Distributed Open Inventor: A Practical Approach to Distributed 3D Graphics

Gerd Hesina, Dieter Schmalstieg, Anton Fuhrmann, and Werner Purgathofer
Vienna University of Technology, Austria
{hesina|schmalstieg|fuhrmann|purgathofer}@cg.tuwien.ac.at

ABSTRACT
Distributed Open Inventor is an extension to the popular Open Inventor toolkit for interactive 3D graphics. The toolkit is extended with the concept of a distributed shared scene graph, similar to distributed shared memory. From the application programmer's perspective, multiple workstations share a common scene graph. The proposed system introduces a convenient mechanism for writing distributed graphical applications based on a popular tool in an almost transparent manner. Local variations in the scene graph allow for a wide range of possible applications, and local low latency interaction mechanisms called input streams together with a sophisticated networking architecture enable high performance while saving the programmer from network peculiarities.

Keywords
Distributed graphics, concurrent programming, scene graph, distributed virtual environment, computer supported cooperative work, virtual reality

1. Introduction
The rapid evolution of high performance computer networks - in particular the Internet - has created the opportunity for the development of distributed graphical applications. On the one hand, vertical distribution is used to enhance the performance of graphical applications by executing on an ensemble of separate, communicating machines, exploiting the resulting parallelism [10]. Such a configuration, often called decoupled simulation [24], is commercially available via tools like Performer [20]. On the other hand, horizontal distribution is used to enable collaborative applications, that allow multiple users to work together, possibly over large distances. Particularly successful domains are Computer Supported Cooperative Work (CSCW) and Distributed Virtual Environments (DVE). However, these systems are generally based on distributed databases and proprietary protocols which are both application specific and thus fail to provide a general mechanism for graphics.

Current general purpose graphics libraries are engineered around the concept of a *scene graph*, a hierarchical object-oriented data structure. 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, which is the amount of flexibility we desire. Unfortunately, these toolkits have no built-in support for distribution.

A general-purpose distributed graphics toolkit should not place programming complexity on the programmer, or it will not be used. In particular, the programmer should not be forced to change the usual work style because of distribution. Obviously, a straight forward approach to achieve this requirement is to extend a toolkit that programmers are already familiar with to support distribution in a transparent way so that existing code continues to work with no or only minor modifications and new applications can be written without learning a new framework.

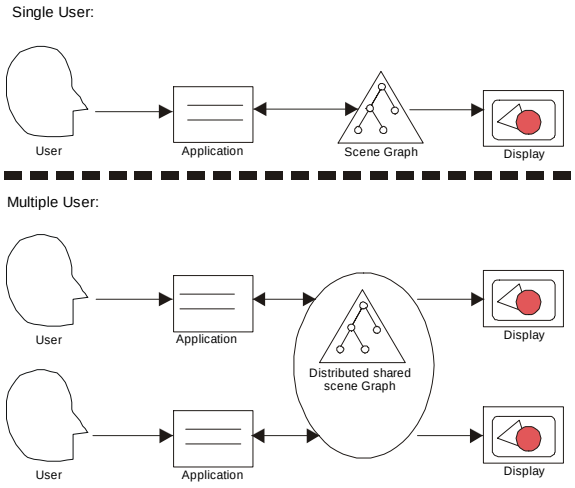


Figure 1. A single user's view of an interactive graphical application (top) is extended with the concept of a distributed shared scene graph (bottom) for multiple users.

We achieve this goal by extending a popular mainstream graphics toolkit, Open Inventor [28] (OIV). This toolkit is widely available and popular with graphics programmers, and is based on the most widely accepted programming language for graphics (C++). Our approach - *Distributed Open Inventor* (DIV) - extends the basic software to support a distributed shared scene graph, comparable to distributed shared memory (Figure 1). The implementation is almost transparent to the application programmer. Distributed programs generally execute efficiently, and the programmer need not deal with network peculiarities. Our approach is particularly interesting from a software engineer's perspective, as OIV is commercially available software not available as source code, and so we cannot rely on any techniques that require modification of the underlying code base.

2. Distributed shared scene graph

2.1 Motivation and overview

A scene graph is a hierarchical data structure of graphical objects. The application builds and maintains the scene graph, and the graphics toolkit uses it to create images. DIV's scene graph has the semantics of a database held in distributed shared memory [14]: Multiple workstations in a distributed system can make concurrent updates to the system, and all updates are reflected at each workstation's view of the scene graph. The scene graph represents the shared state of the distributed systems to both the application, and to the users via the images rendered from it.

