Flexible Parametrization of Scene Graphs

Gerhard Reitmayr*and Dieter Schmalstieg†
Vienna University of Technology
Favoritenstrasse 9-11/188/2, 1040 Vienna, Austria

ABSTRACT

Scene graphs have become an established tool for developing interactive 3D applications, but with the focus lying on support for multi-processor and multi-pipeline systems, for distributed applications and for advanced rendering effects. Contrary to these developments, this work focusses on the expressiveness of the scene graph structure as a central tool for developing 3D user interfaces. We present the idea of a context for the traversal of a scene graph which allows to parameterize a scene graph and reuse it for different purposes. Such *context sensitive scene graphs* improve the inherent flexibility of a scene graph acting as a template with parameters bound during traversal. An implementation of this concept using an industry standard scene graph library is described and its use in a set of applications from the area of mobile augmented reality is demonstrated.

CR Categories: I.3.6 [Computer Graphics]: Methodology and Techniques—graphics data structures and data types, languages D.2.11 [Software Engineering]: Software Architectures—domain-specific architectures H.5.1 [Information Interfaces and Presentation]: Multimedia Information Systems—Artificial, augmented, and virtual realities

Keywords: scene graphs, software design, 3D user interfaces

1 INTRODUCTION

Scene graphs have become an established tool for developing interactive 3D applications. They offer an object-oriented and structured approach to describing the application's graphical needs and interactions with the 3D presentation.

Scene graphs have been extended to cope with additional requirements. Shared scene graphs were developed to support distributed applications. Multi-threaded extensions efficiently use SMP hardware and drive multiple rendering sub-systems by parallel render traversals or staged multi-threaded pipelines. Another direction of research investigates how to incorporate new rendering methods such as multi-pass methods within the scene graph. Some results are discussed in the following section on related work.

All these developments address technical issues and usually trade-off structural flexibility for improved performance. However, the original idea of the scene graph is to allow for rapid and flexible implementation of 3D user interfaces. Here the emphasis lies more on the expressiveness of the data structure and the effects that combinations of individual nodes can achieve.

Scene graphs are subject to a number of forces in the software design of an application. Content of an application is stored in a scene graph as a set of graphical models consisting of geometric shapes and transformation hierarchies. Dynamic applications need to change this content database either due to users' actions or algorithmic updates.

*e-mail: reitmayr@ims.tuwien.ac.at
†e-mail:schmalstieg@ims.tuwien.ac.at

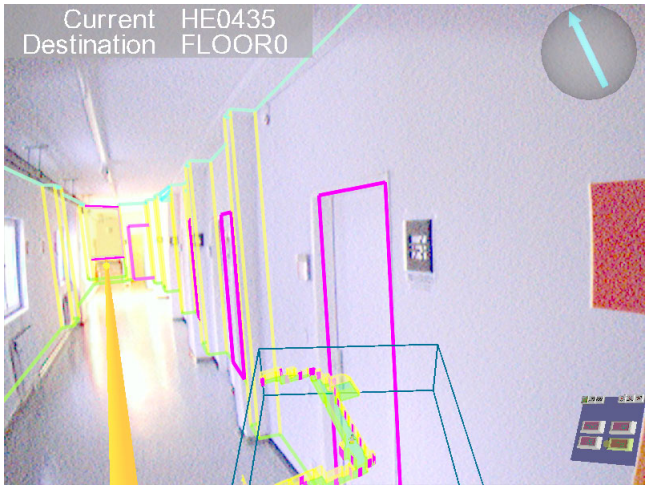

Figure 1: An example of a 3D user interface shows the user's view of an indoor navigation application. The building geometry is augmented, a world-in-miniature display shows her current location in the building and a set of navigation aids points to the next door along the chosen path.

Presentation of the content requires information on materials, render styles, lighting calculation modes, render buffer settings and tests and even render algorithms. Such information is also stored in the scene graph as an additional set of nodes interleaved with the geometry nodes. More complex render algorithms such as hidden line rendering or outline rendering can be composed of several traversals of the same content with different presentation options. An application may require to vary the presentation of all or selected parts of the content geometry. Also, the same content may be rendered with different presentations in one scene, requiring various combinations of presentation and content nodes.

Scene graphs are hierarchical structures, and varying both content and presentation nodes usually requires reassembling new scene graphs from other information or modifying a scene graph by replacing or changing all relevant nodes. The latter either requires a list of all relevant nodes or extensive information on how to find them in the scene graph. Both solutions usually add a substantial layer of complexity to a scene graph based application.

There are two limitations to traditional scene graphs that give raise to the described complexity. (1) The scene graph is a static structure and can only be changed via an API requiring the logic to drive the change within the application. (2) The structure is given by the close coupling of nodes created by the limited communication means between them. Nodes in a scene graph need to communicate state information to achieve the variability of different combinations of such nodes. Traditionally, there are two ways to transfer such information.

