Distributed Applications for Collaborative Three-Dimensional Workspaces

Dieter Schmalstieg, Gerhard Reitmayr, Gerd Hesina
Vienna University of Technology, Austria

Abstract

This paper focuses on the distributed architecture of the collaborative three-dimensional user interface management 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, load balancing, ad-hoc collaboration, and ubiquitous computing.

1 Introduction

In contrast to most distributed virtual environments (DVEs) and networked games (e. g., [18, 26] 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 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 [30] and Columbia's EMMIE [9] projects.

In previous work [35] on the *Studierstube* system (Figure 1), we have demonstrated how such a collaborative 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 alternately by a single user (applications as a set of complementary tools)
- multiple heterogeneous input and output media (e. g., HMD vs. desktop display) can be used to accommodate several user interface styles.

In this work, an extension of [36], 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 [46]. 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 [37]. 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") [40] 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 [39, 29, 20]. This development is mirrored in the DVE area in concepts such as subjective views [38], privacy widgets [8], or even dead reckoning techniques [23]. While users of relaxed WYSIWIS can suffer from a lack of mutual location awareness and have to use tools like telepointers [34, 39], 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 [34]. 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 [4]. 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 [29] or floor control [22] 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., [16, 17, 27]. The architecture presented in this paper also tries to exploit domain specific properties using a hybrid approach.

Besides static workgroup topology, some research also considers dynamic changes to the workgroup and client migration [5, 11], 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 [10], recent work favors the image copy approach [42]. This is partly due to novel architectures that make it easy to marshal complex runtime structures [3], 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 [6] features collaborative augmented reality, but is limited by its lack of an underlying distributed system.

CRYSTAL introduces multi-tasking to virtual environments [43]. The closest relative to our approach is EMMIE [9], which provides a similar platform, but does not include dedicated application management. Other prominent collaborative user interfaces, such as mediaBlocks [44] or multi-computer interaction [32, 33] 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 [17].

However, this design has two distinct disadvantages: Additional effort is spent on keeping application state and graphical objects synchronized (called "dual database problem" in [24]), 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 [15], Repo-3D [24], Avango [42], SGAB [47]) 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) [19] is based on the popular Open Inventor (OIV) toolkit [41]. It utilizes OIV's notification mechanism to automatically trigger an "observer" callback whenever an application changes something in the observed scene graph similar to [20]. 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 [45, 25].

Studierstube uses object-oriented runtime extension through subclassing [35]. 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.

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 a significant advantage of centralized DVE systems [12, 16, 26]: 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 [23], where hosts manage the locally controlled entity and remote entities are represented as "ghosts" [7].

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.

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 node 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 3.5) that autonomously respond to user input in possibly complex ways, for example using gesture recognition [13]. 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 [24]. 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 deregister 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.

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 demarshaled (SoDB::readAll) and added to the local scene graph, so it can resume operation.

To complete migration, the source host must deregister 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.

4.3 Migration through tangible application objects

The migration capabilities make it easy to place application instances anywhere in a virtual workspace spanning a larger area, and lend themselves to ubiquitous computing where information is always available through a variety of user interfaces. However, management of application instances themselves (which, where) through menus or other traditional user interface techniques is cumbersome and not appropriate for the type of user experience we are aiming for. We have therefore chosen to use a tangible interaction metaphor similar to the one proposed in [21] to manipulate application instances: Application instances are bound to physical marker objects, and these tangible application objects move along as the markers are moved. In [21], the bound virtual objects were passive 3D models, and interaction was limited to simple proximity operations of markers. These restrictions were party due to the use of only a single type of display (video see-through head mounted display), and a lack of an underlying distributed system.

For our approach, we use the same optical tracking library (ARToolKit) to detect the makers, but use it to display the 3D graphics of a *Studierstube* application instance above the flat marker, which thereby becomes the physical basement of the application object. Using the *Studierstube* distributed system, the application instance can be displayed anywhere in a heterogeneous workspace. Because the distributed system knows where application instances are located, any form of interaction can be used. Although it is a purely passive and inexpensive object, the marker becomes a physical embodiment of the application instance, which resides purely in the distributed system. Migration can thus be controlled by carrying the marker around. Our approach therefore combines the advantages of tangible augmented reality (real time 3D position tracking, 3D graphics, direct manipulation) [21] and mediaBlocks [44] (multi-modal input and output, location independence).

