Die approbierte Originalversion dieser Dissertation ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).



TECHNISCHE UNIVERSITÄT WIEN

DISSERTATION

Distributed Collaborative Augmented Reality

ausgeführt zum Zwecke der Erlangung des akademischen Grades
eines Doktors der technischen Wissenschaften unter der Anleitung von

A.o. Prof. Dipl.-Ing. Dr. Michael Gervautz
Institut Nr. 186
Institut für Computergraphik
und Univ.-Ass. Dipl.-Ing. Dr.techn. Dieter Schmalstieg
als betreuendem Assistenten

eingereicht an der Technischen Universität Wien
Fakultät für Technische Naturwissenschaften und Informatik

von

Dipl.-Ing. Gerd Hesina

Abstract

Augmented Reality is the mixture of computer graphics and text with real world and/or video images. This thesis presents improvements for collaborative augmented reality environments. We present a toolkit, which uses a distributed shared scene graph approach to enable transparent distribution at the lowest level. Application migration is used to distribute computational load within the system. We demonstrate how migration can be applied and used for load balancing, dynamic workgroup management, remote collaboration, and even "ubiquitous computing". The resulting system is a complex distributed collaborative augmented reality work environment, which is used to address the question of how to use three-dimensional interaction and new media in a general work environment, where a variety of tasks are carried out simultaneously by several users. The implementation was done in the *Studierstube* system, a collaborative augmented reality environment. At its core, the system uses collaborative augmented reality to incorporate true 3D interaction into a productivity environment. This concept is extended to include multiple users, multiple host platforms, multiple display types, multiple concurrent applications, and a multi-context (i. e., 3D document) interface – into a heterogeneous distributed environment. All this happens almost totally transparent to the application programmer.

Kurzfassung (German Abstract)

Augmented Reality ist die Überlagerung von Computergraphik und Text mit der wirklichen Umgebung und/oder Videobildern. Diese Dissertation behandelt Verbesserungen für kollaborative Augemented Reality (AR) Umgebungen. Wir stellen ein Werkzeug vor, dass einen verteilten, mehrfach benutzten Szenegraphen, als Ansatz verwendet, um eine transparente Verteilung auf der untersten Ebene zu ermöglichen. Application migration wird verwendet um Berechnungen innerhalb des Systems aufzuteilen, eine dynamische Anwenderverwaltung zu erreichen, Kollaboration zwischen weit entfernten Anwendern zu ermöglichen und um "ubiquitous computing" zu unterstützen. Das resultierende System ist eine komplexe kollaborative AR Arbeitsumgebung, welche dazu verwendet wird, um die Frage, wie man dreidimensionale Interaktion in einer üblichen Arbeitsumgebung, in der verschiedenste Aufgabenstellungen gleichzeitig durch mehrere Benutzer bearbeitet werden, zu behandeln. Die Implementierung wurde im *Studierstube* System ausgeführt, welches eine kollaborative Augemented Reality Umbgebung ist. Als Kern verwendet das System kollaborative AR um echte 3D Interaktion in die Arbeitsumgebung einzubinden. Dieses Konzept wird um jeweils multiple Benutzer, Computer Plattformen, Anzeigegeräte, parallele Applikationen und Kontextschnittstellen (d.h. 3D Dokumente), zu einer heterogenen verteilten Umgebung erweitert. All dies geschieht beinahe völlig transparent für den Applikationsprogrammierer.

Acknowledgments

While it will be impossible to thank all those who have contributed in some way to this work, there are certain folks who must be acknowledged. First and foremost are the members of the Institute of Computer Graphics and Algorithms at Vienna University of Technology, where this work was undertaken, especially my advisors Dieter Schmalstieg and Michael Gervautz. Thanks to Werner Purgathofer for giving me a chance to work at the institute. Numerous others at the institute influenced this work over the years, especially Anton Fuhrmann.

Thanks to Markus Krutz, Rainer Splechtna, Hermann Wurnig and Andreas Zajic for their implementation work. Special thanks to Meister Eduard Gröller for spiritual support throughout the whole project.

I would like to thank my family and friends who have encouraged and supported me over the many years spent on this and previous work. My family has never wavered in their support, or in their conviction that I could and would one day complete this dissertation. Especially I would like to thank Gudrun for endless patience and love and my little daughter Alexandra for giving me a new reason to live.

This work was supported by the Austrian Science Foundation (FWF) under project no. P-12074.

Mit einer Weisheit,
die keine Träne kennt,
mit einer Philosophie,
die nicht zu lachen versteht,
und einer Größe,
die sich nicht vor Kindern verneigt,
will ich nichts zu tun haben.
— Khalil Gibran

Table of Contents

1 INTRODUCTION 8

2 STRUCTURE OF THE THESIS 11

3 RELATED WORK 13

3.1 FUNDAMENTALS OF NETWORKING FOR VIRTUAL ENVIRONEMNTS 15
3.2 DISTRIBUTED SYSTEMS EXAMPLES 22
3.3 SHARED SCENE GRAPHS 31
3.4 COLLABORATIVE AUGMENTED REALITY 33

4 STUDIERSTUBE OVERVIEW 35

4.1 BACKGROUND 35
4.2 DISTRIBUTED STUDIERSTUBE 37
4.3 SUMMARY 41

5 DISTRIBUTED OPEN INVENTOR 42

5.1 INTRODUCTION 42
5.2 DISTRIBUTED SHARED SCENE GRAPH 43
5.3 REPLICATED SCENE GRAPH PROTOCOL 45
5.4 LOCAL VARIATIONS 47
5.5 NETWORKING 49
5.6 APPLICATION IN OUR AUGMENTED REALITY ENVIRONMENT 50
5.7 IMPLEMENTATION 52
5.8 RESULTS 55
5.9 DISTRIBUTED OPEN INVENTOR, VERSION 2 57
5.10 SUMMARY 58

6 BRIDGING MULTIPLE USER INTERFACES WITH AUGMENTED REALITY 59

6.1 INTRODUCTION 59
6.2 MULTIPLE USERS 59
6.3 MULTIPLE CONTEXTS 60
6.4 MULTIPLE LOCALES 61
6.5 INTERACTION DESIGN 63
6.6 IMPLEMENTATION 65
6.7 RESULTS 72
6.8 SUMMARY 77

7 CONTEXT MIGRATION 79

7.1 INTRODUCTION 79
7.2 CONTEXTS AND MIGRATION 79

7.3 ACTIVATION MIGRATION 81
7.4 APPLICATION MIGRATION 82
7.5 USAGE OF MIGRATION 82
7.6 RESULTS 87
7.7 SUMMARY 89

8 CONCLUSIONS AND FUTURE WORK 90

8.1 DISCUSSION 91

9 REFERENCES 94

10 APPENDIX 102

1 Introduction

Technical progress in recent years gives reason to believe that virtual reality (VR) has a good potential as a user interface of the future. At the moment, VR applications are usually tailored to the needs of a very specific domain, such as a theme park ride or a virtual mock-up for design inspection. We believe that augmented reality (AR), which is the mixture of computer graphics and text with real and/or video images, and sometimes called the less obtrusive cousin of VR, has a better chance to become a viable user interface for everyday productivity applications, where a large variety of tasks has to be covered by a single system.

This work has mainly been motivated by the fact that current AR and VR environments support only a limited number of users and a fixed number of applications. The investigation of such systems revealed more limitations and problems. Display devices are not interchangeable or able to be mixed and many systems lack a powerful interaction metaphor. Furthermore we did not find systems which utilize the power of multiple user interface dimensions like multiple users, multiple concurrent applications, multiple display devices, multiple host computers, and multiple operating systems.

To address the aforementioned problems, this thesis proposes new techniques and tools to build truly distributed systems that allow for larger collaborative workgroups in augmented reality. The proposed solution runs software on a network of graphics workstations - one for each user and allows different display technologies. A sophisticated networking set-up provides minimal latency and low bandwidth requirements so that the quality of the shared experience is not adversely affected. This solution is scalable: extension to any reasonable number of users can be done by adding another module without system modifications.

The presented techniques are not limited to AR/VR environments. Many systems and applications, like networked games, computer supported cooperative work systems, web-based applications, distributed mobile systems and even distributed databases could take advantage of the proposed approaches because those systems have often the same (or at least a portion of the) aforementioned problems.

This thesis focuses on problems, which arise if non-distributed systems are turned into distributed ones (e.g. to improve scalability, or to enable a larger user-base). Furthermore it tries to find new tools and techniques to interact with distributed systems.

We developed *Studierstube*, which is a research AR environment. *Studierstube* is the study room where Goethe's famous character, Faust, tries to acquire knowledge and enlightenment. We chose this term as the working title for our efforts to develop user interfaces for future work environments. In the Studierstube project we try to address the question of how to use three-dimensional interaction and new media in a general work environment, where a variety of tasks are carried out simultaneously. In essence, we are searching for a 3D interaction metaphor as powerful as the desktop metaphor for 2D.

The original *Studierstube* architecture (Schmalstieg et al., 1996) that has been developed makes use of a relatively powerful workstation with multiple graphics outputs. Hence only a limited number of users, which wear head mounted displays (HMD) is supported. While this approach allows the construction of a multi-user system with little overhead, it is not truly scalable, i.e. more than a certain number of users can definitely not be supported due to hardware limitations. Furthermore collaborators are forced to be in the same room and are limited to HMD displays.

In order to research, develop and verify the new techniques and tools, we have chosen to enhance the *Studierstube* system. We also propose to extend our design to connect multiple remote *Studierstube* sites with each other for combined local/tele-collaboration. Local users may collaborate in augmented reality at the local site, and simultaneously interact with another user group at a site anywhere on the Internet. For example, *Studierstube* could be used as a utility for exploring scientific visualisation systems. Several groups are able to share their workspace within our environment. The resulting architecture enables advanced features for collaborative virtual environments, allowing multiple concurrent applications as well as multiple users - both local and remote.

Furthermore we introduce light weight application migration to be able to shift computational load of replicated applications from one host to another, while application migration streams live applications from host to host in a way that is transparent to the application programmer and user(s). We demonstrate how these tools can be applied for load balancing, dynamic

workgroup management, remote collaboration, and even *ubiquitous computing* (Weiser, 1991). This thesis presents our approaches to turn *Studierstube* into a distributed system.

2 Structure of the Thesis

We introduce related work in chapter 3. Chapter 4 describes basic concepts of our distributed collaborative augmented reality system *Studierstube* and chapter 5 presents the major building block, which enables distribution of the system. In chapter 6 we have a close look at the whole system and describe new techniques, which are used to bridge multiple user-interface dimensions. New tools for application handling are introduced in chapter 7, and chapter 8 concludes this thesis.

This thesis contains material previously published in:

G. Hesina, D. Schmalstieg, A. Fuhrmann, W. Purgathofer: Distributed Open Inventor: A Practical Approach to Distributed 3D Graphics, Proc. VRST '99, London, pp. 74-81, Dec. 1999.

D. Schmalstieg, A. Fuhrmann, G. Hesina: Bridging Multiple User Interface Dimensions with Augmented Reality, Proceedings of the 3rd International Symposium on Augmented Reality (ISAR 2000), pp. 20-30, Munich, Germany, Oct. 5-6, 2000.

D. Schmalstieg, A. Fuhrmann, G. Hesina., Zs. Szalavári, L. M. Encarnação, M. Gervautz, W. Purgathofer: The Studierstube Augmented Reality Project. To appear in: Augmented Reality: The Interface is Everywhere, SIGGRAPH 2001 Course Notes, Los Angeles CA, USA, ACM Press, August 2001.

D. Schmalstieg, G. Hesina: Converging User Interface Paradigms Using Collaborative Augmented Reality, to appear in Proc. of 9th International Conference on Human-Computer Interaction 2001 (HCII), New Orleans, USA, August 5-10, 2001.

D. Schmalstieg, G. Hesina: Application Migration for Virtual Work Environments, available as Technical Report, submitted to UIST 2001, Orlando, Florida, November 11-14, 2001.

During the development of the thesis the following additional papers have been published, which are partially relevant to the work presented here:

G. Hesina, D. Schmalstieg: A Network Architecture for Remote Rendering, Proceedings of Second International Workshop on Distributed Interactive Simulation and Real-Time Applications, pp. 88-91, Montreal, Canada, July 1998. IEEE Computer Society.

F. Faure, C. Faisstnauer, G. Hesina, A. Aubel, M. Escher, F. Labrosse, J. Nebel, J. Gascuel: Collaborative Animation over the Network, IEEE Proceedings of Computer Animation 1999 (CA'99), 26-28 May, Geneva, Switzerland, 1999.

A. Fuhrmann, G. Hesina, F. Faure, M. Gervautz: Occlusion in collaborative augmented environments, Proceedings of the 5th EUROGRAPHICS Workshop on Virtual Environments (EGVE 1999), Vienna, June 1-2, 1999. Extended version appeared in: Computers & Graphics 23(6), pp. 809-819, 1999.

3 Related Work

The main attraction of Computer Graphics during the last years was the creation of interactive three-dimensional applications, since they provide the basis for other scientific disciplines and the entertainment industry. This interest in interactive graphics applications started just a few years ago, because earlier the hardware was not capable to deal with the huge amount of data to be processed in realtime. But in the last years even machines in lower price ranges are able to render three-dimensional scenes in realtime and graphics standards made it possible for vendors to produce hardware solutions as commodity items.

Now three-dimensional real-time rendering is a fast developing section of computer graphics, like computer science itself. In the beginning were three-dimensional objects, which could be rendered in a *viewer* and be moved with the mouse. But this solution is only satisfying, when one just wants to view a single object and not a whole scene. Therefore more realism and power was added to this model. That is, one does not just watch a scene from an outside position, but becomes more and more immersed in the world itself. The hardware used for interaction became more and more complex as well. To get a realistic feeling of being integrated in the virtual world, devices like the head mounted display (HMD) were introduced and made a high impression on the users. These output devices are still not standard equipment of every computer for economic and ergonomic reasons, but in the future this may change.

Semantic structuring of the scenes was introduced to allow architectural walkthroughs, city walkthroughs, fly-overs and more. In such applications, it is interesting to allow more than just one participant in such an environment. To make so-called multi-user virtual reality (VR) possible it is necessary to distribute the whole system over a network. This reveals the true power of the system, since the number of the participants is no longer restricted and they are able to collaborate.

The idea to support collaboration of human users lead in two directions: remote collaboration (Bryson, 1993) and local collaborative virtual environments where users join a world and can interact and communicate in a natural way. In the latter category two very successful approaches have been developed: The CAVE (Cruz-Neira, 1993) and the workbench

(Responsive Workbench, (Krüger et al., 1995), Virtual Workbench (Obeysekare et al., 1996), Virtual Table (Encarnação et al., 1999)).

CAVE (Computer Automated Virtual Environment, see Figure 1) is a scientific data visualization system which projects stereoscopic images on the walls of a room. The observer needs to wear LCD shutter glasses. This approach assures superior quality and resolution of viewed images and wider field of view in comparison to HMDs. The CAVE is essentially a five-sided cube. The participant stands in the middle of the cube, and images are projected onto the walls in front, above, below, and on both sides (left, right) of the participant.



Figure 1: This figure shows a CAVE system, which has been used at the SIGGRAPH 1998 conference.

The Workbench (see Figure 2) is essentially a table on which computer generated images are projected resulting in a typical set-up used by e.g. surgeons, engineers and architects. The resource requirements are less demanding than those of the CAVE and the horizontal workspace is very useful for manipulation with hand-held tools. Both systems suffer from the drawback that true stereoscopic images can only be rendered for one “master” user wearing the head tracker - users have to remain close to the master because distortions increase proportional to their distance to the tracked point of view. Applications in which users surround an object won’t work in the CAVE and are only possible for two participants in an enhanced version of the workbench (Agrawala et al., 1997).



Figure 2: Two users simultaneously view a shared virtual environment on the Responsive Workbench. Note that the image on the Workbench is rendered for the point of view of the camera.

In order to describe distributed virtual environments (DVE) we need to introduce frequently used terms and discuss some networking issues.

3.1 Fundamentals of networking for virtual environemnts

Building a distributed virtual environment implies that data must be transmitted over a network. As mentioned in the previous section it should be possible to support many users, which can interact in a virtual world. Hence the system should be scaleable. That is, it should be possible to support a larger number of users and adding new users should not cause a redesign of the whole system.

Virtual environments should be *responsive*. Users should be able to interact in such an environment in (nearly) real-time. Therefore every time consuming function must be optimized to achieve the best performance. Networking in every form is very time consuming and therefore a problem, but careful design and optimizations in transmission functions are used to keep the required time low.

If needed data is not available the whole system may stall, but if all data is delivered just in time no stalling caused by the network should occur. To achieve this, problems caused by the network must be analyzed. The following introduces frequently used terms within networked virtual environments.

Objects

This term is often used to describe properties of a virtual world. The definition ranges from C++ objects to a description as parameters, that could define position, orientation, acceleration, color, texture, surfaces, topology, temporality and some other parameters. Sometimes objects are called *entities* in virtual worlds.

Actor, Avatar, Scene, Scenegraph

A particular type of object is called actor. It has the ability to interact with other objects in an environment. An avatar is the (geometric) representation of a user. If a virtual environment provides the functionality of avatars then it is possible that users of this environment can see each other via avatars. A scene is a geometric description of many objects (e.g. a virtual world). The hierarchical structuring of a scene (and therefore its objects) is often based on a graph structure, which is called scenegraph.

Latency

Latency is the time measured from the setting of input until corresponding output is manifested. Sending messages over a network introduces latency and this conflicts with needed concurrent execution. The latency is not mainly introduced due to physical limitations of a network. It is rather a cause of network and software issues. For example, a message that is sent over a network may pass many network interfaces that process this message and convert it into an individual recognizable format. The fastest network is useless if the processor that read messages is slow.

Limited Bandwidth

Bandwidth is the capacity that a telecommunications medium has for carrying data. For voice communication (e.g., telephone), bandwidth is measured in the difference between the upper and lower transmission frequencies expressed in cycles per second, or hertz (Hz). For digital communication, bandwidth and transmission speed are usually treated as synonyms and measured in bits per second. Bandwidth is limited by physical limitations of the communication medium and the speed of used processors.

Unpredictable Bandwidth

The actual speed or transmission time of a message from source to destination depends on a number of factors. For example, World Wide Web transmissions travel at very high speed on fiber optic lines most of the way but lower bandwidths on local loops at both ends, and server processing time add to the overall transmission time. Therefore bandwidth is said to be unpredictable. It is possible that a link between a source and destination computer breaks down. Such events are always unpredictable.

Scalability

Scalability of a system does not only mean that it should function well in the rescaled situation, but it should actually take full advantage of it. For example, an application program would be scalable if it could be moved from a smaller to a larger operating system and take full advantage of the larger operating system in terms of performance (user response time and so forth) and a larger number of users that could be handled.

Network Topology

A topology describes the configuration or arrangement of a (usually conceptual) network, including its nodes and connecting lines. It describes the relation of computers within a network and defines a routing algorithm, which is used to transmit packets. By choosing a certain network topology, the system can exploit the inherent benefits such as prevention of cycles. It is important to use a network topology that fulfills quality of service requirements like latency. Another important reason to choose network topology carefully is scalability.

Bridge

In a network, a bridge is a hardware device or software that copies a data-link (physical network level) packet from one network to the next network along the communications path. For example, two local area networks (LANs) might be interconnected with a bridge, a connecting wide area network (WAN) link, and a bridge at the other end.

Router

On the Internet, a router is a device or, in some cases, software in a computer, that directs information packets to the next point toward their destination. The router is connected to at least two networks and decides which way to send each information packet based on its current understanding of the state of the networks it is connected to. A router creates or

maintains a table of the available routes and their conditions and uses this information along with distance and cost algorithms to determine the best route for a given packet. Typically, a packet may travel through a number of routers before arriving at its destination.

Unicast: 1 to 1

A UNICAST packet is addressed to a particular, single network address. Only the recipient will recognize this packet since its network interface knows about its own particular address. All the other stations on the subnet will not read this packet since the packet destination address differs from their own addresses.

Unicasts from one subnet to another one cross bridges transparently. Since the bridges know about the network topology (i.e. where the source and destination are located), only the segments that have to support the traffic will forward the packet. Routers only forward unicast packets.

Unicast network topologies do not scale well because a single message is sent to all other nodes. So the message transfer has $O(N^2)$ complexity.

Broadcast: 1 to all

It is possible to send broadcast packets on a network if, and only if the network supports broadcasting. Broadcast packets can place a high load on a network since they force every host on the network to service them. This transmission technique is typically used for two reasons: it is desired to find a resource on a local network without prior knowledge of its adress, or important functions such as routing require that information be sent to all accessible neighbors. Broadcast messages are used to send updates to all other nodes at once. The O(N) messages transferred over the network must be processed from all other nodes.

Multicast: 1 to many

Multicast is a form of data transmission that facilitates transmission from one point to many points more or less simultaneously. It is, for example, used for programming on the MBONE, a system that allows users at high-bandwidth points on the Internet to receive live video and sound programming. In addition to using a specific high-bandwidth subset of the Internet, Mbone multicast also uses a multicast protocol that allows signals to be encapsulated as TCP

packets when passing through parts of the Internet that can not handle the multicast protocol directly.

With multicasting it is possible to send messages to a subset of participating workstations. The basic idea behind multicasting in virtual environments is to map entity properties to multicast groups and send entity updates only to relevant groups. Therefore it is possible to define a multicast group in which a client can be seen by others. A client sends information updates only to the relevant group. Furthermore clients listen only to “interesting” groups, that is, only groups that can be seen.

A MULTICAST packet is addressed to a subset of nodes on a subnet. The destination address is particular to the group of systems it wants to reach: this is called a multicast group. Modern network interfaces are only listening to the groups the system should listen to (requested by the applications on the node). Unfortunately, most network interfaces listen to all multicast packets and the application has to decide whether each multicast packet is interesting for it or not.

Bridges forward multicasts and since they cannot know where the potential destinations are located, the multicast packets are sent to all interfaces. This behavior is called flooding because all segments see all packets. Multicasts do not cross routers unless they are routed using a special multicast routing protocol.

Tunneling

Many routers do not support IP multicast routing yet. Therefore multicast packets often travel over long distances across so called “tunnels”. A tunnel is a unicast (point-to-point) virtual link that may cross several bridges and routers. Tunnel endpoints can be routers supporting multicast routing, or workstations running special daemons to emulate multicast routing. These systems listen to multicast packets, encapsulate them into unicast packets, and send them over one or more tunnels. Each remote tunnel endpoint can then send the encapsulated multicasts over other tunnels, or restore the multicast packets and put them on their local LAN.

MBONE

MBONE stands for Multicast backBONE. This is a world-wide virtual network defined over the Internet to support routing of multicast packets. These packets are carrying mainly audio and video data from video-conferences over the net. MBONE is a virtual infrastructure of tunnels linking networks that can directly support IP multicast.

Being an overlay network of the Internet, MBONE is a shared medium providing a limited bandwidth. People sending data over the MBONE are supposed to follow the MBONE etiquette. In particular, a video session must consume less than 128 kbit/s.

To avoid saturating the Internet, the MBONE network has its own rate limiting system. This is a static mechanism implemented on MBONE routers which drops packets when the global throughput over a given tunnel might exceed a predefined value. Therefore the MBONE cannot support more than 2 or 3 parallel international broadcasts without dropping packets.

Client-Server network topology

Clients and servers play special roles. A client serves the end user directly and requests info from a server, which offers the info and responds to requests. Servers should be reliable. That is, requests must be served and delivered. However, this means not that client-server communication must be reliable. Sometimes servers must be fault tolerant. This can be achieved by either using recovery mechanisms or to use backup servers (e.g., database servers).

In such systems there are no direct client-client connections. Messages are sent to servers that route them to other clients and/or servers in the distributed simulation system. The main advantage is that it is possible to cull, augment or modify messages at a server before propagating them to other clients. This model of a distributed virtual environment minimizes the network traffic.

Hybrid topology

A hybrid network topology consists usually of peer-to-peer and client-server communication. That is, a host is able to transmit data to some hosts directly and to other ones through a central server. For example a LAN-based client will typically communicate with other hosts on the LAN in a peer-to-peer fashion but would communicate with a low-bandwidth

connected host (e.g. modem connection) through a server that can aggregate and compress packets for delivery.

