7. International Immersive Projection Technologies Workshop
9. Eurographics Workshop on Virtual Environments
J. Deisinger, A. Kunz (Editors)

The blue-c Distributed Scene Graph

Martin Naef[1], Edouard Lamboray[1], Oliver Staadt[2], Markus Gross[1]

[1]Computer Graphics Laboratory, Swiss Federal Institute of Technology, Zurich
[2]Computer Science Department, University of California, Davis
{naef, lamboray, grossm}@inf.ethz.ch, staadt@cs.ucdavis.edu

Abstract

In this paper we present a distributed scene graph architecture for use in the blue-c, a novel collaborative immersive virtual environment. We extend the widely used OpenGL Performer toolkit to provide a distributed scene graph maintaining full synchronization down to vertex and texel level. We propose a synchronization scheme including customizable, relaxed locking mechanisms. Our distributed scene graph includes both locally stored nodes for static scene data as well as nodes shared across multiple sites, thus minimizing synchronization overhead. We discuss the performance and demonstrate the functionality of our toolkit with two prototype applications in our high-performance virtual reality and visual simulation environment.

Keywords

Distributed graphics, scene graph, collaborative virtual environments, networked virtual reality

CR Categories and Subject Descriptors

I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism — Virtual Reality

1. Introduction

Over the past decade, the field of virtual reality has evolved rapidly and has proven to be a powerful technology for a wide range of application areas such as science, engineering and medicine. Recent advances in networking technology and the increasing availability of high-speed network connections enable researchers to interconnect immersive virtual reality systems at remote locations [15].

Distributed virtual environments can either be constructed based on underlying distributed databases [5, 4, 8] describing the world in abstract terms or scene graphs including the geometrical representation. Database approaches are most frequently used in virtual environments comprising a very large number of nodes. Due to rendering performance issues, however, a direct database traversal is less adequate for complex immersive environments with tight real-time constraints.

For that reason, most immersive VR applications are based on scene graph toolkits [21, 21, 15] which provide a hierarchical object-oriented scene representation. Toolkits used in stand-alone VR systems are usually not immediately suited for distributed applications due to the lack of built-in mechanisms for sharing application data in a consistent fashion across multiple sites. Thus, distributed scene graphs [23, 7, 13] have been developed to solve this problem. Figure 1 depicts a typical application scenario where a distributed scene graph is employed to share application data over a network.

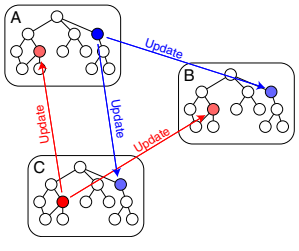

Figure 1: *Multiple sites sharing a common scene and propagating their respective changes.*

Ideally, a distributed scene graph should provide all features available in stand-alone scene graph toolkits without adding unnecessary programming complexity. Hence, adding distributed scene graph functionality to a widely-used toolkit ensures a large degree of backward compatibility to existing applications and provides a familiar programming environment for new applications.

We have developed the *blue-c Distributed Scene Graph* (bcDSG) in the context of the blue-c project [20, 10] – a collaborative tele-presence environment with simultaneous acquisition of 3D video [24] and immersive projection. We employ OpenGL Performer [21], which is one of the most widely-used toolkits for high-performance immersive VR applications, as the underlying scene graph. This allows us to use Performer features including file loaders, multi-processing, and flexible scene graph structures while maintaining high system performance. Although we will refer to Performer throughout the paper, the concepts presented here are as well applicable to other scene graphs such as OpenSG (www.opensg.org) or Open Scene Graph (www.openscenegraph.org).

The bcDSG shared nodes are replicated and, as opposed to other approaches, fully synchronized down to vertex and texel level. Furthermore, we do not rely on specific notification mechanisms and do not change the API paradigm of the underlying scene graph.

Note that this approach differs from shared memory approaches, where a single copy of the scene graph can be stored centrally, and from distributed immediate mode rendering, where rendering or interaction commands are sent in one direction to individual nodes in rendering clusters [15, 17, 1].

