Distributed Applications for Collaborative Augmented Reality

Dieter Schmalstieg
Vienna University of Technology, Austria
email: schmalstieg@ims.tuwien.ac.at

Gerd Hesina
Vienna University of Technology, Austria
email: hesina@cg.tuwien.ac.at

Abstract

This paper focuses on the distributed architecture of the collaborative augmented reality system Studierstube. The system allows multiple users to experience a shared 3D workspace populated by multiple applications using see-through head mounted displays or other presentation media such as projection systems. The system design is based on a distributed shared scene graph that alleviates the application programmer from explicitly considering distribution, and avoids a separation of graphical and application data. The idea of unifying all system data in the scene graph is taken to its logical consequence by implementing application instances as nodes in the scene graph. Through the distributed shared scene graph mechanism, consistency of scene graph replicas and the contained application nodes is assured. Multi-user 3D widgets allow concurrent interaction with minimal coordination effort from the application. Special interest is paid to migration of application nodes from host to host allowing dynamic workgroup management, such as load balancing, late joining and early exit of hosts, and some forms of ubiquitous computing.

1 Introduction

In contrast to most distributed virtual environments (DVEs) and networked games (e. g., [16, 23] that are based on an egocentric *virtual world* metaphor, collaborative augmented reality allows multiple co-located users to experience a shared *virtual workspace* through see-through head mounted displays (HMDs) or projection environments. A virtual workspace makes natural communication as well as interaction with both virtual and real objects is possible. This approach fits well into a conventional office environment that is augmented with heterogeneous media, as exemplified in UNC's office of the future [26] and Columbia's EMMIE [8] projects.

In previous work [31] on the *Studierstube* system (Figure 1), we have demonstrated how a collaborative augmented reality system can provide convergence of several aspects of user interfaces:

- Multiple users can be accommodated simultaneously;
- multiple applications can be used concurrently by multiple users or alternatingly by a single user (applications as a set of complementary tools)

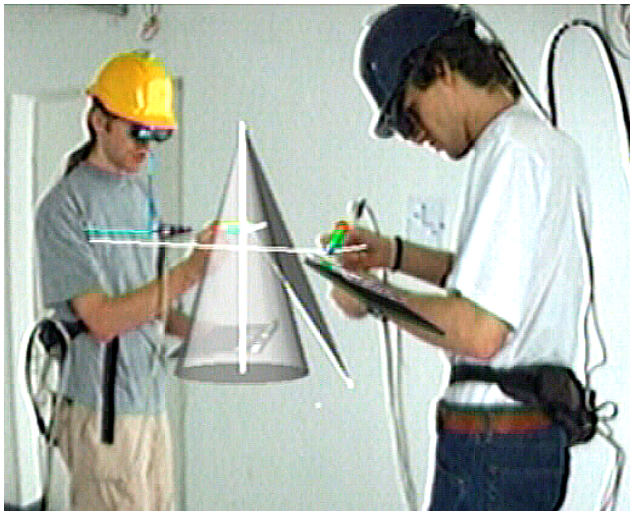

Figure 1: The distributed virtual workspace Studierstube supports multiple users and applications using tracked head mounted displays and hand-held tracked props. The image shows two users engaged in a geometry education task (live video overlay).

- multiple heterogeneous input and output media (e. g., HMD vs. desktop display) can be used to accommodate several user interface styles.

In this work, we focus on the design of *Studierstube's* underlying distributed system, which tries to accommodate the networking requirements of a virtual workspace. It manages a distributed shared scene graph that hides the details of networking from the application programmer. A key contribution is the unification of all application specific graphical and non-graphical data in the scene graph through the implementation of application instances as nodes in the scene graph. Such application nodes are distributed through the same mechanism as conventional scene graph nodes. Applications also rely heavily on multi-user 3D widgets, which ensure consistency but also enhance responsiveness through controlled consistency relaxation. A second key contribution is application node migration which allows dynamic workgroup management, in particular late joining, early exit, load balancing and some degree of ubiquitous computing [41]. Please note that the presented distributed system techniques were designed for face-to-face collaboration in collaborative augmented reality, but are applicable to any kind of distributed virtual environment.