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

The simplest approach to a synchronous view on shared data is to store the data only once and redirect any access via remote procedure calls (e. g. Sun RPC [30], Java RMI [29], CORBA [2], DCOM [22]). However, interactive graphical applications, in particular virtual environments, require that the data required for rendering is stored locally at the workstation, or interactive frame rates will be impossible. Therefore pure client-server approaches are infeasible for our purposes.

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

2.2 Communication path for interactive graphics applications

Interactive graphical applications place the human user in a loop with the computer. A simple model of this loop is composed of the following stages (Figure 2):

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

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

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

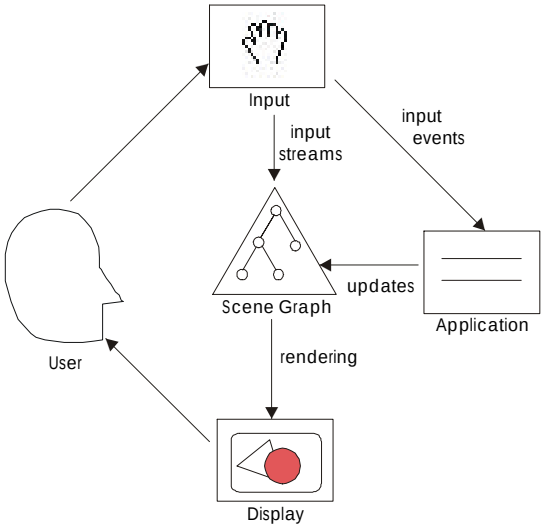


Figure 2. Typical communication path in an interactive graphical application placing the human in a feedback loop.

Some modifications of the scene graph do not require complex computations by the application, but can perform simple changes to scene graph attributes directly related to the input, but with highest possible responsiveness. The graphics toolkit allows to set up such interactions (e.g. dragging, camera movement) to work within the runtime software at maximum performance, without involving user written computation code (comparable to nervous reflexes which do not involve the human brain). We call such communication paths *input streams* (Figure 2).

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

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

2.3 Network topology

We observe than since some kind of distribution must take place, it can also be utilized to achieve a certain degree of vertical distribution by balancing the load of computationally intensive task between multiple

workstations. The computationally intensive tasks are rendering and computation. While rendering places together with processing input streams necessarily places a high load on each user's workstation, application specific computation can be assigned to another workstation.

Such a simultaneous horizontal and vertical distribution is achieved by distinguishing *application servers* and *rendering clients*. A reasonable system is composed of at least one application server and at least two rendering clients (Figure 3). An application server stores the master copy of a scene graph and performs application specific computation. A rendering client stores a replica of the scene graph and renders the image for the user. Updates that a server makes to its master scene graph are distributed to all clients holding a replica using multicasting.

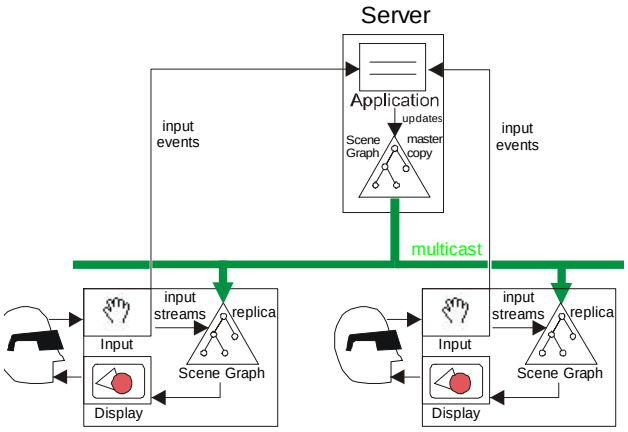


Figure 3. The system architecture separates most of the application specific computation from the hardware intensive rendering executed by graphical clients.

Optionally, an application server may also perform rendering for monitoring purposes or if an insufficient number of workstations is available. Note that typically the graphics hardware is the most expensive part of the system. If no rendering is required on the server side, inexpensive workstations without high performance graphics can be used.