2. Related work

Various methods have been proposed to build networked virtual environments, such as NPSNET [12], RING [6], DIVE [5] and DIS/HLA [4, 8]. A detailed overview of these and similar systems can be found in [19]. These systems focus on large-scale virtual environments with synchronization happening at the application level, updating the logical world definition as opposed to the geometric representation. Recently, shared scene graph architectures have been proposed for cluster-based rendering [15, 17]. However, their design is optimized for distributing the scene graph to cluster nodes for rendering purposes only. Distributed modifications and subsequent synchronization of changes for collaborative applications is not supported. Architectures more closely related to the approach taken in bcDSG include Avango [23], Distributed Open Inventor [7], and Repo-3D [13].

Distributed Open Inventor is based on the Open Inventor toolkit and relies on its inherent notification scheme. Although this allows for an elegant implementation of scene graph distribution, the issue of *locking* is not addressed. The underlying Inventor toolkit, however, is not optimized for large real-time virtual environments and visual simulation applications and provides no built-in support for multi-processing and multi-pipe rendering. Furthermore, other scene graphs such as OpenGL Performer [21] or OpenSG [15] do not provide a similar notification mechanism.

Avango solves the problem of detecting changes by adding "Inventor-style" fields to Performer, replacing the original attribute access methods. This is achieved by subclassing Performer nodes and implies that only encapsulated Performer features can be used in distributed applications. Avango's encapsulation is not complete, e.g. the distribution of geometry leaf nodes is not implemented.

The Repo-3D architecture combines Modula-III, which provides native support for distributed objects, with a custom graphics solution. Although this results in an elegant solution, wide acceptance of this system for application development is unlikely in comparison to an architecture based on a widely-used scene graph toolkit and a mainstream programming language such as Performer and C++.

3. System overview

The blue-c API provides an application development environment which offers flexible access to all blue-c features, including graphics and sound rendering, device input, 3D video and scene distribution. These subsystems are provided as services and managed by the blue-c core (Figure 2).

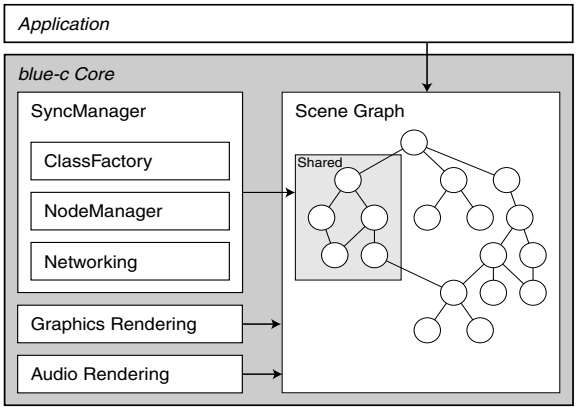

Figure 2: *System overview.*

The blue-c distributed scene consists of the following base components:

- The shared scene is built using special shared node classes. These shared nodes are derived from Performer objects and inherit an additional synchronization interface (Section 3.2).
- The synchronization service traverses the shared portion of the scene once per frame (Section 3.3). It generates, sends and handles scene operation messages. The synchronization service also includes a class factory and node and ID management (Section 4).
- The consistency, locking and ownership management mechanisms (Section 5) are implemented as part of the synchronization service.

- The network interface sends and receives scene operation messages. It also provides session and ID management (Section 6).

3.1. Scene Graph

The main data structure for all world-related data is a scene graph. It is based on OpenGL Performer and enhanced with customized nodes for 3D video rendering of the user representation [24], and contains audio and animation support.

The Performer scene is a directed graph built from different types of group nodes, geometry nodes and rendering attributes such as material, texture and lighting. These nodes and attributes are C++ objects. All nodes and attributes can be referenced several times within the graph. This allows for efficient memory usage and sorting to minimize the number of state changes when rendering the scene.

The blue-c distributed scene graph system follows a split approach. It divides the scene graph into a *shared* and a *local* partition. For performance reasons, only the shared partition is traversed by a synchronization service and kept consistent across all participating sites. The local partition is completely under the control of the local application. It is typically used for geometry that is loaded from files and forms the static environment of the collaborative application. Groups inside the shared partition of the scene can include non-synchronized children by using special reference nodes.

3.2. Shared nodes and connectivity

The shared portion of the scene must be built completely using customized nodes that include a node identifier, sharing state information and a serialization interface.