Connection types

Two types of data transport mechanisms can be distinguished:

- connection-oriented: A *virtual connection* is set up between a sender and receiver and is used for the transmission of a stream of data.
- connectionless: Datagrams are transmitted to specified destinations without prior knowledge of a path.

Connection oriented services are based on a *virtual connection* which is a logical channel between a sender and receiver. This connection must be set up before any data is transmitted and it is closed when no longer needed. Once a connection has been opened data may begin to flow. The data items are usually bytes, and the stream may be of any length. The transport layer software is responsible to subdivide the stream of data for transmission and to deliver it reliable in the correct sequence to the receiver.

Connectionless services may be unreliable. Therefore the application layer is responsible for detecting lost or out-of-order datagrams and to force retransmissions or other error recovery mechanisms. However, both types are needed for some applications because each mechanism offers performance and programming benefits. Datagrams are fast delivered but the transmission is unreliable. Communication via streams is slow but it is reliable.

Stream vs. Datagram

Communication via streams is connection oriented. That is, communication takes place after a connection between two participants is established. It is ensured that packets are transferred in the right order and that the whole transfer is always uncorrupted. The most important property of a stream is, that no message boundaries exist. Streams are used if reliable transfer is required and maybe the size of the whole transfer is unknown.

Datagrams are only used if connectionless communication is used. That is, communication takes place without a pre set up connection between two participants and every message is addressed individually. The delivery of datagrams is not guaranteed because packets may be unable to leave the sender or may be dropped by routers or other busy machines. If a datagram arrives it is ensured that its contents are uncorrupted. Datagrams have message boundaries

which depend on individual configurations of involved routers. Communication with datagrams is used if speed is important and unreliable transfer is acceptable.

Replication

Many objects in a distributed virtual environment must be replicated, rather than just shared, because the programs using the data cannot afford the overhead of remote access. A good example is the description of a graphical scene (e.g. scene graph). The programs that update the displays must redraw their scenes as often as possible. Therefore it is necessary to have the scene graph locally available.

3.2 Distributed systems examples

Many researchers have built common virtual places in which users can interact with each other and also with responsive applications. On the one hand, vertical distribution is used to enhance the performance of graphical applications by executing on an ensemble of separate, communicating machines, exploiting the resulting parallelism (Gelernter, 1992). Such a configuration, often called decoupled simulation (Shaw et al., 1993), is commercially available via tools like Performer (Rohlf & Helman, 1994). On the other hand, horizontal distribution is used to enable collaborative applications, which allow multiple users to work together, possibly over large distances. To get a more detailed insight into distributed virtual environments and to introduce some networking issues we describe and discuss some well-known systems, which influenced our work in some way.

SIMNET

SIMNET (Pope, 1989) is a distributed military virtual environment and is sometimes called “the mother of networked virtual environments”. SIMNET was begun in 1983 and the goal was to develop a low-cost networked virtual environment for training small units to fight as a team. The SIMNET network software architecture consists of an object-event architecture, a notion of autonomous simulation nodes, and an embedded set of predictive modeling algorithms called “dead reckoning” (Miller & Thorpe, 1995). While broadcast is used to distribute event messages to other hosts, multicast is used to concurrently run multiple independent *excersises*. A dedicated simulation protocol is used for object updates. The successor of this simulation protocol is DIS (distributed interactive simulation), which has been standardized by IEEE (IEEE, 1993).

Advantages

Dead reckoning is used to reduce the transmitted number of packets. This also makes packet loss less of a problem because objects continue to move in the direction of its last known heading and at its last known speed.

Disadvantages

The broadcast mechanism places many packets onto the network, which limits scalability. SIMNET requires the use of dedicated high-performance networks and its proprietary implementation needs specialized hardware (SGI). Hence it is not usable by a broad range of researchers and it is impossible to support the simulation of different types of participants on different types of machines.

NPSNET

While the early NPSNET-I to III (Zyda et al., 1992) systems used broadcasting and an proprietary simulation protocol, NPSNET-IV (Macedonia et al., 1995) was the first three-dimensional environment that incorporated both the DIS application protocol and the IP multicast network protocol.

DIS is used for application level communication among independently developed simulators (e.g. aircraft simulators and constructive models). IP multicast is used to support large-scale distributed simulation over internetworks. NPSNET utilizes heterogeneous parallelism (decoupled simulation) for system pipelines (e.g. draw, cull, application, and network) and for the development of a high-performance network software interface.

The target application set is distributed battlefield simulation. Therefore much attention is paid to consistent updates of battlefield units. The best effort approach to distributed consistency relies on the DIS communication library. NPSNET uses a geographic approach to define multicast groups whereby the world is partitioned into hexagonal areas each associated with a multicast group.

Advantages

Decoupled simulation is used to utilize capabilities of powerful machines. NPSNET IV uses a vicinity-based area of interest filtering, based on a subdivision of the environment in 2D hexagonal cells. Furthermore the DIS protocol is used to achieve a more open system design and to allow communication among independently developed simulators.

Disadvantages

Due to the system architecture, NPSNET supports only a limited number of participants. Furthermore it is not possible to filter out or group specific network messages (this fact should not be confused with the aforementioned partitioning into cells).

MR Toolkit

The MR Toolkit (Shaw & Green, 1993) peer package implements a simple shared virtual memory model. Raw memory locations can be marked as shared and local changes explicitly flushed to the other copies, which must then explicitly receive the changes. The system is based on an unreliable best-effort protocol (UDP). It ignores lost packets and hopes that there is sufficient redundancy in packet transmission.

Advantages

MR does not use additional heartbeat packets, because it relies on frequent sending of packets. The use of an unreliable protocol improves the network delay over a reliable one because the overhead of error correction and retransmission of the same packet is economized. However, the frequent sending of packets places a high load on the network.

Disadvantages

MR Toolkit has no features to handle heterogeneous architectures. It provides a single, fully replicated VE, in which each process has an exact copy of the same world. MR maintains a complete graph connection topology, which results in $O(N^2)$ messages (in respect of the number of participants). This limits the total number of participants to four or less because of the packet loads.

DIVE

The original DIVE architecture (Carlsson & Hagsand, 1993) used the ISIS toolkit concept of process groups (Birman, 1993) to simulate a large shared memory over a network. A process group is a set of processes that are addressed as a single entity via multicasting. A more recent version of DIVE (Frecon & Stenius, 1998) makes heavy use of a scalable reliable multicast approach (instead of ISIS) for the exchange of events in order to keep the views consistent that multiple users on a network have of the world. Designed around metaphors like a white board or conference table, DIVE focuses on the development of new ways of computer supported cooperative work in three dimensions.

Advantages

DIVE uses a distributed, fully replicated database, which is dynamic and has the capability to add new objects and to modify the existing database in a reliable and consistent way.

Disadvantages

DIVE uses reliable multicast protocols and concurrency control via a distributed locking mechanism to accomplish database updates. This adds significantly to the communication costs. Because of this software architecture, it is difficult to scale DIVE beyond 16 or 32 participants. However, it does well in situations where database changes must be guaranteed and accurate at each participant's site.

NetEffect

NetEffect (Das et al., 1997) is a client-server architecture with multiple servers, which aims to support more than 1000 users. A client is linked to a server via a connection-oriented reliable connection. Each server manages one or more user-groups (so-called communities), which can be populated by user controlled avatars, and manages the transmission of update messages between the various clients. It uses "group dead reckoning", based on a visibility which is pre-determined by the system designer by defining object groups, which are in the same building or room. The communication between the servers is limited by managing all objects in a "community" on the same server, independent to the physical location of the connected clients managing those objects. Dynamic load balancing is used to create a uniform distribution of users over the servers. A similar approach is used within our system, which is described in chapter 7.

Advantages

Packet transmission is reduced by using a "group dead reckoning" approach for pre-defined object groups. Since NetEffect aims to support a very large number of geographically dispersed users it does not use multicast. This enables modem users to participate without additional software or hadware requirements. In order to distribute the load among the servers load balancing is used to support a greater number of users.

Disadvantages

NetEffect uses one master server, which maintains a databases of users. If the total number of users becomes extremely large, the master server may slow down a lot as the look-up time for the database increases significantly. A way to solve this problem is to replicate the master server. The user-database can be divided under multiple master servers, which makes the architecture more scalable.

RING

The RING system (Funkhouser, 1995) represents a virtual environment as a set of independent entities each of which has a geometric description and behavior. Entities can be either static (e.g. terrain, buildings, furniture) or dynamic. Latter can be either autonomous (e.g. robots) or controlled by a user (e.g. vehicles) via input devices.

Every entity is managed by exactly one client workstation. Furthermore clients maintain surrogates for some entities managed by other clients (remote entities). Such surrogates contain representations for the entity's geometry and behavior. But it is possible that representations are simplified. When a client receives a message for a remote entity, it updates the geometric and behavioral model for the entity's local surrogate.

RING is a client-server based application. Therefore no direct client-client connections exist. A client sends a message to the corresponding server which forwards this message to other client and server workstations participating in the same distributed simulation. The main advantage of this network topology is that servers can cull, augment and alter messages before sending them to other clients or servers.

Server based message culling is implemented using pre-computed line-of-sight visibility information. The virtual environment is partitioned into a spatial subdivision of cells and servers keep track of which cells contain which entities by exchanging automatic update messages when entities move around and cross cell boundaries. RING uses an unreliable network protocol to speed up communication and to transmit position update messages. Therefore the application layer is responsible for delivery guarantee of important messages.

Advantages

Client workstations can not limit the entities in the entire distributed simulation because the storage, processing, and network bandwidth requirements of each client are independent

among them. Every client must store and handle update messages only for the subset of entities visible to one of the client's local entities. High-level management of the virtual environment may be performed by servers without the involvement of every client. For example, adding and removing an entity requires notification of only one server. This server handles the notification of other involved clients or servers.

This client-server network topology enables the use of efficient networks and protocols available between server workstations, but not available to all client workstations. For example, clients may use a slow modem connection to a server, but the servers may use high-bandwidth links for server-server connections.

Disadvantages

The message routing through servers introduce extra latency. Because of no direct client-client connections, every message from a client routes through at least one server and possibly two. Some high-level decisions in a server (e.g. movement in a cell) increase the latency. However, Funkhouser claims that extra latency due to server processing has not been noticeable during experiments.

Spline

Spline (Waters et al., 1997) provides a convenient architecture for implementing multi-user virtual environments that are based on a shared world model. The world model is stored in an object-oriented database. Applications interact with each other by making changes to the world model and observing changes made by other applications. The system distributes the world model maintaining a partial copy of the model locally in each Spline process. Such a copy contains the parts of the model that are near to the point of interest. To maintain approximate consistency between the world model copies, Spline sends update messages when necessary.

A spline process is structured into the following modules: application support, world model, and inter-process communication. The latter sends out multicast messages describing changes in the local world model copy made by the local application and receives messages from other Spline processes about remote changes. The API of Spline consists primarily of operations for creating/deleting objects in the world model and reading/writing data fields in these objects. The application support module contains some tools that allow interaction between an application and the local world model copy.

Locales are simple collector classes and are the central organizing principle of the Spline world model. Every object is in exactly one locale. Messages about an object are sent only to the multicast address of the containing locale. If someone wants to know something about an object but the corresponding locale is unknown there is only one method to retrieve the address of the locale. This method uses so-called beacons to locate objects. Beacons contain a tag and the multicast address of the locale. The key feature of a beacon is that in addition to broadcasting messages about itself via the multicast address of the locale it is in, it broadcasts messages about itself via a special beacon multicast address. Spline features multiple locales that correspond to activities (for example, chat takes place in a street café, while train rides take place on a train). We used an adapted variation of locales in our system (see chapter 6).

Advantages

Messages about an object are sent to the multicast address of the containing locale. This makes it possible to filter incoming messages primarily by opening connections to the multicast addresses of the locales a process is currently interested in and not opening connections to the multicast addresses of other locales. Therefore the main filtering of messages is performed at the hardware level of routers and network cards. The processor load of any individual user is reduced.

Disadvantage

A great disadvantage is the fact that MBONE is not public useable. The usage of multicasting is very convenient and efficient but not everyone is able to use it. Another disadvantage is that, if too many actors are within a multicast group, the produced overhead in message processing by the nodes is unacceptable. Small multicast groups force actors to change their associated group very often. This additional overhead of entering and leaving groups introduces high work-load to the whole virtual environment.

CRYSTAL

The purpose of CRYSTAL (Tsao and Lumsden, 1997) is to create a VE that can be quickly adapted for many different types of scientific investigations. CRYSTAL allows for on-the-fly expandability by using a modular architecture to link various pieces of execution code dynamically to alter the VE's function and appearance. To let the modules interact with one another in the VE, CRYSTAL segments the VE into 3D volumes called crystals. Each module possesses one or more crystals, and draws virtual objects in the corresponding space. They are similar to desktop windows. Whereas desktop windows are completely independent, crystals

can interact frequently with one another. Our *Studierstube* framework uses and extends those ideas to support multiple users, multiple applications, multiple locales, and a multiple document interface as described in chapter 6.

Advantages

Crystals may also be completely independent. In this case, the VE becomes a general-purpose, multi-context workspace. e.g. one crystal is a 3D graph, another is a clock, etc. CRYSTAL allows any modular configurations so that the VE as well as the hardware components to control the VE can be customized on the fly.

Disadvantages

The CRYSTAL system does not incorporate true multi-user operation, which is clearly a limiting factor for a DVE. It can only be used in quasi multi user mode using a virtual workbench or CAVE.

CSpray

CSpray (Pang & Wittenbrink, 1997) is a collaborative 3D visualization system, which uses different levels of information sharing, an intuitive control strategy for coordinating access to shared resources, and several 3D visualization tools. This system allows a small group of geographically distributed scientists to share their data and to interactively create visualizations. It uses a stream based networking approach.

Advantage

Cspray uses a playback mechanism to save a trace of a collaborative session. This can later be read back and fed to the system to create a playback of the visualization process. Furthermore, one can also collaborate during playback.

Disadvantage

The design of Cspray limits the system to SGI machines. Hence it is not possible to use other platforms to participate.

Virtual environment construction tools

To speed up the development of distributed virtual environments tools were developed (e.g. Bamboo (Watsen & Zyda, 1998) and Octopus (Hartling et al., 2001)). Bamboo is a cross-platform toolkit for developing dynamically extensible, real-time networked virtual environments. By using the plugin metaphor utilized in commercial packages like Adobe

Photoshop, applications can load and unload modules at runtime, which allows the system to reconfigure itself dynamically. These modules can define geometry, textures, sounds, behavior, interfaces, etc. (Singhal and Zyda, 1999). Bamboo adds a security model to ensure correct, safe behavior of a collaborative virtual environment. Beyond this, it has a component for area-of-interest management to reduce the information individual sites have to process (Abrams et al., 1998). Finally, it provides a persistent universe so that shared environments can be "discovered" any time.

Octopus is a tool for enabling the development of collaborative VR applications. The main design goal is to mask the details of the underlying networking from the programmers. It is independent of the software used to create the VR environment, hence it can be integrated in existing application development toolkits. Octopus provides object sharing and a framework for using and adding avatars. Its treatment of shared objects as user-defined structures provides more flexibility than large collaborative combat-related solutions. However, Octopus supports only explicit distribution and lacks a consistent marshall/unmarshalling (flattening data-structures for network transmission) strategy. Furthermore it is unusable in heterogeneous networks because the object sharing mechanism is built on top of the runtime type information (RTTI) from C++.

3.2.1 Discussion

This section is a comparison between client-server and peer-to-peer network topologies designed for virtual environments. A central server is clearly a bottleneck because of resource consuming operations like many network i/o operations or client management. A server usually means that there is some finite limit to the number of participants and therefore the system is definitely not scaleable, but allows to achieve persistence of the whole system. Let us assume that the last user of a peer-to-peer based virtual environment leaves the world. What would happen? If no additional mechanisms of data warehousing are used then the system would be dissolved.

However, since a peer-based design avoids the potential bottleneck of a central resource, simple transmission techniques like broadcasting (send a message to all participants, whether they are interested or not) may introduce high network-load. Some systems use multicasting for communication. The greatest disadvantage of multicasting is that not all networks are able to support this technique (e.g. modem connections). Furthermore it is not so easy to choose a

right sized number of participants (or objects) that are associated with a multicast group. The size of a multicast group may dynamically change (participants join/leave). This operations introduce overhead to the system.

So, it is not easy to choose between server and peer-based design approaches. Hence some hybrid systems were introduced (e.g. RING: (Funkhouser, 1995), NPSNET-IV: (Singhal and Zyda, 1999), DWTP: (Broll, 1998)). RING introduced a hybrid system design including peer-to-peer and client-server communication. Users are able to connect to servers, which manage the regions for them. The client-server communication is connection-less. Therefore it does not matter if a user changes servers very often because connection-less datagrams introduces not much overhead. The server-server communication is realized with peer-based multicast. That is of no interest for users because they do not realize the multicasts between servers. But with this hybrid design a new problem needs to be solved. If a server is highly loaded (e.g. too many connected clients) it should be possible to migrate them. One solution of this problem might be load balancing (see chapter 7).

While there are many other distributed virtual environments, we have only discussed those that heavily influenced the networking strategies of our system presented in this thesis. Other prominent DVE systems that are not immediately relevant to our work include dVS (Grimsdale, 1991), WAVES (Kazman, 1993), AVIARY (Snowdon & West, 1994), VEOS (Bricken & Coco, 1994), BrickNet (Singh et al., 1995), MASSIVE (Greenhalgh & Benford, 1995), VLNET (Pandzic et al., 1995), Community Place (Lea et al., 1997), and Ultima Online (Origin, 1997).

The next section presents work, which is very important to understand some ideas (in respect of distribution) behind our own system.

3.3 Shared Scene Graphs

Distributed Virtual Environments often separate the visual representation of objects from the application semantics. While this increases modularity in the design, it also creates a “dual database” problem. Some architectures including recent work on DIVE (Steed et al., 1999), Avango (Tramberend, 1999), SGAB (Zeleznik et al., 2000) and Repo-3D (MacIntyre & Feiner, 1998) address this problem in a manner very similar to Distributed Open Inventor (DIV), which is presented in chapter 5, and (Hesina et al., 1999). DIV is an extension to the

popular Open Inventor (OIV) toolkit (Strauss & Carey, 1992) with the concept of a distributed shared scene graph, similar to distributed shared memory (Levelt et al., 1992).

As MacIntyre and Feiner put it, “Keeping these dual databases synchronized is a complex, tedious, and error-prone endeavor. In contrast, some non-distributed libraries, such as Inventor, allow programmers to avoid this problem by using the graphics scene description to encode application state”. Repo-3D addresses the problem using Modula-III with language embedding of distributed objects together with a custom graphics solution (Obliq-3D). While Modula-III is certainly a good choice for language-level embedding of distributed objects, in our opinion the user acceptance of Avango (Tramberend, 1999) - a solution based on mainstream choices (C++, Performer (Rohlf & Helman, 1994)) - would be higher.

However, Avango relies on subclassing Performer to mix in the desired transparent support for distribution. This implies that Avango applications can only use those features of Performer made available through subclassing. Furthermore, many architectural features of Avango - such as field contained in scene graph nodes and connections between fields - are standard features of OIV, but not part of Performer.

The Scene Graph as Bus (SGAB) approach, is a proposed mechanism for mapping between heterogeneous scene graphs, in a cross-platform manner. It maps scene graphs from different toolkits to an internal representation and is therefore more or less not restricted to specific scene graph toolkits.

Recent work on DIVE (Steed et al., 1999; not to be confused with DIV) introduced a scene-graph based database extension to avoid unnecessary network messages. So-called holder objects are able to generate a cascade of database modifications, instead of generating many network packets describing the modifications.

While groupware applications from the *computer supported cooperative work* field (CSCW) share some concepts of distributed objects (in our case: distributed shared scene graphs) with the aforementioned systems, inconsistencies tend to arise from multiple users attempting to perform *conflicting* actions: the results are usually obvious to the users and can be corrected using social protocols. However, this might be an acceptable solution for local collaboration (e.g. a virtual conference in the same room) but definitely not an acceptable solution for

remote collaboration. Although CSCW systems share fundamental semantic problems (e.g. consistency) with the aforementioned systems, solutions differ significantly due to different technical environments *(2D vs 3D, conventional desktop applications vs virtual reality interface).*

Last but not least, we examine some work and collaborative augmented reality, which is highly related to our work.

3.4 Collaborative Augmented Reality

Almost a decade ago, Weiser introduced the concept of *ubiquitous computing* as a future paradigm on interaction with computers (Weiser, 1991). In his vision, computers are constantly available in our surrounding by embedding them into everyday items, making access to information almost transparent. In contrast, *augmented reality* systems focus on the use of personal displays (such as see-through head-mounted displays) to enhance a users perception by overlaying computer generated images onto a user's view of the real-world.

Collaborative augmented reality enhances AR with distributed system support for multiple users with multiple display devices, allowing a co-located joint experience of virtual objects (Billinghurst et al., 1998b; Schmalstieg et al., 1996; Szalavári et al., 1998a). Some researchers are experimenting with a combination of collaborative AR, ubiquitous computing and other user interface concepts. Prominent examples include EMMIE developed at Columbia University (Höllerer et al., 1999; Butz et al., 1999), work by Rekimoto (1998), and the Tangible Bits Project at MIT (Ishii and Ullmer, 1997; Ullmer et al., 1998). These systems share many aspects with our approach for a collaborative augmented reality system making use of a variety of stationary as well as portable devices.

Working with such a system will require transfer of data from one computer's domain to another. For that aim, Rekimoto (1997) proposes *multi-computer direct manipulation,* i. e. drag and drop (or pick and drop, as Rekimoto calls it) across system and display boundaries. To implement this approach, a physical prop (in Rekimoto's case, a pen) is used as a virtual "store" for the data, while in reality the data transfer is carried out via the network using the pen only as a passive locator. Similar transfer functions are available in EMMIE (Butz et al., 1999). Such use of passive objects as perceived media containers is also implemented by the Tangible Bits group's mediaBlocks (Ullmer et al., 1998).

An issue that inevitably arises when multiple users are collaborating is that of privacy – users do not necessarily want all their data to be public (Szalavári et al., 1998a; Butz et al., 1998). A solution for the privacy issue is possible for every architecture that supports independent display to multiple users, be it via separate desktop displays (Smith and Mariani, 1997), hand-held displays (Rekimoto, 1998), head-mounted displays (Schmalstieg et al., 1996; Butz et al., 1999) or time-interlacing displays (Agrawala et al., 1997). So called *subjective views* can be employed for displaying local variations only to one user, if they are useless or distracting to other users, such as local highlighting or annotations, or if privacy is desired. Subjective views are also part of our *Studierstube* environment, and will be further exploited for the research proposed in this thesis. The *Studierstube* system is introduced in the next chapter.

4 Studierstube Overview

This chapter gives a short introduction to our *Studierstube* system. We present some background information on the original *Studierstube* framework and give an overview of the distributed version, which is used to experiment with distributed collaborative augmented reality.

4.1 Background

The original *Studierstube* architecture (Schmalstieg et al., 1996; Szalavári et al., 1998b) was a collaborative augmented reality system allowing multiple users to gather in a room and experience the sensation of a shared virtual space that can be populated with three-dimensional data. Head-tracked see-through head-mounted displays (HMDs) allow each user to choose an individual viewpoint while retaining full stereoscopic graphics. This is achieved by rendering the same virtual scene for every user's viewpoint (or more precisely, for every user's eyes), while taking the users' tracked head positions into account.

Collaborators may have different preferences concerning the chosen visual representation of the data, or they may be interested in different aspects. It is also possible to render customized views of the virtual scene for every user that differ in aspects other than the viewpoint (for example, individual highlighting or annotations). At the same time, co-presence of users in the same room allows natural interaction (talking, gesturing etc.) during a discussion. The combination of real world experience with the visualization of virtual scenes yields a powerful tool for collaboration (Figure 4).



Figure 4: Two collaborators wearing see-through displays are examining a virtual object. Note that the system supports independent views on shared objects.

