EUROGRAPHICS 2000 / M. Gross and F.R.A. Hopgood
(Guest Editors)

Volume 19 (2000), Number 3

Scene-Graph-As-Bus: Collaboration between Heterogeneous Stand-alone 3-D Graphical Applications

Bob Zeleznik, Loring Holden, Michael Capps, Howard Abrams, and Tim Miller

{bcz, lsh, tsm}@cs.brown.edu *{capps, howard}@cs.nps.navy.mil*
Brown University, Providence, Rhode Island
Naval Postgraduate School, Monterey, California

Abstract

We describe the Scene-Graph-As-Bus technique (SGAB), the first step in a staircase of solutions for sharing software components for virtual environments. The goals of SGAB are to allow, with minimal effort, independently-designed applications to share component functionality; and for multiple users to share applications designed for single users. This paper reports on the SGAB design for transparently conjoining different applications by unifying the state information contained in their scene graphs. SGAB monitors and maps changes in the local scene graph of one application to a neutral scene graph representation (NSG), distributes the NSG changes over the network to remote peer applications, and then maps the NSG changes to the local scene graph of the remote application. The fundamental contribution of SGAB is that both the local and remote applications can be completely unaware of each other; that is, both applications can interoperate without code or binary modification despite each having no knowledge of networking or interoperability.

1. Introduction

A continuing software engineering challenge for virtual environments (VE's) is to enable independently-designed, and possibly pre-existing, software components to be shared with minimal effort. This sharing can take two forms: the creation of hybrid applications from independent application components, and the distribution of a single-user application to enable multiple participants.

Unfortunately, given the immaturity of VE's, software components for interaction techniques or geometric behaviors are rarely, if ever, re-used outside the systems in which they were created. Furthermore, despite the progress in collaborative virtual environments, many existing applications are designed to support only a single user.

Those systems that do support sharing typically require a common standardized interface for component interoperability and distribution. This approach can lead to efficient and elegant solutions, especially if components are not designed independently. However, agreeing upon and enforcing a standard is often not possible; and for scalability, components must be designed independently at least to some degree.

The alternative to sharing via a standardized interface is to unify the inherently similar data of different applications. Patterns of data that differ in structure, but serve the same purpose, can theoretically be translated between to effect the appearance of a unified data structure. However, the quality and efficiency of applications that share data with this technique may be limited by lossy mappings and low-level communication.

Our long-term goal is to provide the highest quality sharing while still allowing for maximum independence between computer graphics components. We recognize that such a goal may be impossible to realize ideally, so instead we envision a solution consisting of a staircase of gradual steps whereby the first step easily will allow some sharing between independent components, and subsequent steps will improve the degree and quality of sharing at the cost of additional work and less component-independence.

This paper outlines the first step in the staircase of so-

© The Eurographics Association and Blackwell Publishers 2000. Published by Blackwell Publishers, 108 Cowley Road, Oxford OX4 1JF, UK and 350 Main Street, Malden, MA 02148, USA.

lutions, whereby independently-designed applications built atop heterogeneous frameworks can cooperate, to a degree, in a shared virtual environment.

2. Overview

Despite vast differences between computer graphics applications, there is one general commonality—a scene graph. Although applications may use different scene graph libraries, the underlying features of each scene graph are generally quite similar and reasonable mappings between them can be made. Furthermore, many scene graphs provide a monitoring facility where one or more callbacks can be installed for notification whenever elements of the scene graph are modified. Therefore applications which share the similar concept of a scene graph and that additionally support scene graph monitoring can be made to communicate implicitly through their scene graphs.

We enable scene graph sharing by our Scene-Graph-As-Bus (SGAB) approach. SGAB dynamically interjects callbacks into an application's scene graph in order to monitor changes, and then when changes occur, SGAB heuristically maps those changes to a common Neutral Scene Graph (NSG). SGAB distributes the NSG changes to remote SGAB peers, which in turn heuristically map the changes to the remote application's scene graph. SGAB also enables scene graph nodes to be annotated with contention codes that specify how scene graph nodes are distributed and how modifications can occur. These contention codes, for example, can be used to optimize networking protocols for a particular node, or to ensure local-only scene graph elements are not distributed.

Although SGAB is applicable to new and existing applications alike, our focus has been to enable two forms of sharing between existing applications without modification of program code or binaries:

- Networking a shared virtual environment between multiple heterogeneous participant applications that were designed for single users.
- Forming hybrid applications out of independently-designed applications.

3. Prior Work

Computer-supported collaboration, and distribution of graphical data, are mature areas of computer science. A number of previous efforts share some portion of the goals of this project, but we know of no system to date that has embodied all of our objectives. In this section we discuss the relevant prior art, and delineate the innovations of the SGAB technique.