As Silicon Graphics provides no access to the Performer source code, the most elegant way to add the necessary interfaces is to derive new node classes from both the related Performer classes and the shared node interface as shown in Figure 3. Multiple inheritance allows to add an additional interface without affecting the original Performer methods. The shared classes are named after their Performer counterparts, replacing the prefix “pf” with our own “bc” (e.g., *pfDCS* becomes *bcDCS*).

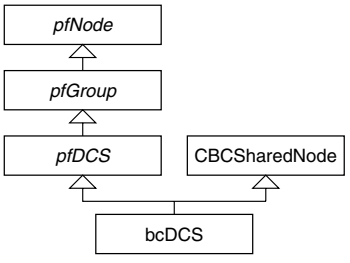

Figure 3: *Class diagram of a bcDCS node.*

Derived nodes include both the traditional scene graph nodes (such as groups, transformations, switches and geometry containers) as well as attribute classes (such as materials, textures and highlighting). Attribute objects are treated similar to children of group nodes, allowing for shared attributes between different nodes.

Both the traditional parent-child relationships in the graph and references to rendering attributes, e.g. texture objects, are defined as the *connectivity* information.

3.3. Synchronization service

The synchronization service traverses the shared portion of the scene once per rendering frame. Each visited node is checked for dirty flags. The necessary scene update messages are generated and enqueued for network transmission as explained in Section 4. The synchronization service also receives and handles messages from the partner sites.

The synchronization service expects reliable transmission of messages with a guaranteed order between two sites. It does not make assumptions about global message ordering, i.e. messages from different sites may arrive in any order.

In order to avoid temporary inconsistencies, such as rendering new geometry before material properties have been transmitted, messages from a single site are always processed in batches of a full frame.

The lack of global message ordering may lead to situations where updates for the same node may overtake each other (e.g. site A creates a new node, passes ownership to site B, which in turn sends an update. The update from site B may arrive at site C before the original create message from site A). In such cases, incoming messages may not always be processed immediately. The message is then put into a delay buffer and reinserted into the incoming message queue after the next incoming batch. A counter is increased each time the message is delayed, allowing to detect if there is a serious problem somewhere in the network.

The efficient handling of messages requires two additional subsystems that are part of the synchronization service:

- The *node manager* provides a fast node identification service, returning a pointer to the node data for an abstract ID.
- The *class factory* provides services to create a new node by specifying the class type name or Performer custom type information. It is also used for transforming Performer scene nodes into shared nodes.

4. Scene operations

The scene graph is kept consistent among the participating sites by sending messages with node modification operations. This section explains the message types and the node identification system.

4.1. Node identification

The usual node referencing using pointers does not work across local machine boundaries since no global address space is provided. Hence, a globally unique, abstract node identifier is required.

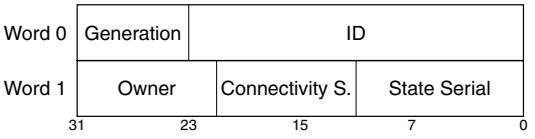

Figure 4: *Node identification structure.*

Each shared node has its own NodeID structure which consists of a 24 bit main identification number (ID) and an 8 bit generation number (Figure 4). The generation number allows reusing the ID number after the corresponding node has been deleted without risking wrong identifications caused by inconsistent message ordering across sites. New, globally unique NodeIDs are provided by the networking system (Section 6.4).

The ID number is used as direct index into an ID-to-node-reference table that is kept inside the node manager. This direct mapping allows very fast lookups, while ID reuse helps keeping the table reasonably small.

The ID structure also encodes node ownership as a 10 bit system ID. A quick ownership test is therefore based solely on the ID structure, and ownership requests can be sent directly to the current owner.

To make sure that outdated update messages do not override the most recent state, a serial number is stored for both connectivity and attribute state. Update messages always include the respective serial number. Only updates with higher serial numbers are processed by the synchronization service.

Since state serial numbers are only 12 bits long, they may wrap over in actual applications. Therefore, the comparison always uses “windows” to decide which number is more recent. In theory, this could break the consistency. In practice, there is still slack for almost a minute network delay in the case of an update every frame at 50 frames per second. Connectivity updates are much less frequent than state updates, hence shorter serial numbers are sufficient.

As shown in Figure 4, all ID information is encoded into two 32 bit words to keep the additional memory footprint per node minimal.