5 Demonstrations

5.1 Load balancing

One straightforward application of activation migration is dynamic workgroup management with load balancing. Late joining hosts need to receive copies of already running application instances through application migration. Exiting hosts that hold a master copy of an application instance need to pass on the master property to another host before deleting the object. Every master copy of an application instance places load on a host resulting 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 or hosts changes over time, computational load may become unevenly distributed.

As a countermeasure, we have implemented a simple load balancing mechanism (Figure 5a-d). A single session manager runs as a dedicated process in the environment and is responsible for monitoring the computational load. When the load changes due to modifications to the set of application instances or hosts, the session manager initiates appropriate activation or application migration procedures. 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 and host capacity.

In a test setup, we experimented with a number of distributed application instances running on host of different capability (SGI 4-CPU Onyx2 vs. SGI O2 vs. 500MHz PC workstation). The test confirmed that load balancing could off-load the less capable O2 and PC platforms by moving master properties to the Onyx2, and increased frame rate by an average 30% on 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 Mobile collaborative augmented reality

In [31] we describe initial experiments for a shared space experience involving a user wearing a mobile AR kit. In this initial demonstration the mobile user could only enter a stationary *Studierstube* environment and collaborate with a user of the stationary AR setup in a predetermined location by on-demand replication of the stationary user's application instance(s). This demonstration made some limited use of the locale concept from [2]. In our system the term locale describes an independent coordinate system that is used to group application instances based on geometric or semantic properties. An application instance can be a member of multiple locales, although this is generally only useful if it is not shown twice on the same display. The stationary and mobile users in [31] used originally independent locales, which were locked in place to enable collaboration.

Since then we have used the management tools for tangible application objects, locales, and migration to provide a more complete user experience of mobile collaborative augmented reality: Two users wearing mobile AR kits equipped with wireless networking can meet in an arbitrary place and connect. Each user has a private locale populated with application instances that are not visible to the other user. By putting down a special marker, a locale representing a public shared space for both users is created. Users can move application instances from their private locales to the public one and vice versa by moving the application's marker objects to and from the vicinity of the public locale marker (Figure 6 and video figure 2). Moving an application instance into the public locale makes it visible/interactive for the other user. Tracking is done using helmet mounted as well as overhead cameras. If an application instance is bound to a particular marker and this marker is first sighted by one of the camera, it will request a copy of the corresponding application instance through migration.

Internally, application instances are arranged in the scene graph in one of two locale nodes, and moved between the locales according to the users' actions. Only the content of the public locale is shared. If an application is moved from private to public by the first user, it migrates (is copied) to the second user, but the master property is kept at the first user. However, if the application is subsequently moved to the private locale of the second user, activation migration to the second user (and removal from the first user's host) is performed.

On a side note, an interesting alternative for a private locale is using a body stabilized locale that does not map applications to marker objects but rather moves along with a roaming user.

5.3 Scalable panorama display

A recent trend in visualization is the construction of tiled displays from inexpensive projectors driven by clustered PC workstations [28]. Although we have not yet implemented seamless tiling and genlocked rendering for a fully integrated display wall, a *Studierstube* cluster can at least work as a scalable panorama display using conventional displays placed side by side or in arbitrary configurations. Each display provides a window into a portion of the virtual workspace by depicting the content of a locale associated with that sub volume. Overhead cameras pick up how users move markers in the workspace, and make the corresponding application instances move accordingly (Figure 7 and video figure 3).

Since each host/display combination uses a locale with a finite extent, it need not know about the content of other locales. Hence as long as application instances remain stationary, there is no need to communicate with other hosts about the interaction regarding application instances contained in the locale, thus preserving network bandwidth and improving scalability through exploiting locality. An application instance need only be shared if it spans multiple locales because it happens to lie on the border or is very large.

Using the tangible marker objects, the panorama setup can accommodate interaction in the style of Rekimoto's "multi computer drag and drop" operations [32]. A user can move an application instance across the workspace, and the corresponding application will migrate from locale to locale, thus preserving the principle of locality.