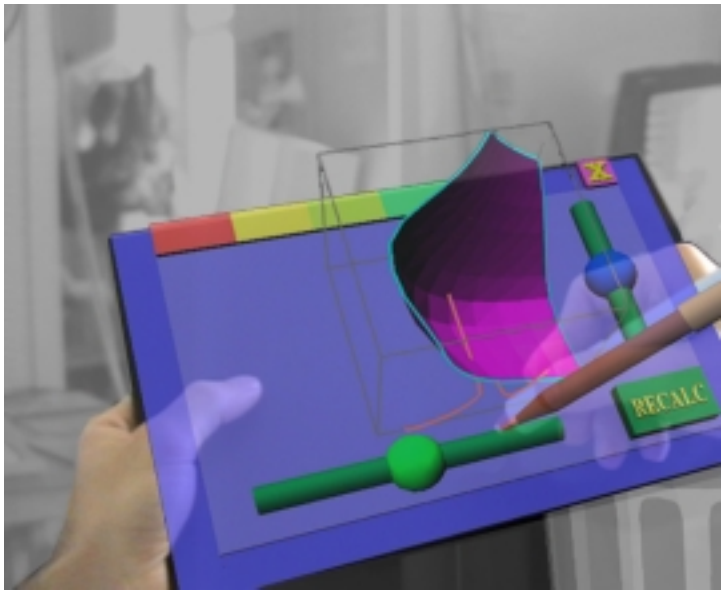


Figure 5:The Personal Interaction Panel combines tactile feedback from physical props with overlaid graphics to form a two-handed general purpose interaction tool.

We use the Personal Interaction Panel (PIP) as an input device for our system. The PIP (see Figure 5) is a two-handed interface (Szalavári & Gervautz, 1997) that is composed of two lightweight hand-held props, a pen and a panel, both equipped with magnetic trackers. Via the see-through HMD, the props are augmented with computer generated images, thus instantly turning them into application-defined interaction tools similar in spirit to the virtual tricorder of Wloka & Greenfield (1995), only using two hands rather than one. The pen and panel are the primary interaction devices.

The props' familiar shapes, the fact that a user can still see his or her own hands, and the passive tactile feedback experienced when the pen touches the panel make the device convenient and easy to use. Proprioception (Mine et al., 1997) is readily exploited by the fact

that users quickly learn how to handle the props and can remember their positions and shapes. A further advantage is that users rarely complain about fatigue as they can easily lower their arms and look down on the props.

This version of our *Studierstube* framework was able to render independent views for multiple users but it was limited to only one host (in respect of rendering). The next section describes the distributed version of *Studierstube*, which resulted from the work presented in this thesis.

4.2 Distributed Studierstube

While the *Studierstube* architecture from (Szalavári et al., 1998b) incorporated simple distribution mechanisms to provide graphics from multiple host computers and shared data from a separate device (tracker) server, the initial networking approach later turned out to be insufficient for the evolving distribution requirements. An even more limiting factor was that the toolkit allowed to run only a single application and a single application at a time. To address these problems and to enhance *Studierstube* we developed several extensions. We added multi-user capabilities and features from desktop systems (multitasking, multi document interface).

Multiple users

The first extension is to allow multiple users to collaborate (e.g. Figure 16, Figure 17). Collaboration of multiple users implies that the system will typically incorporate multiple host computers. However, we also allow multiple users to interface with a single host (e.g. via a large screen display), and a single user to interface with multiple computers at once. On a very fundamental level, this means that we are dealing with a distributed system. Hence we need a mechanism to run applications in a distributed manner. Since *Studierstube* is based on the Open Inventor scene graph toolkit we use a distributed shared scene graph approach called DIV (see chapter 5, and Hesina et al., 1999). Additional capabilities which stem from distribution arise: multiple types of output devices such as HMDs, projection-based displays, hand-held displays etc. can be handled, and the system can span multiple operating systems.

Multiple applications

To support multiple applications we need loadable application objects, which are written as separate shared objects, and dynamically loaded into the runtime framework. This is achieved by embedding applications in the scene graph. Applications use the concept of so-called contexts, which are the fundamental units from which the *Studierstube* environment is

composed. A context is a union of *data* itself, the data's *representation* and an *application* which operates on the data.

Multiple document interface

In a conventional desktop system, the data representation of a document is typically a single 2D window. Analogously, in our three-dimensional user interface, we define a context's representation as a three-dimensional structure contained in a certain volume – a *3D-window*. Unlike its 2D counterpart, a context can be shared by any group of users, and even more importantly, can be present in *multiple locales* simultaneously by replication.

Multiple locales

Locales correspond to coordinate systems in the virtual environment. They usually coincide with physical places (such as a lab or conference room, or parts of rooms), but they can also be portable and associated with a user, or used arbitrarily – we even allow (and use) overlapping locales in the same physical space. We define that every display used in a *Studierstube* environment shows the content of exactly one locale. Every context can (but need not) be replicated in every locale; these replicas will be kept synchronized by *Studierstube's* distribution mechanism.

Application migration

In section 3.1 we outlined problems of networking for distributed virtual environments. To address some of them and to further enhance our framework we developed tools, which are able to migrate applications from one host to another (independently of the operating system). These tools enable us to support dynamic user groups. That is, users are able to late-join a collaboration session or to leave at any time. Application migration is mainly used to perform load balancing among the participating hosts, which yields to better scalability. Furthermore migration is used to support and enhance remote collaboration (migration of privileges to modify a context and therefore its scenegraph).

4.2.1 Distributed shared scene graph

Current high-level graphics libraries are engineered around the concept of a *scene graph*, a hierarchical object-oriented data structure of graphical objects (see section 3.3). Such a scene graph gives the programmer an integrated view of graphical and application specific data, and allows for rapid development of arbitrary 3D applications. While most DVE systems use a scene graph for representing the graphical objects in the application, many applications

separate application state from the graphical objects. This application state is then distributed, while the graphical objects are kept locally. This allows custom solutions that optimize network utilization through minimal sharing of application state, but has two distinct disadvantages: Replicated application state and graphical objects must be kept synchronized (called “dual database problem” in (MacIntyre & Feiner, 1998)), and the distribution is not transparent to the application developer, who may even be forced to actively send synchronization messages in some replication schemes.

An alternative solution now popularized by a number of research groups (DIVE: Frecon & Stenius, 1998; Repo-3D: MacIntyre & Feiner, 1998; Avango: Tramberend, 1999; SGAB: Zeleznik et al., 2000) overcomes these disadvantages by introducing a distributed shared scene graph using the semantics of distributed shared memory. Distribution is performed implicitly through a mechanism that keeps multiple local replicas of a scene graph synchronized without exposing this process to the application programmer or user. By embedding application specific state in the scene graph, applications can now be developed without taking distribution into account, unless special multi-user features are desired.

Our own implementation of this concept is Distributed Open Inventor (DIV) (see chapter 5 and Hesina et al., 1999) based on the popular Open Inventor (OIV) toolkit (Strauss & Carey, 1992). It utilizes OIV‘s notification mechanism to automatically trigger an observer callback whenever an application changes something in the observed scene graph. These changes are then propagated to all scene graph replicas using *reliable* multicast. Network transparent access to input (tracker) data is provided through a similar mechanism based on multicast from a tracker source.

4.2.2 Runtime extension through application objects

Even dedicated end-user applications such as today‘s computer games incorporate some kind of extension mechanism that allows to add new content to the application not contained in the original distribution of the program. In virtual environments that feature a virtual world metaphor, these extensions are introduced as new objects and places. In contrast, a virtual work environment metaphor is extended through adding of new applications and services rather than objects.

However, the conceptual boundary between objects and applications is fuzzy. Extensions can have various forms of implementation, from passive geometric datasets to entities whose

behavior is determined through some scripting mechanism to native binary modules. In an object oriented framework, it is good practice to extend a system through deriving new objects from a foundation class, so that they can inherit a standard interface that will allow the surrounding simulation framework to talk to them in a meaningful way. This is used within our *Studierstube* framework to provide an interface to application programmers. Some approaches take this idea to the extreme by only providing a kernel capable of loading extensions (Bamboo: (Watsen & Zyda, 1998); Jade: (Oliveira et al., 1999)).

4.2.3 Contexts in Studierstube

Our virtual environment *Studierstube* (Schmalstieg et al., 2000) combines object-oriented runtime extension through subclassing and scene-graph based 3D work environments. An application in the Studierstube system is developed as a context that is embedded as a node in the scene graph. Surprisingly, we are not aware of any other extension mechanism that uses this particular approach.

Context classes are derived from a context foundation class that extends the basic scene graph node interface of OIV with a fairly capable application programmer‘s interface (API). This API allows convenient management of 3D user interface elements and events, and also supports a multiple-document interface – each document gets its own 3D window. Multiple documents are implemented through application instances embedded as separate nodes in the scene graph. However, they share a common application code segment, which is loaded on demand. Naturally, multiple applications can be loaded concurrently for convenient multi-tasking.

As the scene graph is distributed using DIV, so are the applications embedded in it. A newly created context instance will be added to all replicas of a scene graph, and will therefore be distributed. The programming model of making application instances nodes in the scene graph also implies that all application specific data – i. e., data members of the application instance – are part of the scene graph, and thus implicitly distributed by DIV (see Figure 6).

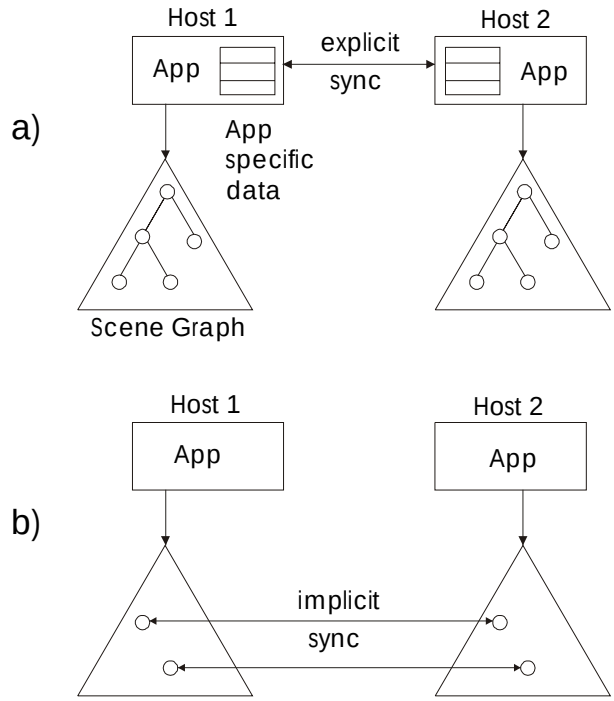


Figure 6: (a) Traditional distributed virtual environments separate graphical and application state, and synchronize only application state. (b) A distributed shared scene graph achieves replication that is transparent to the application.

4.3 Summary

This chapter introduced the *Studierstube* system. Some background information on the original *Studierstube* framework and an overview of the distributed version were presented. We outlined the requirements, which are necessary to build a distributed version and proposed solutions. The core component of the distributed system (DIV) is presented in chapter 5. Multiple applications, multiple users, multiple locales, and the support for a multiple document interface are described in chapter 6. Chapter 7 presents application migration and some applications of it. Results are presented at the end of every chapter. Chapter 8 draws conclusions.

5 Distributed Open Inventor

5.1 Introduction

In order to achieve our goal to enhance the *Studierstube* system and to build a distributed collaborative augmented reality environment we need a "toolkit" that is responsible for distribution of the graphical content. The graphical part of *Studierstube* is built on top of the Open Inventor (OIV) toolkit (Strauss & Carey, 1992) which is an object-oriented scene graph storing both geometric information and active interaction objects. This toolkit is widely available and popular with graphics programmers, and is based on the most widely accepted language for graphics (C++).

To ease the distribution and to eliminate consistency problems at the scene graph level we have developed a tool that distributes OIV's scene graph by detecting changes and to distribute only that differences. This approach is able to distribute the *Studierstube* system in a transparent and easy way. Furthermore this approach yields to a general-purpose tool for distributed graphics. That is, legacy OIV applications, which use this tool, are also able to achieve distributed execution in a transparent way. This solution has two advantages:

1. Most of the required work to enhance *Studierstube* to support collaboration via network is taken care of by DIV.
2. It is transparent to the *Studierstube* application programmer. That is, programs can be written without distribution in mind.

Our approach - *Distributed Open Inventor* (DIV) - extends the basic software to support a distributed shared scene graph, comparable to distributed shared memory (Figure 7). The implementation is almost transparent to the application programmer. Distributed programs generally execute efficiently, and the programmer need not deal with network peculiarities.

The first version of DIV utilized the notification scheme of OIV to observe a scene graph for changes. At that time OIV was commercial software and not available as source code. The successor of this DIV version uses the freely available open-source version of OIV. We have modified the underlying code base to improve performance and to track changes more accurately. The following chapters describe the first approach. At the end the second version is introduced and compared against the first one.

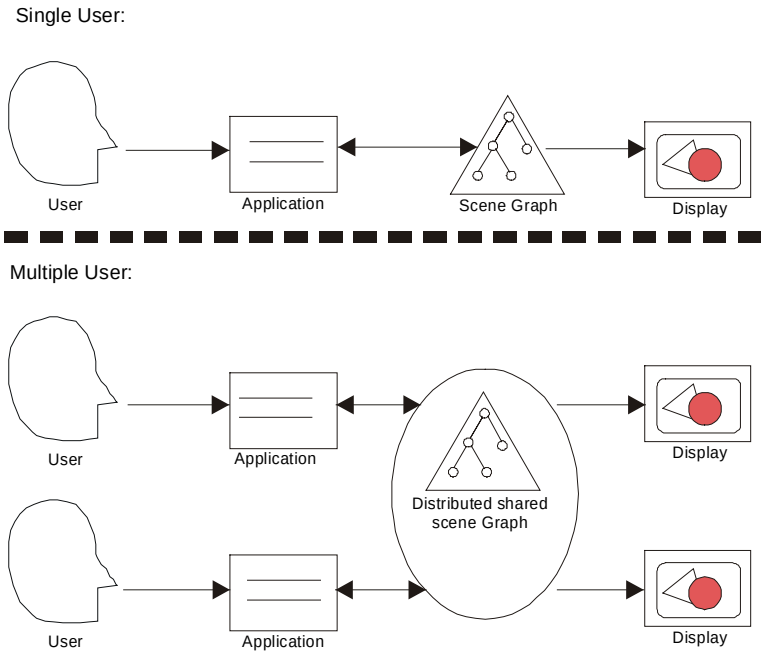


Figure 7: A single user's view of an interactive graphical application (top) is extended with the concept of a distributed shared scene graph (bottom) for multiple users.

5.2 Distributed shared scene graph

5.2.1 Motivation and overview

A scene graph is a hierarchical data structure of graphical objects. The application builds and maintains the scene graph, and the graphics toolkit uses it to create images. DIV's scene graph has the semantics of a database held in distributed shared memory (Levelt et al., 1992): Multiple workstations in a distributed system can make concurrent updates to the system, and all updates are reflected at each workstation's view of the scene graph. The scene graph represents the shared state of the distributed systems to both the application, and to the users via the images rendered from it.

The DIV runtime system takes care that all views are updated in a timely fashion, and that conflicts arising from simultaneous or near simultaneous updates of the same data entity are resolved so the consistency of the shared scene graph is not compromised.

The simplest approach to a synchronous view on shared data is to store the data only once and redirect any access via remote procedure calls (e. g. Sun RPC (Sun, 1988), Java RMI (Sun, 1998), CORBA (Ben-Natan, 1995), DCOM (Rubin & Brain, 1999)). However, interactive graphical applications, in particular virtual environments, require that the data used for

rendering is stored locally at the workstation, or interactive frame rates will simply be impossible. Therefore pure client-server approaches are infeasible for our purposes.

Instead, our approach relies on replication of the scene graph (or at least, the relevant portion) at every workstation and keep these replicas synchronized. In this section, we give an overview about how this goal is achieved. First an analysis of the paths that data flows in an interactive graphics application is given. We then consider the characteristics of these paths, in particular, which path must be fast and therefore optimized (such as the transfer from the graphical data base to the rendering hardware mentioned above).

5.2.2 Communication path for interactive graphics applications

Interactive graphical applications place the human user in a loop with the computer. A simple model of this loop is composed of the following stages (Figure 8):

- *Input* from the user
- Application specific *computation*
- The *scene graph* representing the visual state of the system
- *Display* of the scene graph

This model features the following principal communication paths within the computer system:

- Propagation of *input events* from the input devices to the computation module
- *Updates* to the scene graph as a result of computation
- *Rendering* of a 3D image from the scene graph

Some modifications of the scene graph do not require complex computations by the application, but can perform simple changes to scene graph attributes directly related to the input, but with highest possible responsiveness. The graphics toolkit allows to set up such interactions (e. g. dragging, camera movement) to work within the runtime software at maximum performance, without involving user written computation code (comparable to nervous reflexes which do not involve the human brain). We call such communication paths *input streams* (Figure 8).

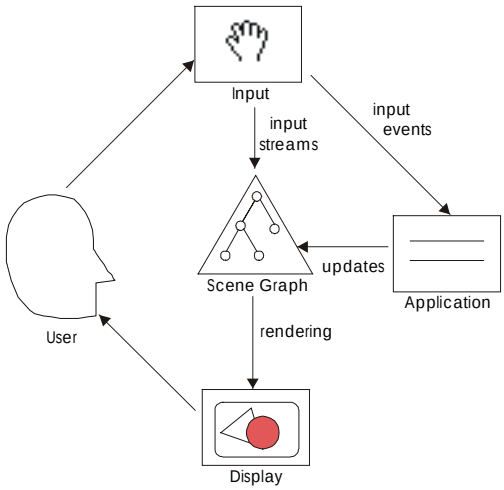


Figure 8: Typical communication path in an interactive graphical application placing the human in a feedback loop.

Because of performance requirements, input streams cannot be distributed over the network - the interaction would be too slow and the network load too high. Therefore, input streams are allowed to make local modifications to the scene graph, with mandatory synchronization only taking place after the input stream has been disabled (optionally updates can be made for synchronization purposes with lower frequency).

For the design of DIV, we must distinguish which communication paths must be fast and hence require the communicating components to reside at the same workstation. Clearly, rendering must be as fast as possible, which requires the scene graph to be stored locally and thus created the need for replication in the first place. Additionally, dividing interactions into input streams and input events allows to keep input streams locally, and distribute only input events. We have followed these design principles throughout our work.

5.3 Replicated scene graph protocol

This section explains the protocol necessary to synchronize two copies of a scene graph. Let us first examine the properties of the data structure we are dealing with. A scene graph is an object-oriented hierarchical structure reflecting the semantic relationships of graphical objects in the scene. It is composed of *nodes*, which are implemented as first class objects in the toolkit's underlying object-oriented host language (C++ in the case of Open Inventor). The toolkit typically offers a large variety of node classes for all purposes of the application. Each node is composed of fields that store that attribute data for a particular node class. A directed acyclic graph is constructed from group nodes that store links to their children. Rendering is a by-product of traversing the scene graph and executing each node's rendering method.

The vocabulary of operations possible on a scene graph consists of relatively few messages. The state of every node is determined by a node's fields. Reading a field's value does not change the state of the scene graph and therefore need not be distributed. The most common operation that must be propagated is an update of a field's value. Fields store a basic data type such as numerical values, boolean flags, vectors, matrices etc. The information necessary to encode such an update can be encoded in fixed size messages and efficiently transmitted over the network.

A special case occurs when the structure of the scene graph itself changes - nodes may be added or removed. Special messages are reserved to create and delete nodes. Note that while a typical graphical application frequently performs field updates such as changing the position of an object, changes to the scene graph's structure are relatively rare. However, if node creation occurs, there is a tendency to create a whole sub graph at once, consisting of a substantial amount of data. To make this process more efficient, applications often load whole sub graphs from a file. Our implementation generalizes this approach by introducing a message which allows all participating workstations to load a sub graph either from file (if a common file service exists) or from a URL. This solution is convenient for application programmers and also more efficient than creating node by node.

Deletion of group nodes is always recursive, i. e. if a parent node is deleted and its children are not referenced elsewhere in the scene graph, the children are also deleted, hence no message for deleting sub graphs is necessary.

Per default, nodes in OIV are anonymous unless the programmer explicitly specifies a name. However, references to nodes in messages require a unique node identifier. Therefore a message for naming a node (the node is identified by indicating the path from the root) is introduced.

A summary of the messages necessary to keep scene graph replicas synchronized is given in Table 1.

Message	parameters
Update field	Node id, field id, value
Create node	Node type, parent node name, child index
Delete node	Node name
Create sub graph	File name or URL, parent node name, child index
Set node name	Path to node, new node name

Table 1: protocol to keep scene graphs synchronized

5.4 Local variations

Most applications will just require to share a scene graph. However, a potentially much larger range of distributed graphics applications can be constructed by allowing local variations in the scene graph. Local variations (Figure 9) can be useful in a variety of ways:

- Individual content per user: Each user may operate on a variety of data sets, and choose to share only some of them, or decide on-line which data sets can be seen by other users and which not. Reasons may include privacy and security (compare (Pang & Wittenbrink, 1997)), individuality (e. g. a private shelf or clip board) or work flow (only "polished" data is shared).
- The same data may be viewed differently by multiple users, which is different to the above in that structurally identical or at least similar data is shown with different attributes to different users. Reasons to change the representation of one particular data set for individual users can be motivated by their roles. For example, a customer sees a simpler representation than the sales manager, or a teacher sees solutions to problems that the students may not see. Sometimes part of the data (such as labels) may also be intentionally hidden from other users, for example in multi- player games (Szalavári et al., 1998a) (see Figure 10).

- Individual viewpoints are a special case of individual content. This concept is particularly useful for virtual environments (see section 5.6) where head tracking on a per-user base determines the position of a virtual camera.

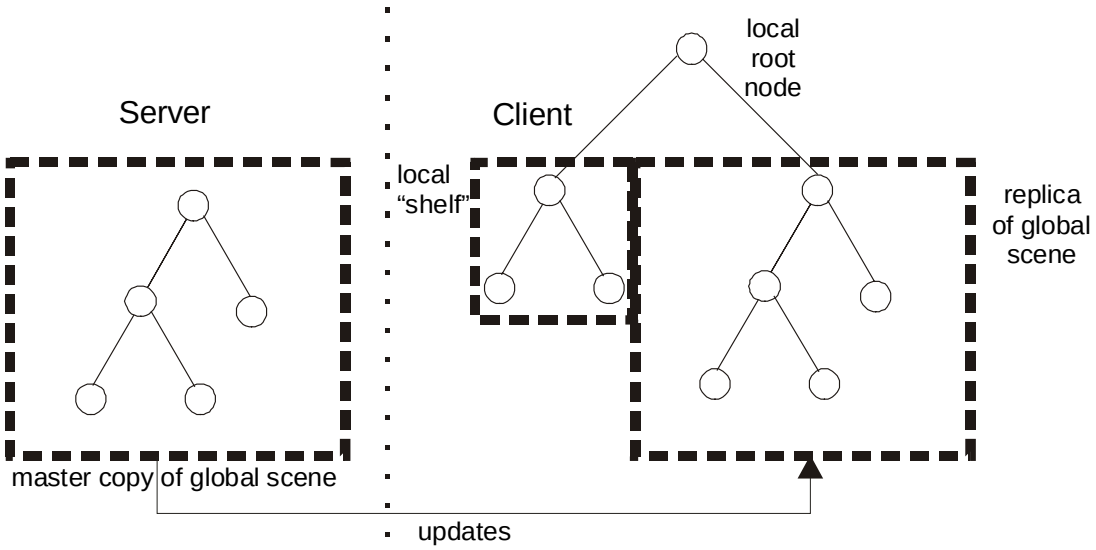


Figure 9: Local variations (such as a "shelf") allow to customize the behavior for each user.

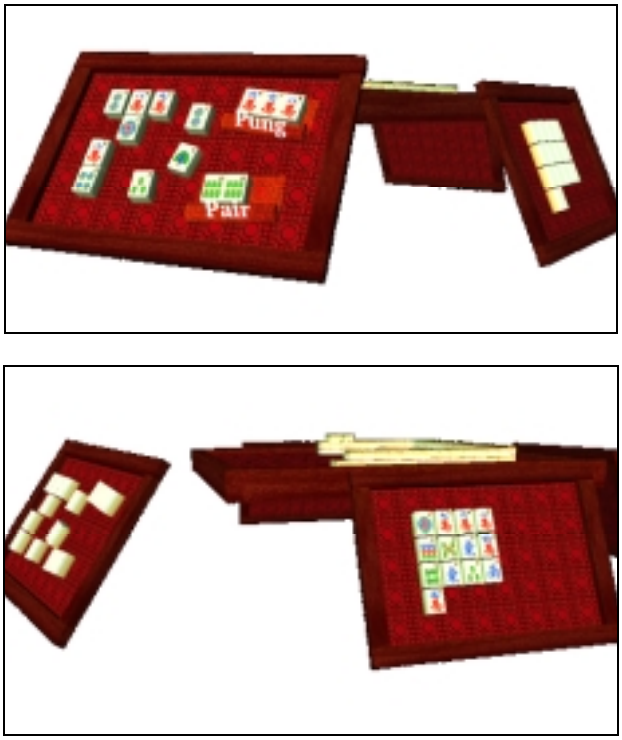


Figure 10: Personal displays secure privacy when playing Mahjongg – the left player (top view) cannot see his opponent's tile labels and vice versa (bottom view)

Some typical editing operation such as high lighting, selection, dragging, or cursor display require locally varying graphics. Note that these interaction concepts work in conjunction with low-latency input streams (see section 5.2.2) that short cut the distributed communication paths.