4.2. Node status and traversal

Each shared node keeps a set of flags, encoded in a single integer. These flags are:

- *New:* The node has just been created, only the local system knows of its existence.
- *State Dirty:* Attributes of the node have changed and must be broadcast.
- *Connectivity Dirty*: The connectivity of the node has changed. This especially applies to group nodes, however, material properties are also considered children of the node.
- *Request Ownership:* The local system wants to become owner of the node. No request has been sent yet.
- *Ownership Request Pending:* An ownership request for the current node has been sent. There is no answer yet.

These flags only reflect state information for the local copy of the node. They are never transmitted across the network. Additional state information such as the owner and update generation is encoded in the node ID presented above.

The application programmer is responsible for setting the state flags whenever Performer attributes or node connectivity has changed.

The shared partition of the scene graph is traversed once per rendering frame. The traversal is exhaustive, i.e., all nodes, including attributes such as textures, are traversed regardless of visibility. A node is visited only once per traversal by checking against a frame counter which is stored inside the node and updated during the first visit.

4.3. Operations on the scene

During the traversal, scene update messages are created according to the state of the visited node. The following subsections describe the message types and how they are handled by the receiver.

Create node. Whenever a new node is detected (i.e. the *New* flag is set), a *CreateNode* message is generated and broadcast. The message includes the class name in plain text and the ID of the new node.

At the receiver side, the class factory creates the required node object instance according to the provided class name and registers the node with the node manager.

Update state. For nodes with the *State Dirty* flag set, an *UpdateState* message is generated and broadcast. Shared nodes provide a streaming interface to serialize all Performer node attributes into the message transmission buffer. To guarantee consistency, only the owner of a node is allowed to send *UpdateState* messages (see Section 5). For each state update, the state serial number of the node is increased.

The receiver side first tries to locate the node by the ID. If the node ID is unknown, the *UpdateState* message is delayed. Only if the state serial number stored in the message is larger than the locally stored serial number, the node restores its state through the serialization interface. Otherwise, the update message has been superseded by another update message and can safely be discarded. Ownership information is also updated by the *UpdateState* message.

UpdateState messages always contain all Performer node attributes. Since the blue-c system is targeted at high-bandwidth networks, some redundancy in the transmission is compensated by less state bookkeeping overhead and therefore more efficient per-frame handling. This compromise also makes it easier to update the derived classes whenever new Performer features are added.

Update connectivity. Connectivity updates are similar to state updates. An *UpdateConnectivity* message is generated whenever the *Connectivity Dirty* flag is set. The list of the IDs of the child nodes is streamed into the message buffer. References to material property objects (e.g. pfGeoState and their children in Performer terminology) are treated just as regular children with their own ID.

Connectivity updates follow the same rules regarding serial number and ownership handling. The receiver side only updates the connectivity of the node if all child nodes are found, otherwise the message is delayed.

Delete node. Old nodes should be deleted to avoid memory leaks. If the application deletes a node, a *DeleteNode* message is generated. Only the owner of the node is allowed to delete a node.

4.4. Detecting orphaned nodes

After each scene traversal, orphaned nodes can easily be detected by iterating through the list of nodes in the node manager and checking the frame number of the last visit in the shared node data. Nodes that are not referenced in the scene have an old traversal number and can be deleted.

There may be situations where objects are only temporarily removed from the scene. To avoid automatic cleanup in this case, the application receives a delete request message where it can set a deny/grant flag for each object.

5. Consistency and locking

Any distributed system has to provide methods to guarantee data consistency across the nodes. Depending on the target application, there are different consistency requirements, e.g. some systems provide totally consistent data at all times whereas others allow for some transient differences between the sites.

Strict locking schemes that guarantee a consistent state at all times usually result in either relatively high latencies, scalability restrictions (e.g. traditional client-server models with a single master node) or highly complex systems. Strict consistency requirements are typical for applications where the participants compete against each other.

Temporary inconsistencies, however, are acceptable for many applications. Especially in collaborative applications where the users try to achieve a common goal and where they have additional communication channels at their disposal, e.g., a voice connection, minor glitches are perfectly acceptable as long as they do not result in permanent inconsistencies.

For the blue-c distributed scene graph, a relaxed locking scheme based on object ownership was implemented. By default, it provides immediate response to user interface actions by allowing for local modifications of nodes before ownership is acquired. For scenarios where a strict locking scheme is appropriate, the application developer may easily change the semantics as required.