5.4 Portable window

Rather than moving application instances across display boundaries, the system can also support the inverse operation of supporting portable – tracked – displays that provide a head-mounted or hand-held window into the workspace, comparable to [14]. Locality can be exploited by displaying only the content of a finite space around the tracked display. In our demonstration we have chosen to position separate spatially limited locales in a similar arrangement to the panorama display described above. The portable display renders the content of the locale closest to the display's tracked position. When the portable display enters a locale, the application instances migrate (are copied) to the entering display (Figure 8 and video figure 4). The source of this migration operation is either a pre-existing host holding the master copies of the applications (note that in this case, the source host and the portable host will be sharing the locale), or a persistent storage server if no actively participating host is available (see section 5.6 below).

Note that unlike [14], [9], or [33], the separation of the workspace into locales allows us to break with a globally continuous representation of space at will. For example, we have implemented a "pack-and-go" feature allowing the locale to be bound to the portable display and take the locale's content away (Figure 9).

"Pack-and-go" is done by creating a new display-stabilized locale and moving the application instances into this locale. If another *stationary* host is subscribed to the initial locale, it can either discard the application instances, or retain copies. In the latter case, the application instances will be present in two different locales, leading to the next example of remote collaboration (section 5.5), albeit over a rather small distance.

5.5 Remote collaboration

For collaboration at a distance, be it in geographically separated sites or in larger rooms with multiple non-adjacent displays, an application instance can be represented in multiple disjoint locales simultaneously. Note that the selection of application instances and their overall layout within the locale can be determined for each locale independently.

Although there is no shared space, the replication of the application instance's scene graph implies that the state of the application instance copies in each locale is shared – changes made to one copy will be reflected by all other copies. However, sharing of input data is only useful in a joint reference coordinate system, which is given by a particular locale. Since hosts sharing a locale will usually be physically close (and on the same network segment), their sharing of input data can be implemented efficiently. For each application instance we can partition the set of subscribed hosts according to the locale to which each host is subscribed to. Then input data is only shared within these subsets to again exploit locality.

Subscribers of the locale, among which the master property of the application instance is held, can actively change the application, while subscribers of the other locale are passive observers. However, active and passive roles can easily be changed by activation migration, for example with a simple "click-to-activate" strategy using a pointing device.

When application instances are bound to marker objects, an existing application can be copied onto another marker object, which can then be used in a different locale for remote collaboration on the shared application instance (Figure 10 and video figure 5).

5.6 Persistent storage

Using the migration tools referred to throughout this paper, application instances can easily be made persistent. We distinguish two forms of persistency: 1) Live persistency, meaning that the application instance continues to be executed by a host. The application does not receive interactive input, but is allowed to advance, for example to continue graphical animations and bookkeeping tasks. This form of persistency can easily be implemented by a host serving a special locale not visualized on any display. 2) Dormant persistency, meaning that the application is linearized as if it were to migrate to another host, but is rather written to a file for later retrieval. Obviously, this form of persistency can be trivially implemented with a standard shared file system.

Retrieval of persistent application instances is easy as long as they are bound to unique markers – a user simply needs to retrieve the right marker and show it to one of the sensor cameras associated with a particular display (Figure 11 and video figure 6). Using a session manager facility which stores a directory of all application instances, the system can distinguish between dormant application instances that need to be retrieved from physical storage and instantiated, and application instances simply not present in the current locale. The latter are present in another locale (or possibly the special persistency locale) and can be retrieved through migration alone.

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 between hosts using lightweight activation migration or by streaming linearized scene graphs. We have shown how to use these tools for workgroup management, load balancing, ad-hoc collaboration, clustered rendering 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 aim at unifying and extending the service components of our system (session management, persistent storage and device servers) into a central knowledge facility used as a ubiquitous computing server throughout the environment.

Acknowledgments

This project was sponsored in part by Austrian Science Fund (*FWF*) contract P-14470-INF, EU contracts IST-2000-28610 and IST-2001-34204, and an Austrian government "infrastructure lab" grant. Special thanks to A. Fuhrmann for many fruitful discussions, and to all of the *Studierstube* development team, in particular R. Splechtna, J. Prikryl, A. Zajic and F. Ledermann.

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

Video figures

We have provided video figures to illustrate the demonstrations presented here. These videos can be downloaded from the following web page: http://www.studierstube.org/projects/migration/

The following table gives an overview of the content of the video figures and to which section and figure they relate.