A number of virtual environment systems were designed to include networked collaboration. For example, the Reality Built for Two[4], SPLINE[17], NPSNET[10], and DIVE[6] systems all have a notion of shared graphical objects and communication of state changes to those objects. Each of these projects takes a different approach to the distribution of initial object state, network topology, and collaboration paradigms, but all assume homogeneous client software. The Distributed Interactive Simulation (DIS)[1] and High-Level Architecture (HLA)[9] standards enable cooperation between heterogeneous clients, as long as they follow a set of network protocols. The SGAB approach is instead intended for bridging the informational gap between independently-designed stand alone systems, with minimal or no modification to those systems. SGAB is specifically not intended to be a preferred communication infrastructure, and would not compete with other VE frameworks in that regard. A review of networked virtual environment architectures, and a tutorial for these standard methods of information sharing, can be found in Singhal and Zyda's text.[14]

Forming hybrid applications out of independently-designed applications, without a previously-arranged communications interface, has been performed with some success in non-3D environments. For instance, the Artifact-Based Collaboration system from the University of North Carolina[13] included the ability to incorporate independently-designed applications. Modified text and image editors could work with shared objects to allow implicit collaboration.

The X Teleconferencing and Viewing (XTV) system[2] is one of a number of systems that allows users at multiple sites to view and manipulate the output from a single X-based application. A floor control model allows one or more sites to give mouse and keyboard input to the application, and the output view is broadcast to all participating sites. XTV is similar to the SGAB approach, in that it uses the 2-D bitmap (rather than the 3-D scene graph) as the shared data element. Most applications require no modification for XTV unless complex collaboration semantics and protocols are desired, again similar to SGAB. Other CSCW systems incorporate more complex data distribution, but do not lend themselves to sharing graphical 3-D content.

Communications in SGAB incorporate a mapping-layer approach, in which each client's internal data representation is mapped to a special representation for transmission, and then mapped to the recipient's format. The Polylith[5] system incorporates a software bus that facilitates communication between applications on heterogeneous architectures. Data packets are similarly converted from client-specific representations to an internal transmission format, and then converted back before delivery. The CORBA standard[3] takes a similar approach, specifically for language- and architecture-independent inter-process communication. Both methods are networking libraries which require the application to be modified pervasively to send and receive data.

The Distributed Open Inventor (DIV) project[8], which was developed simultaneously with SGAB, embodies many of the design goals of our system. DIV uses the scene graph as a

© The Eurographics Association and Blackwell Publishers 2000.

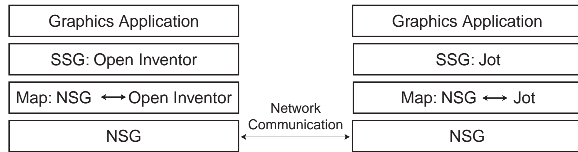

Figure 1: *Interaction between SGAB layers in a heterogeneous environment.*

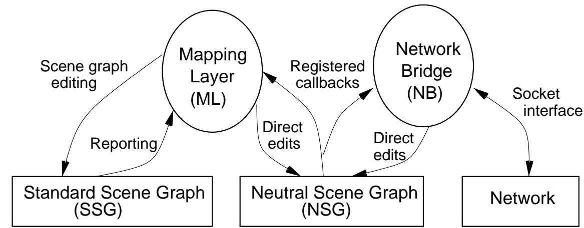

Figure 2: *The SGAB architecture depicts how the mapping layer (ML) bridge interacts with the NSG (Neutral Scene Graph) and how the network bridge (NB) maps the NSG to the network.*

shared memory structure, and it encourages the authoring of graphical applications that are distributed in a manner nearly transparent to the programmer. The system also includes excellent high-performance networking facilities. GMD's Avocado system[16] similarly distributes data by transparent replication of the scene graph, in this case that of the Performer graphics library, on SGI systems. The primary difference between these systems and SGAB is that SGAB offers similar replication functionality between a variety of graphical libraries, in a cross-platform manner.

4. SGAB Theory and Design

SGAB is intended to extend the functionality of scene graphs so that they behave as a shared data structure. To effect this distributed behavior, SGAB uses a single shared data hierarchy, known as the abstract or neutral scene graph (NSG). In order to minimize its impact on application design, SGAB provides mappings between this abstract NSG and the specific scene graph used by each application. The number of scene graphs is much smaller than the number of applications, so many applications can be integrated by writing only a small number of mapping layers.