2 Related work

Synchronous groupware and distributed virtual environments have much in common in terms of user requirements, but technical solutions have sometimes surprisingly little overlap. DVEs typically try to minimize communication costs at the expense of generality by specialized protocols and minimal sharing of application state [32]. In contrast to DVEs, synchronous groupware tries to introduce collaborative tools to a conventional 2D desktop environment, which requires a more general approach to distribution. In particular, collaboration transparent systems try to provide shared use of applications that were originally intended for a single user, following a WYSIWIS ("what you see is what I see") [35] paradigm.

Later relaxed variants of WYSIWIS were introduced that allow users to share individual windows rather than the complete desktop, or set an individual non-shared viewpoint for a specific window [34, 25, 18]. This development is mirrored in the DVE area in concepts such as subjective views [33], privacy widgets [7], or even dead reckoning techniques [20]. While users of relaxed WYSIWIS can suffer from a lack of mutual location awareness and have to use tools like telepointers [30, 34], collaborative augmented reality allows users to truly share a 3D space, providing excellent location awareness.

Building collaboration aware applications that have true multi-user interface elements should be only "slightly harder" than building conventional applications, or application programmers will be reluctant to do so [30]. In object oriented frameworks, a feasible approach is therefore to provide components (widgets) that have built-in collaboration facilities, and can readily be (re-)used by application programmers or even retro-fitted to legacy applications [3]. Our framework offers similar possibilities through application nodes and multi-user 3D widgets.

In both DVE and groupware literature there is a continued debate over centralized vs. replicated architectures. Replication is often associated with better performance because processing can be carried out locally at every host and is available immediately without going through a possibly congested network first. In fact, for real-time rendering local availability of graphical models is compulsory. However, a pure replicated architecture makes it difficult to deal with non-deterministic and time-dependent application behavior, which causes replicated state to diverge. Some applications optimistically neglect this issue, while others impose object locking [25] or floor control [19] schemes. Regardless of mechanism, the price for consistency is paid by some additional network load and latency. Therefore, the key to a successful implementation lies in choosing the right trade-offs for the given application, and the most promising schemes are often hybrids, e. g., [14, 15, 24]. The architecture presented in this paper also tries to exploit domain specific properties using a hybrid approach.

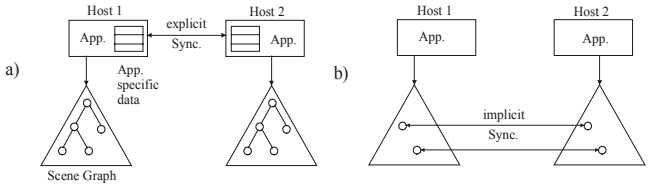

Figure 2: (a) Traditional distributed virtual environments separate graphical and application state, and synchronize only application state. (b) A distributed shared scene graph achieves replication that is transparent to the application.

Besides static workgroup topology, some research also considers dynamic changes to the workgroup and client migration [4, 10], in particular accommodation of late-comers that need to be updated on the current state of the session. Two competing solutions are replaying all previous events to the newcomer vs. transmitting a current image of application state. Because the history of previous events can become arbitrarily large despite potential for compression [9], recent work favors the image copy approach [37]. This is partly due to novel architectures that make it easy to marshal complex runtime structures [2], and is also the foundation for our application migration facility. It should be noted, however, that this kind of migration in a constrained runtime environment is not comparable to full operating system level process migration.

Finally, several projects on collaborative user interfaces inspired our work. SharedSpace [5] features collaborative augmented reality, but is limited by its lack of an underlying distributed system. CRYSTAL introduces multi-tasking to virtual environments [38]. The closest relative to our approach is EMMIE [8], which provides a similar platform, but does not include dedicated application management. Other prominent collaborative user interfaces, such as mediaBlocks [39] or multi-computer interaction [28, 29] anticipate many of our goals, but do not incorporate stereoscopic 3D graphics.

3 Distributed system architecture

3.1 Distributed shared scene graph

Current high-level 3D graphics libraries are engineered around the concept of a *scene graph*, a hierarchical object-oriented data structure of graphical objects. 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 systems separate application state from the graphical objects. The application state is then distributed, while the graphical objects are kept locally (Figure 2a).

This allows custom solutions that optimize network utilization through minimal sharing of application state. In groupware systems, which also rely on sharing of data structures, the separation of graphical and application state

is considered beneficial because it allows independent handling of core application state and graphical "view", which can be exploited for relaxed WYSIWIS [15].