Video figure	Section	Figure
1 - migration and interaction	4 Migration	
2 - mobile collaboration	5.2 Mobile collaborative augmented reality	Figure 6
3 - panorama display	5.3 Scalable panorama display	Figure 7
4 – portable window	5.4 Portable window	Figure 8, Figure 9
5 – remote collaboration	5.5 Remote collaboration	Figure 10
6 – persistent storage	5.6 Persistent storage	Figure 11

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. Barrus, J. W., Waters, R. C., & Anderson, D. B. (1995). Locales and Beacons: Precise and Efficient Support For Large Multi-User Virtual Environments (Technical Report TR95-16): Mitsubishi Electric Research Laboratories.
3. 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.
4. 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.
5. Bharat K. A., L. Cardelli (1995). Migratory Applications. Proc. ACM User Interface Software and Technology (UIST'95), pp. 133-142.
6. Billinghurst M., H. Kato, (1999). Collaborative Mixed Reality. Proc. International Symposium on Mixed Reality (ISMR'99), Yokohama, Japan.
7. Blau B., C. Hughes, M. Moshell, C. Lisle, (1992). Networked virtual environments. Proc. 1992 ACM Symposium on Interactive 3D Graphics, pp. 157–164.

8. 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.
9. 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.
10. Chung G., K. Jeffay, H. Abdel-Wahab (1993). Accommodating late-comers in shared window systems. IEEE Computer, 26(1), pp.72-74.
11. 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.
12. 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.
13. 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.
14. Fitzmaurice, G. W. (1993 July). Situated Information Spaces and Spatially Aware Palmtop Computers. Communications of the ACM, 36(7), 38-49.
15. Frécon E, M. Stenius (1998). DIVE: A Scaleable network architecture for distributed virtual environments. Distributed Systems Engineering Journal, 5(3), pp. 91-100.
16. Funkhouser T. (1995). RING: A Client-Server System for Multi-User Virtual Environments. ACM Symposium on Interactive 3D Graphics, pp. 85- 92.
17. 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.
18. Greenhalgh C., S. Benford (1995). MASSIVE, A Collaborative Virtual Environment for Teleconferencing. ACM Transactions on Computer-Human Interaction, 2(3), pp. 239- 261.
19. 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.
20. 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.
21. Kato, H., Billinghurst, M., Poupyrev, I., Imamoto, K., Tachibana, K., Virtual Object Manipulation on a Table-Top AR Environment. Proceedings of International Symposium on Augmented Reality. 2000.
22. 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.
23. 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.
24. MacIntyre B., S. Feiner (1998). A Distributed 3D Graphics Library. Proc. SIGGRAPH'98, pp. 361-370.

25. 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.
26. Origin (1997). Ultima Online. Commercial online computer game, http://www.owo.com/.
27. Patterson J, M. Day, J. Kucan (1996). Notification servers for synchronous groupware. Proc. ACM Computer Supported Cooperative Work (CSCW'96), pp. 122-129.
28. Pavlakos C., R. Frank, A. McPherson, G. Humphreys, M. Eldridge, A. Finkelstein, A. Heirich: Commodity-based Scalable Visualization. SIGGRAPH 2001 course notes #37, 2001.
29. 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
30. 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.
31. 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.
32. 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.
33. 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.
34. Roseman M., S. Greenberg (1996). Building Real-Time Groupware with GroupKit, A Groupware Toolkit. ACM Trans. Computer-Human Interaction, 3(1), pp. 66-106.
35. Schmalstieg D., A. Fuhrmann, G. Hesina, Zs. Szalavari, L. M. Encarnação, M. Gervautz, W. Purgathofer: The Studierstube Augmented Reality Project PRESENCE - Teleoperators and Virtual Environments, Vol. 11, No. 1, pp. 32-54, MIT Press.
36. Schmalstieg D., G. Hesina: Distributed Applications for Collaborative Augmented Reality. Proceedings of IEEE Virtual Reality 2002, pp. 59-66, Orlando, Florida, March 24-28, 2002.
37. Singhal S., M. Zyda (1999). Networked Virtual Environments, Addison-Wesley, New York NY.
38. Smith G., J. Mariani (1997). Using Subjective Views to Enhance 3D Applications, Proc. ACM Virtual Reality Software and Technology (VRST '97), pp. 139-146.
39. 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.
40. 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.
41. Strauss P., R. Carey (1992). An Object-Oriented 3D Graphics Toolkit. Proc. SIGGRAPH'92, pp. 341-349.
42. Tramberend, H (1999). Avocado: A Distributed Virtual Reality Framework. Proc. IEEE Virtual Reality '99.
43. Tsao J., C. Lumsden (1997). CRYSTAL: Building Multicontext Virtual Environments. Presence, 6(1), pp. 57-72.
44. Ullmer B., H. Ishii, D. Glas (1998). mediaBlocks: Physical Containers, Transports, and Controls for Online Media. Proc. SIGGRAPH'98, pp. 379-386.