First, via traversal state during traversals of the graph. Traversals present the common way to work with scene graphs. They compute and accumulate state and output information while walking the scene graph. A visited node uses information stored in the current state to implement its functionality. For example, a shape node

uses the current drawing style and color to draw itself correctly. A transformation node updates the current transformation by multiplying a local change with the current and storing the result in the state. Such type of the state information is closely coupled to the functionality of the traversal such as a rendering traversal. Also its effect and range is usually dictated by the functionality. The color node will only affect a well defined set of shapes nodes defined by proximity in the scene graph.

Second, nodes can be directly connected via some other mechanism. For example, Open Inventor [18] provides a data flow graph between fields of nodes allowing for arbitrary communication of nodes. This is essentially an implementation of the Observer pattern [5] and requires knowledge of the identity of the sender and receiver to establish the connection. Therefore such a solution requires references of all nodes that need to receive a certain piece of data, which adds considerable complexity to an implementation.

We propose a new way to pass on information through a scene graph that does not depend on local order or direct connections. Furthermore, we achieve this without adding another software construct to the scene graph, but rather rely on a well understood component of it, the traversal state.

The core of the idea is to extend the state construct by exposing a generic state to the application. Such state can be used to store application specific information that adds to evaluating the scene graph through traversals. Nodes can set and evaluate the generic state allow to interact with it without the need to create new node types or traversals. Thus, the overhead of implementing a new state type is eliminated.

A further, second extension builds on the first one. By storing sub-scene-graphs in the state, whole nodes and sub-graphs can be reused at various spots within a traversed scene graph. As the binding to the nodes stored in the traversal state happens only during traversal itself, the nodes can be changed for each traversal and provide a very flexible way of building scene graphs.

2 Related work

The basic components of scene graphs have been described succinctly by Strauss & Carey [18]. These components are the hierarchical structure of nodes aggregating attributes, traversal based on the Visitor pattern [5] to compute results and data flow between attributes to implement animations and interactive behavior.

Distributed systems are required to implement collaborative applications. Using the scene graph as a distributed and structured shared memory provides a transparent way of implementing collaborative applications. Implementations of shared scene graphs synchronize the local scene graph structures of participating application instances. The level of transparency to the programmer varies between different implementations. Avango [20] requires explicit instantiation of distributed data flow between shared instances, Repo-3D [9] distributes full scene graphs but requires the implementation of various callbacks to notify the application programm of changes. Distributed Inventor (DIV) [6] transparently distributes Open Inventor scene graphs and relies on the framework's features to communicate with the application programm. The Scene-Graph-As-Bus work [23] uses an abstract scene graph to translate between different implementations in a distributed system on the fly.

To address the requirements of realistic real-time rendering more constrained and optimized scene graphs were developed. IRIS Performer [15] provides a staged multi-process pipeline to efficiently use multi-processor machines. OpenSceneGraph [1] and OpenSG [12] are new developments that provide thread-safe execution and support for multiple rendering pipelines. The general trend to special purpose scene graph engines is evident in the gaming industry which relies on optimized scene descriptions for maximum performance and visual fidelity. These scene graph implementations abstain from providing a dedicated file format and only provide import filters for geometrical and luminance data.

Some work addresses the fixed traversal structure in hierarchical scene graphs. The Virtual Rendering System [3, 7] extends the general traversal notion with software objects that can perform specialized traversals for multi-pass rendering techniques. However, they only provide support through the programming API and do not have a general high-level language to describe these traversals. Reiners [11] describes a framework that supports different traversal orders by decoupling the visiting of nodes from the selection of the next nodes to visit.

Closer to our view of the scene graph is VRML [2] which abandons the programming API and only provides a declarative language to build scenes and animations. Embedded scripts can provide more complex computations than the predefined nodes but do not have access to the rendering system itself.

Another approach is to wrap the programming API in a high-level language. Obliq-3D [10] and Avango provide bindings for interpreted languages which allow for rapid iterative development. An approach similar to our work is Fran [4] which is explicitly developed as an embedded domain specific language [21] for developing interactive 3D animations. However, it is based on the pure functional host language Haskell which has excellent features for research but probably does not appeal to a wide audience of developers.

The work closest in spirit is described by Schmalstieg and Gervautz [17] who modeled parallel term rewriting systems with cyclic scene graphs. However, it did not provide for a generic language feature reusable for other purposes as well.

To our knowledge, no work up to know has been focusing on extending the expressive power of the scene graph structure itself. We view a scene graph as an embedded domain specific language that raises the level of abstraction for the application programmer and simplifies the task of implementing an application. Our implementation is based on Open Inventor which is a long standing standard for developing interactive 3D applications and relies only on standard features of the library. Therefore it is directly reusable by any application implemented in Open Inventor. The research presented here was developed in the context of a software framework for augmented reality applications called *Studierstube* [16].

3 Concepts

Traversal is the common method of computing results from a scene graph. A software object called action traverses the scene graph by recursively iterating through all nodes and calling a method on each node. The overall result depends on the structure and content of the scene graph as well as on the performed operations during traversal. The structure is usually static and the traversal occurs in a predefined order that may depend on the action and nodes as well.