Computational scalability is achieved by introducing multiple application servers holding mutually exclusive sub scene graphs. Such sub scene graphs can correspond to the content of different 3D windows [8], applications or data sets. Increased flexibility is obtained by allowing a client to choose to replicate all sub scene graphs, or select any subset, depending on application semantics and user preferences. A disadvantage of multiple servers is that exchange of information between applications (such as drag and drop operations) require special tailored solutions.

The application server implicitly performs serialization of events generated by multiple users. Via the multicasting of scene graph changes to the rendering clients, a consistent view of all scene graph replicas is maintained. Placing application specific computation at a server avoids redundant computation at each user's workstation. Server bottlenecks are avoided by allowing for multiple servers (Figure 4) and by managing high performance input streams locally.

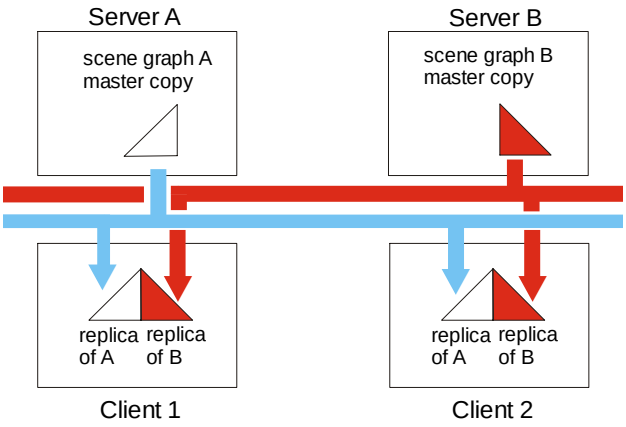


Figure 4. The system allows to distribute computation among multiple application servers as long as the scene graphs managed by each server are disjoint.

3. Replicated scene graph protocol

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

The vocabulary of operations possible on a scene graph consists of relatively few messages. The state of every node is determined by a node's fields. Reading a field's value does not change the state of the scene graph and therefore need not be distributed.

The most common operation that must be propagated is an update of a field's value. Fields store a basic data types such as numerical values, boolean flags, vectors, matrices etc. The information necessary to encode such an update can be encoded in fixed size messages and efficiently transmitted over the network.

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

workstations to load a sub graph either from file (if a common file service exists) or from a URL. This solution is convenient for application programmers and also more efficient than creating node by node. Deletion of group nodes is always recursive, i. e. if a parent node is deleted and its children are not references elsewhere in the scene graph, the children are also deleted, hence no message for deleting sub graphs is necessary.

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

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

message	parameters
update field	node id, field id, value
create node	node type, parent node name, child index
delete node	node name
create sub graph	file name or URL, parent node name, child index
set node name	path to node, new node name

Table 1: protocol to keep scene graphs synchronized

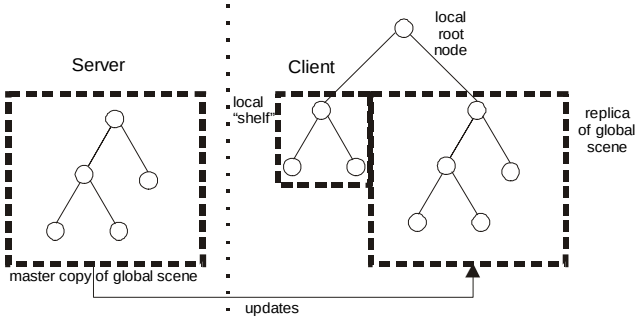
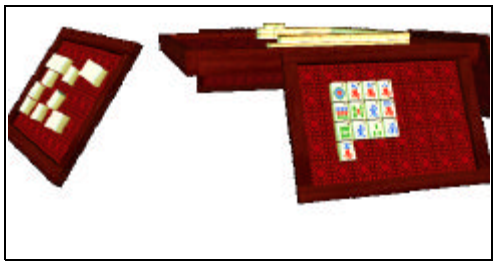


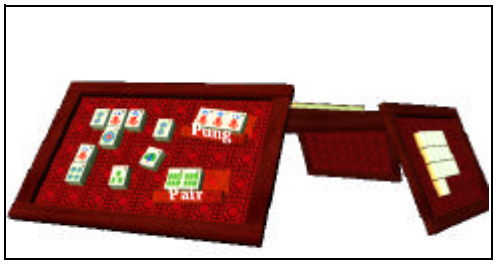
Figure 5. Local variations (such as a "shelf") allow to customize the behavior for each user.