Using DIV to construct a scene graph that is partially distributed is straight forward: The scene graph used locally can vary from workstation to workstation. The only restriction is that those portions that are distributed must be replicated at all workstations, which does not affect applicability in practice.

5.5 Networking

Apart from basic connectivity, a key issue in distributing changes to a shared database like DIV's scene graph is how consistency among the participating processes is guaranteed. Several approaches to this problem have been investigated in the context of distributed virtual environment, and can be loosely categorized into pure client-server solutions (often found in Internet gaming such as Ultima Online (Origin, 1997) or Everquest (Sony, 1999)), pure peer-

to-peer communication (such as DIVE (Frecon & Stenius, 1998)) and hybrid topologies (such as RING (Funkhouser, 1995)).

Trade-offs in designing ideal network support are application specific and it is therefore difficult to design a distributed graphics toolkit that performs well under all circumstances while still be sufficiently suited for general purposes. We have therefore designed networking support in DIV as a configurable module to be prepared for future needs. The currently supported implementation is intended for high performance and scalability for applications that require high bandwidth such as immersive virtual environments with body tracking.

For achieving consistency, we employ a similar approach like Repo-3D (MacIntyre & Feiner, 1998): a sequencer process performs serialization of events generated by multiple users. Changes to the scene graph are then distributed via reliable multicasting (based on UDP with negative acknowledgments) to the participants, so that a consistent view of the scene graph replicas is maintained. There may be more than one sequencer present to avoid overloading one process. Typically the scene graph is coarsely divided into several logically coherent chunks (sub scene graphs) such as the content of different 3D windows (Schmalstieg et al., 2000), applications or data sets, which are then associated with separate sequencer processes. Increased flexibility is obtained by allowing a participant to choose to replicate all such sub scene graphs, or select any subset, depending on application semantics and user preferences.

Using this approach, it is possible to perform application specific computation either locally at each participant, or once in the sequencer process (the sequencer is then funtionally equivalent to an application server). The latter allows a certain degree of vertical distribution - for example, application specific computation can be performed by a compute server with multiple CPUs, while the participating workstations can focus on 3D rendering. It is also possible to create asymmetric master-slave configurations (for example, public demonstrators or location based entertainment).

5.6 Application in our Augmented Reality environment

Virtual environments differ from desktop-based interactive graphical applications primarily in their choice of input and output devices. While output is shown - usually in stereo - on a head-mounted display, or in a CAVE, input is generated using a 6 degree of freedom (6DOF) tracking system such as an Ascension Flock of Birds.

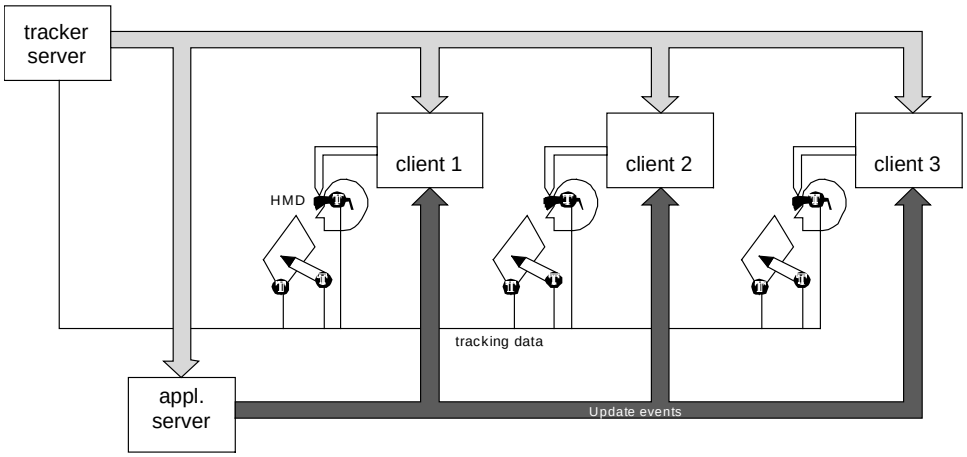


Figure 11: The *Studierstube* virtual environment uses DIV together with a tracker server that multicasts tracker data over the network.

While there is no principle difference of tracker data from input received from a mouse or keyboard, the high data rate (6DOF x multiple stations x 120 updates/sec) makes it necessary to consider the work load placed on each part of the distributed system when processing input from 6DOF trackers.

Furthermore, virtual environments typically demand a high- performance, low latency setup. For example, head tracking should directly control the virtual camera used to render the user's view. Such a requirement is directly equivalent to our input streams in that the communication path from input source to final image should be as fast as possible. Unfortunately, tracking multiple users requires that tracker data is sent over the network at some point, as only a single workstation can be connected to the tracker (typically via a serial line).

To test the applicability of the first version of DIV within our Augmented Reality system (*Studierstube*) we did some experiments (Schmalstieg et al., 2000) to use DIV (Figure 11). We resolve the issue of short communication path by distinguishing a tracker server, one or multiple application servers, and rendering clients. The tracker server uses its own multicast group to transmit tracker data over the network to *both* application servers and rendering clients. An additional benefit of this approach is that computationally intensive filtering and prediction tasks applied to the tracker data can be carried out by the tracker server without consuming resources on other workstations.

The way the tracker data is treated by the rendering clients is quite different from the application servers:
The rendering clients use the tracker data directly as an input stream for continuous actions, for example to control the virtual camera or to control interaction widgets such as the rubber band shown in Figure 15.

The application servers transform the tracker data into input events. For example, the server notes when the tracker hits a button area in 3D and passes a “press button” event to the application code, which then reacts appropriately.

Creating interaction elements that execute in such a hybrid client/server style requires a little effort, but it keeps communication paths as shorts as possible. Tracker data is always directly delivered to the workstation that needs it, no matter whether it is a client or server.

5.7 Implementation

5.7.1 Software architecture

Open Inventor was a commercial software product available for most graphics platforms, (including most Unix variants and Windows NT) and uses OpenGL for rendering. Note that OIV is now open-source. It was chosen because of its popularity, flexibility and we built *Studierstube* on top of it. Furthermore we have many legacy OIV applications available in our lab. OIV is implemented as an object-oriented class hierarchy in C++ and a library for runtime binding. Refer to Figure 12 for an overview.

The obvious choice of adding distribution properties to a class hierarchy is to modify one of the base classes to take care of distribution, so that this property is inherited throughout the class hierarchy. Unfortunately, Open Inventor as a commercial product was not available in source code, which ruled out this approach.

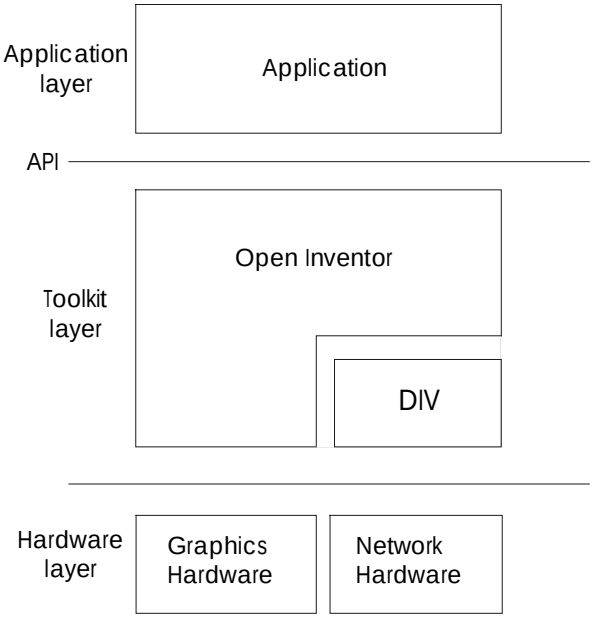


Figure 12: DIV is software that plugs into a standard graphics solution – Open Inventor – to provide distribution.

Instead, we resorted to a different approach which is equally feasible and works even if no source code is available: OIV has a built-in concept of notification that is used to propagate updates upwards in the scene graph hierarchy if a node is modified. These notification events can be monitored with a so-called node sensor. A user-specified callback function is executed whenever something changes in the sub graph associated with the node sensor. The callback receives as parameters references to the field which has changed and to the node containing the field. Update messages can trivially be constructed from this information, as only the new absolute value of the field needs to be transmitted (idempotent messages). Recording the modifications made to a scene graph by an application implicitly serves as a serialization mechanism if the application receives input events from multiple users.

A slightly more complicated situation arises if the structure of the scene graph itself changes, i. e. a node is added or deleted. In this case, the node sensor still calls the user's function, indicating the group node whose children have changed, but does not indicate which child has been added or removed. We resolved this matter by caching the hierarchical structure with a “shadow” scene graph that consists of copies of only the group nodes, while leaf nodes are referenced. When a group's children change, the group node is compared to its shadow to evaluate what change has been made. The shadow data structure is not included in the scene and thus not visible. It has also a small memory footprint and little computational overhead as it contains only links.

DIV itself uses a similar approach as Avango (Tramberend, 1999) to handle late-joining users. A new user has no knowledge of the current shared application state and therefore it is impossible to participate without an atomic state transfer from an old user to the new one. During this atomic state transfer all other communications within that certain group is suspended until the transfer completes.

5.7.2 Lazy naming

As mentioned in section 5.3, every message refers to a node and thus needs to uniquely identify the node. OIV has a built-in naming scheme for nodes based on a hash table, which is highly efficient and ideal for our purposes. It also lets users specify names for nodes in geometry files (.iv) which is a convenient way for applications to identify nodes and also works when the geometry file is distributed. However, it frequently occurs that applications modify anonymous nodes and these modifications have to be distributed.

In case of such an event, DIV automatically detects that the node is nameless and resolves the problem: The node is assigned a synthetic unique name composed of a prefix and the path from the root. This name is distributed (hence the set node name message), and then the update message refers to the newly named node. This lazy naming scheme creates extra network traffic only the first time a node is modified. As the working set of nodes that are modified in the life cycle of an application is typically small, the resulting overhead is negligible and independent of a potentially huge scene graph.

5.7.3 Usage example

In order to demonstrate the ease of transformation of exiting OIV applications into distributed applications based on DIV, we give a code example. Shown are the few modifications necessary to achieve a simple master-slave configuration. The first step is to create a DIV manager object for master or slave operation:

```
div = new CDivMain(ipAddress, port, masterOrSlave);
```

The next step is to create a root node for the scene graph at the master and enable sharing:

```
root = new SoSeparator;
root->ref();
div->shareNode(root, “myRootNode”);
```

As an example we add a sphere to the scene graph (this action is already shared):

```
Root->addChild(new SoSphere);
```

The parameter “myRootNode” is required to identify the corresponding root nodes in the master and slave process. The slave has several options to build a corresponding scene graph - either create it locally, or load it from a file or via the network. In any case, it must name its root node corresponding to the master:

```
root->setName(“myRootNode”);
```

Finally, both master and slave call their main loop. For a slave, DIV provides a modified main loop which compensates the fact that OIV is not thread safe and can therefore not be used for asynchronous processing of network updates. Figure 13 shows an example of an update.

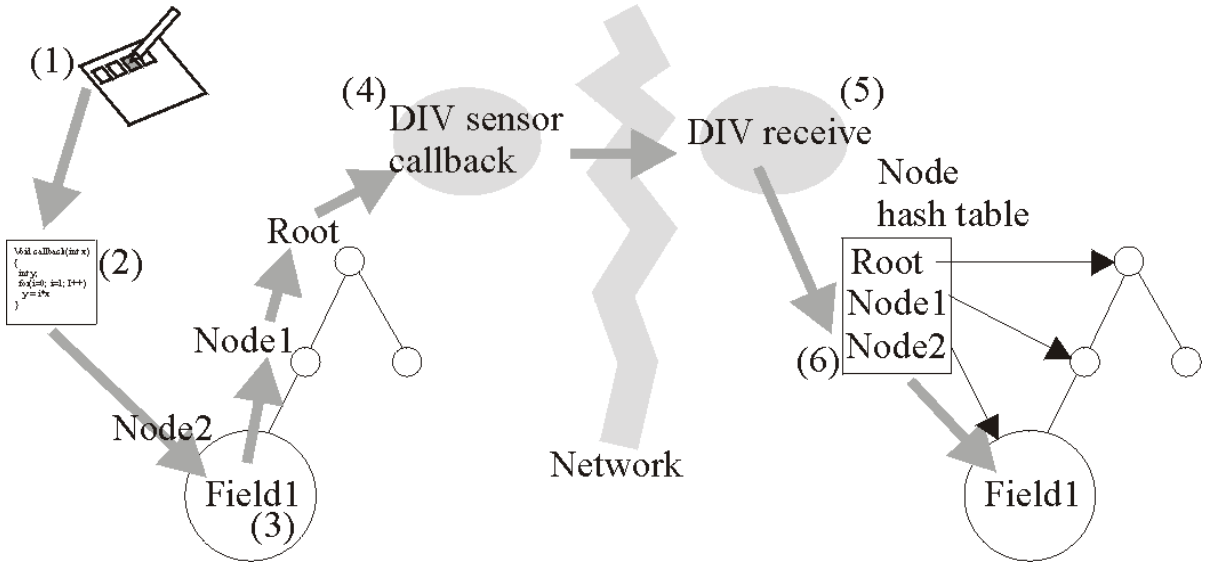


Figure 13: Example of a field update in a master-slave configuration. (1) User triggers an action by pressing a button. (2) Corresponding callback is executed and modified field1 of node2. (3) Event notification is propagated upwards in scene graph and observed by sensor. (4) Sensor transmits message to slave host. (5) Receiver picks up message and looks up corresponding node in internal hash table. (6) Slave node is modified.

5.8 Results

Several distributed multi-user applications were implemented with DIV. To verify that DIV indeed provides a programming environment that is convenient for programmers familiar with scene graph toolkits, and that distribution is almost transparent, we have extended existing single user applications written for OIV. The fact that DIV is mostly equivalent to OIV allowed to realize our test applications in a few days.

The first example that was chosen for distribution is the maze game (Figure 14) featuring a hand-held labyrinth toy which can be tilted to make a ball roll through the corridors. The objective is to guide the ball to the goal while avoiding the holes in the maze's floor. The

game was distributed for multiple users, allowing each user to see and manipulate the maze. Updates were intentionally made relative so that the resulting tilt is equal to the sum of the steering motions of all users, which creates an interesting and entertaining collaborative task. Users can also see each other's point of view represented by a simple avatar, a feature which makes use of a locally varied scene graph (each user's scene graph contains avatars for the other users, but not for the user).

A second example was constructed from a multi-user painting application implemented in our virtual environment *Studierstube*. Multiple users can collaboratively apply 3D paint into a common work volume. Each user wears a head-tracker and a tracked “brush” tool; the data from the head and tool tracker is directly fed as an input stream to the virtual camera and cursor, respectively.

Parameters such as paint color, size of paint droplets and paint pressure are controlled with local interaction widgets, which represent local variations of the scene graph - each user can have an individual current color etc. Furthermore, we make use of local variations combined with input streams for the line drawing utility (Figure 15), which displays a rubber band while the user is dragging. When the rubber band is released, a line of paint droplets is created and added to the shared scene graph.

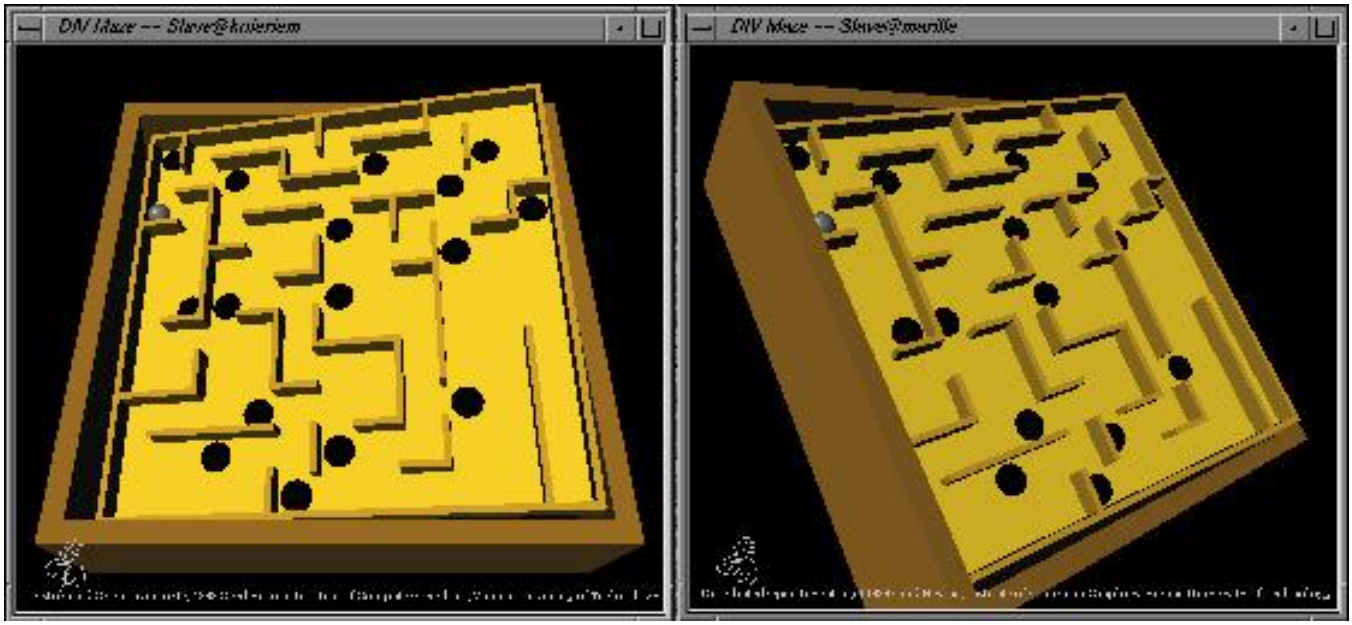


Figure 14: The shared maze game allows users to collaborate (or work against each other) using multiple workstations.

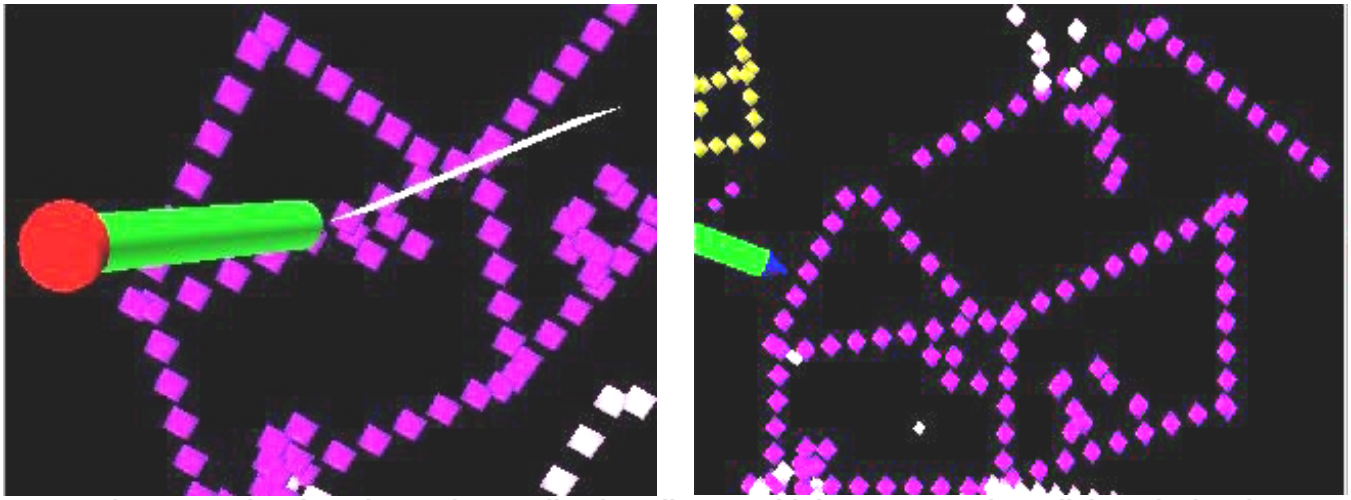


Figure 15: The shared spraying application allows multiple users to paint collaboratively. The top image shows a user drawing a rubber band, which is an example of a local graphical variation connected to an input stream. Note how the second user's view (bottom image) does not show the rubber band.

5.9 Distributed Open Inventor, Version 2

As mentioned before the previous sections described the first version of DIV. On August 15, 2000, SGI released Open Inventor to the open source community. This move enabled us to redesign DIV and to use a different approach to achieve our main goal: transparent distribution.

A drawback of the first approach is the need of a "shadow" scene graph structure which is required because OIV does not provide a mechanism to correctly identify structural scene graph changes. The "shadow" structure was used to save a snapshot of the scene graph and to compare it against the actual state. During tests we identified some problems with that approach. For example custom OIV nodes might disable the copy mechanism which prevents copying into the "shadow" structure.

To resolve this issue and to eliminate the "shadow" structure we slightly redesigned DIV. Open Inventor is now really open and available as source code. Therefore we were able to modify the base class of the group node to store additional information about a structural change. This information consists of three parts: type of change, index of modified or new child and a pointer to the modified or new child.

The notification mechanism is now able to retrieve that information and to report it via the usual callback mechanism. This little change was enough to remove the shadow structure and to improve DIV's overall performance.

Additional modifications to multiple valued fields (array fields) were done to optimize those field updates. OIV is not able to report the index of a changed array entry within a multiple valued field. Instead OIV reports that the whole field was modified resulting in a large DIV message if the array has many entries. To overcome this issue the base class for multiple valued fields store the index of the modified entry.

5.10 Summary

This chapter has introduced a practical approach to distributed graphics, realized as DIV, the Distributed Open Inventor library. DIV is founded on the notion of a distributed shared scene graph, a powerful data structure that unifies graphical and application data with distributed control. Our implementation extends the popular Open Inventor toolkit and thus allows programmers to continue software development in a familiar style and software development environment. We have built DIV to enable distributed collaborative work within our *Studierstube* system. Our approach is almost completely transparent to the application programmer and allows existing applications to be distributed with very little effort. DIV is used in *Studierstube* but can also be used with legacy OIV applications as well. The next chapter describes the design philosophy of our distributed collaborative augmented reality version of *Studierstube*, as well as the underlying software and hardware architecture.

6 Bridging Multiple User Interfaces with Augmented Reality

6.1 Introduction

While the previous chapter described the toolkit (DIV) that we have developed to distribute our *Studierstube* system on a low (graphical attribute) level, this chapter introduces our work to enhance our system on an application level to enable distributed collaborative augmented reality.

As mentioned in the distributed *Studierstube* overview section 4.2, *Studierstube* started as a pure augmented reality setup (Schmalstieg et al., 1996; Szalavári et al., 1998b) that focused on experimenting with the possibilities of new user interfaces that incorporate AR. This architecture incorporated simple distribution mechanisms to provide graphics from multiple host computers and shared data from a separate device (tracker) server. It turned out that this approach was insufficient to provide a framework for distributed collaborative augmented reality. An even more limiting factor was that the toolkit allowed to run only a single application at a time. This chapter describes our efforts to convert *Studierstube* to a distributed collaborative augmented reality environment.

For efficient experimentation, we have implemented a framework that generalizes over multiple user interface dimensions, allowing rapid prototyping of different user interface styles. The *Studierstube* user interface spans the following dimensions:

6.2 Multiple users

The system allows multiple users to collaborate (e.g. Figure 16, Figure 17). While we are most interested in computer-supported face-to-face collaboration, this definition also encompasses remote collaboration. Collaboration of multiple users implies that the system will typically incorporate *multiple host computers*. However, we also allow multiple users to interface with a single host (e.g. via a large screen display), and a single user to interface with multiple computers at once. On a very fundamental level, this means that we are dealing with a distributed system. It also implies that *multiple types of output devices* such as HMDs, projection-based displays, hand-held displays etc. can be handled and that the system can span *multiple operating systems*.