Actions implement a double-dispatch technique. Each type of action manages a table of functions for each type of node. Upon visiting a node, the action looks up the corresponding function and passes the node and itself as parameters. Thus, it allows to vary the operation executed depending both on the type of action and the type of node. The function table technique also allows to easily extend the framework with new nodes by adding a new entry to the function table of existing actions or with new actions by defining a new function table and appropriate functions.

Nodes use traversal state to communicate with each other during traversal. For example, shape nodes need to know the current color value to draw themselves correctly. The current color is stored in a dedicated state element and is manipulated by material property nodes that set the state during traversal. Typically, a scene graph

library provides a number of dedicated state elements for rendering attributes, transformation matrices and similar information.

3.1 Context state

We propose to add an independent and generic *context* element to any traversal state. Usually, the type and operation of these elements are fixed and tailored to specific nodes and actions. The new context element augments the state with a general purpose element. The new element is tracked for all actions and can be set and used by all nodes. Consequently, a set of additional nodes can influence traversal based on the new element but independent of the type of action.

The additional traversal element is called *context* and a scene graph annotated with context is a *context sensitive scene graph*. The context is a modeled as an associative array mapping index strings to value strings. For each index either the associated value is returned or the empty string if the index is not present in the array. The mapping itself is implemented using the map data structure from the standard template library of C++. The empty string is returned from the state, if the given index is not stored as a key in the map.

The context's value needs to be set and traversals or nodes need to react to the context. These two operations are embedded in special nodes that are part of the scene graph. A dedicated property node allows the modification of the context during traversal and implements the following operations on the context:

ADD inserts a set of (index, value) pairs into the current context. Older entries with the same keys are overwritten by the new values.

SET sets the context mapping to a given set of (index, value) pairs replacing all former entries.

REMOVE deletes a set of entries defined by a set of indices from the map. If an entry for a given index is not present, the index is ignored.

CLEAR resets the context to the default state which contains no entries at all.

Other nodes react on the current context during traversal. It would be possible to directly map different entries to different behavior such as selecting a color from a given list, based on the value of certain index in the context. However, such an approach requires implementing a new node for every aspect that the application would like to influence by setting the context.

In effect the context element exposes the usually hidden state mechanism to the application on the level of the scene graph configuration. Up to now applications had to implement their own state elements and access nodes to have access to the same facilities of the traversal framework. Using the proposed context element an application can directly use the information transport mechanism embodied by action state and build scene graphs whose components react to application state. Nevertheless, there is no need for a direct link between the application state and the acting node because the transport mechanism of action state effectively decouples the the state-changing nodes from the influenced ones.

3.2 Node references in the context

A second extension, building upon the exposed state, stores node references in the context and accesses them during traversal. References to scene graph nodes are stored in the context for later retrieval. Upon retrieval they are traversed to execute their specific functionality at the current traversal position.

Such a mechanism allows to construct a scene graph as a template with named slots that are filled during traversal from the context. The slots can carry different nodes for each reference to the template scene graph or each traversal. For example, different sets of property nodes that influence render style and material colors can be configured to be used in specified places in the scene graph. Thus, the same scene graph can be rendered in various styles. Moreover, the actual location of the slots is encoded in the scene graph itself and decoupled from the application code which only has to take care of setting the node references before traversal.

4 Implementation

The implementation of the context sensitive scene graph is straightforward based on Open Inventor's built in extension mechanisms. Open Inventor stores the state during traversal as a set of individual elements for different parts of the state. For each type of element a dedicated stack tracks the current value of the element. The framework provides for extending the traversal state with new elements independently of any action and to enable the use of these elements during all actions. A more detailed exposition on using this mechanism can be found in [22], chapter 2.

4.1 Basic context management

A new element *SoContextElement* was implemented that stores the current map describing the context. Accessor methods to add, set, remove elements to and from the context and to clear it were implemented together with the necessary interfaces of the framework. Moreover, the use of the new element together with every action was enabled.

```
SoContext {
    SFEnum   mode     ADD
    MFString index    []
    MFString value    []
}
```

Figure 2: Specification of the SoContext node. The first column specifies the type of the field, the second column the name and the third the default value. `index` and `value` define the (index, value) pairs and `mode` the operation to execute.

A dedicated property node *SoContext* was created that modifies the context during traversal. The node uses the element's accessor methods to update it according to its own parameters which are set using a number of the node's fields. The exact specification of the node is given in Figure 2. The enumeration field `mode` can take the values `ADD`, `SET`, `CLEAR` and `CLEAR_ALL` which correspond to the four operations on the context. The multiple value fields `index` and `value` denote (index, value) pairs for the `ADD` and `SET` operations while the `CLEAR` operation only uses the `index` field.

Two nodes react on the current context during traversal. An *SoContextReport* node simply reports subsets of the context. It mirrors the fields `index` and `value` of the SoContext node and sets the field `value` to the values associated with the indices specified in the field `index` within the context in the current traversal (see Figure 3).