The abstract architecture of SGAB can be summarized as three logical layers (as seen in Figure 1):

- **The standard scene graph (SSG) layer.**
  This is an unmodified scene graph library, such as Open Inventor [15]. This layer may also contain, as a sub-layer, any extensions to the SSG necessary to allow the mapping layer to perform its function. For example, IRIX Performer [12] does not provide facilities for reporting scene graph changes, and therefore a Performer mapping layer would have to be extended.
- **The mapping layer (ML).**
  This layer maps between operations on the SSG and operations on the NSG. It includes functions, callbacks, and mapping data. This layer additionally sets default contention codes in the NSG layer to specify the naïve distributed application behavior. Since this layer is SSG-specific, there will be a different version of this layer for each SSG supported.
- **The neutral scene graph (NSG) layer.**
  The NSG is a special abstract scene graph that it is not displayed; it exists only as a intermediate 3-D graphical data structure. An identical NSG is shared between all participants in a networked SGAB session. In addition to the data formats and contention resolution mechanisms, it must also specify the scene graph data semantics to be implemented in the translation of the mapping layer. Note that the NSG is necessarily memory-independent from the SSG; e.g. a transformation matrix must be stored redundantly in the SSG and the NSG. For large scenes, this can be a significant increase in memory usage.

The abstract NSG is designed to communicate directly with the application's scene graph, and encapsulates all knowledge of how data is distributed over the network. Thus, the typical application programmer programs directly to an unmodified scene graph interface, ignoring any complexities of scene graph distribution. However, for greater control of efficiency and shared scene graph behavior, application programmers may also directly modify the NSG. In particular, the NSG consists of abstract scene graph nodes augmented by distribution-specific *contention codes*. Non-naïve (SGAB-aware) applications can modify the contention codes for NSG scene graph nodes, through an API in the mapping layer, to refine the distributed behavior of the application.

5. SGAB Implementation

The SGAB implementation is grounded upon the use of bridges to map changes between independent data representations. In particular, each SGAB client consists of two bridges depicted in Figure 2: a mapping layer (ML) bridge between an NSG and an application's SSG and a network bridge (NB) between a local NSG and a remote NSG via the network. Both bridges are able to directly call the APIs of their bridged data, however to preserve their independence and to enable different bridge implementations, the bridged data cannot know about the bridge. Instead the bridge data supports reporting – a process by which the bridge can register functors to be called on the occurrence of a specific type of event, such as the creation or modification of a scene graph node. Thus, when a change is made to an element of a bridged data representation, the bridge will be notified to convert the data to the other bridge endpoint. An essential implementation detail is to insure that bridges are anti-

© The Eurographics Association and Blackwell Publishers 2000.

looping; that is when the bridge itself writes a change to one representation it should not be automatically notified of the change causing it to write the change back to the other representation.

Although each ML bridge can be designed differently, the pattern we have found to be effective is to treat the ML as a collection of functors that map between similar low-level scene graph concepts, such as a transformation matrix, a color, or a geometry. The ML as a whole is implemented as a single C++ class as are each of the mapping functors. The ML, a collection of functor instances, is responsible for invoking the appropriate functor in response to callback events from either the NSG or SSG. Each functor implements the specialized mapping behavior for converting changes between the SSG and NSG. Thus when the application makes a change to the SSG, a functor registered by the ML to handle that change is invoked to map the change to the NSG using the NSG's API. To achieve specialized behavior, applications are allowed to call the ML functors directly, but otherwise are unaware of either the ML or the NSG.

The NSG implementation reflects a simplified, fully populated, graphical scene graph, but without rendering facilities. The NSG purpose is twofold: to serialize NSG changes with services for byte ordering, pointer-to-uniqueID translation, etc; and to provide all the possible context for any given ML to map between NSG and local SSG changes. This latter issue is critical because different scene graphs may store data of different granularity. For example, a change to a single transform node in one SSG may map to changes in multiple objects in a different scene graph. Since complete copies of the NSG are maintained for each SGAB client, ML's have sufficient context to map any NSG change to their local SSG.

SGAB transmits changes between NSG clients using the NB. The NB captures all NSG changes (via NSG's reporting mechanisms), serializes them, and then distributes them over the network to remote NB's. Remote NB's receive and unpack changes from the network, and then apply them to the NSG, which in turn invokes the ML to edit the SSG.