However, this design has two distinct disadvantages: Additional effort is spent on keeping application state and graphical objects synchronized (called "dual database problem" in [21]), and the distribution is not fully transparent to the application developer, who may even be forced to actively send synchronization messages in some replication schemes. In our opinion, this makes it more than "slightly harder" to build a collaborative application.

An alternative solution recently popularized by a number of research projects (DIVE [13], Repo-3D [21], Avango [37], SGAB [42]) 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 (Figure 2b). 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, Distributed Open Inventor (DIV) [17] is based on the popular Open Inventor (OIV) toolkit [36]. It utilizes OIV's notification mechanism to automatically trigger an "observer" callback whenever an application changes something in the observed scene graph similar to [18]. These changes are then propagated to all scene graph replicas using reliable multicast.

3.2 Input processing

Most distributed architectures assume a remote collaboration situation where one user is equivalent to one host with designated input and output facilities, and network bandwidth is uniformly scarce. The face-to-face collaboration we are considering is quite different to this assumption implicit in both groupware and DVE applications. For example, the collaborative session in Figure 1 uses one host and HMD per user, but has a dedicated server for the magnetic tracking system that processes input for all users. Also consider a virtual workspace for design reviews, composed of a large curvilinear "dome" display driven by projections from three networked workstations, with input for multiple users coming from an optical tracker connected to one of the hosts. Neither of these configurations is symmetric, and input, output or hosts cannot be directly assigned to users.

The general assumption of groupware systems and many DVE systems that user input is available at a user's local host does not apply to these situations. Instead, real-time rendering depends on the input data to be delivered to all hosts quickly, in particular for head tracking. The problem is further acerbated by the fact that unlike input from a keyboard and mouse, tracking for multiple users produces a substantial amount of data. Fortunately, the co-located setup of users allows us to assume a high-performance local area network (LAN) in which such data can be efficiently distributed via multicast. Because of high update frequency and idempotent semantics, simple and fast unreliable multicast is sufficient for our purposes.

By comparison, "output" events propagating changes to the shared database after application processing generate only a small volume of data by comparison, but require reliable distribution to prevent replicated state from diverging. However, as graphical state is already replicated via the distributed shared scene graph, a simple reliable multicasting scheme is sufficient and scales reasonably well. This situation stands in gross contrast to shared windowing systems based on centralized design, where all graphical state is considered as application output and must be distributed, which is problematic for such systems.

3.3 Application objects

All DVE platforms, even dedicated end-user applications such as current computer games incorporate require some kind of extension mechanism. 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. Some approaches take this idea to the extreme by only providing a kernel capable of loading extensions [40, 22].

Studierstube [31] uses object-oriented runtime extension through subclassing. New node classes for OIV are loaded and registered with the system on the fly. Using this mechanism, we can take the scene graph based approach that avoids a dual database (graphical + application data) to its logical consequence by *embedding applications as nodes in the scene graph*. Applications in *Studierstube* are not written as monoliths linked with a runtime library, but as new *application classes* that derive from a base application node. Application classes are loaded as binary objects on the fly during system execution, and instances of application objects are embedded into the scene graph. Naturally, multiple application nodes can be present in the scene graph simultaneously, which allows convenient multitasking. Surprisingly, we are not aware of any other extension mechanism that uses this particular approach.

Application classes are derived from an application 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.

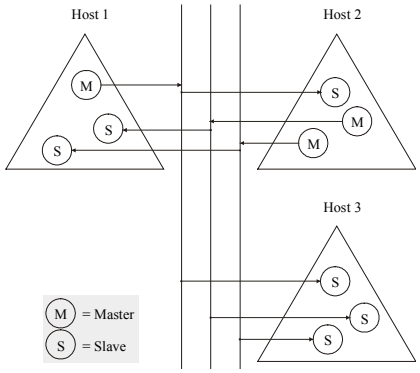

Figure 3: Replicated application instances embedded in a distributed shared scene graph run in master mode at exactly one host and in slave mode at all other hosts.

As the scene graph is distributed, so are the applications embedded in it. A newly created application instance will be added to all replicas of a scene graph, and will therefore be distributed. With the application node all data contained in attributes will be replicated – a sub scene graph of graphical objects, but also attributes that are not visible objects but represent other application data. Non-graphical attributes are simply added as additional "fields" of the application node that do not directly contribute to rendering. We have found this unified treatment of graphical and non-graphical data to drastically simplify application development.

This has the advantage that application specific computations, typically callbacks triggered by 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. Using this scheme, application specific computation is distributed over the workgroup.