The SoContextReport node is a generic interface to the context during traversal. By connecting its `value` field to the field of an-

```
SoContextReport {
    MFString index    []
    MFString value    []
}
```

Figure 3: Specification of the SoContextReport node. The field `index` specifies the values to read from the current context and the field `value` outputs the values for further use.

other node following in the traversal order, any context value can be used to set arbitrary parameters of nodes. This functionality is enabled by Open Inventor providing implicit conversion between field data types where applicable.

Another node provides a short cut to directly change the traversal depending on the context. The *SoContextSwitch* node uses the entries in the context map to compute which children to traverse (see Figure 4 for the full specification). The field `index` sets the index to use in looking up a value in the context. The returned value is then interpreted as an integer specifying the index of the child to traverse. The field `defaultChild` sets the index of the child to traverse, if the index is not present in the context and is therefore mapped to the empty string.

```
SoContextSwitch {
    SFString index          INT32_MIN
    SFInt32  defaultChild   -1
}
```

Figure 4: Specification of the SoContextSwitch node. `index` specifies the entry in the context to read out and `defaultChild` the behavior if the index is not present.

4.2 Nodes as context values

A more complex extension of the above implementation allows to store and retrieve references to nodes in the context element. Again, a pair of dedicated nodes provide the access to the state. The node references are simply encoded in the value strings and an additional list in the context implementation stores the indices that point to nodes rather than strings.

```
SoNodeContext {
    SFEnum   mode     ADD
    MFString index    []
    MFNode   value    []
}
```

Figure 5: Specification of the SoNodeContext node. Again `mode` specifies the operation. Values are now references to scene graph nodes and are stored as such in the context element.

A new property node *SoNodeContext* now stores references to the nodes configured in its `value` field in the context. The fields `mode` and `index` work as in the original SoContext node.

```
SoContextReport {
    MFString index    []
    MFField  value    []
    SFBool   traverse TRUE
    SFBool   report   TRUE
}
```

Figure 6: Specification of the SoNodeContextReport node. The field `index` specifies the values to read from the current context and the field `value` outputs the values for further use. The boolean variables `traverse` and `report` control the node's behavior.

A *SoNodeContextReport* node simply reports node references stored in the context. It mirrors the fields `index` and `value` of the SoNodeContext node and sets the field `value` to the node references associated with the indices specified in the field `index` within the context in the current traversal (see Figure 6). If an index does not reference an node, the respective entry is set to NULL. Furthermore the SoNodeContextReport can also traverse the nodes retrieved from the context allowing to add nodes to the scene-graph for the purpose of traversal only. Two boolean fields `traverse` and `report` control the actual traversal or reporting of nodes in the `value` field.

5 Using context in scene graphs

The nodes described above allow the construction of scene graphs that can be controlled without detailed knowledge of their structure. Such scene graphs act like function objects that are reusable and change their behavior based on arguments set in the context. Some examples of using the context mechanism are described in the following.

5.1 Decoupling of model and control

A simple setup embeds SoContextSwitch nodes throughout the scene graph to only traverse partial subgraphs. For example, a scene graph might store different representations of objects locally for each object separately but as children of SoContextSwitch nodes. The nodes are configured to use the same index and the order of the children corresponding to different presentations is the same for each object. Then an application can control the presentation by manipulating a single SoContext node above the scene graph to set the common index to the desired value. Figure 7 shows the schematic structure of such a scene graph.

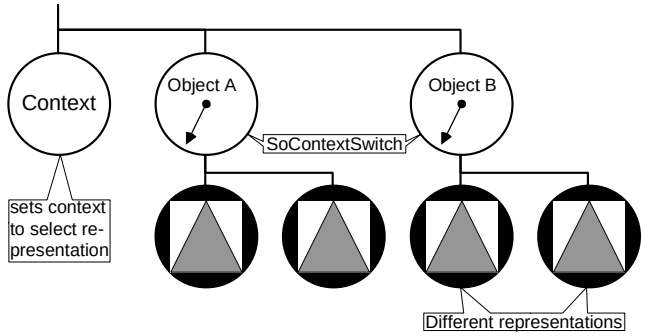

Figure 7: A context sensitive scene graph to select between differen representations of a set of objects.

Several of such indices can be overlaid to produce a matrix like structure of options. Typically combinations can be arranged by simply serializing SoContextSwitch nodes configured with different indices for different aspects. For example, one index could select the color of an object and a second one the render style such as lines, flat shaded polygons or Phong-shaded polygons. Because they can be set independently, the switches can be arranged in any order and do not depend on each other. Figure 8 details the structure of a scene graph allowing to switch between two independent options.

More complex computations can be implemented by alternating switches and SoContext nodes in the scene graph. Then, the choice of one or more indices can influence the setting of another set of indices, implementing a general mapping from a tuple of indices to another tuple. Recursively building such mappings can lead to powerful computations by simply arranging scene graphs in the appropriate way.

5.2 Direct use of context values

The SoContextReport node allows to directly use context values to set the parameters of scene graph nodes. Figure 9 shows how to read out a color value from the context and apply it to a subsequent geometry.