5.1. NSG Implementation

The NSG is implemented by a set of nodes that contain fields. Each node represents a low-level semantic element of a scene graph such as color, transformation, or polygonal geometry. Fields are the "instance" data for the nodes and are implemented by a static, nonextensible set of enumerated primitive types shown in Table 1. The only field that is non-trivial is the `NSGfield_noderef`. Unlike all other fields which are intended to represent the same data in the same way on all networked NSGs, the `NSGfield_noderef` localizes the same node data for specific NSG instances. The `NSGfield_noderef` is a group node in which each node reference is augmented with an `NSGclient_id` application identifier. When the `NSGfield_noderef` is encountered during traversal, only the node references with `NSGclient_id`'s matching the local NSG are traversed. Thus although each NSG client has identical `NSGfield_noderef` data, that data is interpreted differently by each NSG client. This allows local-only node references, which enables each SGAB client to have unique scene graph variations. A specific occasion when such unique variations are necessary is the top-level `NSGsep_node`, representing the root of the NSG scene graph on each SGAB client. Since some applications are hard-coded to treat a specific, irreplaceable node as the root of their SSG, each SGAB client must be allowed to treat a different node as the NSG root. However, by using local-only `NSGfield_noderefs`, each SGAB client can make their different root nodes appear the same as shown in Figure 3.

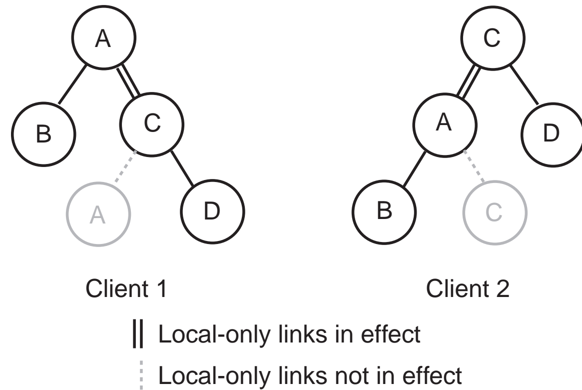

Figure 3: *The scene graph sharing process. Each client stores remote scene graphs as a child of its local root node.*

Field	Description
`NSGfield_double`	64 bit floating point number
`NSGfield_int`	32 bit signed integer
`NSGfield_string`	String of ASCII characters
`NSGfield_vec3`	Three doubles
`NSGfield_matrix`	4x4 matrix
`NSGfield_noderef`	A reference to a node

Table 1: *Neutral Scene Graph (NSG) fields*

NSG nodes consist of the following:

- `name` - the globally unique name for the node
- `tag` - node type
- `fields` - current field settings
- `oldfields` - last field settings
- `ccode` - contention flags governing network behavior
- `creator` - identify of SGAB client that created this node

Node names uniquely identify nodes across the network so that changes to local NSG nodes can be applied to corresponding remote NSG nodes. The tag on the node is used by the ML to determine the node's semantics, as shown in

© The Eurographics Association and Blackwell Publishers 2000.

Node	Description
NSGsep_node	Children (NSGfield_noderef list)
NSGcolor_node	Color in RGB (NSGfield_vec3)
NSGxform_node	Transformation Matrix (NSGfield_xform)
NSGcone_node	Cone primitive
NSGcube_node	Cube primitive
NSGcyl_node	Cylinder primitive
NSGsphere_node	Sphere geometry primitive
NSGmesh_node	Vertices & triangles, optionally per-vertex color & UV coords.
NSGtexture_node	An image

Table 2: *Neutral Scene Graph (NSG) nodes*

Contention code	Description
Remote Callback	Call remote change callbacks
Remote Edit	Accept remote change
Local Callback	Call local change callbacks
Distribute	if node creator: distribute change; else: send change to creator
Local Edit	Accept local change

Table 3: *NSG contention codes*

Table 2, in order to decide how to map the NSG change to the SSG. The `fields` value of a node is a list of the current "instance" data for the node and directly corresponds to the data that is stored in the SSG. The `oldfields` value of a node is the previous `fields` value for the node, and is used with the contention codes to cancel a node edit when a node is locked.

The most complicated field on the NSG node is the `ccode`, an ordered set of boolean flags that express the behavior of an NSG node when it is changed. The five boolean contention codes shown in Table 3 can be set independently in order to effect a range of different behaviors. For example, the concept of a lock can be effected by setting the Local Edit contention codes on one SGAB client and setting the Remote Edit (but not the Local Edit) code on all other clients. If an SGAB client modifies an SSG nodes that maps to an NSG node with only the Remote Edit contention codes set, then the NSG automatically undoes the change by overwriting the application's SSG change with the value stored in the NSG node. However, given the success of using default contention codes that enable Distribute and Remote and Local Edit, we have not yet implemented mechanisms for explicitly coordinating the contention codes between different clients.