45. 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.
46. Weiser M (1991). The Computer for the twenty-first century. Scientific American, 265(3), pp. 94-104.
47. 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.

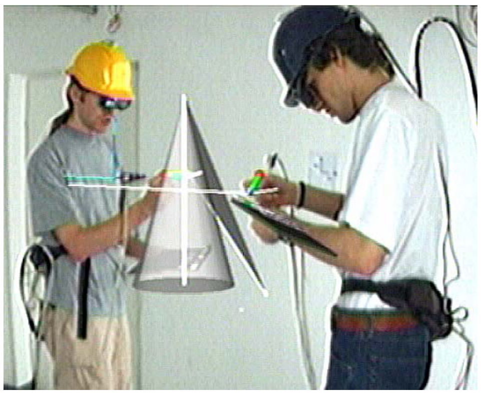


Figure 1: The distributed virtual workspace Studierstube supports multiple users and applications using tracked head mounted displays and hand-held tracked props. The image shows two users engaged in a geometry education task (live video overlay).

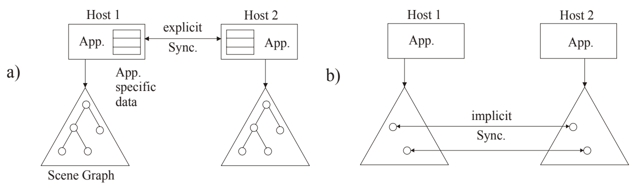


Figure 2: (a) Traditional distributed virtual environments separate graphical and application state, and synchronize only application state. (b) A distributed shared scene graph achieves replication that is transparent to the application.

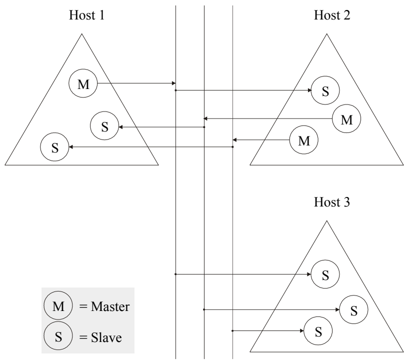


Figure 3: Replicated application instances embedded in a distributed shared scene graph run in master mode at exactly one host and in slave mode at all other hosts.

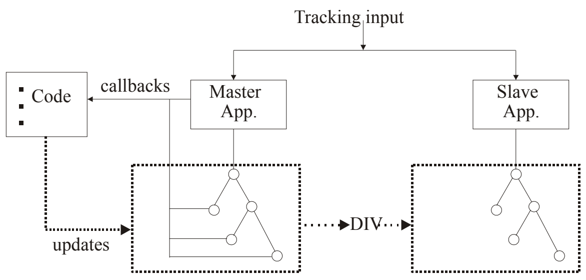


Figure 4: A user's interactions trigger callbacks that modify the scene graph. Changes are propagated to remote replicas through DIV.

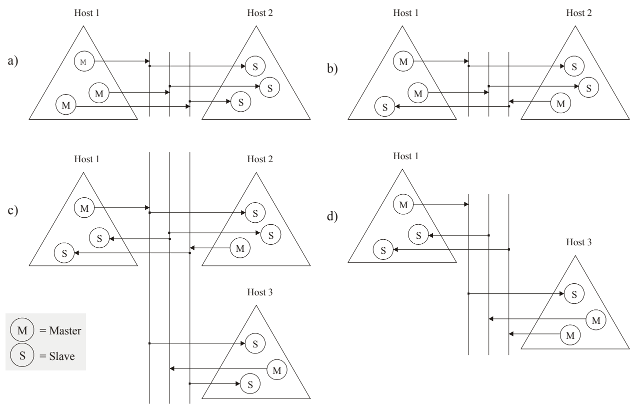
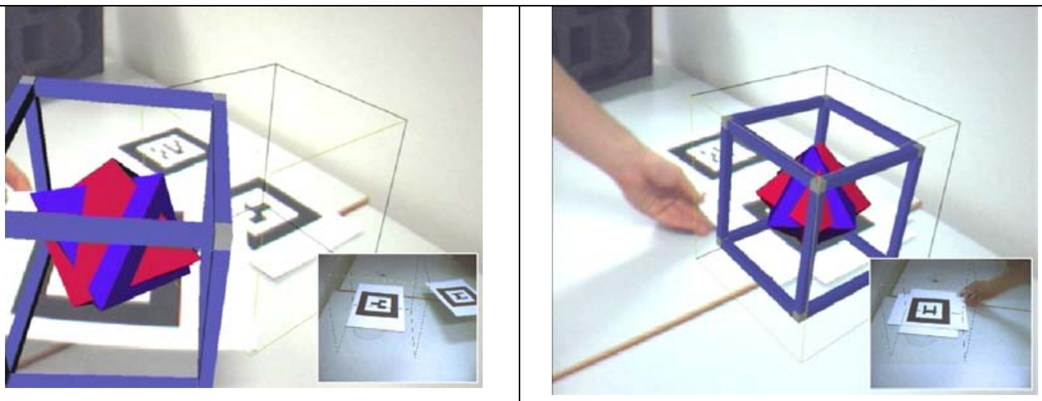