This approach shares the a significant advantage of centralized DVE systems [11, 14, 23]: serialization of updates is implicitly performed, which removes the need for a special consistency protocol and simplifies distribution semantics.

At the same time, 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 (Figure 3). 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 replicated DVE systems [20], where hosts manage the locally controlled entity and remote entities are represented as "ghosts" [6].

3.4 Event processing

Like most interactive systems, *Studierstube* works event-driven, i. e., user input – usually through tracked props – is translated into 3D events.

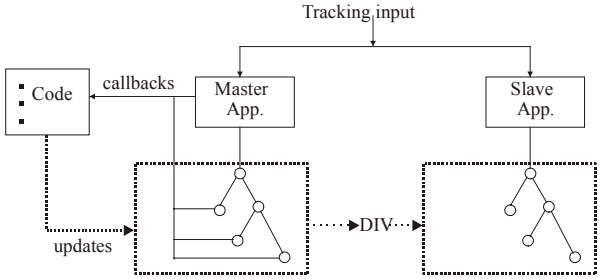

Figure 4: A user's interactions trigger callbacks that modify the scene graph. Changes are propagated to remote replicas through DIV.

Nodes in the scene graph express interest in events through registering callbacks with the system, which are triggered as events are cascaded into the scene graph by the runtime system and consumed by nodes as appropriate. Because it is a regular node in the scene graph, an application nodes receives events without additional measures.

However, an application is not a leaf in the scene graph, but rather a group node that manages an application specific sub graph, usually the content appearing in the 3D window and a set of application controls mapped to hand-held props. Many of the nodes contained in this application-specific sub graph are themselves event-aware widgets (section 0) that autonomously respond to user input in possibly complex ways, for example using gesture recognition [12]. The application node itself is mostly responsible for higher level functions – managing its scene graph –, while most of the interaction callbacks are deferred to contained widgets.

As pointed out above, only the master copy of a replicated application instance needs to perform application specific computation (Figure 4). 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 application's 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 network messages that are automatically created when a node's state changes.

In addition to reactive behavior triggered by event, an application can also be proactive, for example to service independently animated objects. Fidelity of this feature is limited to time-triggered "tick" and "idle function" processing by Open Inventor's runtime model, but nevertheless works well within the distributed framework: Changes to the shared scene graph that occur through independent processing of a master application are immediately communicated to the slaves. As only the master application performs the proactive computations, computing capacity is preserved and no inconsistencies can occur.

3.5 Multi-user 3D widgets

The system architecture as outlined above allows implementations of simple collaborative applications, but does not address two important issues:

- Concurrent input of multiple users to one application
- 3D direct manipulation such as dragging creates excessive "output" updates that congest the network

Let us first consider concurrent input. As pointed out above, input from multiple users is available at any host. It is therefore up to the application to consider appropriate multi-user behavior. To ease development, the 3D widget nodes available in *Studierstube's* interaction library have reasonable default behavior. Often per-widget locking is sufficient – for example, it usually does not make sense to allow multiple users to drag an object simultaneously into opposite directions. Other behaviors may be more specific – for example, a color selector may store one selected color per user. This does not even imply relaxed consistency, as the states for all users can be stored separately in the widget. Local variations will only be produced in the final rendering depending on which user the rendering is intended for. In general, such multi-user behavior will be encapsulated within the widget, so an application programmer need not be concerned with it.

3D widgets are also useful to address the second problem: When a user directly manipulates an object, a large amount of output events will be generated as the scene graph is modified at the frequency of tracking events. The responsible master will then try to notify the slaves of all these updates and flood the network. To overcome this issue, consistency of affected widget attributes is temporarily relaxed [21]. Rather than linking the widget's attributes using network messages, master and slave widgets both perform local computation directly from user input, which can lead to slight deviations of state. This may be seen as a generalized form of dead reckoning. Like dead reckoning, it remains the master widgets responsibility to ensure that all replicas are finally synchronized, usually through periodic correction updates. This scheme significantly reduces the amount of update messages sent and improves the system's scalability without affecting long-term consistency.

Again, the internal workings of 3D widgets are shielded from the application programmer. Moreover, a protocol that ensures consistency of all widget replicas despite temporary deviations can be built entirely from the system's capability of updating individual attributes of nodes without requiring access to lower level networking. It is therefore straight forward for an application programmer to add new widgets that use these advanced features.