4. Local Variations

Most applications will just require to share a scene graph. However, a potentially much larger range of distributed graphics applications can be constructed by allowing local variations in the scene graph. Local variations (Figure 5) can be useful in a variety of ways:



View of user 1



View of user 2

Figure 6. Mahjongg is a multi-user game. Note how the play tile labels of user 1 are hidden in the view of user 2 and vice versa.

- Individual content per user: Each user may operate on a variety of data sets, and choose to share only some of them, or decide on-line which data sets can be seen by other users and which not. Reasons may include privacy and security (compare [18]), individuality (e. g. a private shelf or clip board) or work flow (only "polished" data is shared).
- The same data may be viewed differently by multiple users, which is different to the above in that structurally identical or at least similar data is shown with different attributes to different users. Reasons to change the representation of one particular data set for individual users can be motivated by their roles. For example, a customer sees a simpler representation than the sales manager, or a teacher sees solutions to problems that the students may not see. Sometimes part of the data (such as labels) may also be intentionally hidden from other users, for example in multi-player games [31] (see Figure 6).
- Individual viewpoints are a special case of individual content. This concept is particularly useful for virtual environments (see section 5), where the position of a virtual camera is determined by head tracking on a per-user base.
- Some typical editing operation such as high lighting, selection, dragging, or cursor display require locally varying graphics. Note that these interaction concepts work in conjunction with low-latency input streams (see section 2.2) that short cut the distributed communication paths.

Using DIV to construct a scene graph that is partially distributed is straight forward: The scene graph used by the client can vary from the scene graph stored at the server.

The only restriction is that it must be a super set of the server's distributed scene graph, which does not affect applicability in practice.

Using local variations in the scene graph implies that not the whole application specific computation can be carried out at the application server. All program logic directly affecting local nodes must be carried out locally at the workstation. As both client and server execute the OIV runtime system, there is no problem carrying out application specific code dealing with local variations at the client. The only consequence is that the application programmer must distinguish between client and server code, so complete transparency of distribution can no longer be maintained.

5. Application in a virtual environment

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

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

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

Our solution is based on the Studierstube virtual environment [8] modified to use DIV (Figure 7). We resolve this issue by introducing a new tracker server, which uses its own multicast group to transmit tracker data over the network to *both* application servers and rendering clients. An additional benefit of this approach is that computationally intensive filtering and prediction tasks applied to the tracker data can be carried out by the tracker server without consuming resources on other workstations.

The way the tracker data is treated by the rendering clients is quite different from the application servers:

- The rendering clients use the tracker data directly as an input stream for continuous actions, for example to control the virtual camera or to control interaction widgets such as the rubber band shown in Figure 10.
- The application servers transform the tracker data into input events. For example, the server notes when the tracker hits a button area in 3D and passes a "press button" event to the application code, which then reacts appropriately.

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

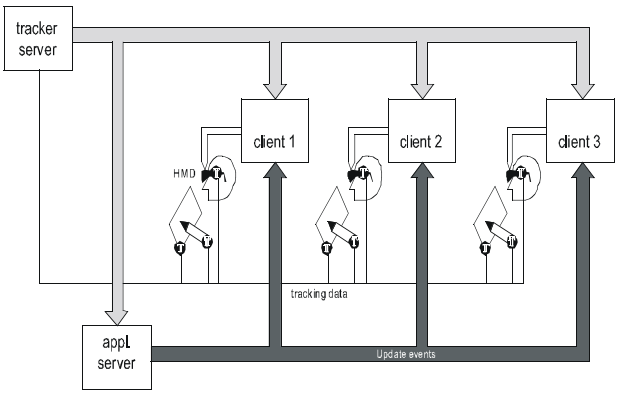


Figure 7. The Studierstube virtual environment has been modified to use DIV together with a tracker server that multicasts tracker data over the network.

6. Implementation

6.1 Software architecture

Open Inventor is a commercial software product available for most graphics platforms, (including most Unix variants and Windows NT) and uses Open GL for rendering. It was chosen because of its popularity, flexibility and because of legacy applications available in our lab. OIV is implemented as an object-oriented class hierarchy in C++ and a library for runtime binding. Refer to Figure 8 for an overview.