Figure 5: (a) Uneven distribution of load on hosts 1 and 2. (b) Load balancing moves one master privilege to host 2. (c) Host 3 joins late and receives one master privilege from host 1. (d) Host 2 exits early and passed its master privilege to host 3.



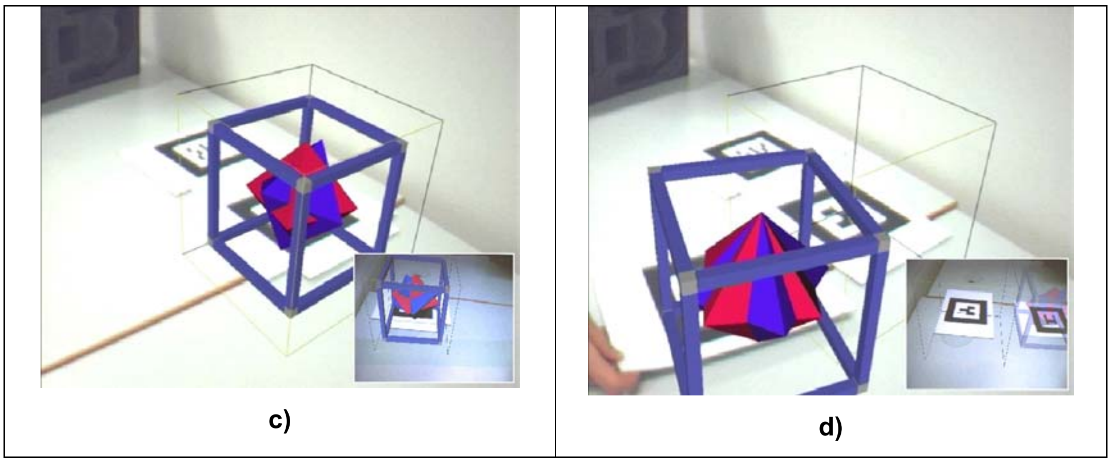


Figure 6: These images show the views of two mobile users collaborating in a public locale. The main images are recorded from user 1, while insets in the bottom right corner show the view of user 2. These images are just a demonstration of the perceived effect. They were not recorded in the same session. (a) User 1 inspects a private application. (b) The private application is moved to the public locale rendered as a wire frame cube. (c) The application becomes public and is also perceived by user 2. (d) As the application is moved out of the public locale, it disappears again the second display.