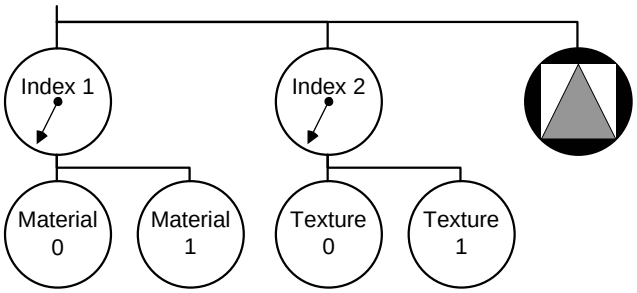

Figure 8: A context sensitive scene graph to select between multiple options at the same time to allow different combinations.

The SoContextReport node named `Color` reads out the value of the context index `"myColor"` and the following material node sets its diffuse color to the value provided by the SoContextReport node. The following sphere geometry will then be rendered in the specified color. Reusing this scene graph fragment throughout a larger scene graph allows to set the rendering color by specifying an SoContext node with the parameters index set to `"myColor"` and value to the desired color.

```
Group {
  DEF Color SoContextReport {
    index "myColor"
  }
  Material {
    diffuseColor = USE Color.value
  }
  Sphere {}
}
```

Figure 9: Using an SoContextReport to set arbitrary field values in the scene graph.

5.3 Scene graph templates

The mechanism of storing nodes in the scene graph provides the greatest flexibility. A scene graph can be fitted with SoNodeContextReport nodes that act as placeholders for the nodes that are to be traversed during a later traversal of the overall scene graph. Therefore, various details and configuration options can be left during construction and are only added at traversal time enabling a kind of late-binding of parts of the scene graph. Figure 10 shows a simple example of the principle.

Furthermore, the application does not need to keep track of the location within the scene graph of the placeholders or manage references to them. Only during creation or change of the scene graph the correct structure needs to be created.

Late-binding and referential transparency simplify the creation of applications that require dynamic scene graphs. For example, a mobile AR application in a wide area environment will only load parts of the required world model for display purposes. The subset of the world model will change while the user moves about the environment and the scene graph within the application will be updated by a central service to reflect the relevant subset. If the central service creates the required slots in the provided scene graph, the application does not have to deal with the structure of the graph to set the required rendering attributes, but rather simply specifies the nodes to fill the template slots. On the other hand, the central service does not need to know about the desired rendering attributes. Therefore, the template mechanism effectively decouples the content from the presentation.

```
DEF SG Separator {
  SoNodeContextReport {
      index "RenderStyle"
      traverse TRUE
  }
  Sphere {}
}
...
SoContextReport {
    index "RenderStyle"
    mode ADD
    value Material { diffuseColor 1 0 0 } # red color
}
USE SG # draw the Sphere in red color
SoContextReport {
    index "RenderStyle"
    mode ADD
    value Group {
       Material { diffuseColor 0 0 1 } # blue color
       DrawStyle { style LINES }       # and wireframe
}
USE SG # draw the Sphere in blue wireframe
```

Figure 10: A template scene graph traversed with two different values set for its parameter.

5.4 Meta-programming of scene graphs

The context sensitive scene graph mechanism shifts the complexity of managing multiple presentations from the application code to the scene graph data structure. However, it also enforces a unified approach to dealing with multiple representations and dynamic switches between them. That is, if several objects require to provide the same parametrization, the representing scene graphs will follow a common pattern and vary in the specific geometry and other details of the objects. To enforce the common pattern on all these objects some forms of meta-programming of the scene graph are required. Such meta-programming can be applied on three levels: scene graph implementation level, scene graph language level and scene graph meta-language level.

1. Open Inventor applications can codify common scene graph patterns by implementing so-called *nodekits*. Nodekits form a special class of scene graph nodes that encapsulate a fixed sub-scene-graph and allow to enforce additional constraints such as fixed field values and connections between fields. They also hide the internal scene graph and only provide access to specified sub-nodes and fields. Nodekits are C++ classes and thus are nodes with a fixed behaviour configurable only as far as the developer intended. They also become new elements of the scene graph language itself and can be used in subsequent creation of scene graphs.

2. The particular implementation of Open Inventor this work is based on [19] also supports the VRML97 syntax and allows to mix and match node types from both language sets because the VRML97 nodes are implemented as Open Inventor node types. VRML97 features a concept for reusable components similar to the nodekits described before. A *PROTO* is a node that is described by a prototype scene graph which is instantiated wherever a node of the defined type is encountered. The definition and reuse of a PROTO is part of the file format and therefore happens at the language level itself. Because the context enabling nodes can be used in the definition of a new PROTO, it becomes possible to define common scene graph patterns in the file format itself. We also developed a Python binding for Open Inventor and implemented a Script node supporting scripts written in Python. A flexible Script node can provide ”glue code” which is often required to calculate complex functions of context state for further use.