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

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

A slightly more complicated situation arises if the structure of the scene graph itself changes, i. e. a node is added or deleted. In this case, the node sensor still calls

the user's function, indicating the group node whose children have changed, but does not indicate which child has been added or removed. We resolved this matter by caching the hierarchical structure with a "shadow" scene graph that consists of copies of only the group nodes, while leaf nodes are just referenced. When a group's children change, the group node is compared to its shadow to evaluate what change has been made. The shadow data structure is not included in the scene and thus not visible. It has also a small memory footprint and little computational overhead as it contains only links.

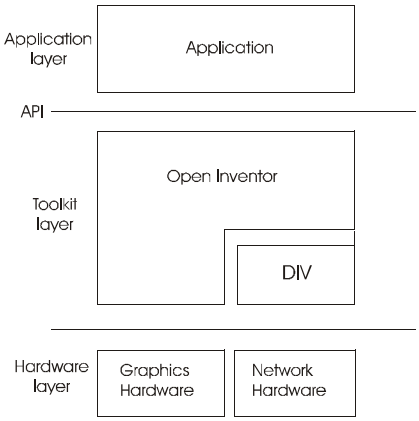


Figure 8. DIV is a software that plugs into a standard graphics solution - Open Inventor - to provide distribution

6.2 Networking

DIV also relies on a thin network abstraction layer constructed on top of UDP with multicasting. A network service object provides services for reliable multicasting of messages as well as utility functions such as connection setup etc. Most of the functions performed in the networking layer are transparent to the application programmer, who only needs to indicate whether a scene graph needs to be distributed - ideally a single initialization call for a simple application that does not require local interaction or other advanced features.

UPD was chosen as transport protocol for performance reason and since multicast support for UPD is readily available. However, propagation of event messages must be reliable, which is a service that UPD alone cannot provide. Therefore a simple reliability mechanism was added that does not suffer from the overhead of high-level protocols such as TCP which are aimed at more general network applications. Packets are numbered by the server - note that each server can have its own numbering sequence as packets from multiple servers do not interfere with each other. The server stores a customizable number of messages in a history buffer, and if clients miss a packet, they may request retransmission (although this rarely happened in our test setup without artificially introducing network errors for testing purposes).

6.3 Lazy naming

As mentioned in section 3, every message refers to a node and thus needs to uniquely identify the node. OIV has a built-in naming scheme for nodes based on a hash table, which is highly efficient and ideal for our purposes. It also lets users specify names for nodes in geometry files (.iv) which is a convenient way for applications to identify nodes and also works when the geometry file is distributed. However, it frequently occurs that applications modify anonymous nodes and these modifications have to be distributed.

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

7. Results

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

The first example that was chosen for distribution is the maze game (Figure 9) featuring a hand-held labyrinth toy which can be tilted to make a ball roll through the corridors. The objective is to guide the ball to the goal while avoiding the holes in the maze's floor. The game was distributed for multiple users, allowing each user to see and manipulate the maze. Updates were intentionally made relative so that the resulting tilt is equal to the sum of the steering motions of all users, which creates an interesting and entertaining collaborative task.

Users can also see each other's point of view represented by a simple avatar, a feature which makes use of a locally varied scene graph (each user's scene graph contains avatars for the other users, but not for the user).

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

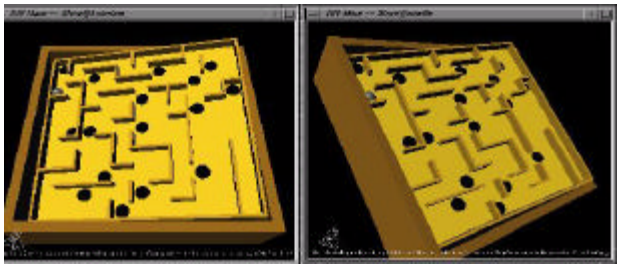


Figure 9. The shared maze game allows to users to collaborate (or work against each other) using multiple workstations.