5.1. Operations

The core concept of the consistency management is node ownership. To negotiate and pass ownership between different sites, three message types are defined:

Request ownership. A site requesting ownership of an object sends a *RequestOwnership* message to the site that is known as the last owner of the node.

If the receiving node is no longer the owner of the requested node, it forwards the request according to the locally stored ownership information. The message is forwarded until the current owner is found. In actual applications, repeated forwarding rarely occurs since new owners typically broadcast update messages soon after acquiring ownership, informing all other sites of the new ownership situation as a side effect.

Pass ownership. The *PassOwnership* message is sent as an answer to a request ownership message. It is sent directly to and transfers the ownership of the defined node to the requestor site.

The receiver node updates the local ownership information and clears the *Ownership Request Pending* flag of the node. It also signals the successful ownership transfer to the application.

Decline ownership. If ownership cannot be passed, a *DeclineOwnership* message is sent.

The receiver node clears the *Ownership Request Pending* flag of the node and informs the application of the declined request. If the local node is dirty, it may request a state update from the owner to restore the original state.

5.2. Relaxed locking and application interaction

The relaxed locking scheme provided by the system allows an application to modify a node at any time, even if it is not the owner. This results in instant visual feedback for editing operations on the scene. If a modified node does not belong to the site, ownership is automatically requested to acquire the right to broadcast an update message (Figure 5).

The current owner always sends its last update message before it passes ownership. The guaranteed ordering between two nodes then ensures that the new owner always knows the most recent state and therefore never propagates changes based on an outdated state.

Update messages arriving for dirty nodes override the local settings and clear the dirty flags. This results in a "jumping" object if multiple sites try to modify the node at the same time. However, the inconsistencies are only temporary, the owner always "wins", returning the system to a consistent state after a delay of some frames. This behavior is perfectly acceptable for a collaborative modelling scenario. For applications where this behavior is undesirable, the solution is simply not to update a node until ownership is acquired.

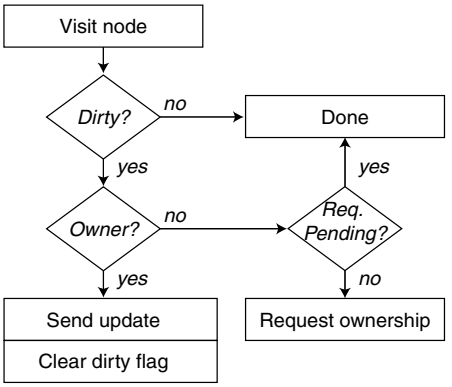

Figure 5: *Locking scheme: Per frame processing.*

By default, the system receiving the ownership request passes ownership immediately if its local instance is not dirty. If it is dirty, the request is delayed until the state update has been broadcast.

An optional "application loop" gives additional control over the ownership management to implement application specific locking schemes. For each ownership request, the system sends an ownership request information message to the application. The application may then decide to decline the request by setting a flag in the message. A typical implementation would decline an ownership request if the node is currently selected for modification.

6. Networking

Since our tele-immersive collaborative virtual reality system requires a versatile and platform-independent networking layer, we decided to build the blue-c communication services upon a distributed object computing standard. We chose a CORBA-based framework for designing and implementing the various communication tasks, which also include the distribution of a scene graph. The core networking functionality is based on the CORBA Audio/Video Streaming Service [14, 3]. Within this approach, the communication channels are initialized using CORBA remote method calls. The actual payload, however, is directly streamed to the network interface, e.g., using IP sockets or ATM adaptation layers.

The core part of the connection setup is implemented in a central session server, which is also used for ID distribution (see Section 6.4). Furthermore, we extended the basic Audio/Video Streaming Service implementation for our specific needs. Data sources and sinks are handled through link endpoints, which locate their communication partners using the CORBA Naming Service. The link endpoints are responsible for establishing a connection according to a given specification, which includes attributes concerning reliability, the underlying transport protocol and unicasting or multicasting. Figure 6 shows an overview of the network configuration.

