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 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 [6]. Such a configuration, often called decoupled simulation [13], is commercially available via tools like Performer [11]. 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 often 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.

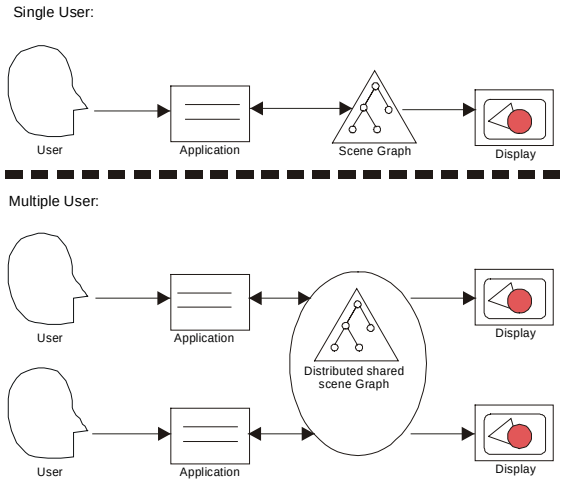

Figure 1: A single user's view of an interactive graphical application (top) is extended with the concept of a distributed shared scene graph (bottom) for multiple users.

We achieve this goal by extending a popular mainstream graphics toolkit, Open Inventor [17] (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 commercial 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 [7]: Multiple workstations in a distributed system can make concurrent updates to the system, and all updates are reflected at each workstation's view of the scene graph. The scene graph represents the shared state of the distributed systems to both the application, and to the users via the images rendered from it.

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

The simplest approach to a synchronous view on shared data is to store the data only once and redirect any access via remote procedure calls (e. g. Sun RPC [19], Java RMI [18], CORBA [1], DCOM [12]). However, interactive graphical applications, in particular virtual environments, require that the data used for rendering is stored locally at the workstation, or interactive frame rates will 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).

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

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).

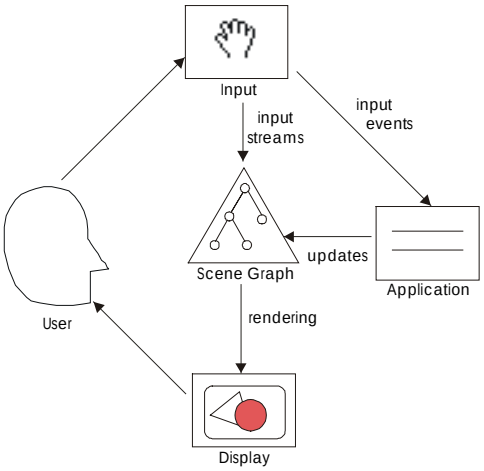

Figure 2: Typical communication path in an interactive graphical application placing the human in a feedback loop.

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

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

3. REPLICATED SCENE GRAPH PROTOCOL

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

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

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

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

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

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

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

Message	parameters
Update field	node id, field id, value
Create node	node type, parent node name, child index
Delete node	node name
Create sub graph	file name or URL, parent node name, child index
set node name	path to node, new node name

Table 1: protocol to keep scene graphs synchronized

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 3) can be useful in a variety of ways:

- Individual content per user: Each user may operate on a variety of data sets, and choose to share only some of them, or decide on-line which data sets can be seen by other users and which not. Reasons may include privacy and security (compare [10]), 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 [20] (see Figure 4).
- Individual viewpoints are a special case of individual content. This concept is particularly useful for virtual environments (see section 6), where head tracking on a per-user base determines the position of a virtual camera.

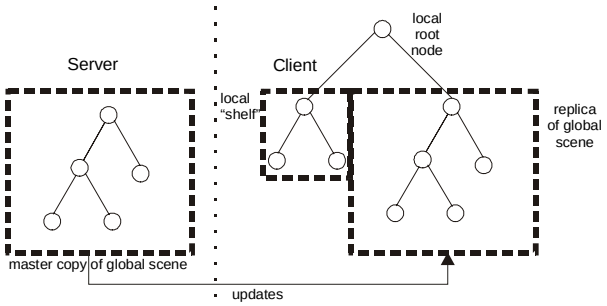

Figure 3: Local variations (such as a "shelf") allow to customize the behavior for each user.

View of user 1

View of user 2

Figure 4: 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.

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 locally can vary from workstation to workstation. The only restriction is that those portions that are distributed must be replicated at all workstations, which does not affect applicability in practice.

5. NETWORKING

Apart from basic connectivity, a key issue in distributing changes to a shared database like DIV's scene graph is how consistency among the participating processes is guaranteed. Several approaches to this problem have been investigated in the context of distributed virtual environment, and can be loosely categorized into pure client-server solutions (often found in Internet gaming such as Ultima Online [9]), pure peer-to-peer communication (such as DIVE [3]) and hybrid topologies (such as RING [5]).