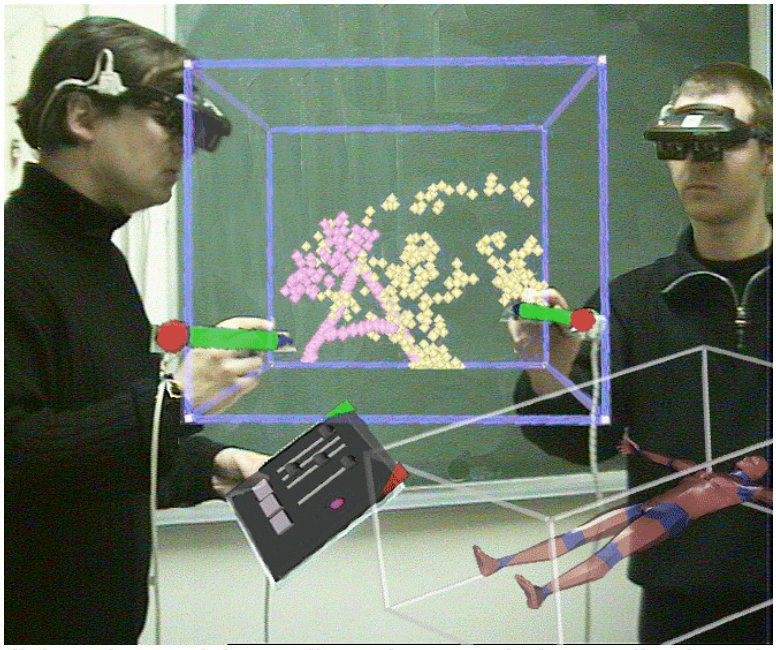


Figure 16: Collaborative work in *Studierstube*: 3D painting application window (with focus, middle) and object viewer window (without focus, lower right).

This has the advantage that application specific computations, typically callbacks triggered through events created through user input, need not be repeated at every host. Instead, for every application instance, a master host is determined, which is responsible for performing all execution of application code. The updates to the application state resulting from these computations are then replicated in the slaves‘ replicas of the application instance through DIV.

6.3 Multiple contexts

As pointed out above, applications are used to construct contexts. Contexts are structured along the lines of the model-view-controller (MVC) paradigm known from Smalltalk’s windowing system (Goldberg & Robson, 1983): *Studierstube*’s data, representation, and application correspond to MVC’s model, view, and controller, respectively. Not surprisingly, this structure makes it straightforward to generalize established properties of 2D user interfaces to three dimensions.

Every context is an instance of a particular application type. Contexts of different types can exist concurrently, which results in *multi-tasking* of *multiple applications*, a feature which is well established within the desktop metaphor, but rarely implemented in virtual environments. Moreover, *Studierstube* also allows multiple contexts of the same type to co-exist, allowing a single application to work with multiple data sets. In the desktop metaphor, this feature is generally known as a *multiple document interface*. Note that it differs from simply allowing multiple instances of the same application which are unaware of each other. Multiple contexts

of the same type are aware of each other can share features and data. For example, consider the shared “slide sorter” from section 6.7.

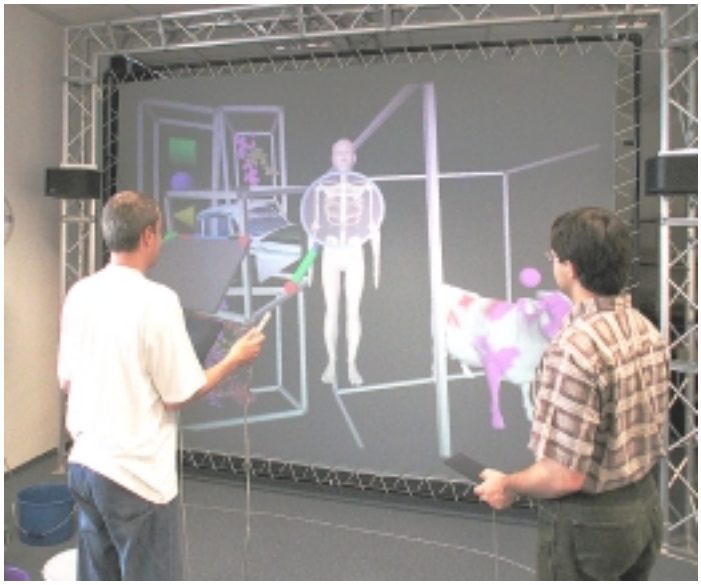


Figure 17: Two *Studierstube* users working jointly on multiple applications in front of a large screen, usually with passive stereo glasses (not shown)

6.4 Multiple locales

Locales correspond to coordinate systems in the virtual environment. They usually coincide with physical places (such as a lab or conference room, or parts of rooms), but they can also be portable and associated with a user, or used arbitrarily – we even allow (and use) overlapping locales in the same physical space. We define that every display used in a *Studierstube* environment shows the content of exactly one locale. Every context can (but need not) be replicated in every locale; these replicas will be kept synchronized by *Studierstube’s* distribution mechanism.

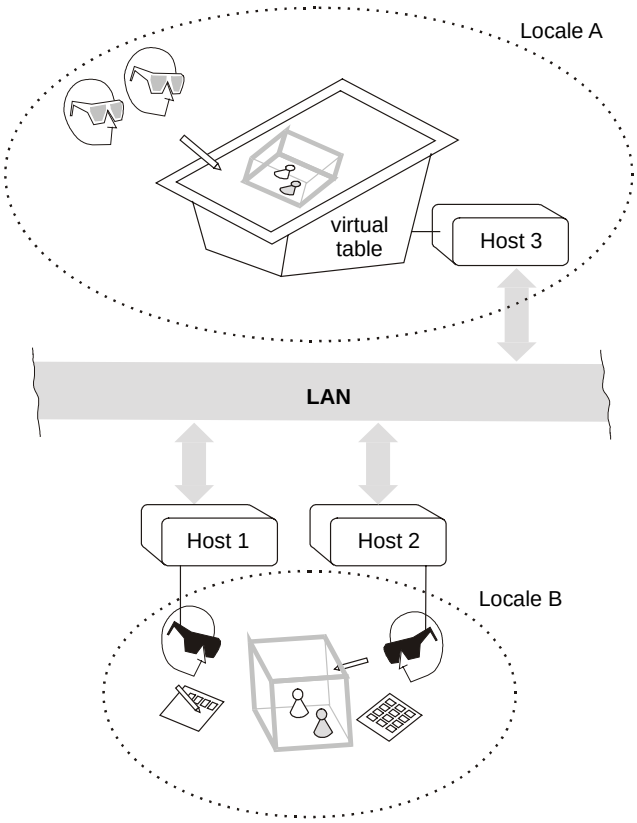


Figure 18: Multiple locales can simultaneously exist in *Studierstube*. They are used to configure multiple different output devices and/or to support remote collaboration

To understand why the separation of locales and contexts is necessary, consider the following examples:

- Multiple users are working on separate hosts. They can share contexts, but can layout the context representations (3D-windows) arbitrarily according to screen format and personal preferences. This is made possible by defining separate locales, as the position of 3D-windows is not shared across locale boundaries (Figure 18). The hosts can be in separate buildings for remote collaboration, or they can be placed side by side. In the latter case, locales would probably overlap, as users might see several or all screens.
- A user wearing a see-through HMD is looking at a large projection screen *through* the HMD. Both display devices (HMD, projection screen) can be set to use the same locale, so the graphics in a user's HMD may *augment* the projection screen's output. Of course this setup is view-dependent and work for only one user, so alternatively, the projection screen may use a separate locale, and present graphical elements which are complementary to the HMD output.

By separating locales (geometric relationships) from contexts (semantic relationships), we achieve a great amount of flexibility in the configuration of displays.

The system presented in this thesis must be understood as an experimental platform for exploring the design space that emerges from bridging multiple user interface dimensions. It can neither compete in maturity and usability with the universally adopted desktop metaphor nor with more streamlined, specialized virtual environment solutions (e. g., CAVEs). However, *Studierstube* demonstrates a design approach for next generation user interfaces as well as solutions on how to implement these interfaces.

6.5 Interaction design

In this section, we give a more detailed explanation of important features and concepts of our user interface.

6.5.1 3D-windows

The use of windows as abstraction and interaction metaphor is a long-time convention in 2D GUIs. Its extension to three dimensions seems logical (Feiner & Beshers, 1990; Tsao & Lumsden, 1997) and can be achieved in a straightforward manner: Using a box instead of a rectangle seems to be the easiest way of preserving the well-known properties of desktop windows when migrating into a virtual environment. It supplies the user with the same means of positioning and resizing the display area and also defines its exact boundaries. Obvious differences of these 3D windows (”boxes”) to their desktop counterparts can in many cases be resolved easily. Positioning a box by grabbing a designated part of its geometry may of course include the rotation of the window to an arbitrary orientation. Resizing is achieved by grabbing a corner and repositioning it with 3DOF, thereby changing all measurements of the box in one movement.

6.5.2 Contexts and sharing

A context encapsulates visible and invisible application-specific data together with the responsible application. The notion of an application is therefore completely hidden from the user, in particular, users never have to “start” an application, they simply open a context of a specific type. This approach simplifies operation and is successfully implemented in today’s *Personal Digital Assistants* such as the PalmPilot, albeit with a limited set of applications. Compared to the desktop metaphor, this approach is much closer to the concept of an *information appliance*, which is always “on”, as desired by several authors (Billinghurst et al., 1998a; Mann, 1997).

Studierstube supports multi-tasking of *different* applications (e.g. a painting application and a 3D modeler), but also multiple concurrent contexts associated with the *same* application (Figure 19). This approach is similar to popular desktop systems such as the *multiple document interface*.

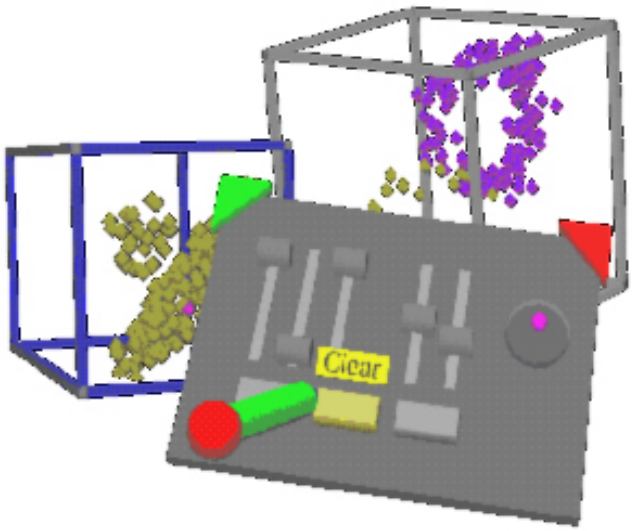


Figure 19: Multiple document interface in 3D – the left window has the user's focus and can be manipulated with the current PIP sheet.

Depending on the semantics of the associated application, ownership of a context may or may not privilege a user to perform certain operations on the information (such as object deletion). Per default, users present in the same locale will share a context. A context – represented by its 3D-window - is owned by one user, and subscribed by others. Per default, a context is visible to all users and can be manipulated by any user in the locale.

6.5.3 Subjective views

A user owning a context may decide to declare that context as private, so that it is hidden to other users. Some contexts may allow special subjective views with a finer granularity than the simple choice of completely visible vs. completely hidden, e. g. object shown with or without textures, such as proposed in (Szalavári et al., 1998a). Basic support for subjective views is built into *Studierstube* in the form of shared/non-shared data. However, while private contexts are a standard feature of *Studierstube*, the semantics of such custom subjective views must be defined in the application associated with a context.

6.5.4 Multiple locales

Multiple locales can exist concurrently. This concept is very powerful, as it not only allows to connect multiple *Studierstube* environments over a network for remote collaboration, but also

to set up an environment with multiple co-located, i. e., overlapping locales. Consider as a scenario a spacecraft mission control center with dozens of collaborating operators assembled in a large hall. Every involved user will assume a specific role and require specific tools and data sets, while some aspects of the mission will be shared by all users. A naïve approach of embedding all users in a single locale means that users in close proximity can work in a shared virtual space, while other users who desire to participate are too far away to see the data well, and are not within arm's reach for manual interaction. By separating contexts from locales, a remote user can import the context into a separate locale, and interact with it conveniently. While our available resources do not allow us to verify such large-scale interaction, in section 6.7 we present some results that back up our considerations.

6.6 Implementation

6.6.1 System overview

Our software development environment is realized as a collection of C++ classes built on top of the Open Inventor (OIV) toolkit (Strauss & Carey, 1992). The rich graphical environment of OIV allows rapid prototyping of new interaction styles. The file format of OIV enables convenient scripting, overcoming many of the shortcomings of compiled languages without compromising performance. At the core of OIV is an object-oriented scene graph storing both geometric information and active interaction objects. Our implementation approach has been to extent OIV as needed, while staying within OIV's strong design philosophy.

This has lead to the development of two intertwined components: A toolkit of extensions of the OIV class hierarchy (mostly interaction widgets capable of responding to 3D events), and a runtime framework, which provides the necessary environment for *Studierstube* applications to execute. Together, these components form a well-defined application programmer's interface (API), which extends the OIV API, and also offers a convenient programming model to the application programmer (section 6.6.5). Applications are written and compiled as separate shared objects (.so for IRIX, .dll for Win32), and dynamically loaded into the runtime framework. A safeguard mechanism makes sure only one instance of each application is loaded into the system at any time. Besides decoupling application development from system development, dynamic loading of objects also simplifies distribution as application components can be loaded by each host whenever needed. All these features are not unique to *Studierstube*, but rarely found in virtual environment software.

6.6.2 3D-windows

A context is normally represented in the scene by a 3D-window, although we allow a context to span multiple windows. The 3D-window class is a container associated with a user-specified scene graph. This scene graph is normally rendered with clipping planes set to the faces of the containing box, so that the content of the window does not protrude from the window's volume. Nested windows are possible, although we have found little use for them. The window is normally rendered with associated "decoration" that visually defines the windows extent and allows it to be manipulated with the pen (move, resize etc). The color of the decoration also indicates whether a window has a user's focus (and hence receives 3D event from that user). Like their 2D counterparts, 3D-windows can be minimized (replaced by a three-dimensional icon to save space in a cluttered display), and maximized (scaled to fill the whole work volume and receive input events exclusively). Typically, multiple context of the same type will maintain structurally similar windows, but this decision is at the discretion of the application programmer.

6.6.3 PIP sheets

Studierstube applications are controlled either via direct manipulation of the data presented in 3D-windows, or via a mixture of 2D and 3D widgets on the PIP. A set of controls on the PIP – a *PIP sheet* - is implemented as an OIV scene graph composed primarily of *Studierstube* interaction widgets (such as buttons etc.). However, the scene graph may also contain geometry (e. g., 2D and 3D icons) that are useful to convey user interface state or merely as decoration. Note that all 3D widgets of Studierstube (e.g. buttons, sliders, checkboxes, dials) are able to distinguish between 3 different operation modes:

1. Normal mode (default): if Studierstube is used in non-distribution mode. Widgets react with their default behavior.
2. Master mode: if the associated context is a master context. Widgets do not send updates via DIV unless a "commitment event" occurs (e.g. slider: button release).
3. Slave mode: if the associated context is a slave context. Widgets react with their default behavior. Updates via DIV are possible (master mode widgets send updates after a "commitment event" has occurred).

Every type of context defines a PIP sheet template, a kind of application resource. For every context and user, a separate PIP sheet is instantiated. Each interaction widget on the PIP sheet

can therefore have a separate state. For example, the current paint color in our artistic spraying application (Figure 19) can be set individually by every user for every context. However, widgets can also be shared over all users, all contexts, or both. Consequently, *Studierstube's* 3D event routing involves a kind of multiplexer between windows and users' PIP sheets (Figure 20).

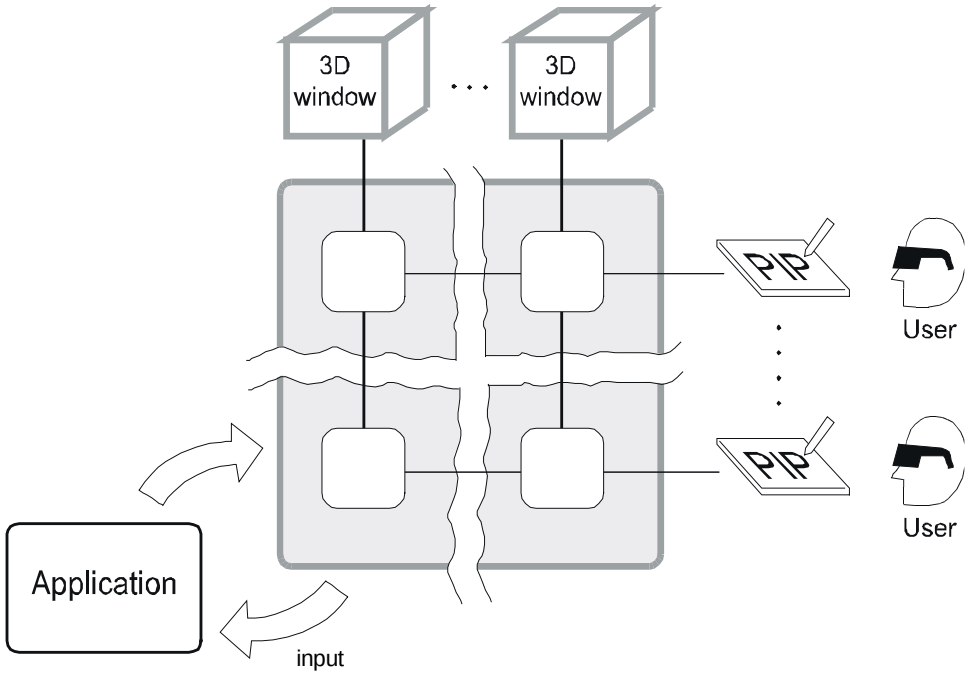


Figure 20: Multiplicity relationships in *Studierstube* - control elements on the PIP are instantiated separately for every (user, 3D-window) pair

6.6.4 Distributed execution

The distribution of *Studierstube* requires that for each replica of a context all graphical and application-specific data is locally available at each host which has a replica. In general, applications written with OIV encode all relevant information in the scene graph, so replicating the scene graph at each participating host already solves most of the problem.

For that aim, we have created Distributed Open Inventor (DIV) (section 5; Hesina et al., 1999) as an extension (more a kind of plug-in) to OIV. A scene graph need not be totally replicated – local variations (compare MacIntyre & Feiner, 1998) in the scene graph can be introduced, which is among others useful for fine-tuning low-latency operations such as dragging.

More importantly, local variations allow us to resolve distribution on a *per-context* base. A context is owned by one workstation (called a master context), which will be responsible of

processing all relevant interaction on the application, while other workstations (in the same locale and in other locales) may replicate the context (as a slave context).

The roles that contexts may assume (master or slave) affect the status of the context's application part. The context data and its representation (window, PIP sheet etc.) stay synchronized over the whole lifespan of the context for every replica. The application part of a master context is active and modifies context data directly according to the users' input. A slave context's application is dormant and does not react to user input (for example, no callbacks are executed if widgets are triggered). Instead, a slave context relies on updates to be transmitted via DIV. Note that context replicas can swap roles (e. g., by moving master contexts to achieve load balancing), but at any time there may only be one master copy per replicated context.

This has the advantage that application specific computations, typically callbacks triggered through events created through user input, need not be repeated at every host. Instead, for every application instance, a master host is determined, which is responsible for performing all execution of application code. The updates to the application state resulting from these computations are then replicated in the slaves‘ replicas of the application instance through DIV.

This approach shares the most significant advantage of DVE client-server systems such as NetEffect (Das et al., 1997), RING (Funkhouser, 1995), or Ultima Online (Origin, 1997): serialization of updates is implicitly performed, which removes the need for a special consistency protocol and simplifies distribution semantics. In fact, a master/slave pair of application instances has similar semantics like network objects (Birrell et al., 1993), or even X windows (Scheifler & Gettys, 1983) applications that separate user interface from application execution, only that Studierstube applications execute the user interface on both client and server.

Once the low-level replication of context data is taken care of by DIV, the high-level context management protocol is fairly simple: A dedicated session manager process serves as a mediator among hosts as well as a known point of contact for newcomers. The session manager does not have a heavy workload compared to the hosts running the *Studierstube* user interface, but its directory services are essential. For example, it maintains a list of all active

hosts and which contexts they own or subscribe, it gets to decide about policy issues such as load balancing etc.

The master host can be determined for every application instance separately. This implies that a single host can be master for one application instance, but slave for another. Coarse grained parallelism is introduced by distributing the master responsibilities over the hosts according to some scheme. This dual role of every host as master/slave for application instances can be seen as a generalization of peer-to-peer DVE systems such as NPSNET (Zyda et al., 1992), where hosts maintain a master copy of the locally controlled entity and slave (or "ghost" (Blau et al., 1992)) copies for all other.

It is noteworthy that the assignment of master host to application is not performed per application class, but per application instance. This commonly leads to situations where the master copies of two application instances of the same application are maintained by two different hosts. This implies a truly distributed execution of that application, which is handed by the system in a manner completely transparent to the application programmer and the application‘s user(s). Since application instances correspond to documents maintained by the application, the system also implements a distributed multi-document interface.

In combination, the embedding of application instances into a distributed shared scene graph allows to combine attractive properties of client-server and peer-to-peer systems into a coherent whole.

Finally, input is managed separately by dedicated device servers (typically PCs running Linux), which also perform the necessary filtering and prediction. The tracker data is then multicast in the LAN, so it is simultaneously available to all hosts for rendering. Every uses this input data to construct 3D Events.

As pointed out above, only the master copy of a replicated application instance needs to perform application specific computation. Therefore, only the master copy of an application node registers event callbacks with the runtime system, and this rule applies recursively to all event-aware nodes (widgets) contained in that applications sub graph. As a consequence, if an event occurs only the master copy of an application instance will react to it directly, regardless whether the event processing is done directly by the application or indirectly by a contained

widget. Slave copies receive their updates through DIV messages that are automatically created when a node‘s state changes.

However, as pointed out in (MacIntyre & Feiner, 1998), some interaction such as highlighting or dragging styles require highest performance feedback and cannot rely on a regular distribution mechanism. For that purpose, a temporarily relaxed consistency is introduced on a per-widget level. Both master and slave widgets are allowed to directly process input events for visual display (e. g., update an object‘s position while dragging it), but after the local operation is finished, only the master widgets is allowed to “commit” the action by setting the final state, which is then propagated to slave widgets (see also section 6.6.3). Using this protocol, no inconsistencies can occur, while high performance of local operations is ensured. Note that this kind of distributed behavior is implicit in the widgets, and need not concern an application programmer that uses these widgets in an application.

6.6.5 Application programmer’s interface

The *Studierstube* API imposes a certain programming model on applications, which is embedded in a foundation class, from which all *Studierstube* applications are derived. By overloading certain polymorphic methods of the foundation class, a programmer can customize the behavior of the application. The structure imposed by the foundation class makes sure the application allows multiple contexts to be created (i. e., offers the equivalent to a multiple document interface), each of which can be operated in both master mode (normal application processing) and slave mode (same data model, but all changes occur remotely through DIV).