4 Migration

A workgroup of hosts executing a collaborative session should be able to accommodate dynamic changes, for example, provide the current state of applications to late-comers. In this section, we describe migration mechanisms that allow migration of applications within the workgroup.

4.1 Activation migration

It is straight forward to implement a light-weight form of application migration that we call *activation* migration: At any point between the processing of two events by an application instance, the instance's master can be changed from one host to another. This has many similarities with migration of serialization objects in replicated objects systems such as [1]. All that is required is that the master 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 5 details how activation migration is used to build support for load balancing, early exit, and even some forms of ubiquitous computing.

4.2 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).

Since all application state is encoded in the scene graph, marshalling an arbitrary application into a memory buffer becomes a standard operation of OIV (SoWriteAction). The application's complete live state – both graphical and internal – is captured in a buffer, and can be transmitted over the network to the target host (using a reliable TCP connection), where it is unmarshaled (SoDB::readAll) and added to the local scene graph, so it can resume operation.

To complete migration, the source host must 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 (the binary must either be available at the destination host, e. g., via a shared file system, or must be sent along with the marshaled application). The destination host then registers the application's event callbacks so it can become a master copy. Alternatively, both copies can be kept, and either can be master.

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 in a distributed virtual workspace. We tested most of our implementation using a workgroup of three hosts located in the same LAN.

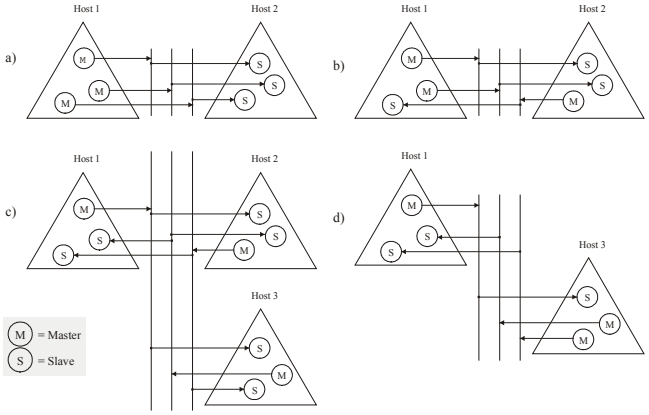

Figure 5: (a) Uneven distribution of load on hosts 1 and 2. (b) Load balancing moves one master privilege to host 2. (c) Host 3 joins late and receives one master privilege from host 1. (d) Host 2 exits early and passed its master privilege to host 3.

The workgroup consisted of Host A, an SGI Onyx2, host B, an SGI Octane, and host C an SGI Visual Workstation PC. Every host ran instances of a "spraying" application and a "painting" application shown in Figure 8.

5.1 Load balancing

One straight forward application of activation migration is load balancing. Every master copy of an application instance places load on a host from input processing and proactive computations. Even if no interaction is intended, tracking data from the user's input devices continues to arrive and must be checked for possible interactions. These computations need not be performed if a host has only a slave copy. Subdividing the responsibilities for master copies among hosts allows a better utilization of computational resources. As the set of application instances changes over time, computational load may become unevenly distributed.

As a countermeasure, we have implemented a simple load balancing mechanism which utilizes activation migration (Figure 5a, b). A session manager which runs as a dedicated process once in the environment is responsible for monitoring the computational load. When the load 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 load to each host based on a simple ad-hoc weighting of applications: load = M * number of master + S * number of slaves (we used M=1, S=1/2).

In our test, User A (at host A) started an instance of the test application (spraying). The session manager computed the load of each host and assigned master privileges to host A and slave privileges to hosts B and C. After some work, user A started another application (painting), and application load was recomputed. As host B executed only a slave instance, it was assigned master privileges for the new instance.

Figure 6: A game of chess with a mobile user after spontaneous application streaming

We compared the overall frame rate with a scenario without load balancing where host B managed all master instances. 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. Load balancing was successfully able to offload the less capable single-CPU host B. For comparison, we also let host A (the 4-CPU Onyx2) manage all master copies, which resulted in an approximate 30% performance increase at all hosts. Note that having dedicated servers for particular applications can also be useful in heterogeneous environments where binaries are not available for all platforms.

5.2 Late joining and early exit

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 (Figure 5c). This is easily achieved through the application migration mechanism described in section 4.1. Whether or not the new copy becomes master or slave is determined by the load balancing policy.