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

For achieving consistency, we employ a similar approach like Repo-3D [8]: a sequencer process performs serialization of events generated by multiple users. Changes to the scene graph are then distributed via reliable multicasting (based on UDP with negative acknowledgments) to the participants, so that a consistent view of the scene graph replicas is maintained. There may be more than one sequencer present to avoid overloading one process. Typically the scene graph is coarsely divided into several logically coherent chunks (sub scene graphs) such as the content of different 3D windows [4], applications or data sets, which are then associated with separate sequencer processes. Increased flexibility is obtained by allowing a participant to choose to replicate all such sub scene graphs, or select any subset, depending on application semantics and user preferences.

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

6. 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 [4] modified to use DIV (Figure 5). We resolve the issue of short communication path by distinguishing a tracker server, one or multiple application servers, and rendering clients. The tracker server uses its own multicast group to transmit tracker data over the network to *both* application servers and rendering clients. An additional benefit of this approach is that computationally intensive filtering and prediction tasks applied to the tracker data can be carried out by the tracker server without consuming resources on other workstations.

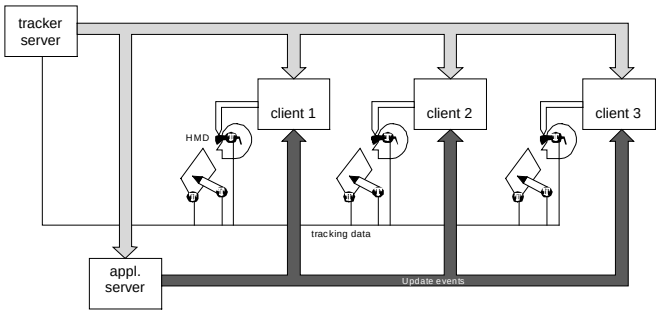

Figure 5: The *Studierstube* virtual environment has been modified to use DIV together with a tracker server that multicasts tracker data over the network.

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 9.

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.

7. IMPLEMENTATION

7.1 Software architecture

Open Inventor is a commercial software product available for most graphics platforms, (including most Unix variants and Windows NT) and uses OpenGL 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 6 for an overview.

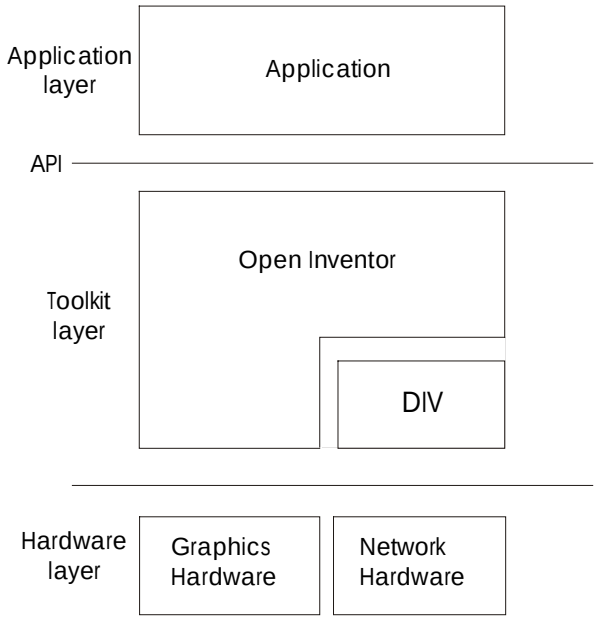

Figure 6: DIV is software that plugs into a standard graphics solution – Open Inventor – to provide distribution.

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 application implicitly serves as a serialization mechanism if the application receives input events from multiple users.

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

DIV uses a similar approach as Avocado [21] to handle late-joining users. A new user has no knowledge of the current shared application state and therefore it is impossible to participate without an atomic state transfer from an old user to the new one. During this atomic state transfer all other communications within that certain group is suspended until the transfer completes.

7.2 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.

7.3 Usage example

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

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

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

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

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

```
root->setName("myRootNode");
```

Finally, both master and slave call their main loop. For a slave, DIV provides a modified main loop which compensates the fact that OIV is not thread safe and can therefore not be used for asynchronous processing of network updates. Figure 7 shows an example of an update.

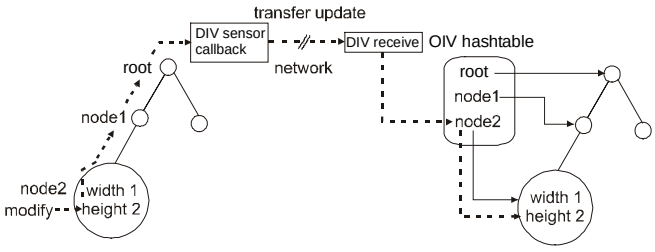

Figure 7: Example of a modified field update in a master-slave configuration. The field "height" of node in the master's scene graph is modified. Via the notification mechanism, the change is propagated first to the shared root, then via DIV's networking layer to the slave process. At the receiver, DIV finds the corresponding node from OIV's hash table and updates the corresponding field.

8. RESULTS

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

The first example that was chosen for distribution is the maze game (Figure 8) featuring a hand-held labyrinth toy which can be tilted to make a ball roll through the corridors. The objective is to guide the ball to the goal while avoiding the holes in the maze's floor. The game was distributed for multiple users, allowing each user to see and manipulate the maze. Updates were intentionally made relative so that the resulting tilt is equal to the sum of the steering motions of all users, which creates an interesting and entertaining collaborative task.