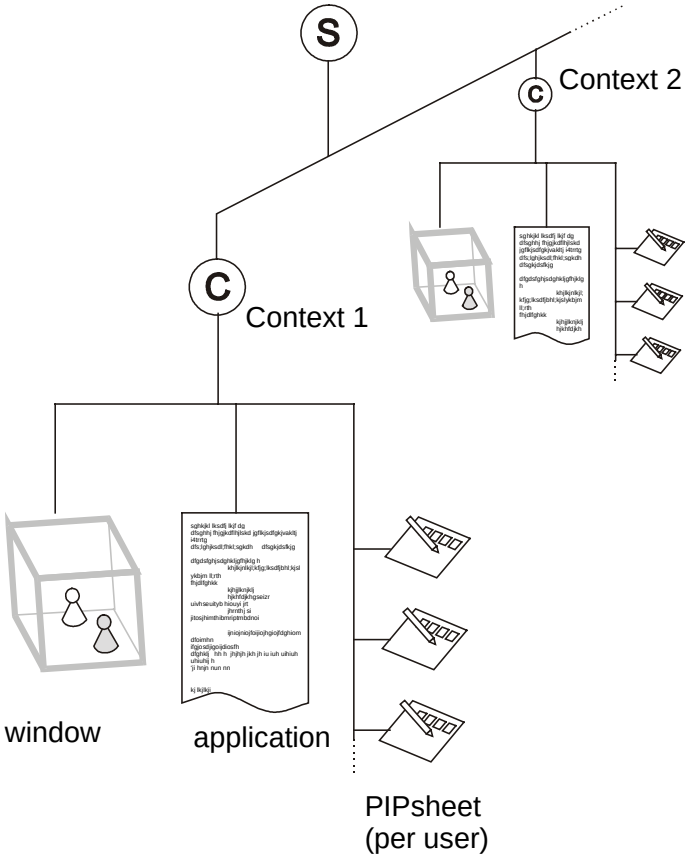


Figure 21: A context is implemented as a node in the scene graph, as are windows and pip sheets. This allows to organize all relevant data in the system in a single hierarchical data structure.

The key to achieve all this is to make the context itself a *node* in the scene graph. Such context nodes are implemented as OIV *kit* classes. Kits are special nodes that can store both fields, i. e., simple attributes, and child nodes, both of which will be considered part of the scene graph and thus implicitly be distributed by DIV. Default parts of every context are at least one 3D-window node, which is itself an OIV kit and contains the context's "client area" scene graph, and an array of PIP sheets, which are also special scene graphs. In other words, data, representation, and application are all embedded in a single scene (Figure 21), which can be conveniently managed by the *Studierstube* framework.

To create a useful application with all the properties mentioned above, a programmer need only create a subclass of the foundation class and overload the 3D-window and PIP sheet creation methods to return custom scene graphs. Typically, most of the remaining application code will consist of *callback* methods responding to certain 3D events such as button press or 3D direct manipulation events. Although the programmer has great freedom to use anything that the OIV and *Studierstube* toolkits offer, it is a requirement that any instance data is stored in the derived context class as a field or node, or otherwise it will not be distributed. However,

this is not a restriction in practice, as all basic data types are available in both scalar and vector format as fields, and new types can be created should the existing ones turn out to be insufficient (a situation that has not occurred to us yet).

Note that allowing a context to operate in both master and slave mode has implications on how contexts can be distributed: It is not necessary to store all master contexts of a particular type at one host. Some master contexts may reside on one host, some on another host – in that case, there will be corresponding slave contexts at the respective other host, which are also instances of the same kit class, but initialized to function as slaves. In essence, our API provides a distributed multiple document interface.

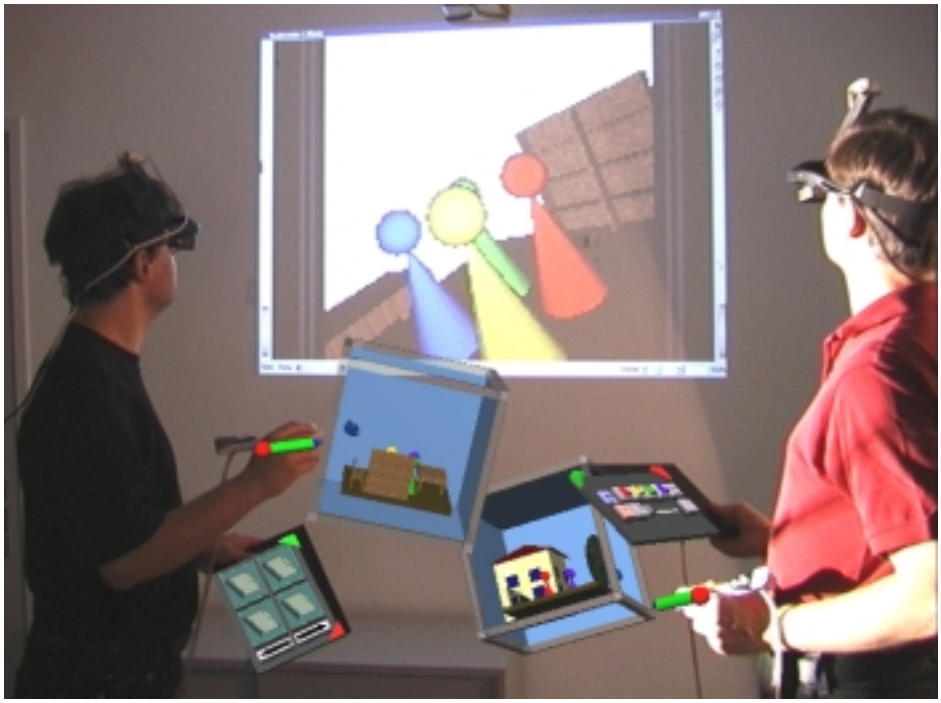


Figure 22: Storyboard application with two users and two contexts as seen from a third "virtual" user used for video documentation. In the background the video projection is visible.

6.7 Results

To demonstrate our framework, we chose the application scenario of *Storyboard design*. This application is a prototype of a cinematic design tool. It allows multiple users to concurrently work on a storyboard for a movie or drama. Individual scenes are represented by their stage sets, a kind of *world in miniature* (Pausch et al., 1995). Every scene is represented by its own context, and embedded in a 3D-window. Users can manipulate the position of props in the scene as well as the number and placement of actors (represented by colored board game figures), and finally the position of the camera (Figure 22, Figure 23).

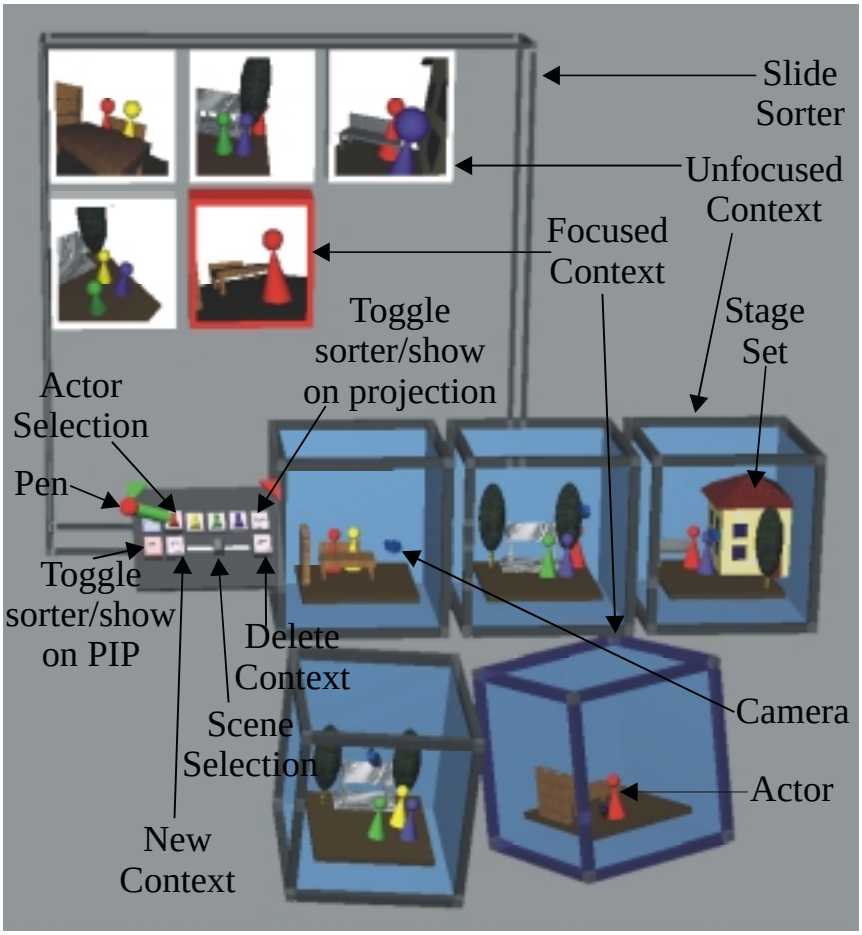


Figure 23: The Storyboarding application allows the 3D placement of actors, props, and cameras. The slide sorter shows a storyboard of all camera "shots"

All contexts share an additional large *slide show* window, which shows a 2D image of the selected scene from the current camera position. By flipping through the scenes in the given sequence, the resulting slide show conveys the visual composition of the movie.

Alternatively, a user may change the slide show to a "slide sorter" view inspired by current presentation graphics tools, where each scene is represented by a smaller 2D image, and the sequence can be rearranged by simple drag and drop operations. The slide sorter comes closest to the traditional *storyboard* used in cinematography. It appears on the PIP for easy manipulation as well as on the larger projection screen.

Using the distributed *Studierstube* framework, we ran the Storyboard application in different configurations.

6.7.1 Heterogeneous displays

Our first configuration (Figure 22, Figure 24) consisted of three hosts (SGI Indigo2, Intergraph Wildcat, SGI O2), two users, and two *locales* (Figure 25). It was designed to show

the convergence of multiple users (real ones as well as virtual ones), contexts, locales, 3D-windows, hosts, displays and operating systems.

The two users were wearing HMDs, both connected to the Indigo2's multi-channel output, and seeing head-tracked stereoscopic graphics. They were also fitted with a pen and pad each. The Intergraph workstation was driving an LCD video projector to generate a monoscopic image of the projection screen (without viewpoint tracking) on a projection wall. The slider show/sorter 3D-window was hidden from graphics output on the HMDs, so the users could see the result of their manipulation of the miniature scenes on the large bright projection exploiting the see-through capability of the HMDs. Users were able to perform some private editing on their local contexts, then update the slide show/sorter to discuss the results. Typically, each user would work on his or her own set of scenes. However, we choose to make all contexts visible to both users, so collaborative work on a single scene was also possible. The slide sorter view was shared between both users, so global changes to the order of scenes in the movie were immediately recognizable.

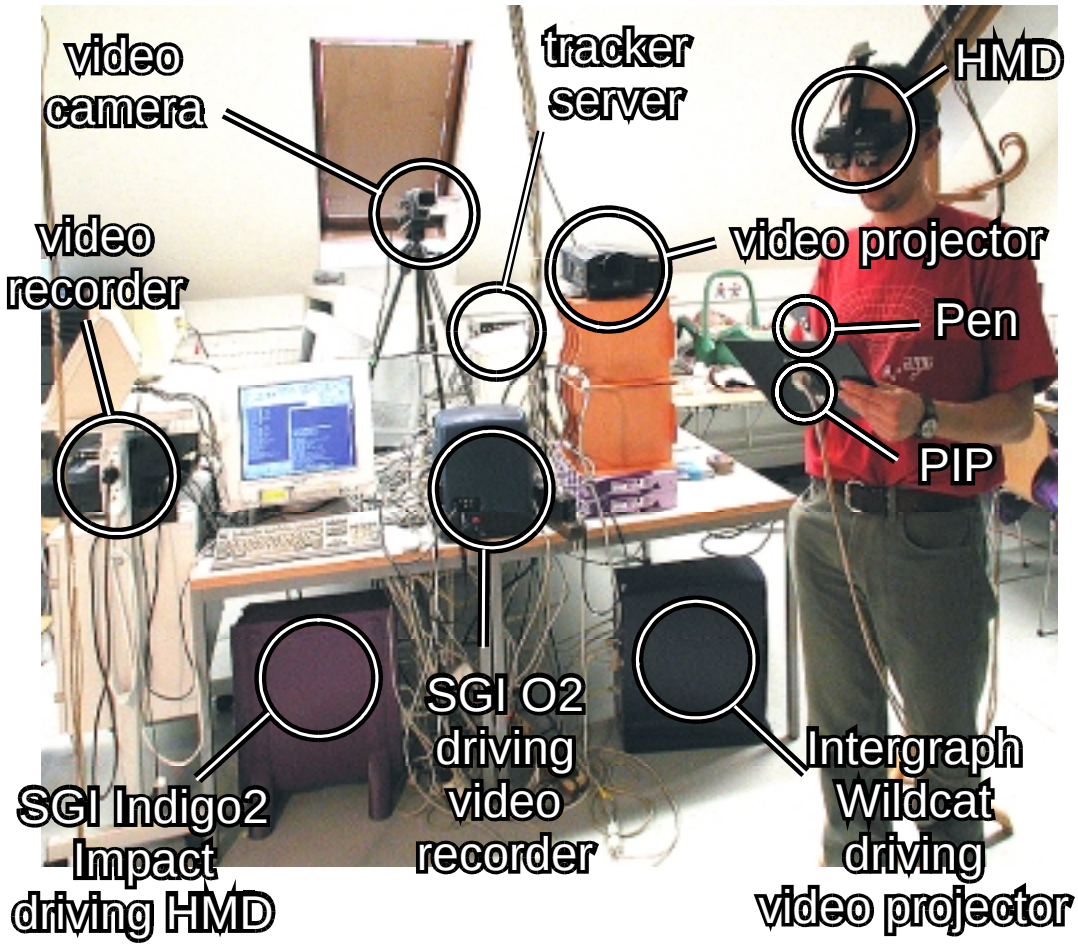


Figure 24: Hardware setup for the heterogeneous display experiment.

The third host – the O2 – was configured to combine the graphical output (monoscopically) from *Studierstube* with a live video texture obtained from a video camera pointed at the users and projection screen. The O2 was configured to render for a virtual user, whose position was identical with the physical camera. This feature was used to document the system on video.
This configuration used two locales, one shared by the two users and the O2, while a separate locale was used for the Intergraph driving the projection screen (again viewed by a virtual user).

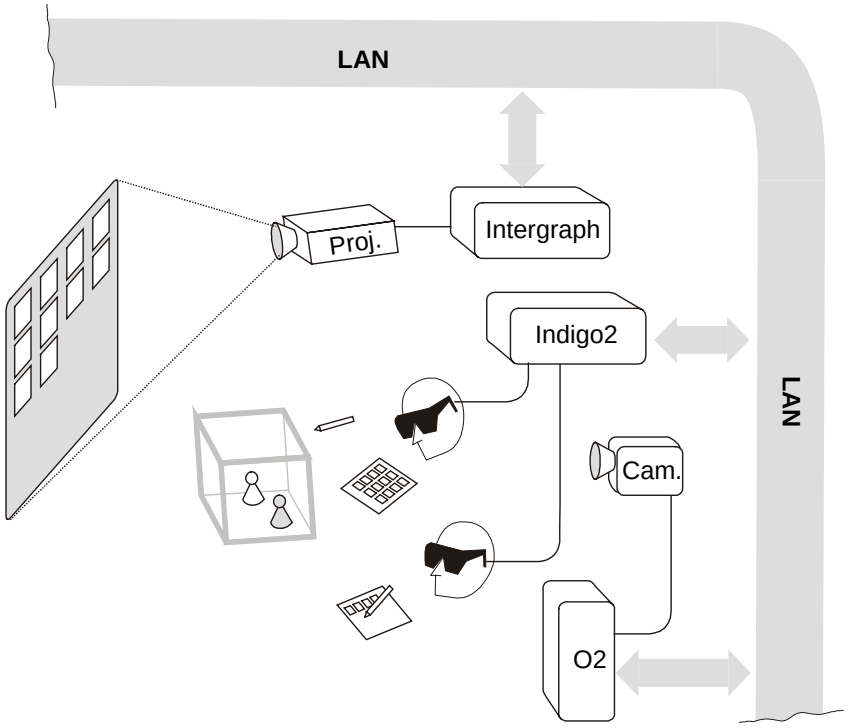


Figure 25: Heterogeneous displays – two users simultaneously see shared graphics (via their see-through HMDs) and a large screen projection

The additional video host allowed us to perform live composition of the users' physical and virtual actions on video, while the video projector driving the projection screen could be freely repositioned without affecting the remainder of the system.

6.7.2 Symmetric workspace

The second example was intended to show multi-user collaboration in pure augmented reality with multiple hosts. The Storyboarding application was executed in a more conventional augmented reality setup consisting of two hosts (Indigo2, Intergraph), two users, and one *locale* (Figure 26).

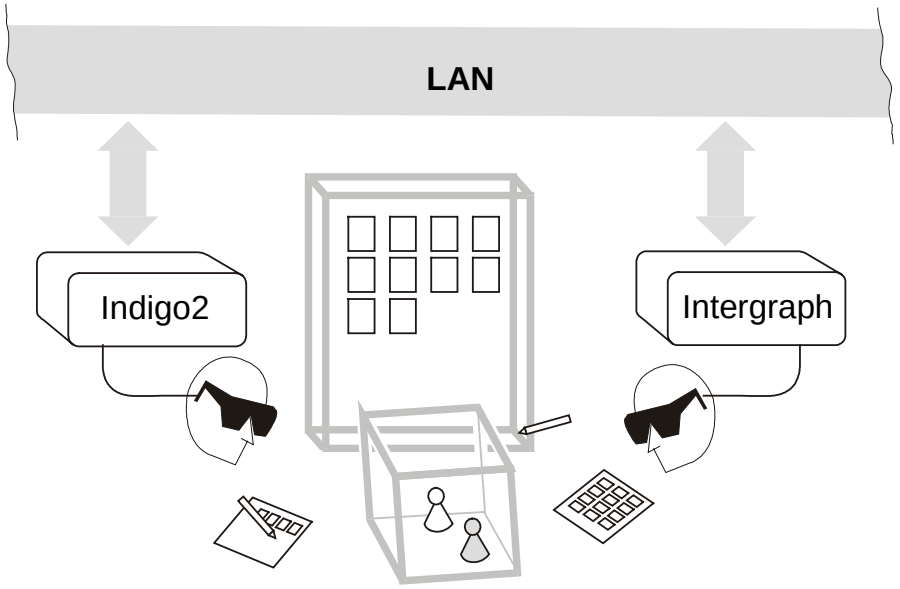


Figure 26: A symmetric workspace configuration uses homogeneous displays (2 HMDs) to present a shared environment to multiple users in a single locale

Both users were wearing HMDs again, but the first user was connected to the Indigo2, while the second user was connected to the Intergraph. In this configuration, the slide show/sorter was included in the graphics shown via the HMD rather than projected by a separate video projector.

While the obtainable frame rate was significantly higher than for the first configuration, since rendering load for the two users was distributed over two hosts, no high resolution wide field-of-view projection was available for the slide show/sorter. Consequently, only a single locale was necessary since both users shared the same physical space.

6.7.3 Remote collaboration

The third example was created to show remote collaboration of multiple users. In this setup, we built a second *Studierstube* environment in the laboratory next door to experiment with the possibilities of remote collaboration. We then let two users collaborate remotely using the Storyboard application.

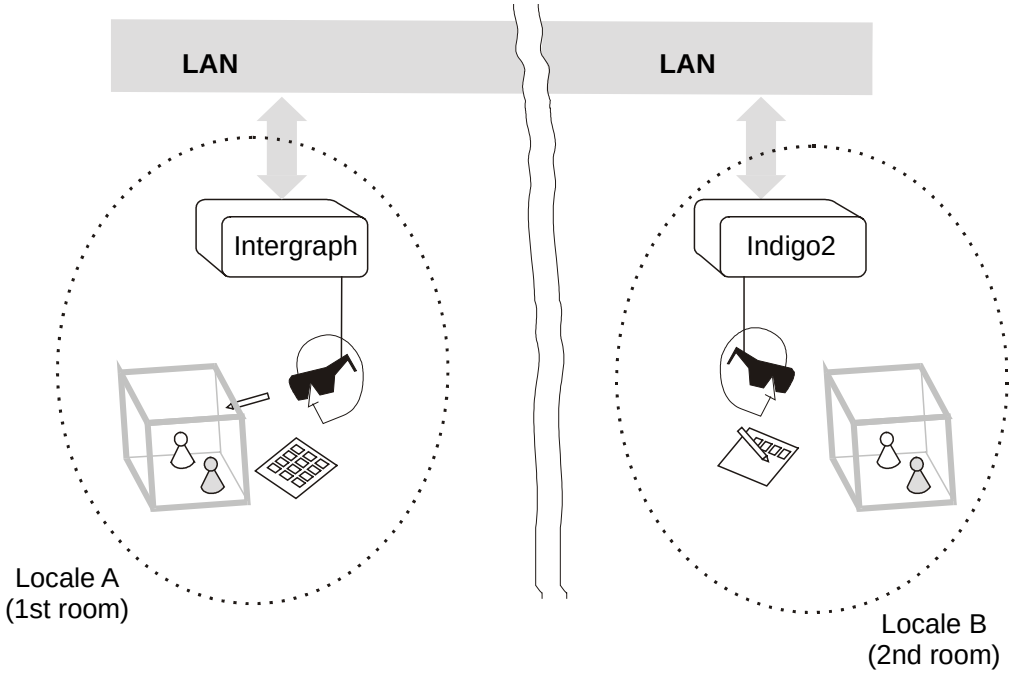


Figure 27: Remote collaboration: Two geographically separated users experience a shared environment

Note that the results are preliminary in the sense that all hosts were connected to the same LAN segment, and network performance is thus not representative of what one would get over a wide area network connection. However, this was not the current focus of investigation.

The system consisted of two hosts (Indigo2 in the first laboratory, O2 in the second), two users and two locales (Figure 27). Each user was wearing a HMD connected to the local workstation. In contrast to configuration from section 6.7.2, two locales were used as the users did not share a physical presence. The sharing of context, but not locale, allowed them to rearrange their personal workspace at their convenience without affecting collaboration.

6.8 Summary

This chapter presented distributed *Studierstube*, a prototype user interface that uses distributed collaborative augmented reality to bridge multiple user interface dimensions: Multiple users, context, and locales as well as applications, 3D-windows, hosts, display platforms, and operating systems. Distributed *Studierstube* supports collaborative work by coordinating a heterogeneous distributed system based on a distributed shared scene graph and a 3D interaction toolkit.

Our implementation prototype shows that despite its apparent complexity, such a design approach is principally feasible, although much is left to be desired in terms of quality and maturity of hard- and software. However, this is out of scope of this thesis.

The next chapter introduces further enhancements to our framework, which enable support and use of dynamically changing user groups as well as load balancing strategies, locales for remote collaboration via WANs and application streaming which is used to support late joiners.

7 Context Migration

7.1 Introduction

In chapter 6 we introduced the distribution enhanced version of Studierstube. The following describes further improvements: Contexts (applications) are able to change their current status (either master or slave). This enables us to implement load balancing strategies and locales for remote collaboration. Furthermore we present a concept for collaboration via WANs (e.g. *the Internet*) and application streaming which is used to support late joiners in our framework.

7.2 Contexts and Migration

As mentioned in section 6.6.4, the roles that contexts may assume affect the status of the context's application part. The application part of a master context is active and modifies context data directly according to the users' input. In contrast, a slave context's application is dormant and does not react to user input. For example, no callbacks are executed if widgets are triggered. Instead, a slave context relies on updates to be transmitted via DIV. When the application part changes the scene graph of the master context, DIV will pick up the change and propagate it to all slave contexts to keep them in sync with the master context. This process happens transparently within the application, which uses only the master context's scene graph.

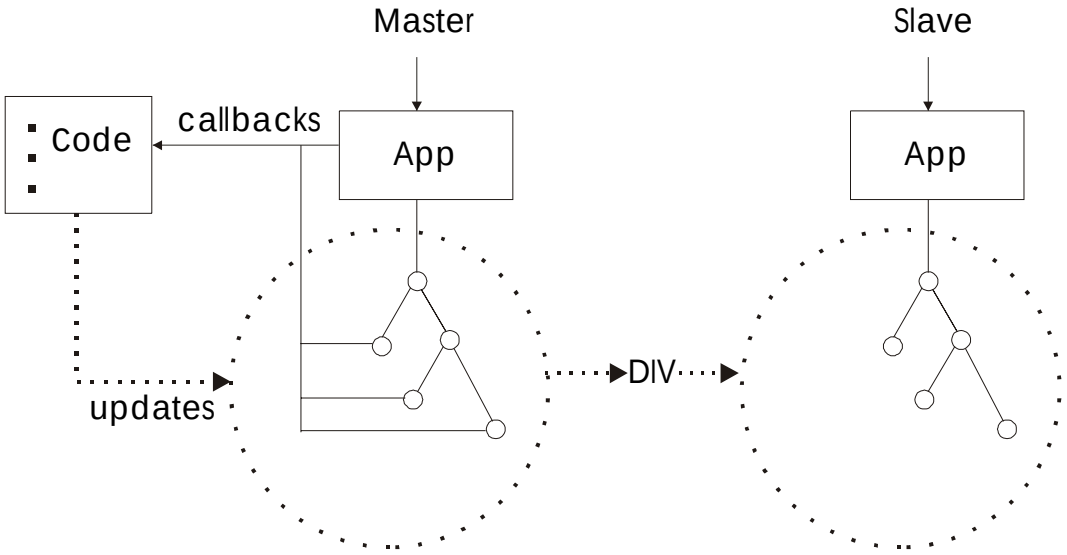


Figure 29: Master and slave context synchronization via DIV updates.