The `creator` field of an NSG node identifies the SGAB client that created the node. NSG uses the `creator` field to avoid cycles when nodes are edited. When an SGAB client modifies an NSG node, the change is distributed only to the SGAB client that created the node. The creator then may accept, modify, or reject the change. If the change is not rejected, the creator will distribute the change to all other SGAB clients including the client originating the change. When SGAB clients receive network changes to a node they did not create, they apply the change but do not distribute it.

5.2. Open Inventor

We have created an ML for Open Inventor [15] that can be used by existing Open Inventor applications without code modification or recompilation. The ML attaches to an Open Inventor scene graph by using a run-time binding feature allowing a shared library to interpose on application function calls intended for a different shared library. Our ML intercepts application calls to both `SoXtRenderArea::setSceneGraph()` and `SoXtViewer::setSceneGraph()`, and before invoking the intended Open Inventor function translates the Open Inventor scene graph to NSG and installs callbacks using `SoNodeSensor`. Through the installed callbacks, the ML gets updates whenever the application modifies the scene graph, enabling the ML to map the changes to the NSG.

Since, as it happens, state in the NSG is inherited top-down, left-right (the same as Open Inventor), the mapping between these scene graphs is simplified. Certain Open Inventor nodes have a relatively trivial one-to-one mapping back and forth to NSG, such as `SoBaseColor`, `SoTexture2`, `SoCube`, `SoCylinder`, and `SoSphere`. `SoIndexedTriangleStripSet`, on the other hand, has a one-to-one mapping with NSG only in certain cases. The one-to-one mapping is possible when all vertex, triangle, color, and texture UV coordinate state for the triangle mesh is encapsulated in the Open Inventor node, as opposed to being inherited from other nodes. This is similarly true for many other nodes, such as `SoFaceSet`, `SoQuadMesh`, and `SoTriangleStripSet`.

A node mapping special case is the mapping between Open Inventor's `SoSeparator` and NSG's `NSGsep_node`. Both versions need to be kept consistent, but finding changes are non-trivial since some children of the `NSGsep_node` don't belong in the `SoSeparator` and vice versa. For example, client-specific children on the NSG side are only manifest on a particular SGAB client, and shouldn't be mapped to `SoSeparator`'s on other clients. Similarly, certain Open Inventor nodes can't or shouldn't be mapped to NSG. An added wrinkle is that these unmapped nodes can not be ignored, since these mappings have to take into account the unmapped nodes when comparing the two different versions of the nodes to find differences.

© The Eurographics Association and Blackwell Publishers 2000.

5.3. Jot

We also created a mapping layer for Jot, a computer graphics system used for research projects at Brown University. The strong contrast between the scene graph structures of Jot and Open Inventor provides an important foundation for evaluating the generality of the SGAB approach. Open Inventor's scene graph uses a hierarchical state inheritance model in which scene graph nodes represent pieces of rendering state (geometry, color, etc), whereas Jot's scene graph is a flat list of objects that each encapsulate all necessary rendering state including color, geometry, texture, and transformation.

Unlike the run-time binding interposition needed by the Open Inventor ML, the Jot ML attaches in a very straightforward manner by using a hook in the Jot system for dynamically loading a shared library on startup. Thus, the Jot ML can install callback functors as part of the initialization of the Jot ML upon startup of a Jot application.

Although mapping a Jot object to NSG is a one-to-many operation, Jot reports changes to specific object fields thereby requiring only directly affected subsets of the full mapping to be updated. Thus a change to a Jot object's color field maps directly to the `NSGcolor_node` that is closest in traversal order to the `NSGshape_node` corresponding to the Jot objects' geometry. However, changes to a Jot object's transformation field may map to a cumulative chain of `NSGxform_nodes`. We handle this one-to-many mapping by finding all NSG transformation nodes along the path from the NSG's root to the `NSGshape_node` corresponding to the Jot objects' geometry. Then, we update only the last node in the list (closest in traversal order to the leaf geometry) such that the cumulative effect of all the transformation nodes is the same as the single Jot transformation field. In either case, if no NSG node corresponding to the Jot object's field exists, a new NSG node is created and inserted.

Mapping NSG changes into Jot is simpler. When an NSG color or texture node changes, the Jot ML finds all NSG shape nodes that are affected. Then the NSG color is mapped to all the Jot objects that correspond to the affected NSG shapes nodes. A similar technique is used for changes to NSG transformations, but each affected Jot object is given a cumulative transformation determined by all the transformations that affect it. In the case that no Jot object corresponding to the NSG change exists, a new one is created.

Changes to an NSG separator are currently translated to Jot using a naïve technique. All NSG shapes underneath a changed separator are re-mapped in full back to corresponding Jot objects. When separators are changed, new Jot objects need to be displayed or hidden accordingly. This process is performed in a two step brute-force manner. First, the nodes under the changed NSG separator are traversed to find all Jot objects that need to be displayed. Second, we iterate through all Jot objects and hide those that no longer have corresponding NSG shape nodes in the NSG scene graph.