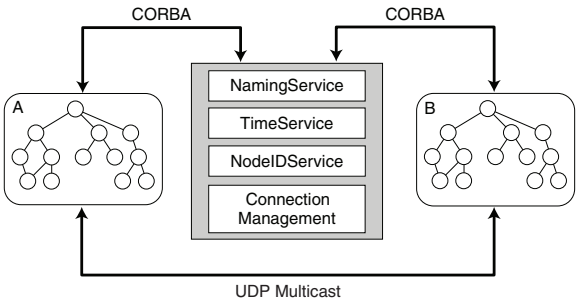

Figure 6: *Networking overview.*

6.1. Communication channel configuration

As explained in Section 4, keeping the blue-c shared scene graph consistent requires a reliable transmission of the scene graph operations from each participating site. However, no total ordering of the scene graph operations is required. The TCP protocol guarantees a reliable transmission, i.e., the data is transmitted loss-free and in order. However, TCP is not very well suited for real-time systems because of its built-in flow control mechanisms. Furthermore, TCP cannot be used in a multicast setup. As a consequence, we implemented an appropriate scheme for reliable data transmission based on the connectionless and unreliable UDP protocol and on explicit positive and negative acknowledgements. The technique we use is similar to the Reliable Multicast Protocol RMP, used in the source-ordered reliability level [23].

The use of IP resource reservation protocols, like RSVP [2], during connection setup, or of UDP-based RTP/RTCP [18] allows for the future support of quality of service features within the system.

6.2. Shared scene graph message transmission

Once the communication link is established, our framework transmits messages containing the actual payload, i.e., the shared scene graph operations. The scene graph operations are streamed into appropriate messages, which are then written into transmission buffers. The format of a shared scene graph message is specified in Table 1.

A transmission buffer can either be directly flushed to the network, or it is automatically sent, after it has been com-

pletely filled with messages. It can contain a single message, several messages, or a frame of a large message that does not fit into a single transmission buffer. Especially messages resulting from scene graph operations including texture updates can be larger than the 4 kB size limit of the transmission buffers.

ATTRIBUTE	SIZE
MessageType	4 bytes
MessageSource	2 bytes
MessageDestination	2 bytes
NodeID	8 bytes
Timestamp	8 bytes
PayloadLength	4 bytes
PayloadData	PayloadLength

Table 1: *Format of a shared scene graph message.*

At the receiver, a dedicated task activates a callback upon buffer reception. In this callback, the messages, or message frames, are extracted from the received transmission buffer. All completely received messages are then forwarded to the upper layers, in which the scene graph operations are reconstructed and applied to the local copy of the shared scene graph.

Note that we also include a timestamp in the shared scene graph message. It is not required for consistent scene graph updates, but it can be exploited by the application. A global time for distributed applications is implemented using the CORBA Time Service.

All memory management for buffers and messages is handled by the memory pools of the communication framework. Typical message and transmission buffers are allocated during application start-up. The communication framework also handles hardware specific issues, like little/big endian encoding. The implementation of the communication framework is built on the TAO/ACE framework (http://www.cs.wustl.edu/~schmidt/TAO.html).

In order to assess the performance of our implementation, we measured the round-trip time of shared scene graph messages between different hosts in the same local area network. We generated transmission buffers of size 4 kB at the scene update rate of 50 frames per second, i.e., a constant traffic of 1.6 megabit per second, and we obtained average round-trip times of less than 3 milliseconds.

6.3. Multicasting issues

As soon as N, the number of participating sites, becomes larger than two, the networked communication can be done more efficiently using multicast protocols. As described in Section 4 and Section 5, we distinguish between two categories of messages:

- Scene operations which are sent to all participating sites.
- Locking operations which need to be transmitted only between the two concerned sites.

This scenario can be translated into a communication link setup which consists of one multicast channel for scene operations and a unicast channel in between each two sites for locking operations. In this setup, the transmission of locking messages will only reach the sites which are directly involved in the operation.

However, the increasing number of point-to-point connections for large N makes this setup not very feasible in practice. Hence, for large N, all messages are transmitted over the multicast channel and locking operations need to be dismissed by the non-involved sites.

6.4. Distributed node identifiers

During application start-up, the participating sites register themselves at a session server, from which they obtain references to their communication partners (see Section 6.1). Furthermore, the main server distributes a different range of authorized node IDs to every client application joining the shared session. The centralized distribution of node IDs guarantees that every client will use distinct IDs. At any time during the application, a given node ID resolves to at most one node in the shared scene graph.