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

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

Parameters such as paint color, size of paint droplets and paint pressure are controlled with local interaction widgets, which represent local variations of the scene graph - each user can have an individual current color etc. Furthermore, we make use of local variations combined with input streams for the line drawing utility (Figure 9), which displays a rubber band while the user is dragging. When the rubber band is released, a line of paint droplets is created and added to the shared scene graph.

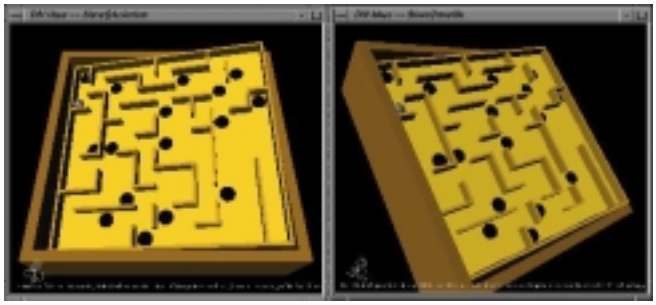

Figure 8: The shared maze game allows users to collaborate (or work against each other) using multiple workstations.

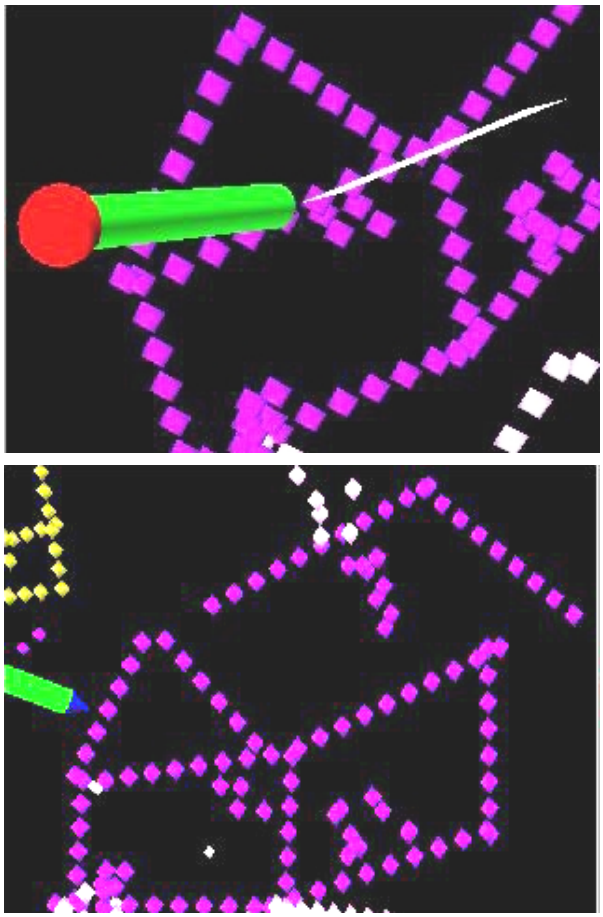

Figure 9: The shared spraying application allows multiple users to paint collaboratively. The top image shows a user drawing a rubber band, which is an example of a local graphical variation connected to an input stream. Note how the second user's view (bottom image) does not show the rubber band.

9. 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 [23], SPLINE [22], AVIARY [15], MR Toolkit [14], NetEffect [2], or RING [5]. 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. In particular, it is common to separate the visual representation of objects from the application semantics. While this increases modularity in the design, it also creates a "dual database" problem. Some architectures including recent work on DIVE [3], Avocado [21] and Repo-3D [8] address this problem in a manner very similar to DIV's distributed shared scene graph.

As MacIntyre and Feiner put it, "Keeping these dual databases synchronized is a complex, tedious, and error-prone endeavor. In contrast, some non-distributed libraries, such as Inventor, allow programmers to avoid this problem by using the graphics scene description to encode application state". Repo-3D addresses the problem using Modula-III with language embedding of distributed objects together with a custom graphics solution (Obliq-3D). While Modula-III is certainly a good choice for language-level embedding of distributed objects, in our opinion the user acceptance of Avocado [21] - a solution based on mainstream choices (C++, Performer [11]) - would be higher.

However, Avocado relies on subclassing Performer to mix in the desired transparent support for distribution. This implies that Avocado applications can only use those features of Performer made available through subclassing. Furthermore, many architectural features of Avocado - such as field contained in scene graph nodes and connections between fields - are standard features of OIV, but not part of Performer. In other words, extending OIV with DIV yields a tool which is less complex, but more complete than Avocado. In particular, backward compatibility with a large set of legacy applications is intriguing.

10. 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. For latest developments and results see http://www.cg.tuwien.ac.at/research/vr/div/.

11. 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.

12. REFERENCES

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

[2] 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.

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

[4] 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/.

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

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

[7] 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.

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

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

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

[11] 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.

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

[13] 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.

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

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

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

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

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

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

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

[21] Tramberend, H. Avocado: A Distributed Virtual Reality. IEEE Virtual Reality, 1999.

[22] 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.

[23] 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.