For early exist of a host, its master copies need activation migration to one of other hosts, so they remain available (Figure 5d). Again, the target of the activation migration can be determined using load balancing.

In recent work described elsewhere [27], we used dynamic streaming of applications to accommodate mobile augmented reality users (powered by a Dell Notebook) that spontaneously connect to a stationary *Studierstube* environment via wireless LAN. Figure 6 shows a mobile and a stationary user playing chess after the chess application has been streamed to the mobile user.

5.3 Ubiquitous computing

A ubiquitous computing environment allows a user to get access to computing services using a variety of interaction platforms. A simple form of this concept is multi-computer direct interaction [28]. 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 Figure 7).

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 destination host. After this migration, the destination host becomes master and the source nodes becomes slave, 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.

We tested multi-computer direct interaction in a different setup with two PC workstations with adjacent desktop displays. Interaction was performed via a tracked stylus (Figure 8). In this demonstration (shown in the accompanying video), a user could interact with both hosts/displays in seamless way.

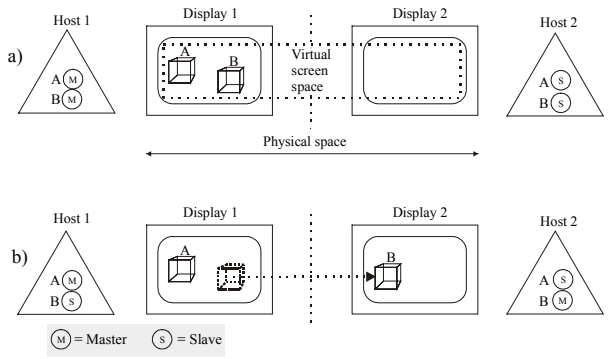

Figure 7: a) Two hosts sharing a single physical space. b) when the user moves application windows across display boundaries, the application is migrated along.

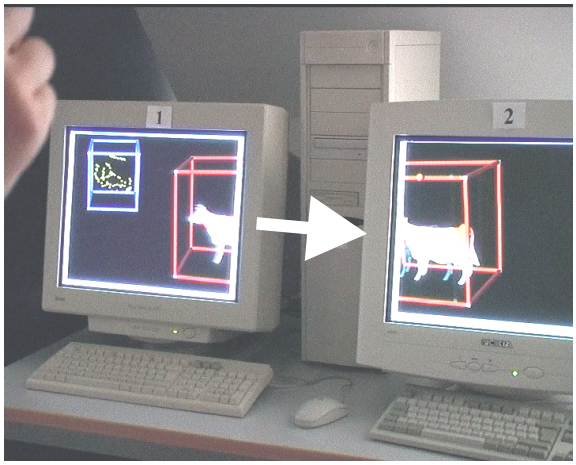

Figure 8: Application migration through dragging one of multiple 3D application windows across display and host boundaries

Both monitors were showing different portions of the virtual environment corresponding to their physical position (see also Figure 7). Using the stylus, the user could interact with applications on either display, and naturally drag and drop 3D-windows between displays. Activation migration followed an application to whatever display it was dragged to, and was visualized using a simple graphical load monitor.

6 Conclusions and future work

We have presented a distributed virtual workspace capable of handling multiple users and applications. It is based on a distributed shared scene graph. Applications are embedded as application nodes in the scene graph and thus implicitly distributed using a hybrid distribution scheme. Applications can be moved among host using light-weight activation migration or through streaming linearized scene graphs. We have shown how to use these tools for workgroup management, load balancing, and ubiquitous computing.

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. PC workstations are very powerful commodity items, but unlike high-end system such as SGI Onyx2, they are usually not very scalable. 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 paper, we can accommodate a large variety of system requirements with 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.

Future work will use application migration for fully mobile AR: Users may leave *Studierstube* sessions at any time with their mobile AR equipment, and meet for instantaneous collaboration anywhere. A leaving user takes (copies of?) running applications onto the road, and a new user may share running applications with others.

Acknowledgments

This project was sponsored by the Austrian Science Fund (*FWF*) under contract P-12074-MAT and P-14470-INF. Special thanks to Anton Fuhrmann for many fruitful discussions, to Klaus Dorfmüller-Ulhaas for his help with the video, to all of the *Studierstube* development team, in particular Rainer Splechtna, Jan Prikryl and Gerhard Reitmayr, and to M. Eduard Gröller for his spiritual guidance.