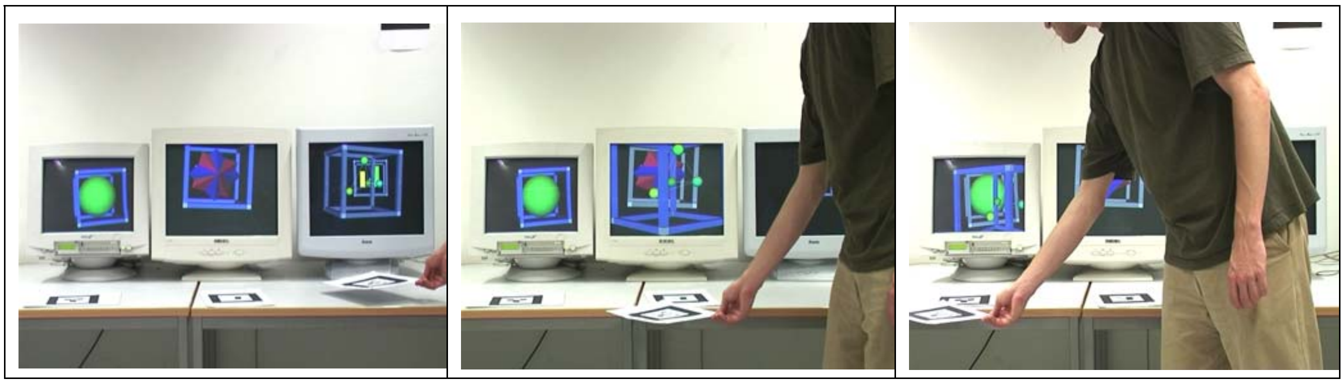


Figure 7: An application is moved across three displays by manipulating the associated marker. The application migrates from host to host in the process.

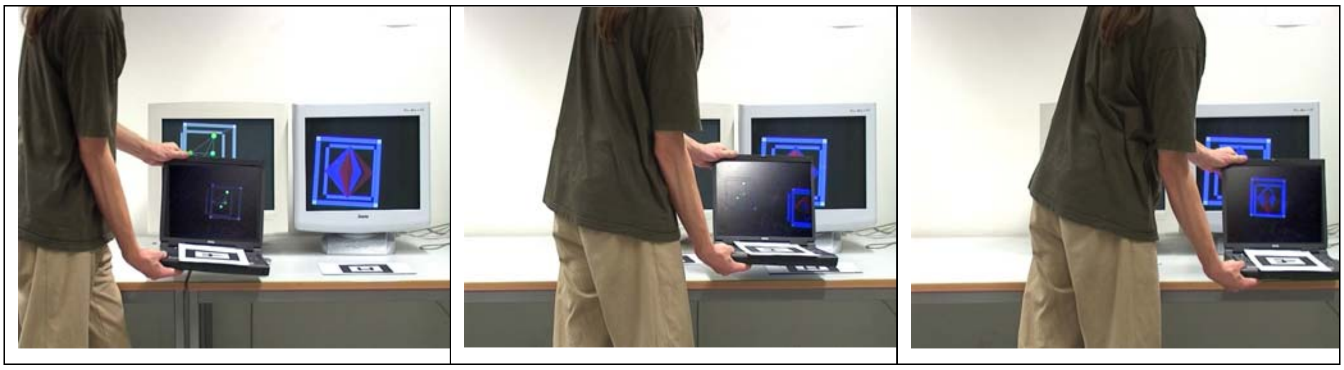


Figure 8: A tracked display is moved across two locales. It always renders the applications present in the locale it is currently part of. Again the applications are migrated to the display as necessary.

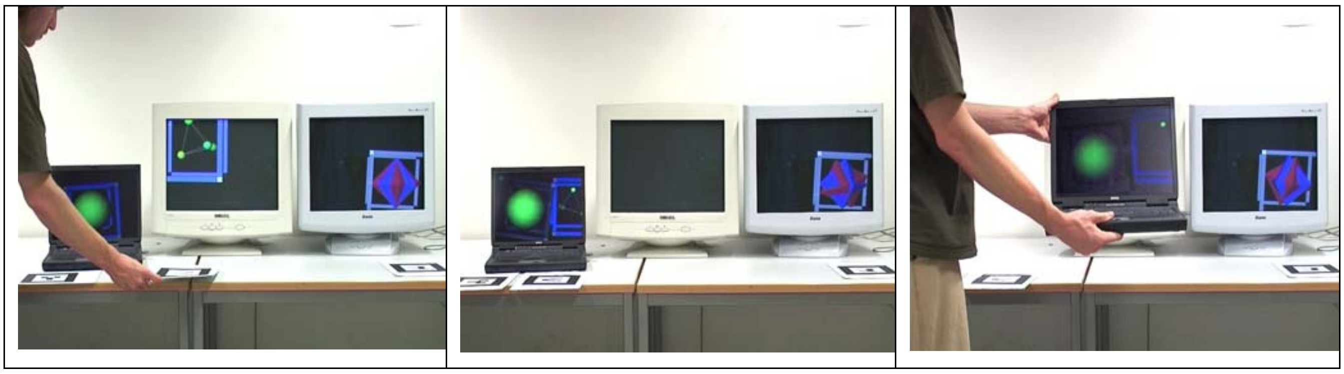


Figure 9: A portable display is added to the panorama screen. As before applications are moved to its locale by manipulating markers. In this demonstration the display copies the applications into its own portable locale and the applications stay on the display as it is moved.