Index name	Description
User	user id associated with the render area
Application	id of the application the current node belongs to.
Window	id of the 3D window containing the current node.
Eye	the stereo-buffer of the current render traversal.
DivMode	the distribution mode for the current application

Table 1: Context information provided by the Studierstube framework.

3. Finally, automatic methods of generating a scene graph file can be applied to cope with the added complexity in the scene graph. We have investigated this approach in a set of location based mobile augmented reality applications [14] that operate within the city of Vienna. A generic XML database stores building models, street networks and information on interesting locations in an application independent format. Then dedicated transformations generate scene graph files for specific applications. The resulting files already contain the necessary sub-structures using context information to switch between different representations.

6 Results and applications

The general mechanism provided by context sensitive scene graphs is used in the *Studierstube* framework to implement functions ranging from providing system level information to supporting the needs of individual applications. Some demonstrated uses are described in the following sections. An extended version describing more results including an implementation of information filtering as described by Julier et. al [8] is available online as a technical report[1].

6.1 System management in Studierstube

Applications within *Studierstube* need different system level information to implement their behavior. *Studierstube* supports multiple users and allows to drive multiple displays simultaneously to provide user-specific views. Displayed information can depend on the user, whether the image is rendered for the left or right eye, the id of the application a node is part of, the collaborative session used to distribute the information and many more parameters. Such information can be transported in a traversal independent manner using the context of a traversal. We cannot employ global variables to transport such information because some items are only applicable to certain subgraphs or may change across different subgraphs. For example, in the overall scene graph structure used by a *Studierstube* process, there are several application identifiers for the different applications running in parallel.

Studierstube defines a set of well-known context indices as parameters (see Table 1). The information about the user id and the selected buffer is only available during a render traversal, because it is not applicable for other actions.

Several nodes in the *Studierstube* framework use the system specific context information to implement adaptive presentations. The *SoWindowKit* node draws window borders in different colors depending on each user's window focus. The focus color can be configured for each user individually. The SoWindowKit node uses the user id information during a render traversal to index into an array of color values storing these user dependent colors and sets the window borders' color to the user's focus color. The context distributes the information on which user is rendered for from all the instances of windows that might be present in a scene graph.

Another optimization concerns the use of the application id. Widgets need to implement special behavior in a distributed setting and need to change field values without the propagation of these changes to other replicas. Fields are selectively put into and removed from filters in the replication mechanism to disable or enable propagating changes. Therefore, a widget needs to know the instance of the session object responsible for its part of the scene graph. The instance can be retrieved with a simple look-up based on the application id provided via the context mechanism. This approach is much more efficient than the alternative, namely to search the scene graph from a common root node for the widget node and then walk the computed path to look for any session objects that are ancestors of the widget node.

6.2 Signpost - attributing of a general model tree

The Signpost application described in [13] is an indoor navigation application based on a wide area tracking solution. It provides navigational hints to a user roaming a building. The user interface provides a number of graphical representations of the building geometry (see Figure 1).

World-in-Miniature A miniature world model with the current location of the user floats in a head stabilized position in front of the user. Its orientation is either fixed to north-up or changed to align with the user's viewing direction.

Augmentation of room geometry The room geometry of the building can be augmented by a wire frame representation. Different rendering modes such as hidden-line removal or full X-ray vision are possible.

Navigation aides The navigation system computes the shortest path fro the current room to the destination room selected by the user. Doors along the way are highlighted to aid the user. A world stabilized arrow points to the next door to pass and a compass-like widget shown in a heads-up display gives overall orientation information.

A core requirement of the software architecture was to build a modular, extensible system that allows to easily add application features. To address this issue we separated the building model from the components that are responsible for different presentations and interactions. A dedicated *model server* component holds the scene graph of the building model and presents an interface for client components to reuse it in their own scene graph for rendering. The model scene graph is annotated with SoContextReport nodes to set various aspects of the presentation. Client components reuse the model scene graph within their own graph and prepend a set of SoContext nodes to provide their specialized rendering parameters.

The model server's scene graph relies on two dedicated nodes, *SoBAUBuiling* to model a whole building and *SoBAURoom* to model a single room contained within a building. The scene graph encapsulated by an SoBAUBuilding node contains a switch node holding all the rooms of the building. The switch node allows to traverse only a subset of all rooms to limit the rendering to a certain selection. The SoBAURoom node contains a scene graph describing the geometry of a room separated into different sets of polygons for wall, ceiling, floor and other surfaces. Also portal geometry representing doors and connections to other rooms are modelled separately. Every set of polygons is rendering individually with its own render parameters. These parameters are set by dedicated SoContextReport nodes that are controlled by individual indices (see Figure 11). A global naming scheme allows to set attributes such as color, lighting model, drawing style and more for each set of polygons individually.

A client of the model server obtains a reference to the top level SoBAUBuilding node. Then it configures a set of SoContext nodes with the well-known names for different attributes of the model. Moreover, it can also select to only traverse a subset of the rooms. The client can repeat this construction for each presentation style and subset of rooms it requires to display.