5.4. Networking Implementation

We built two separate network bridges to explore the differences in implementation, run-time efficiency and application artifacts between client-server and peer-to-peer topologies.

The peer-to-peer NB uses a distribution protocol based on the notion that every NSG node has only one SGAB client as its creator. Thus, whenever a change is made to an NSG node by an SGAB client that wasn't the node's creator, the change is distributed only to the node's SGAB client creator. However, when a node is changed by a SGAB client that is its creator, that change is distributed to all other SGAB clients. All transmissions between clients were done using reliable TCP/IP. We found that the peer-to-peer approach was efficient, although somewhat complex, and inconsistencies could occur during contention for objects.

The client-server based NB was designed to provide a simpler, more consistent network implementation at the cost of additional communications overhead to the server. The client-server NB uses both reliable TCP/IP and unreliable multicast. In order to modify an NSG node, SGAB clients first request a lock for that node from the central server. Depending on the status of the contention codes set on the node, the server may grant the lock or may allow simultaneous access by multiple clients. In any case, the server distributes node changes to all other clients and keeps a current copy of the complete NSG in memory.

The client-server based NB uses a hybrid networking protocol to provide efficient distribution of NSG node modifications. Rather than using reliable and heavy-weight messages for each intermediate update (i.e., each update of an object during direct mouse manipulation), only the lock and unlock messages are sent reliably. Each intermediate update is sent instead by multicast, gaining delivery speed and reducing bandwidth consumption at the cost of reliability since a few lost packets during interaction are assumed to be more acceptable than overall slow interaction.

Finally, to improve the interactivity of networked interactions, we also provided mechanisms in both NB implementations of the "ghosting" technique[7] where modifications to unlocked objects are depicted immediately as a semi-transparent "ghost" version of the original object. During the modification process, a lock is requested of the server; if denied, the ghost disappears and if granted, the semi-transparent ghost becomes opaque indicating the granting of the lock. We implemented "ghosting" on the client where the change was attempted, but not on other clients.

6. Examples

We have only begun to to explore the performance and potential uses of SGAB. An interesting example is shown in Figure 4. The Jot system has an algorithm for dynamically wrapping a tight geometric mesh of Skin[11] over objects in

© The Eurographics Association and Blackwell Publishers 2000.

the Jot scene graph. Since Skin operates by reading the geometry of objects in the scene graph, Skin can be wrapped around objects native to a remote Open Inventor SGAB client. In the figure, texture-mapped skin has been wrapped around an airplane model that was loaded into a standard Open Inventor viewer. Although Skin is a complex dynamic geometry that changes every frame, SGAB was able in real-time to map the geometry to NSG, distribute it over a local area network, and unmap it into the standard, unmodified Open Inventor viewer application.

Figure 5 shows the Open Inventor marble game demonstration application as it appears within the Jot system when the two are connected via SGAB. The marble game application is simply a box with a plane inside of it that can be tilted. A ball moves across the plane, is influenced by its slant and bounces off of objects. The Sketch[18] functionality of Jot can be quickly used to create objects constrained to lie on top of the plane. When the plane moves on the Open Inventor side, the scene graph change is reflected on the Jot side where the constraints are enforced and the resulting scene graph modifications are distributed back to the Open Inventor application. In our first attempt we linked, without code modification, Sketch to the marble demo using SGAB. However, since the original marble demo computed collisions for the marble against an internal representation of the walls not stored in the scene graph, the marble would pass through geometry produced by the Sketch application. To address this problem, we modified the marble game to compute collisions directly against scene graph geometry so that the marble would collide with geometry created by Sketch or any other SGAB application. This highlights the approach of using SGAB to prototype joint application behavior without code modification and then to focus subsequent coding effort on eliminating any unacceptable artifacts. In general, we expect that the most useful joint application behaviors will emerge from applications that treat the scene graph as a primary data structure from which internal representations may be computed and cached. Alternatively, applications that treat the scene graph as an output-only side-effect of their primary internal data structures will not benefit as much from SGAB interoperability.

7. Conclusions and Future Work

This paper introduces the Scene-Graph-As-Bus framework, a novel mechanism for networking and combining 3D scene-graph-based applications with little effort. SGAB notices changes to the scene-graph data structure in each client and communicates those changes to other participants. This procedure can be performed with any scene graph that supports immediate reporting of scene graph changes; changes are automatically translated between varying scene graph formats, hardware architecture, operating systems, and programming languages as needed.

