkit, Addison-Wesley, Reading, Massachusetts, 1993

[Rohlf94] Rohlf, J., Helman, J.: IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics, Proceedings of ACM SIGGRAPH ’94, Orlando, FL, pp. 381-394.

[Samanta00] Samanta, R., et al.: Hybrid Sort-First and Sort-Last Parallel Rendering with a Cluster of PC’s. SIGGRAPH/Eurographics Workshop on Graphics Hardware, Interlaken, Switzerland - August, 2000

[Singhal99] Singhal, S.; Zyda M.: Networked Virtual Environment – Design and Implementation, Addison Wesley acm Press, 1999

[VRML97] The Virtual Reality Modelling Language (VRML) Specification 2.0 ISO/IEC CD 14772, 1997