Web information: **http://www.studierstube.org/**

References

1. Bal H., M. Kaashoek, A. Tanenbaum (1990). Orca: A Language for Parallel Programming of Distributed Systems. IEEE Transactions on Software Engineering, Vol. 18, No. 3, pp. 190-205.
2. Begole J., C. Struble, C. Shaffer, R. Smith (1997). Transparent Sharing of Java Applets: A Replicated Approach. Proc. ACM User Interface Software and Technology (UIST'97), pp. 55-64.
3. Begole J., M. Rosson, C. Shaffer (1999). Flexible collaboration transparency: Supporting worker independence in replicated application-sharing systems. ACM Transactions on Computer-Human Interaction, 6(2), pp. 95-132.
4. Bharat K. A., L. Cardelli (1995). Migratory Applications. Proc. ACM User Interface Software and Technology (UIST'95), pp. 133-142.
5. Billinghurst M., H. Kato, (1999). Collaborative Mixed Reality. Proc. International Symposium on Mixed Reality (ISMR'99), Yokohama, Japan.
6. Blau B., C. Hughes, M. Moshell, C. Lisle, (1992). Networked virtual environments. Proc. 1992 ACM Symposium on Interactive 3D Graphics, pp. 157–164.
7. Butz A., C. Beshers, S. Feiner (1998). Of Vampire Mirrors and Privacy Lamps: Privacy Management in Multi-User Augmented Environments. Proc. ACM User Interface Software and Technology (UIST'98), pp. 171-172.
8. Butz A., T. Höllerer, S. Feiner, B. MacIntyre C. Beshers (1999). Enveloping Computers and Users in a Collaborative 3D Augmented Reality, Proc. International Workshop on Augmented Reality (IWAR'99), pp. 35-44.
9. Chung G., K. Jeffay, H. Abdel-Wahab (1993). Accommodating late-comers in shared window systems. IEEE Computer, 26(1), pp.72-74.
10. Chung G., P. Dewan (1996). A mechanism for supporting client migration in a shared window system. Proc. ACM Symposium on User Interface Software and Technology (UIST'96), pp. 11-20.
11. Das T., G. Singh, A. Mitchell, P. Kumar, K. McGhee (1997). NetEffect: A Network Architecture for Large-Scale Multi-User Virtual Worlds. Proc. ACM Symposium on Virtual Reality Software and Technology (VRST'97), pp. 157-164.
12. Encarnação L. M., O. Bimber, D. Schmalstieg, S. Chandler (1999). A Translucent Sketchpad for the Virtual Table Exploring Motion-based Gesture Recognition. Computer Graphics Forum (Proc. EUROGRAPHICS'99), Milano, Italy, pp. 277-286.
13. Frécon E, M. Stenius (1998). DIVE: A Scaleable network architecture for distributed virtual environments. Distributed Systems Engineering Journal, 5(3), pp. 91-100.
14. Funkhouser T. (1995). RING: A Client-Server System for Multi-User Virtual Environments. ACM Symposium on Interactive 3D Graphics, pp. 85- 92.
15. Graham T, T. Urnes, R. Nejabi (1996). Efficient distributed implementation of semi-replicated synchronous groupware. Proc. ACM User Interface Software and Technology (UIST'96), pp. 1-10.
16. Greenhalgh C., S. Benford (1995). MASSIVE, A Collaborative Virtual Environment for Teleconferencing. ACM Transactions on Computer-Human Interaction, 2(3), pp. 239- 261.
17. Hesina G., D. Schmalstieg, A. Fuhrmann, W. Purgathofer (1999). Distributed Open Inventor: A Practical Approach to Distributed 3D Graphics, Proc. ACM Virtual Reality Software and Technology (VRST'99), London, pp. 74-81.
18. Isenhour P., J. Begole, W. Heagy, C. Shaffer (1997). Sieve: A Java-based collaborative visualization environment. Late Breaking Hot Topics, Proc. IEEE Visualization '97, Phoenix, AZ, pp. 13-16.
19. Lauwers J., T. Joeseph, K. Lantz, A. Romanow (1990). Replicated Architectures for Shared Window Systems: A Critique. Proc. ACM Office Information Systems (COIS'90), Cambridge, MA, pp. 249-260.