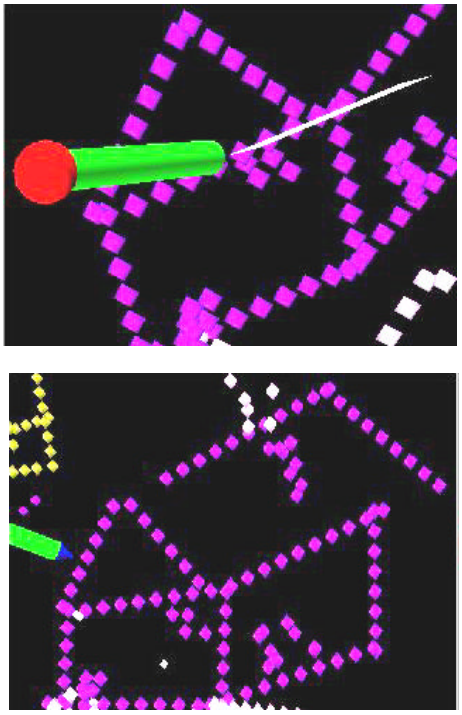


Figure 10. The shared spraying application allows multiple users to paint collaboratively. The top image shows a user drawing a rubber band, which is an example of a local graphical variation connected to an input stream. Note how the second user's view (bottom image) does not show the rubber band.

Parameters such as paint color, size of paint droplets and paint pressure are controlled with local interaction widgets, which represent local variations of the scene graph - each user can have an individual current color etc. Furthermore, we make use of local variations combined with input streams for the line drawing utility (Figure 10), which displays a rubber band while the user is dragging. When the rubber band is released, a line of paint droplets is created and added to the shared scene graph.

8. Related work

A lot of research has been dedicated to building common virtual places in which users can interact with each other and with the underlying simulation. This research has produced a number of platforms such as NPSNET [34], SPLINE [33], AVIARY [26], MR Toolkit [25], DIVE [4], NetEffect [5], or RING [9]. Years of research and experiments with or for these platforms have led to the evolution of several techniques for implementing efficient networked virtual environments. However, these systems are specially designed for the purpose of the DVEs (such as training, playing or scientific visualization) and not designed for supporting general distributed interactive 3D graphics applications. DVEs as well as online games such as Ultima Online [17], Everquest [27], and QuakeWorld [12] and commercial toolkits like dVS/dVISE [11] or World2World [23] only distribute a minimum sub set of application state and rely on a priori distribution of graphical data.

Over the past years VRML [1] has become widely accepted as a file format for exchanging 3D data over the Internet. VRML97 [32] is used for some DVE platforms [13]. However, with few exceptions [3] little work has concentrated on a general purpose programming toolkit for distributed graphics based on VRML.

CSCW such as DistView [19] or GroupKit [21] systems provide automatic distribution of data for desktop tasks. While the application area is similar to ours, these systems are not designed with support for interactive graphics in mind. The distributed mechanisms do not provide support for integration in existing programming paradigm for 3D such as scene graphs. Furthermore, the use of synchronous updates for all operations conflicts with the performance requirements and scalability goals of distributed graphics.

Popular graphics toolkits, e. g., Open Inventor [28], and Java 3D [6], provide a comfortable programming model, but very limited support for distribution. Some of them - like Performer [20] - allow to distribute stages in the graphics pipeline over multiple processes, but no general distribution is available.

Finally, there are some general distributed graphics toolkits which resemble many aspects of out approach. TBAG [7] has an elegant distribution mechanism based on a shared constraint graph. However, assertions regarding constraints must be shared by all processes, which restricts the scalability of the approach. Moreover, a constraint graph is somewhat different from a scene graph in its use. IntelligentBox [16] uses a mechanism of collecting events and propagating it over a network to synchronize a replica, which is similar to our implementation of a shared data structure. However, no details of the underlying network mechanism is given, so it is hard to judge the efficiency. Finally, recent work on Repo-3D [15] shares many of our goals. It uses an implementation based on Modula-III and a replicated object package to achieve distribution. On top of the distributed shared memory model provided by the software system, an interpreted language called Repo and a companion graphics toolkit called Repo-3D form the development system. Unoftunately, from the description given in the paper it is hard to judge the issue of scalability as the underlying networking mechanisms, in particular networking topology, are not explained in detail. Furthermore, while Modula-III is certainly a good choice for language-level embedding of distributed

objects, in our opinion the user acceptance of a mainstream language like C++ would be higher.

9. Conclusions and future work

This paper has presented a practical approach to distributed graphics, realized as DIV, the Distributed Open Inventor library. DIV is founded on the notion of a distributed shared scene graph, a powerful data structure that unifies graphical and application data with distributed control. Our implementation extends the popular Open Inventor toolkit and thus allows programmers to continue software development in a familiar style and software development environment. Our approach is almost completely transparent to the application programmer and allows existing applications to be distributed with very little effort.