The different client components implementing the WIM, augmentation or navigational hints all use one or more SoBAUClient

[1] http://www.ims.tuwien.ac.at/publication_detail.php?ims_id=TR-188-2-2004-03

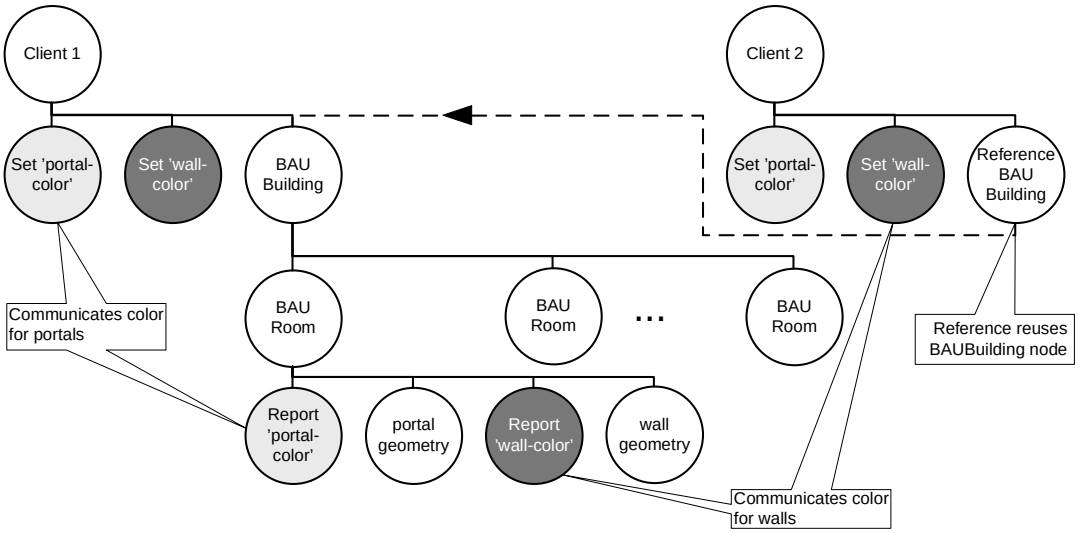

Figure 11: Structure of the BAURoom scene graph and its use by two clients. The BAURoom node annotates the different geometries with report nodes that read a set of parameters from the context such as rendering color or drawing style. Each client holds a reference to the overall building scene graph and prepends it with context nodes to set the value of the individual parameters.

nodes in their scene graph. They all set different presentations styles using the context while being independent of the actual building scene graph. For example, the augmentation component uses an SoBAUClient node with styles to render the polygons in wire frame mode with different colors for walls, ceiling, floor and portals. A second SoBAUClient node can be switched into the scene graph to render filled polygons into the Z-buffer only before rendering the wire frame to achieve a hidden line effect (see Figure 12).

The client applications are separated from the overall building model and do not have any detailed information about its structure. Any changes to the model are transparent and instantaneously available to the applications. Also adding a new application requiring different presentation styles does not affect the existing ones. The modularity also furthers reuse of the individual components in new interfaces.

7 Discussion

The proposed mechanism provides a flexible way to communicate information between nodes in a scene graph and furthermore compose scene graphs on the fly. Different presentations are simply different instances of a template scene graph combined with different presentation parameters. Combinations of content and presentation are created during traversal only which is the actual moment in time they are required.

Global information delivery could be addressed by other means as well. For example a publish-subscribe mechanism or an event bus architecture could provide for the simple means of decoupling the source of information and the scene graph nodes using it.

Finally, changes to the scene graph structure are transparent to the transport mechanism. Therefore, it is more flexible than direct connections such as the observer pattern which require references to the end-points to be set up or destroyed.

The context sensitive scene graph traversal introduces aspects of declarative programming using the structure of the scene graph. While the original scene graph approach already represents a powerful application of the Composite pattern [5], the declarative programming possible by constructing a scene graph was previously limited to a fixed visual appearance dictated by the structure.

A scene can now be made a reusable and parameterizable object using the SoContextReport and SoNodeContextReport nodes. Therefore, it can become a function like object that produces different rendering results based on the parameters set with the SoContext and SoNodeContext nodes. Because Open Inventor provides a human-readable file format, such programming can be done in a declarative style within the file format.

Similar effects can be achieved by using a dynamic language that also provides a scene graph API. For example, the Scheme binding used in Avango [20] allows to directly declare a scene graph in a source file. Language features of Scheme would allow to encapsulate the creation of the scene graph in a function and pass parameters to it that are used in the created instance. By binding such parameters later to different values, a global communication mechanism is established again. However, modifying the scene graph would require the new nodes to be passed the same parameter, again creating some complexity in the application. The binding provided by the context sensitive scene graph is even later than variable binding, it only happens during a specific traversal and can vary between individual traversals.