20. Macedonia M., M. Zyda, D. Pratt, P. Barham, S. Zeswitz (1994). NPSNET: A Network Software Architecture for Large Scale Virtual Environments. Presence: Teleoperators and Virtual Environments, 3(4), pp. 265-287.
21. MacIntyre B., S. Feiner (1998). A Distributed 3D Graphics Library. Proc. SIGGRAPH'98, pp. 361-370.
22. Oliveira M., J. Crowcroft, D. Brutzman, M. Slater (1999). Components for Distributed Virtual Environments, Proc. ACM Virtual Reality Software and Technology (VRST'99), London, pp. 176-177.
23. Origin (1997). Ultima Online. Commercial online computer game, http://www.owo.com/.
24. Patterson J, M. Day, J. Kucan (1996). Notification servers for synchronous groupware. Proc. ACM Computer Supported Cooperative Work (CSCW'96), pp. 122-129.
25. Prakash A., H. Shim (1994). DistView: Support for building efficient collaborative applications using replicated objects. Proc. ACM Computer Supported Cooperative Work (CSCW'94), pp. 153-164
26. Raskar R., G. Welch, M. Cutts, A. Lake, L. Stesin, H. Fuchs (1998). The office of the future: A unified approach to image-based modeling and spatially immersive displays. Proc. SIGGRAPH'98, pp. 179-188.
27. Reitmayr R., D. Schmalstieg (2001). Mobile Collaborative Augmented Reality. Proc. ACM and IEEE International Symposium on Augmented Reality (ISAR'01), New York, pp. 114-123.
28. Rekimoto J (1997). Pick-and-Drop: A Direct Manipulation Technique for Multiple Computer Environments, Proc. ACM User Interface Software and Technology (UIST'97), pp. 31-39.
29. Rekimoto J., M. Saitoh (1999). Augmented surfaces: A spatially continuous work space for hybrid computing environments. Proc. ACM Conference on Human Factors in Computing Systems (CHI'99), pp. 378-385.
30. Roseman M., S. Greenberg (1996). Building Real-Time Groupware with GroupKit, A Groupware Toolkit. ACM Trans. Computer-Human Interaction, 3(1), pp. 66-106.
31. Schmalstieg D., A. Fuhrmann, G. Hesina (2000). Bridging Multiple User Interface Dimensions with Augmented Reality. Proc. International Symposium on Augmented Reality (ISAR'00), Munich, Germany, pp. 20-30.
32. Singhal S., M. Zyda (1999). Networked Virtual Environments, Addison-Wesley, New York NY.
33. Smith G., J. Mariani (1997). Using Subjective Views to Enhance 3D Applications, Proc. ACM Virtual Reality Software and Technology (VRST '97), pp. 139-146.
34. Stefik M, D. Bobrow, G. Foster, S. Lanning, D. Tatar (1987). WYSIWIS Revised: Early Experiences with Multi-User Interfaces. ACM Trans Office Information Systems, 5(2), pp. 147-167.
35. Stefik M., G. Foster, D. Bobrow, K. Kahn, S. Lanning, L. Suchmann (1987). Beyond the Chalkboard: Computer Support for Collaboration and Problem Solving in Meetings. CACM 30(1), pp. 32-47.
36. Strauss P., R. Carey (1992). An Object-Oriented 3D Graphics Toolkit. Proc. SIGGRAPH'92, pp. 341-349.
37. Tramberend, H (1999). Avocado: A Distributed Virtual Reality Framework. Proc. IEEE Virtual Reality '99.
38. Tsao J., C. Lumsden (1997). CRYSTAL: Building Multicontext Virtual Environments. Presence, 6(1), pp. 57-72.
39. Ullmer B., H. Ishii, D. Glas (1998). mediaBlocks: Physical Containers, Transports, and Controls for Online Media. Proc. SIGGRAPH'98, pp. 379-386.
40. Watsen K. M. Zyda (1998). Bamboo - A Portable System for Dynamically Extensible, Real-Time, Virtual Environments. Proc. Virtual Reality Annual International Symposium (VRAIS'98), pp. 252-259.
41. Weiser M (1991). The Computer for the twenty-first century. Scientific American, 265(3), pp. 94-104.
42. Zeleznik B., L. Holden, M. Capps, H. Abrams, T. Miller (2000). Scene Graph As Bus: Collaboration between Heterogeneous Stand-alone 3-D Graphical Applications. Proc. EUROGRAPHICS 2000, pp. 91-98.