An important feature of context replicas is that they are able to swap roles. That is, a master becomes a slave and vice versa. Note that at any time there may be only one master per replicated context. We use a dedicated process ("session manager") to implement a directory service and a known point of contact for late-joiners. The session manager (sman) maintains a

list of all active hosts and which contexts they own and subscribe to. All communications with the sman process is done via bi-directional reliable data transfer through a dedicated distribution-manager object. Figure 30 depicts the integration of the session manager process into our system.

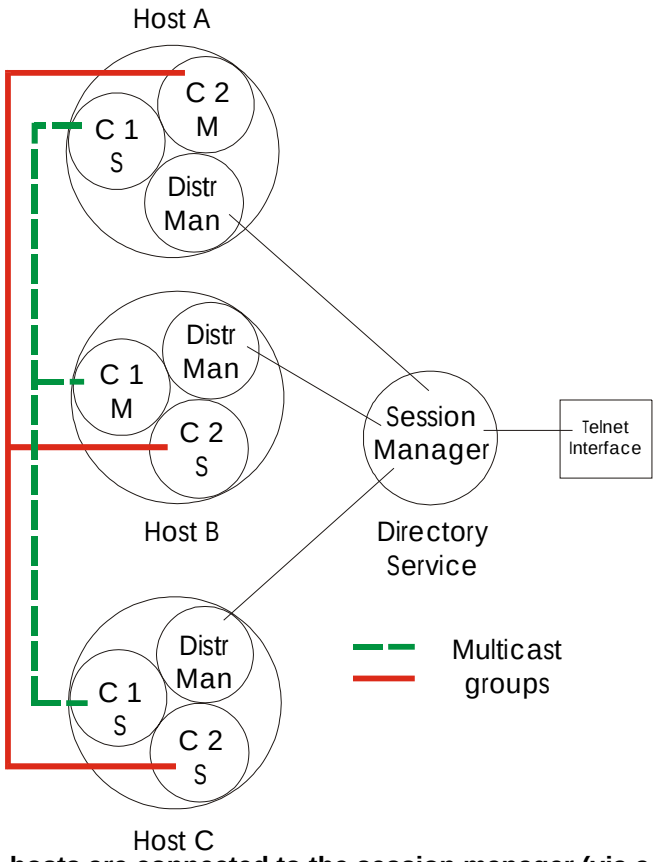


Figure 30: Three hosts are connected to the session manager (via a distribution-manager object). They run two contexts (C1, C2) in either master (M) or slave (S) mode. Through the telnet interface it is possible to perform maintenance tasks or display statistics about the current session. DIV updates are sent via the two multicast groups.

We call the whole process of role swapping migration. We distinguish between two different versions of migration:

- *Application migration* allows to transfer running Studierstube applications from one host to another, maintaining intact the state of the user interface as well as the internal state of the application.
- *Activation migration* is the light-weight variant of application migration: an application executes in a distributed system using a replication mechanism, but the responsibility for the computationally critical portion – called activation – of the application can be handed off from host to host without affecting the application or its user(s).

These tools allow us to address a number of practical issues in user interface and application management for virtual work environments that are otherwise difficult to resolve:

- *Dynamically changing configurations*: Late joining and early exit of users and their respective hosts can be handled by migrating the set of application instances and their activations for each host.
- *Load balancing*: Applications can be migrated from one host to another if the computational load is too high. This mechanism can also account for situations where it is desirable to support asymmetric configurations, e. g., when a powerful compute server is available to offload less capable workstations, or when not all hosts are able to execute a particular application due to platform or hardware constraints.
- *Utilization of heterogeneous network capacity*: Many DVE applications acknowledge variations in network performance by quality of service management and degradation strategies, but make the unrealistic assumption that the networking quality is homogeneous. We observe that small groups of co-located users are also likely to share a high-bandwidth local area network (LAN), while remote collaboration typically relies on a lower performance wide area network (WAN). Activation migration can be used to make the best of both situations simultaneously by allowing finer grained interaction between users sharing a higher bandwidth network.
- *Ubiquitous computing*: Applications can be made to follow a user across physical locations.

7.3 Activation Migration

At any point during the processing of two events by an application instance, the instance‘s master can be changed from one host to another. All that is required is that the application node and its contained sub graph recursively unregister their event callbacks at the old master host, and register callbacks at the new master host. The old master becomes a slave and vice versa; from this moment on the new master host will be responsible for triggering all application specific behavior. This process is transparent to other hosts, the user and even the application itself. Section 7.5 details how activation migration is used to build support for load balancing, early exit, remote group collaboration and even some forms of ubiquitous computing.

7.4 Application migration

Complete application migration requires that a running application instance moves from one host to another, while user interface and internal state are kept intact. This is different to the aforementioned activation migration in that it requires complete transportation of the live application to a host that did not replicate that application instance before (otherwise activation migration would be sufficient).

Building on Studierstube‘s distributed architecture, such application migration is straight forward: All application state is encoded in the scene graph through the application node and its contained sub graph. Marshalling a arbitrary scene graph into a memory buffer is a standard operation of OIV (SoWriteAction). The application is marshalled, so its complete live state – both graphical and internal – is captured in the buffer, and can be transmitted over the network to the target host, where it is unmarshalled (SoDB::readAll) and added to the local scene graph, where it resumes its operation.

For a complete migration, the source host should unregister the application instance‘s event callbacks before migration and delete the application instance after marshalling. Moreover, the destination host must load the application‘s binary object module if not already present in memory, and register the application‘s event callbacks so it can become a master copy.

However, in many cases it is desirable to create a replicated application on both the source and destination host. In this case deletion of the source copy is not performed. The remainder of the procedure depends on whether source or destination are intended to become master, but both is straight forward to accomplish. Section 7.5 describes the implementation of late joining behavior and support for ubiquitous computing using application migration.

7.5 Usage of migration

In this section, we describe the use of our new tools – activation migration and application migration – to implement several interesting behaviors of a distributed virtual work environment.

7.5.1 Load balancing

One straight forward application of activation migration is load balancing. Every host responsible of running a master copy of an application instance must continuously perform

application specific computation as the user generates input events. Even if no interaction is intended, tracking data from the user‘s input devices continues to arrive and must be checked for possibly interactions. These computations need not be performed if a host has only a slave copy, which will be updated only as a consequence of remote computation.

Subdividing the responsibilities for master copies among hosts allows a better utilization of computational resources. However, the set of application instances will change over time as application instances are created and deleted by the users. As a consequence, computational load may become unevenly distributed.

As a countermeasure, we have implemented a simple load balancing mechanism which utilizes activation migration: A session manager which runs as a dedicated process once in the environment is responsible for monitoring the computational load. When it changes due to modifications of the set of application instances, the session manager initiates appropriate activation migration to balance the load. Currently, we have only implemented a very simple load balancing strategy that tries to assign an equal number of master instances to each host. However, an arbitrary policy can be used to decide how to balance the load without changing the underlying mechanism. A simple extension would be to assign weights to application instances depending on their demand for resources and capacity estimates to hosts depending on their processing power. It is also possible to lock activations to specific hosts or group of host, for example if binary modules are available for only some of the used platforms, so not all hosts are technically capable of running master copies of an application.

7.5.2 Late joining

When hosts are added to a Studierstube session after the distributed system is already executing, it is necessary to build a copy of the replicated application instances at the new host. This is easily achieved through the application migration mechanism described in section 7.4 using the variant that does not delete the source application instance. Whether or not the new copy becomes master or slave is determined by the load balancing policy.

7.5.3 Early exit

The opposite operation to late joining of a host is early exit, where one host ceases operation of the distributed system while the remaining hosts continue to execute. In this case, no application migration is necessary, the exiting host simply deletes its application instances. However, any master copies maintained by the exiting host need an activation migration to

one of the remaining host before they are deleted, so one master copy remains available. Again, the target of the activation migration can be determined using load balancing.

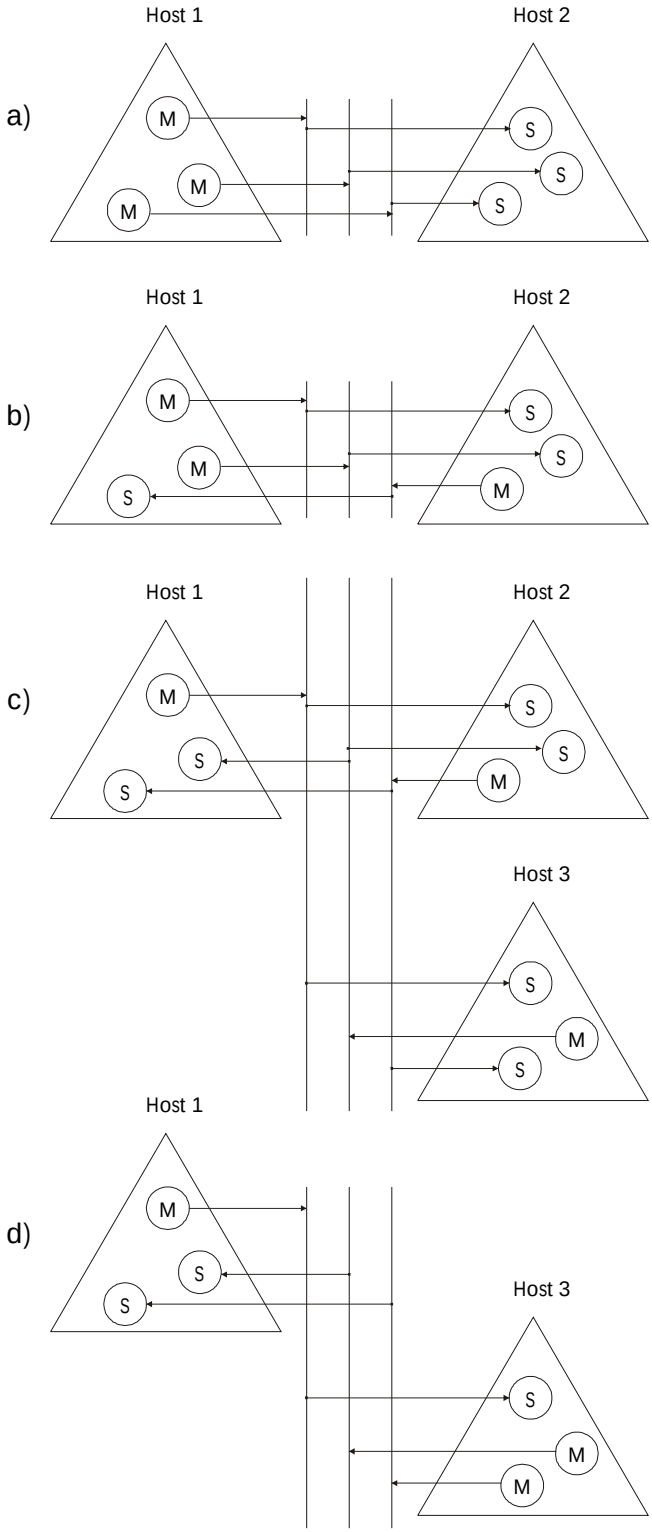


Figure 31: (a) Uneven distribution of load on hosts 1 and 2. (b) Load balancing moves one master privilege to host 2. (c) Host 3 joins late and receives one master privilege from host 1. (d) Host 2 exits early and passed its master privilege to host 3.

7.5.4 Ubiquitous computing

A ubiquitous computing environment allows a user to get access to computing services using a variety of interaction platforms. In our case, we consider only interaction platforms capable of performing 3D interaction with *Studierstube* applications. For example, two non-immersive display platforms (e. g. back-projection table and large desktop monitor) driven by two hosts can be connected using a multi-computer direct manipulation metaphor: By dragging the 3D window that belongs to an application across display boundaries, it can also be migrated (see). This migration can take one of two distinct forms:

1. The application instance was already distributed and shared by the hosts before the manipulation act. Then only the activation needs to migrate to the target host. After this migration, the destination host becomes master, but the application is still distributed and shared.
2. The application is only executing at the source host. Then the manipulation act triggers application migration to the destination host. After that, the application is only executing at the destination host.

The second variant will probably not be used for multi-computer direct manipulation style interaction in most cases, but may be interesting if the display platforms are not physically adjacent. For example, users may want to migrate their applications from office to home and vice versa.

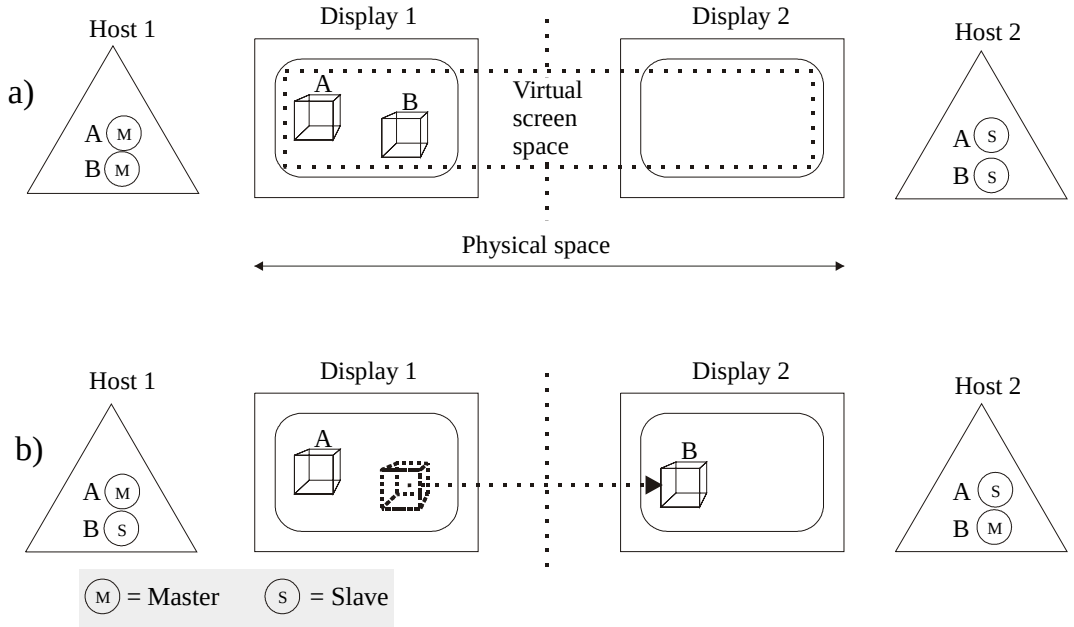


Figure 32: Two hosts sharing a single physical space – when the user moves application windows across display boundaries, the application is migrated along.

7.5.5 Remote collaboration

Remote collaboration can bring together users in a virtual environment that cannot share a physical environment, because they are geographically separated. A reasonable assumption is that the hosts of co-located users share a LAN and can leverage high network performance, which allows for example to share tracking data through multicasting. In contrast, remote collaboration will typically use a WAN with significantly higher latency and lower bandwidth. However, users in separate locations do also not share a geometric frame of reference, so that collaboration will significantly differ from co-located collaboration.

We use these observations to allow for remote collaboration of two (or more) groups of co-located users. It is assumed that user groups at each end of the WAN share a LAN, in which a distributed Studierstube system is executed as previously described. In addition, the two local Studierstube are connected via the WAN through proxy processes that route multicast messages (see Figure 33). While the tracking information is kept separately in each LAN, DIV messages are tunneled through the proxy connection. In that way, replicas of application instances are kept synchronized over the WAN.

However, only users on the side where the application instance‘s master copy resides can interact with the application while the users at the other side of the WAN connection are passive observers (note that they still can choose individual viewpoints, which is a behavior that does not affect the application). This setup can be reversed if users trigger activation migration to move the master copy to the other side of the WAN. This results in a reversal of active and passive user groups. We have chosen to trigger the reversal using a simple click-to-focus event for the 3D window that the application provides. This approach supports also locales at each side (see section 6.4) because the separation of locales from contexts is achieved through the implementation of 3D windows.

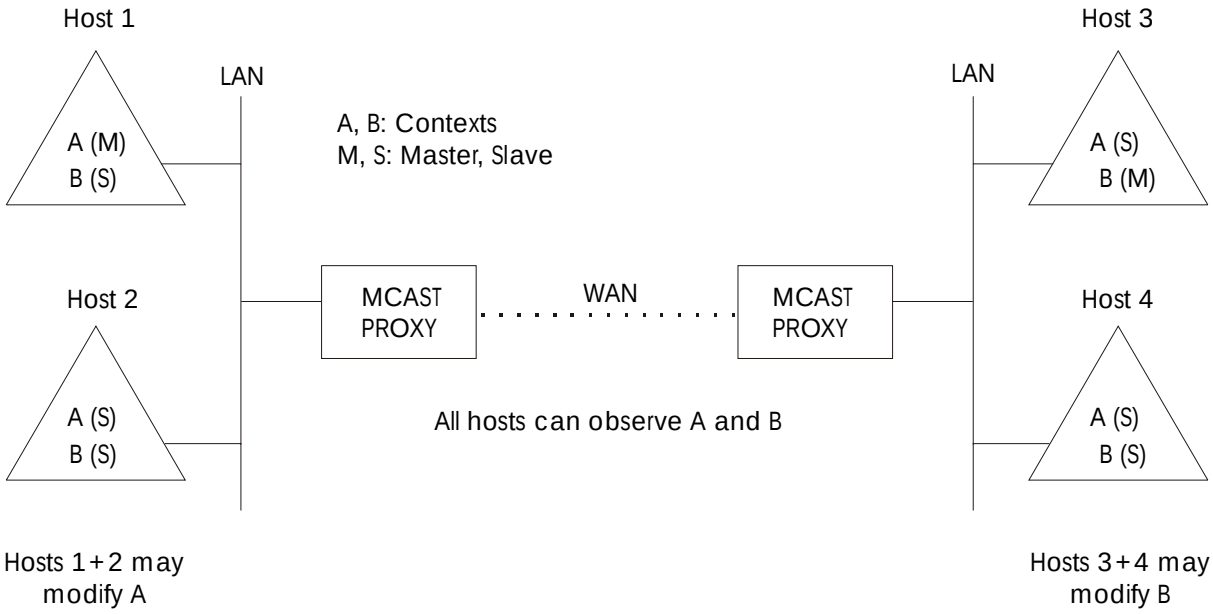


Figure 33: In this scenario, two Studierstube sites are connected remotely. Users at a site owning the master privilege of an application can interact directly with it, users at the slave site can observe but need to request activation migration before interaction.

7.6 Results

The aforementioned new features of Studierstube give us the ability to perform very interesting experiments. We chose three scenarios: Activation migration, locales, and application streaming. For each scenario we defined some experiments which are now described in detail.

7.6.1 Activation migration

We have chosen this scenario to demonstrate our master/slave concept. We used a setup with three hosts (Figure 30) which were located in the same LAN but not in the same room. Host A was an SGI Onyx2 (located in Room A), Host B an SGI Octane and Host C an SGI Visual Workstation PC (both located in Room B). Every host ran a context of a spraying application (Figure 16) and a context of a painting application (Figure 17).

Peer-to-peer load balancing

The first experiment demonstrates load balancing with a peer-to-peer strategy. The session manager uses the following ad-hoc algorithm to perform symmetric load balancing:

1. Compute a load indicator for each host based on a simple formula: load = number-of-master-processes + 0.5 * number-of-slave-processes
2. Assign the master context to the host with the lowest load. Every other hosts get slave privileges for that particular distributed context.

We describe now the realization of this experiment. User A (at host A) starts a context of the test application (“spraying”, Figure 16). The session manager computes the load of each host

and decides that Host A gets master and all other hosts get slave privileges for that context. After some collaboration work User A decides to start another application (painting). Hence the sman process calculates again the load of all hosts and detects that Host B has only a running slave context. Host B gets master and all other hosts slave privileges. We compared the overall frame rate at Host A with a scenario (same setup without load balancing) where Host B (single CPU) managed two master contexts and all other hosts only ones with slave privileges. The result was that with load balancing we gained 30% of rendering performance (frame rate change) at Host B and lost only 20% at host A. This shows that running a mixture of master slave processes is less time-consuming than running multiple master contexts on a single CPU machine. The better performance at Host A results from the fact, that Host A is a multiple CPU machine. Obviously a multiple CPU machine is able to handle master contexts in an optimized way.

Static application server

The next experiment demonstrates load balancing with a *static server* strategy. That is, every new context is started with master privileges on the same pre-defined host, which acts as a so-called application server (every application runs at this host computer with master privileges). We used the same scenario as above but with the *static server* strategy. Host A (SGI Onyx2 with 4 CPUs) acted as our application server. The result was that the frame rate at every host gained 30% compared to the load balancing strategy (see previous experiment). This clearly confirms our observation, that running multiple master contexts at a multiple CPU machine is optimized.

Remote collaboration

The last experiment investigated remote collaboration. We reconfigured our setup and replaced Host B with a PC which was located in another computer lab which was not in the same LAN with Host A and C. Hence Host A and C defined one site and Host B another one. In order to be able to use our reliable multicast mechanism for distribution we tunneled all multicast traffic through interconnected proxies. The focus policy of contexts (and therefore their surrounding 3D-windows) was changed to “click to master”. That is, if someone from a site who does not own master privileges from a specific context clicks into a window the master from the corresponding context at a host at the other site is switched into slave mode and the originator gets master privileges. We used the same scenario as in experiment 1. Host A and B managed one slave and one master context per host. Host C served two slave contexts. After some collaborative work user B clicked into a slave context window and got master privileges. Host A lost the master privileges from that context. Our observation was

confirmed that collaboration through contexts that have a master and a slave copy at the same site is fast.

7.6.2 Locales

To demonstrate our possibility to use locales and remote collaboration we connected Host A from the above experiment to a Virtual Table and repeated the experiment with the same results. This experiment shows our transparent integration of multiple display types.

7.6.3 Application streaming

The last set of experiments is used to demonstrate our application streaming feature. We chose the same setup of experiment 1. After some work with the running applications we added a fourth host (a PC) to the system. Every application was streamed to the new user. The application and PIP-sheet state of the new user's *Studierstube* was correctly distributed and it was possible to collaborate with the others. Another experiment was to test a scenario, where an application follows the user, as he or she changes the current location. We used the Onyx2 from the previous experiments and connected it to a projection wall. The Octane was connected to the Virtual Table and ran a context of the painting application with one user. After some painting the Onyx2 was added to the system and after a commit action from the user (button click), the application was streamed to the Onyx2 while the user was walking to the projection wall. The user was able to present the previous painted object to a large audience, who watched the application on the projection wall.

7.7 Summary

We have presented enhancements to our distributed *Studierstube* system, which is capable of handling multiple users and multiple applications and we have introduced and evaluated two related new tools for moving applications among hosts: light-weight activation migration and application migration through streaming linearized scene graphs. Furthermore we have shown how to use these tools for load balancing, remote collaboration and ubiquitous computing. The next section concludes this thesis and presents some future work.

8 Conclusions and Future Work

Studierstube is a prototype distributed system for building innovative user interfaces that use collaborative augmented reality. It is based on a heterogeneous distributed system based on a shared scene graph and a 3D interaction toolkit. This architecture allows for the amalgamation of multiple approaches to user interfaces as needed: augmented reality, projection displays, ubiquitous computing. The environment is controlled by a two-handed pen-and-pad interface, the Personal Interaction Panel, which has versatile uses for interacting with the virtual environment. We also borrow elements from the desktop, such as multi-tasking and multi-windowing. The resulting software architecture resembles in some ways what could be called an "augmented reality operating system." The work presented in this thesis added many enhancements to the original *Studierstube* system and presented tools that enable the construction of a distributed collaborative augmented reality system. The author wants to stress the point that these tools are not limited to the *Studierstube* system.