During the lifetime of the application, every client keeps a list L of authorized, but not-assigned IDs. In case a client creates a new node, it removes an ID from L and assigns it to the new node. In an analogous way, a client deleting a node increments the corresponding ID's generation counter and reintroduces the ID into its list L of authorized, but not-assigned IDs. Note that only the client triggering the delete operation, i.e., the node's last owner, introduces the ID into L, even if it does not fall into its initial range of possible IDs. Hence, this algorithm allows for efficient reuse of node IDs and avoids situations in which the same ID refers to two different nodes. Furthermore, a client realizing that the size of L, i.e., the number of available IDs, falls below a threshold, requests an additional range of IDs at the session server. Thus, a real shortage of available IDs and any related delays can be avoided.

7. Example applications

We implemented two collaborative example applications based on the blue-c distributed scene graph. The applications take advantage of the full scene graph synchronization, including texture updates, and use different locking schemes.

Figure 7: *Example applications: Distributed chess, collaborative painter.*

7.1. Distributed chess

An immersive, larger-than-life chess game was implemented as a classical example of a multi-user game. The application features all core concepts of the distributed scene system.

The chessboard is located inside an old building. The room geometry and textures are loaded from a file independently by all sites and reside in the non-synchronized partition of the scene. The chessboard is built only by the first site in the shared partition. Its geometry, attribute and texture data is streamed to the other sites using the synchronization features. The geometry of the pieces is loaded from files. They are inserted into the shared partition by reference nodes.

The application includes little game logic. It only allows to pick and move the chess pieces on the board and removes hit pieces with a small animation sequence.

Picking and moving the pieces follows a strict locking scheme: Pieces are only moved after object ownership is acquired. Ownership requests for picked pieces are declined. This makes sure no two players move the same piece at the same time.

The animation sequence for removing pieces from the board simply updates the geometry and sets the dirty flags, using the default relaxed locking scheme implemented by the system.

7.2. Collaborative painter

The collaborative painter tool was inspired by the CavePainting application [9]. The user has the option to create simple geometrical objects (quads) of different colors. He can then paint onto the surface of these quads, i.e., modify the texture. With the exception of the color palette and a visual cue for the tracker position in the form of a spray can, all objects, including textures, are shared between the nodes. All users immediately see all of the scene updates.

8. Discussion and future work

This paper presents a shared, distributed scene graph based on OpenGL Performer. It provides a relaxed locking scheme optimized for collaborative work. The extensions do not impose any penalty on the local rendering performance and require little additional considerations from the application developer.

The system was optimized for collaborative applications in a controlled, high-bandwidth environment. It relies on source-ordered reliable network transmission.

In the current implementation we decided not to encapsulate a few Performer node types such as outdated structures (e.g. pfLightPoints) and very large data objects (e.g ASD terrain nodes and image based rendering objects) that are unlikely to be included in a shared scene. These features can still be used in the local partition of the scene and be referenced from the shared partition if needed.

The current system transfers complete node state information with a single message. This level of granularity is well suited for the standard Performer node types. For performance reasons, however, customized nodes with large and complex internal data structures should be synchronized at the field level. The necessary extensions will be provided in future versions of the blue-c distributed scene graph.

At this moment, the system does not allow late joining of sites. Instead, there is a single rendez-vous point at application startup. Similar to Avango, late joining could be implemented by requesting a complete graph update from a single node in the system and add a global, non-blocking rendez-vous and buffering system to guarantee no pending updates are missed.

Acknowledgements

We would like to thank all members of the blue-c team for many inspiring discussions. This work has been funded by ETH Zurich as a “Polyprojekt” (grant no. 0-23803-00).

References

1. J. Allard, V. Gouranton, L. Lecointre, and E. Melin. “Net Juggler and SoftGenLock: Running VR Juggler with active stereo and multiple displays on a commod-

ity component cluster." In *Proceedings of the IEEE Virtual Reality 2002*. IEEE Computer Society, 2002.