SGAB's primary benefits are transparency and generality: data can be shared without modification between fundamentally different graphic clients. However, this technique has a number of limitations which restrict its application. Most important is that not all scene graphs support reporting; for instance, the popular Performer [12] and Java3D libraries do not. There are three ways to handle these "non-compliant" scene graphs: the scene graph implementation can be modified to provide reporting, the application can provide its own reporting mechanism, or a mapping layer can be written that continuously polls the scene graph to determine application changes. We have not tested the latter approach and believe that it would present interesting research problems for efficient performance.

Even when using a compliant scene graph, applications that are shared using SGAB are subject to considerable system overhead both in speed and space. SGAB duplicates the application's scene graph in the NSG layer which may be a problem for large scenes. In addition, the overhead of copying a scene graph node into the NSG, distributing it, and then converting it to a remote application's scene graph may be significant for highly-interactive applications. Our initial results with SGAB have been for relatively small scenes distributed over a LAN. Although we have been satisfied with SGAB performance under these conditions, we expect that future research would have to be directed toward reducing SGAB's inefficiencies in order to handle a broader class of large-scale applications.

In addition to performance, artifacts may arise when applications are shared through SGAB. These artifacts can come from both lossy ML to NSG translations and assumptions that an application makes about the scene graph. For instance, an application that keeps collision detection information outside of the scene graph must update that data in response to scene graph changes made by remote participants. Failure to do so results in artifacts in SGAB's shared scene graph environment. Furthermore, since SGAB condenses all scene graph information to a least common denominator set of primitives, sharing higher-level semantic information between applications is not well supported. For example, points in a cloud representing population data could be manipulated into a nonsensical arrangement by an unknowing remote participant. While contention codes do offer some control over how nodes are manipulated, semantic exchange layered above the SGAB system may be needed for meaningful interoperability.

To further realize the potential of SGAB, more mapping layers need to be created so that a wider variety of applications can be prototyped. We expect that mapping layers can be created for any scene graph structure, with varying difficulty. We feel that an important next step is the inclusion of a scene graph with top-down state inheritance semantics (such as Performer) as opposed to the left-right state inheritance of Open Inventor.

Despite the overhead of maintaining replicated NSG

© The Eurographics Association and Blackwell Publishers 2000.

scene graphs and the potential artifacts of implicit application sharing, we believe that SGAB is an important tool for rapidly experimenting with application hybrids of independently designed components. In addition, SGAB provides a simple mechanism for gaining distributed application behavior from applications designed to run on a single machine. With additional research, sophisticated mapping layers could infer higher-level information from patterns of changes in the scene graph–thus requiring little or no application modification for communication of semantics.

In conclusion, we believe that SGAB makes it easy to demonstrate the value of distributed networked environments such that, if nothing else, developers can effortlessly prototype application behavior with SGAB and then resort to recoding a more robust system for richer collaborative interaction. This leaves two open research problems: what is the next level of interoperabililty? And is there a simple transition to it from SGAB?

8. Acknowledgments

A special thanks to Jaron Lanier, who had the original SGAB idea and supported the funding of its implementation via Advanced Network & Services and the National Tele-Immersion Initiative. Thanks also to Kent Watsen for major contributions to the design of the SGAB system.