We are currently working on a more complete integration of DIV and Studierstube software. All of the interaction tools designed for Studierstube can be realized with DIV, but current implementation semantics do neither distinguish shared from local state nor input events from input streams, which is essential for distribution with reasonable performance. Further plans involve the development of new interaction styles that make only sense within a truly distributed framework.

10. Acknowledgments

This work has been supported by the Austrian Science Funds (FWF) under project no. P-12074-MAT. Special thanks to Hermann Wurnig for working on the implementation and to Michael Gervautz.

11. References

[1] Bell, G., Parisi, A., Pesce, M. The Virtual Reality Modeling Language, Version 1.0 Specification, 1995. URL: http://www.vrml.org/VRML1.0/vrml10c.html.

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

[3] Broll, W. DWTP-An Internet Protocol for Shared Virtual Environments. In Proc. of ACM VRML'98, 49-56, 1998.

[4] Carlsson, C., and Hagsand, O. DIVE: A Multi-User Virtual Reality System. In Proc. IEEE VRAIS '93, 394-400, Sept. 1993.

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

[6] Deering, M., and Sowizral, H. Java3D Specification. Technical Report, Sun Microsystems, Aug. 1997. URL: http://java.sun.com/products/java-media/3D/.

[7] Elliot, C., Schechter, G., Yeung, R., and Abi-Ezzi, S. TBAG: A High Level Framework for Interactive, Animated 3D Graphics Applications, In Proc. ACM SIGGRAPH '94, 421-434, Aug. 1994.

[8] Fuhrmann, A., and Schmalstieg, D. Multi-Context Augmented Reality. Technical Report TR-186-2-99-14, Institute of Computer Graphics, Vienna University of Technology, 1999. URL: http://www.cg.tuwien.ac.at/research/TR/.

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

[10] Gelernter, D. Mirror worlds. Oxford University Press, 1992.

[11] Ghee, S., Mine, M., Naughton-Green, J., and Pausch, R. Programming Virtual Worlds. SIGGRAPH 94 Course Notes 17, 1994.

[12] Id Software. Quake World, online computer game, 1996. URL: http://www.quakeworld.net/.

[13] Lea, R., Honda, Y., Matsuda, K., and Matsuda S. Community Place: Architecture and Performance. Proceedings of ACM VRML'97, 41-50, 1997

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

[15] MacIntyre, B., and Feiner, S. A Distributed 3D Graphics Library. SIGGRAPH 98 Conference Proceedings, Annual Conference Series, 361-370, 1998.

[16] Okada, Y., and Tanaka, Y. Collaborative environments of IntelligentBox for distributed 3D graphics applications. The Visual Computer, 14(4), 140-152, 1998.

[17] Origin. Ultima Online, online computer game, 1997. URL: http://www.owo.com/.

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

[19] Prakash, A. and Shim, H. S. DistView: Support for Building Efficient Collaborative Applications Using Replicated Objects. In Proc. ACM CSCW '94, 153-162, Oct. 1994.

[20] Rohlf, J., and Helman, J. IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics. In Proc. ACM SIGGRAPH '94, 381-394, 1994.

[21] Roseman, M., and Greenberg, S. Building Real-Time Groupware with GroupKit, a Groupware Toolkit. ACM Transactions on Computer Human Interaction, 3(1):66-106, March 1996.

[22] Rubin, W., and Brain, M. Understanding DCOM. Prentice Hall PTR, 1999, ISBN 0-13-095966-9.

[23] Sense8 Corporation. World2World - Technical Overview. 1997. URL: http://www.sense8.com/.

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

[25] Shaw, C., and Green, M. The MR Toolkit peers package and experiment. In Proc. of VRAIS '93, 463-469, 1993.

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

[27] Sony Corporation. Everquest, online computer game, 1999. URL: http://www.everquest.com/.

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

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

[30] Sun Microsystems. Remote Procedure Call Protocol Specification. Network Working Group RFC1050, April 1988.

[31] Szalavari, Z., Eckstein, E., and Gervautz, M. Collaborative Gaming in Augmented Reality. Proceedings of VRST' 98, 195-204, Taipei, Taiwan, Nov. 2-5, 1998.

[32] The VRML Consortium Incorporated. The Virtual Reality Modeling Language, International Standard ISO/IEC 14772-1:1997, 1997. URL: http://www.vrml.org/Specifications/VRML97/.

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

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