We introduced DIV (distributed Open Inventor), which is the major building block of our distributed system. It extends the popular 3D graphics toolkit Open Inventor to enable transparent distribution at the scene graph level. As a by-product this yielded a universal tool to enable distribution in legacy Open Inventor applications (which were not written with distribution in mind). It is planned that this tool is made available to the public under the LGPL (Lesser Gnu Public License). The integrated network layer supports only a simple reliable multicast transmission technique, which may not be suitable for all future needs. To overcome this issue one needs to replace the network layer with an appropriate solution.

We find that the most important enhancement of our system through the addition of application nodes and associated migration tools is the ability to execute complex and experimental distributed user interfaces in a heterogeneous distributed system with little effort.

As observed by Tsao and Lumsden (1997), in order to be successful for everyday productivity work situations, virtual environment systems must allow "multi-tasking" and "multi-context" operation. By "multi-tasking" they mean that the virtual environment can be re-configured to execute a particular application, i. e., there is a separation of VR system software and application software. Multi-context operation goes beyond that by allowing multiple

applications to execute concurrently rather than sequentially. They also point out that this resembles a development earlier experienced for 2D user interfaces, which evolved from single-application text consoles to multi-application windowing systems. It is no surprise that by "judicious borrowing", many useful results from 2D user interfaces become applicable to 3D, as is evident with *Studierstube's* PIP, 3D-windows, or 3D event system.

Our framework is a user interface that uses collaborative augmented reality to bridge multiple user interface dimensions: Multiple users, contexts, and locales as well as applications, 3D-windows, hosts, display platforms, and operating systems. This architecture allows to combine multiple approaches to user interfaces as needed, so that it becomes easy to create a 3D work environment, which can be personalized, but also lends itself to computer supported cooperative work.

With our approach, we can cater for new system requirements (e. g., to support more users or displays) through the addition of a new workstation that seamlessly fits into the already existing pool. Using an appropriate load balancing policy that uses the mechanisms presented in this thesis, we can accommodate a large variety of system requirements from a limited hardware pool. While we do not claim unbound scalability, we found our system design very useful for the small group collaboration we are investigating.

8.1 Discussion

The CRYSTAL system from (Tsao & Lumsden, 1997) presents the use of "multi-tasking" and "multi-context" operations in a virtual environment system. It does not incorporate true multi-user operation, and consequently has no need for multiple locales. Figure 34 extends the taxonomy from CRYSTAL. For example, MIT's mediaBlocks (Ullmer et al., 1998) allow a user to work with different manipulators, which are dedicated devices for specific applications, and the mediaBlocks themselves are a very elegant embedding for context data. However, although principally possible, no multi-user scenarios were demonstrated.

In contrast, SPLINE (Barrus et al., 1996) is designed towards multi-user interaction. While SPLINE completely immerses a user in a purely virtual world and thus does not meet our definition of a work environment, it features multiple locales that correspond to activities (for example, chat takes place in a street café, while train rides take place on a train).

The closest relative to our work is Columbia's EMMIE (Butz et al., 1999). Except for explicit support of locales, EMMIE shares many basic intentions with our research, in particular concurrent use of heterogeneous media in a collaborative work environment. Like ourselves, the authors of EMMIE believe that future user interfaces will require a broader design approach integrating multiple user interface dimensions before a successor to the desktop metaphor can emerge.

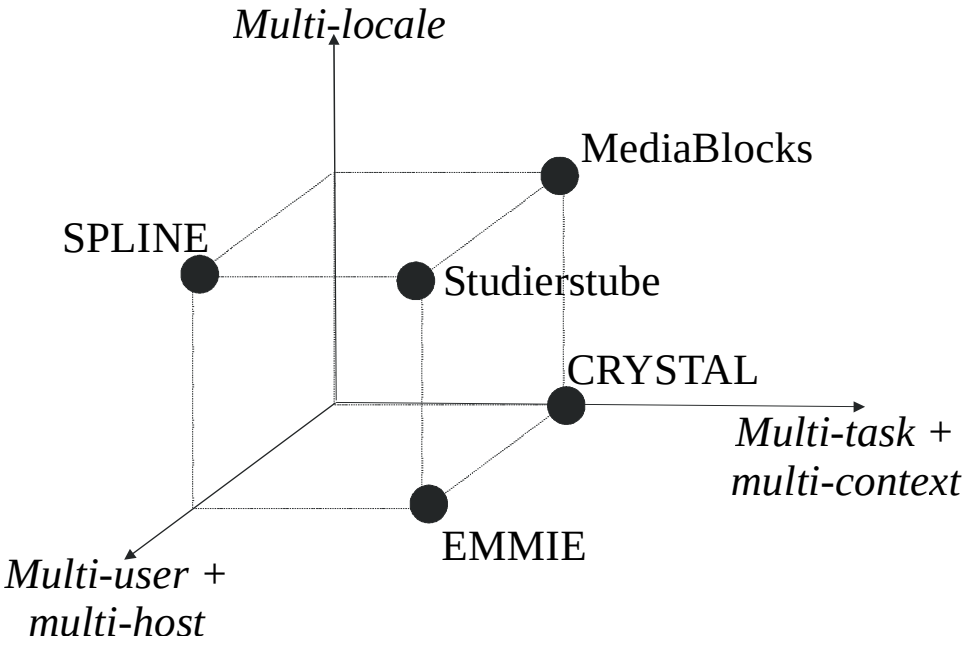


Figure 34: Extended taxonomy for multiple dimensions of user interfaces with some related work (adapted from CRYSTAL). In the original taxonomy, multi-tasking and multi-context were considered separately, which is not necessary. Instead, handling of multiple locales is considered.

The presented ideas are generally useable and are in no way limited to the *Studierstube* platform. The presented approaches to bridge multiple user interface dimensions could be utilized in AR and VR systems to build more general solutions for everyday work environments. The distributed shared scene graph approach (implemented as DIV), which is based on the OIV toolkit is able to transparently turn legacy OIV applications (which were written with no distribution in mind) into distributed ones. Furthermore, the presented technique is not limited to scene graphs. It can be applied to different hierarchical data structures. For example to share and dynamically update distributed HTML pages. Another application could be to use this technique to enable transparent distribution of XML based applications. It would ease distribution of applications and eliminate common mistakes. Because XML is often used as a scripting language to configure a system, it could then be used to transparently synchronize several configuration files in a distributed system.

The migration and load-balancing approaches are very powerful tools to enhance the scalability of systems. One example that comes to mind is the combined usage of some of the presented approaches (DIV, migration tools, load-balancing) to build a CAVE system with several PC workstations. This would yield a scaleable CAVE system in a lower price range. Note that PC workstations are very powerful commodity items, but unlike expensive high-end systems such as SGI Onyx2, they are usually not very scalable.

Future work will use application migration in the context of mobile AR: Users may enter or leave *Studierstube* sessions at any time with their mobile AR equipment, or meet for instantaneous collaboration. A leaving user takes (copies of?) running applications onto the road, and a new user may share running applications with others.
And always remember: “Avatars do it in cyberspace”.

9 References

(Abrams et al., 1998) Abrams H., Watsen K., Zyda M. Three Tiered Interest Management for Large-Scale Virtual Environments, Proceedings of VRST '98, Taipei, Taiwan, November 1998.

(Agrawala et al., 1997) Agrawala M., Beers A., Fröhlich B., McDowall I., Hanrahan P., Bolas M. The Two-User Responsive Workbench: Support for Collaboration Through Individual Views of a Shared Space, Proc. SIGGRAPH '97, pp. 327-332, 1997.

(Barrus et al., 1996) Barrus, J., R. Waters, R. Anderson. Locales and Beacons: Precise and Efficient Support for Large Multi-User Virtual Environments. Proc. VRAIS '96, pp. 204-213, 1996.

(Ben-Natan, 1995) Ben-Natan, R. CORBA: A Guide to the Common Object Request Broker Architecture, McGraw Hill, 1995.

(Billinghurst et al., 1998a) Billinghurst M., Bowskill J., Jessop M., Morphett J. A Wearable Spatial Conferencing Space, Proc. ISWC '98, pp. 76-83, 1998.

(Billinghurst et al., 1998b) Billinghurst M., Weghorst S., Furness III T.: Shared Space: An Augmented Reality Approach for Computer Supported Collaborative Work, Virtual Reality: Virtual Reality - Systems, Development and Applications, 3(1), pp. 25-36, 1998.

(Birman, 1993) Birman, K. The process group approach to reliable distributed computing. Communications of the ACM, 36(12):37-53, December 1993.

(Birrel et al., 1993) Birrell, A., Nelson, G., Owicki, S., and Wobber, E. Network objects. In *Proc. 14th* ACM Symp. on Operating Systems Principles, 1993.

(Blau et al., 1992) Blau, B., Hughes, C. E., Moshell, M. J., and Lisle, C. Networked virtual environments. In *Proc. 1992 ACM Symp. on Interactive 3D Graphic*s, pp. 157–164, 1992.

(Bricken & Coco, 1994) Bricken, W, Coco, G. The VEOS project. Presence: Teleoperators and Virtual Environments, 3(2):111–129, 1994.

(Broll, 1998) Broll W. DWTP: An Internet Protocol for Shared Virtual Environments. Proc. of the 3rd ACM Symposium onVirtual Reality Modeling Language (VRML '98), pp. 49-56, February 1998.

(Bryson, 1993) S. Bryson. The Virtual Wind Tunnel. SIGGRAPH '93 Course, No. 43, pp. 2.1-2.10

(Butz et al., 1998) Butz A., C. Beshers, S. Feiner. Of Vampire Mirrors and Privacy Lamps: Privacy Management in Multi-User Augmented Environments, Proc. ACM UIST '98, pp. 171-172, Nov. 1998.

(Butz et al., 1999) Butz A., Höllerer T., Feiner S., MacIntyre B., Beshers C.. Enveloping Computers and Users in a Collaborative 3D Augmented Reality, Proc. IWAR '99, pp. 1999.

(Carlsson & Hagsand, 1993) Carlsson, C., and Hagsand, O. DIVE – A platform for multi-user virtual environments. Computers & Graphics 17(6):663-669, 1993.

(Cruz-Neira et al., 1993) Cruz-Neira C., Sandin D. J., DeFanti T. A. Surround-Screen Projection-Based Virtual Reality: The Design and Implementation of the CAVE. In Proceedings of SIGGRAPH `93, pp. 135-142, 1993.

(Das et al., 1997) Das, T. K., Singh, G., Mitchell, A., Kumar, P. S., McGhee, K. NetEffect: A Network Architecture for Large-Scale Multi-User Virtual Worlds. In Proc. of the ACM Symposium on Virtual Reality Software and Technology (VRST'97), 157-164, 1997.

(Encarnação et al., 1999) L. M. Encarnação, A. Stork, D. Schmalstieg, O. Bimber. The Virtual Table – A Future CAD Workspace. Proceedings of the 1999 CTS (Autofact) Conference, Detroit MI, Sept. 1999.

(Feiner & Beshers, 1990) Feiner S., C. Beshers. Worlds Within Worlds: Metaphors for Exploring N-Dimensional Virtual Worlds, Proc. UIST '90, pp. 76-83, 1990.

(Frecon & Stenius, 1998) Frécon, E, and Stenius M. DIVE: A Scaleable network architecture for distributed virtual environments. Distributed Systems Engineering Journal (special issue on Distributed Virtual Environments), 5(3), 91-100, Sept. 1998.

(Funkhouser, 1995) Funkhouser, T. RING: A Client-Server System for Multi-User Virtual Environments. 1995 Symposium on Interactive 3D Graphics, 85- 92, April 1995.

(Gelernter, 1992) Gelernter, D. Mirror worlds. Oxford University Press, 1992.

(Goldberg & Robson, 1983) Goldberg A., D. Robson. Smalltalk-80: The language and its implementation. Addison-Wesley, Reading MA, 1983.

(Greenhalgh & Benford, 1995) Greenhalgh C., Benford, S. Virtual reality tele-conferencing: Implementation and experience. In Proc. of the Third European Conference on Computer Supported Cooperative Work (ECSCW '95), Stockholm, 1995.

(Grimsdale, 1991) Grimsdale, C.: dVS Distributed Virtual Environment System. Computer Animation, virtual reality, Visualisation 1991; Blenheim Online, Pinner, Middlesex, pp.163-170, 1991.

(Hartling et al., 2001) Hartling P., Just C., Cruz-Neira, C. Distributed Virtual Reality Using Octopus. Proc. of IEEE Virtual Reality 2001, pp. 53-62, March 13-17, Yokohama, Japan, 2001.

(Hesina et al., 1999) Hesina G., D. Schmalstieg, A. Fuhrmann, W. Purgathofer. Distributed Open Inventor: A Practical Approach to Distributed 3D Graphics. Proc. VRST '99, London, pp. 74-81, Dec. 1999.

(Höllerer et al., 1999) Höllerer T., S. Feiner, T. Terauchi, G. Rashid, D. Hallaway. Exploring MARS: Developing indoor and outdoor user interfaces to a mobile augmented reality system, Computers & Graphics, 23(6), pp. 779-785, 1999.

(IEEE, 1993) IEEE standard for information technology – protocols for distributed simulation applications: Entity information and interaction. IEEE standard 1278-1993. New York: IEEE Computer Society, 1993.

(Ishii and Ullmer, 1997) Ishii H., Ullmer B. Tangible Bits: Towards Seamless Interfaces between People, Bits and Atoms, Proc. CHI '97, pp. 234-241, 1997.

(Kazman, 1993) Kazman R. Making WAVES: On the design of architectures for low end distributed virtual environments. In Proc. IEEE VRAIS '93, pp. 443–449.

(Krüger et al., 1995) W. Krüger, C. Bohn, B. Fröhlich, H. Schüth, W. Strauss, and G. Wesche. The Responsive Workbench: A Virtual Work Environment. IEEE Computer, 28(7):42-48, 1995.

(Lea et al., 1997) Lea R., Y. Honda, K. Matsuda, S. Matsuda. Community Place: Architecture and Performance. Proceedings of ACM VRML'97, pp. 41-50, 1997.

(Levelt et al., 1992) Levelt, W. G., Kaashoek, M. F., Bal H. E., and Tanenbaum, A. S. A Comparison of Two Paradigms for Distributed Shared Memory. Software - Practice and Experience, 22(11), 985-1010, Nov. 1992.

(Macedonia et al., 1995) Macedonia M.R., Zyda M.J., Pratt D.R., Brutzman D.P., and Barham P.T.: Exploiting Reality with Multicast Groups - A Network Architecture for Large Scale Virtual Environments. Proceedings of the IEEE Virtual Reality Annual International Symposium (VRAIS'95), 1995.

(MacIntyre & Feiner, 1998) MacIntyre, B., and Feiner, S. A Distributed 3D Graphics Library. SIGGRAPH 98 Conference Proceedings, Annual Conference Series, 361-370, 1998.

(Mann, 1997) Mann S. Smart Clothing: The Wearable Computer and WearCam, Personal Technologies, 1(1), Springer-Verlag, March 1997.

(Miller & Thorpe, 1995) Miller D., Thorpe J. A. SIMNET: The advent of simulator networking. In Proc. of the IEEE 83(8):1114-1123, August 1995.

(Mine et al., 1997) M. Mine, F. Brooks Jr., C. Sequin. Moving Objects in Space: Exploiting Proprioception In Virtual-Environment Interaction. Proc. SIGGRAPH '97, pp. 19-26, 1997.

(Obeysekare et al., 1996) U. Obeysekare et al.: Virtual Workbench - A Non-Immersive Virtual Environment for Visualizing and Interacting with 3D Objects for Scientific Visualization. Proceedings of Visualization `96, pp. 345-350, 1996.

(Oliveira et al., 1999) Oliveira M., Crowcroft J., Brutzman D., Slater M. Components for Distributed Virtual Environments, Proc. VRST '99, London, pp., Dec. 1999.

(Origin, 1997) Origin. Ultima Online, online computer game, 1997. URL: http://www.owo.com/.

(Pang & Wittenbrink, 1997) Pang, A., and Wittenbrink, C. Collaborative 3D Visualization with CSpray. IEEE Computer Graphics & Applications, 17(2), 32-41, 1997.

(Pandzic et al., 1995) Pandzic, I., Çapin, T., Magnenat Thalmann, N., Thalmann, D. VLNET: A Networked Multimedia 3D Environment with Virtual Humans. Proc. Multi-Media Modeling MMM `95, Singapore, pp.21-32, November 1995.

(Pausch et al., 1995) Pausch R., T. Burnette, D. Brockway, M. Weiblen. Navigation and Locomotion in Virtual Worlds via Flight into Hand-Held Miniatures, Proc. SIGGRAPH ’95, pp. 399-401, 1995.

(Pope, 1989) Pope A. The SIMNET network and protocols. Technical Report 7102. Cambridge, MA: BBN Systems and Technologies, July 1989.

(Rekimoto, 1997) Rekimoto J. Pick-and-Drop: A Direct Manipulation Technique for Multiple Computer Environments, Proc. UIST ‘97, pp. 31-39, 1997.

(Rekimoto, 1998) Rekimoto J. A Multiple Device Approach for Supporting Whiteboard-based Interactions, Proc. CHI ‘98, pp. 344-351, 1998.

(Rohlf & Helman) Rohlf, J., and Helman, J. IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics. In Proc. ACM SIGGRAPH ‘94, 381-394, 1994.

(Rubin & Brain, 1999) Rubin, W., and Brain, M. Understanding DCOM. Prentice Hall PTR, 1999, ISBN 0-13-095966-9.

(Scheifler & Gettys, 1983) Scheifler R. W., Gettys J. The X window system. ACM Transactions on Graphics, 16(8):57-69, August 1983.

(Schmalstieg et al., 1996) Schmalstieg D., A. Fuhrmann, Zs. Szalavári, M. Gervautz. Studierstube - Collaborative Augmented Reality, Proc. Collaborative Virtual Environments ‘96, Nottingham, UK, Sep. 1996.

(Schmalstieg et al., 1999) Schmalstieg D., L. M. Encarnação, Zs. Szalavári. Using Transparent Props For Interaction With The Virtual Table, Proc. SIGGRAPH Symposium on Interactive 3D Graphics ‘99, pp. 147-154, Atlanta, GI, April 1999.

(Schmalstieg et al., 2000) Schmalstieg D., Fuhrmann A., Hesina G., Szalavári Zs., Encarnação L. M., Gervautz M., Purgathofer W.: The Studierstube Augmented Reality Project. To appear in: "Augmented Reality: The Interface is Everywhere", SIGGRAPH 2001 Course Notes, Los Angeles CA, USA, ACM Press, August 2001. Available as technical report TR-186-2-00-22 (ftp://ftp.cg.tuwien.ac.at/pub/TR/00/TR-186-2-00-18Paper.pdf), Vienna University of Technology, December 2000.

(Shaw et al., 1993) Shaw, C., Green, M., Liang, J., and Sun, Y. Decoupled Simulation in Virtual Reality with the MR Toolkit. ACM Transactions on Information Systems, 11(3):287-317, 1993.

(Shaw & Green, 1993) Shaw, C., and Green, M. The MR Toolkit peers package and experiment. In Proc. of VRAIS '93, 463-469, 1993.

(Singh et al., 1995) Singh G., Serra L., Png W., Wong A., Ng H. BrickNet: Sharing Object Behaviors on the Net. In Proc. IEEE VRAIS '95, pp. 19–25, 1995.

(Singhal and Zyda, 1999) Singhal S., Zyda M. Networked Virtual Environments, Addison-Wesley, New York, NY, 1999. ISBN: 0201325578

(Smith and Mariani, 1997) Smith G., J. Mariani. Using Subjective Views to Enhance 3D Applications, Proc. VRST '97, pp. 139-146, New York, NY, Sep. 1997.

(Snowdon & West, 1994) Snowdon, D., and West, A. AVIARY: Design Issues for Future Large-Scale Virtual Environments. Presence, 3(4), 288-308, 1994.

(Sony, 1999) Sony Corporation. Everquest, online computer game, 1999. URL: http://www.everquest.com/.

(Steed et al., 1999) Steed, A., Frecon, E., Avatare, A., Pemberton, D., Smith, G. The London Travel Demonstrator. Proc. VRST '99, London, pp. 50-57, Dec. 1999.

(Strauss & Carey, 1992) Strauss, P. S., and Carey, R. An Object-Oriented 3D Graphics Toolkit, In Computer Graphics (Proc. ACM SIGGRAPH '92), 341-349, Aug, 1992.

(Sun, 1998) Sun Microsystems. Java Remote Method Invocation - Distributed Computing for Java. March 1998. URL: http://java.sun.com/marketing/collateral/javarmi.html.

(Sun, 1988) Sun Microsystems. Remote Procedure Call Protocol Specification. Network Working Group RFC1050, April 1988.

(Szalavári & Gervautz, 1997) Szalavári Zs., M. Gervautz. The Personal Interaction Panel - A Two-Handed Interface for Augmented Reality, Computer Graphics Forum, 16(3), pp. 335-346, Sep. 1997.

(Szalavári et al., 1998a) Szalavári, Z., Eckstein, E., and Gervautz, M. Collaborative Gaming in Augmented Reality. Proceedings of VRST' 98, 195-204, Taipei, Taiwan, Nov. 2-5, 1998.

(Szalavári et al., 1998b) Szalavári Zs., A. Fuhrmann, D. Schmalstieg, M. Gervautz. Studierstube - An Environment for Collaboration in Augmented Reality, Virtual Reality - Systems, Development and Applications, 3(1), pp. 37-49, 1998.

(Tramberend, 1999) Tramberend, H. Avango: A Distributed Virtual Reality. IEEE Virtual Reality, 1999. [Avocado is now known as Avango]

(Tsao & Lumsden, 1997) Tsao J., C. Lumsden. CRYSTAL: Building Multicontext Virtual Environments, Presence, 6(1), pp. 57-72, 1997.

(Ullmer et al., 1998) Ullmer B., Ishii H., Glas D. mediaBlocks: Physical Containers, Transports, and Controls for Online Media, Proc. SIGGRAPH '98, pp. 379-386, July 1998.

(Waters et al., 1997) Waters, R., Anderson, D., Barrus, J., Brogan, D., Casey, M., McKeown, S., Nitta, T., Sterns, I., and Yerazunis, W. Diamond Park and Spline: Social Virtual Reality with 3D Animation, Spoken Interaction and Runtime Extendability. Presence, 6(4), 461-481, 1997.

(Watsen & Zyda, 1998) Watsen K. and Zyda M., Bamboo - A Portable System for Dynamically Extensible, Real-Time, Virtual Environments, Proceedings of the 1998 Virtual Reality Annual International Symposium (VRAIS '98), IEEE, Atlanta, GA, March 1998.

(Weiser, 1991) Weiser M. The Computer for the twenty-first century. Scientific American, pp. 94-104, 1991.

(Wloka & Greenfield, 1995) M. Wloka, E. Greenfield: The Virtual Tricoder: A Uniform Interface for Virtual Reality. Proceedings of ACM UIST'95, pp. 39-40, 1995.

(Zeleznik et al., 2000) Zeleznik, B., Holden., L., Capps, M., Abrams, H., and Miller, T. Scene-Graph-As-Bus: Collaboration between Heterogeneous Stand-alone 3-D Graphical Applications. Eurographics 2000, August 2000

(Zyda et al., 1992) Zyda, M. J., Pratt, D. R., Monahan, J. G., and Wilson, K. P. NPSNET: Constructing a 3D Virtual World. In Proc. 1992 ACM Symposium on 3D Graphics, 147-156, March 1992.

10 Appendix

Curriculum Vitae

Gerd Hesina

Göttschach 56

A-2632 Grafenbach

AUSTRIA

hesina@cg.tuwien.ac.at

1974	Born on the 29th of June in Vienna, Austria
1980-1984	Primary School at Volksschule, 1110 Vienna, Molitorgasse 11
1984-1986	Secondary School at BRG 11, 1110 Vienna, Gottschalkgasse 21
1986-1988	Secondary School at BRG Neunkirchen, 2620 Neunkirchen
1988-1993	Department of Engineering and Electronics at HTL Wiener Neustadt
June 1993	Graduation from HTL Wiener Neustadt
1989-1993	Worked as freelance programmer
October 1993	Begin of studies at the Technical University of Vienna, major field computer science
1996	Started working on diploma thesis: "A Network Library for Multi-User Virtual Environments"
1997	Finished thesis
June 1997	Degree: Diplom-Ingenieur der Informatik (scM)
October 1997	Started working on ph.D. thesis ("Distributed Collaborative Augmented Reality") as research assistant at the Institute of Computer Graphics at the Vienna University of Technology
May 2001	*Finished thesis*