References

1. IEEE standard for information technology–protocols for distribute dsimualtion applications: Entity information and interaction. IEEE Standard 1278-1993. New York: IEEE Computer Society, 1993.
2. Hussein Abdel-Wahab and Kevin Jeffay. Issues, problems and solutions in sharing x clients on multiple displays. *Journal of Internetworking Research and Experience*, 5 (1):1–15, March 1994.
3. R. Ben-Natan. *CORBA: A Guide to the Common Object Request Broker Architecture*. McGraw Hill, 1995.
4. Charles Blanchard, S. Burgess, Young Harvill, Jaron Lanier, and A Lasko. Reality built for two: A virtual reality tool. In *Proceedings of the 1990 Symposium on Interactive 3D Graphics*, March 1990.
5. Michael Capps, David Stotts, Jim Duff, and Jim Purtilo. Distributed interoperable virtual environments. In *Proceedings of the Third International Conference on Configurable Distributed Systems*, pages 202–209, Annapolis, MD, May 1996.
6. C. Carlsson and Olaf Hagsand. Dive-a platform for multi-user virtual environments. *Computers and Graphics*, 17(6):663–9, 1993.
7. Brook Conner and Loring Holden. Providing a low-latency user experience in a high-latency application. In *Proceedings of 1997 Symposium on Interactive 3D Graphics*, 1997.
8. Gerd Hesina, Dieter Schmalstieg, Anton Fuhrmann, and Werner Purgathofer. Distributed open inventor: A practical approach to distributed 3d graphics. In *Proceedings of ACM VRST '99*, London, England, December 1999.
9. Frederick Kuhl, Richard Weatherly, and Judith Dahmann. *Creating Computer Simulation Systems*. Prentice Hall, 1999.
10. Michael Macedonia, Donald Brutzman, Michael Zyda, David Pratt, Paul Barham, John Falby, and John Locke. Npsnet: A multi-player 3d virtual environment over the internet. In *Proceedings of the 1995 Symposium on Interactive 3D Graphics*, pages 9–12, Monterey, California, April 1995.
11. Lee Markosian, Jonathan M. Cohen, Thomas Crulli, and John Hughes. Skin: a constructive approach to modeling free-form shapes. In *Proceedings of ACM SIGGRAPH '99*, pages 393–400, 1999.
12. John Rohlf and J Helman. Iris performer: A high performance multi-processing toolkit for real-time 3d graphics. In *Proceedings of ACM SIGGRAPH '94*, pages 381–394, 1994.
13. Doug Shackelford, John Smith, and John Smith. The architecture and implementation of a distributed hypermedia storage system. In *Proceedings of Hypertext '93*, pages 1–13, 1993.
14. Sandeep Singhal and Michael Zyda. *Networked Virtual Environments - Design and Implementation*. ACM PRess Books, SIGGRAPH Series, 1999.
15. Paul Strauss and Rikk Carey. An object-oriented 3d graphics tool-kit. In *Proceedings of ACM SIGGRAPH '92*, pages 341–349, August 1992.
16. Henrik Tramberend. Avocado: A distributed virtual reality framework. In *Proceedings of IEEE Virtual Reality '99*, pages 14–21, Houston, Texas, March 1999.
17. Richard Waters, David Anderson, John Barrus, D. Brogan, M Casey, S McKeown, T Nitta, and William Yerazunis. Diamond park and spline: Social virtual reality with 3d animation, spoken interaction and runtime extendability. *Presence*, 6(4):461–481, 1997.
18. Robert C. Zeleznik, Kenneth P. Herndon, and John F. Hughes. SKETCH: An interface for sketching 3D scenes. In *SIGGRAPH 96 Conference Proceedings*, pages 163–170. ACM SIGGRAPH, August 1996.

© The Eurographics Association and Blackwell Publishers 2000.

Figure 4: *Skin*[11] *created in the Jot system around an airplane model that was loaded into an Open Inventor viewer and seamlessly mapped to Jot using SGAB. (from Zeleznik, et al., Collaboration between Heterogeneous Stand-alone 3-D Graphical Applications)*

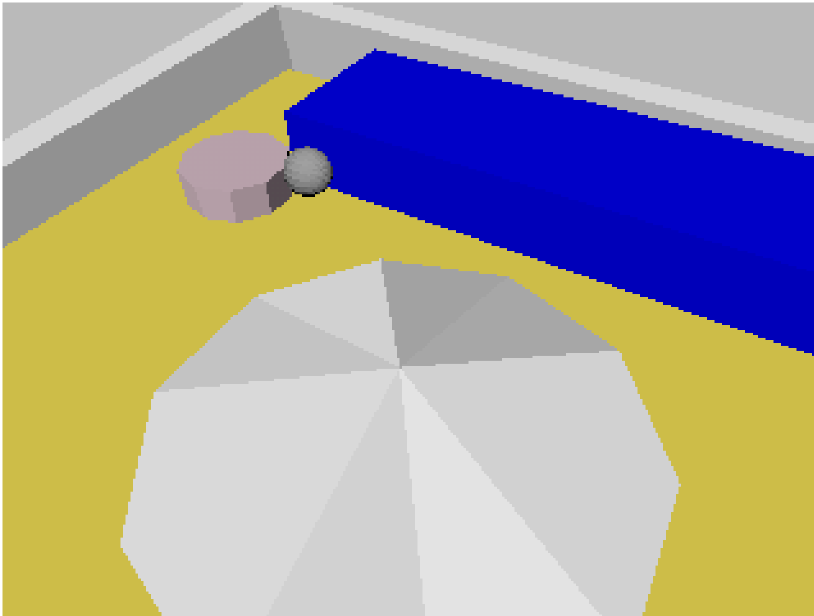

Figure 5: *A ball controlled by an Open Inventor application collides with objects created in a Jot application. (from Zeleznik, et al., Collaboration between Heterogeneous Stand-alone 3-D Graphical Applications)*

© The Eurographics Association and Blackwell Publishers 2000.