2. R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. "Resource ReSerVation Protocol (RSVP)." RFC 2205, Sept. 1997.
3. "Audio/Video Stream Specification." Object Management Group, Jan. 2000.
4. "IEEE standard for information technology - protocols for distributed simulation applications: Entity information and interaction." IEEE Standard 1278-1993, 1993.
5. E. Frécon and M. Stenius. "DIVE: A scaleable network architecture for distributed virtual environemnts." *Distributed Systems Engineering Journal*, 5:91–100, Sept. 1998. Special Issue on Distributed Virtual Environments.
6. T. A. Funkhouser. "RING: A Client-Server System for Multi-User Virtual Environments." In *Computer Graphics (1995 SIGGRAPH Symposium on Interactive 3D Graphics)*, pages 85–92, Monterey, California, April 1995.
7. G. Hesina, D. Schmalstieg, A. Fuhrmann, and W. Purgathofer. "Distributed open inventor: A practical approach to distributed 3D graphics." In D. Brutzman, H. Ko, and M. Slater, editors, *Proceedings of the ACM symposium on Virtual reality software and technology*, pages 74–81. ACM Press, 1999.
8. "Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - Framework and Rules." IEEE Standard 1516, September 2000.
9. D. F. Keefe, D. A. F. Rodrigues, T. Moskovich, D. H. Laidlaw, and J. J. L. Jr. "CavePainting: A fully immersive 3d artistic medium and interactive experience." In *Proceedings of the 2001 Symposium on Interactive 3D Graphics*, pages 85–93, March 2001.
10. A. Kunz and C. Spagno. "Simultaneous projection and picture acquisition for a distributed collaborative environment." In *Proceedings of IEEE Virtual Reality 2002*, pages 279–280, 2002.
11. J. Leigh, A. E. Johnson, and T. A. DeFanti. "CAVERN: A distributed architecture for supporting scalable persistence and interoperability in collaborative virtual environments." *Journal of Virtual Reality Research, Development and Applications*, 2(2):217–237, Dec. 1997. The Virtual Reality Society.
12. M. R. Macedonia, M. J. Zyda, D. R. Pratt, P. T. Barham, and S. Zeswitz. "NPSNET: A Network Software Architecture for Large Scale Virtual Environments." *Presence*, 3(4), Fall 1994.
13. B. MacIntyre and S. Feiner. "A distributed 3D graphics library." In M. Cohen, editor, *Proceedings of SIGGRAPH 98*, pages 361–370. Addison Wesley, 1998.
14. S. Mungee, N. Surendran, Y. Krishnamurthy, and D. C. Schmidt. *The Design and Performance of a CORBA Audio/Video Streaming Service*, chapter in Design and Management of Multimedia Information Systems: Opportunities and Challenges. Idea Group Publishing, 2000.
15. D. Reiners, G. Voss, and J. Behr. "OpenSG - Basic concepts." 1. OpenSG Symposium, 2002.
16. J. Rohlf and J. Helman. "IRIS Performer: A high performance multiprocessing toolkit for real-time 3d graphics." In *Proceedings of SIGGRAPH 94*, ACM SIGGRAPH Annual Conference Series, pages 381–395, 1994.
17. B. Schaeffer. "Networking and management for cluster-based graphics." http://www.isl.uiuc.edu, Mar. 2002.
18. H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. "RTP: A Transport Protocol for Real-Time Applications." RFC 1889, Jan. 1996.
19. S. Singhal and M. Zyda. *Networked Virtual Environments: Design and Implementation*. ACM Press - SIGGRAPH Series. Addison-Wesley, 1999.
20. O. G. Staadt, A. Kunz, M. Meier, and M. H. Gross. "The blue-c: Integrating real humans into a networked immersive environment." In *Proceedings of ACM Collaborative Virtual Environments 2000*, pages 201–202, San Francisco, Sept. 2000. ACM Press.
21. P. S. Strauss and R. Carey. "An object-oriented 3D graphics toolkit." In *Proceedings of SIGGRAPH 92*, ACM SIGGRAPH Annual Conference Series, pages 341–349, 1992.
22. H. Tramberend. "Avocado: A distributed virtual reality framework." In *Proceedings of IEEE Virtual Reality 99*, pages 14–21, 1999.
23. B. Whetten, T. Montgomery, and S. M. Kaplan. "A High Performance Totally Ordered Multicast Protocol." In *Dagstuhl Seminar on Distributed Systems*, pages 33–57, 1994.
24. S. Wuermlin, E. Lamboray, O. G. Staadt, and M. H. Gross. "3D video recorder." In *Proceedings of Pacific Graphics '02*. IEEE Computer Society Press, 2002.

Figure 7: *Example applications: Distributed chess, collaborative painter.*