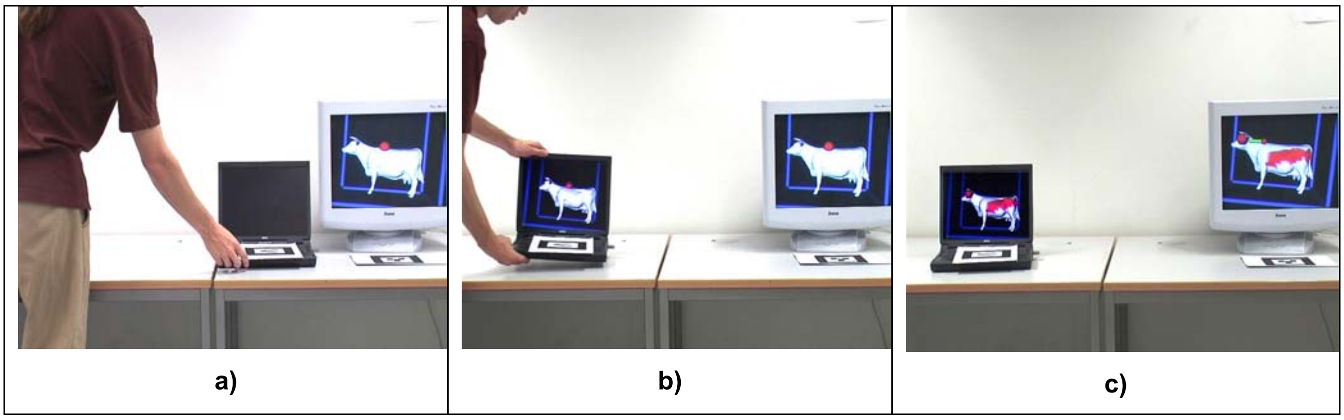


Figure 10: (a) A portable display is placed next to a locale and the running application is copied to the portables locale. (b) The display is moved to another location taking its copy along. (c) Interaction with the application in the original locale also updates the shared copy on the portable display.

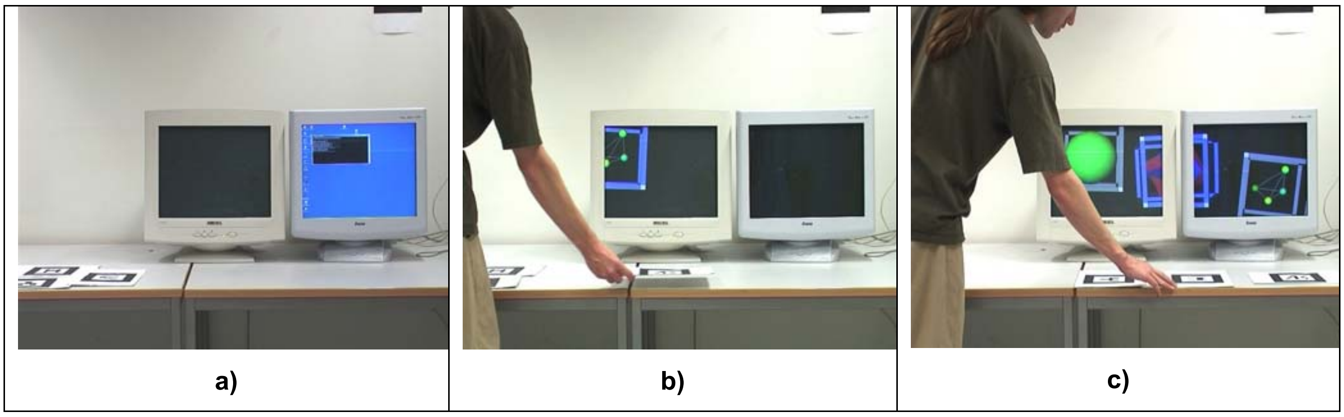


Figure 11: (a) Several applications are stored in a persistent locale to the left. The display hosts are started. (b) Applications are moved to the display hosts by a simple drag operation. (c) Finally all applications migrated to the two display hosts.