7.1 Application to top-down scene graphs

Top down scene graphs group transport state only along traversal paths to allow reordering the graph to apply optimizations. Some state is set by intermediary nodes, e.g. model matrix state is set

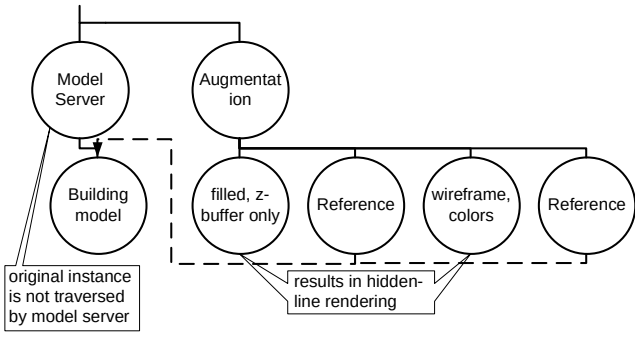

Figure 12: Using a BAUBuilding for multi-pass rendering. The model scene graph is reused twice, once to render the filled polygons into the Z-Buffer only and a second time to render the colored wire frame model resulting in a hidden-line drawing.

by transformations along the path and projection matrix state by a camera somewhere else in the traversal. Similarly, context state can be set along the path as well, for example by implementing dedicated group nodes that set the context for their children.

Accessing context is harder, because most top-down scene graph libraries will not support the field connection concept of Open Inventor. Therefore, a set of nodes is required which take parameter values from the context rather than fixed attributes. For example, a dedicated context aware material node would read out color values from a set of context entries which are configured in the node's attributes. Such a scheme would still permit reordering the scene graph, because the final value will only depend on the traversed path and not on any nodes to the left of it. However, the implementation would generate a larger set of new nodes.

7.2 Performance

As the context is implemented in terms of the standard scene graph state mechanism, it directly supports the caching mechanisms that build on it. The caching mechanisms determine based on state values whether a certain subgraph needs to be re-rendered or a stored display-list can be used instead. While a pure global data structure could achieve a similar effect to the context, such a solution would not be transparent to the existing scene graph implementation and therefore would require changes to the caching mechanisms.

The changes to the SoContextReport node's fields during traversal would signal the library to destroy any caches created. To avoid the resulting performance loss, we created another container node that stops any such signals created during traversal.

8 Conclusions and future work

The interpretation of the scene graph structure as a declarative language provides the developer with a new tool set. Programming the scene graph structure directly raises the level of abstraction and frees the developer from much bookkeeping and overhead work in creating the scene graph. The interpreted nature of describing the used scene graph structure allows for quick turn around times during iterative development and is thus well suited for developing user interfaces. It also lends itself naturally to data-driven application designs.

The structural complexity of a context sensitive scene graph can increase dramatically with the number of objects and number of features that are varied. This development can be countered relying on meta-programming of the scene graph with automated methods. A data-driven program architecture will allow to generate the required scene graph from given data and a template pattern to apply to the set of objects passed in. The programmer then only needs to create the template instead of manually applying the same structure to all objects. We have started to investigate an architecture [14] that realizes such a method for large-scale mobile AR applications.

In future work we will continue to apply the context sensitive scene graphs to user interfaces for mobile augmented reality. The implementation itself is complete and available as part of the *Studierstube* augmented reality framework at `http://www.studierstube.org/`

Acknowledgments

The authors would like to thank Anton Fuhrmann for his early input to the concept of context sensitive scene graphs. The presented work was sponsored by the Austrian Science Foundation *FWF* under contracts no. P14470 and Y193, and Vienna University of Technology by Forschungsinfrastrukturvorhaben TUWP16/2002.

References

[1] Don Burns and Robert Osfield OpenSceneGraph http://openscenegraph.sourceforge.net/index.html, April 6th, 2004

[2] Rikk Carey and Gavin Bell. *The Annotated VRML 2.0 Reference Manual*. Addison-Wesley, 1997.

[3] Jürgen Döllner and Klaus Hinrichs. A generalized scene graph. In *Proc. VMV 2000*, pages 247–254, Saarbrücken, Germany, 2000. Akademische Verlagsgesellschaft.

[4] Conal Elliott. Modeling interactive 3d and multimedia animation with an embedded language. In *Proc. of the first conference on Domain-Specific Languages*, Santa Barbara, CA, USA, October 15–17 1997. USENIX, The USENIX Association.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, USA, 1995.

[6] Gerd Hesina, Dieter Schmalstieg, and Werner Purgathofer. Distributed open inventor : A practical approach to distributed 3D graphics. In *Proc. ACM VRST'99*, pages 74–81, London, UK, December 1999.

[7] Jürgen Döllner and Klaus Hinrichs. A generic rendering system. *IEEE Transactions on Visualization and Computer Graphics*, 8(2):99–118, April–June 2002.

[8] Simon Julier, Marco Lanzagorta, Yohan Baillot, Lawrence Rosenblum, Steven Feiner, and Tobias Höllerer. Information filtering for mobile augmented reality. In *Proc. ISAR 2000*, pages 3–11, Munich, Germany, October 5–6 2000. IEEE and ACM.

[9] Blair MacIntyre and Steven Feiner. A distributed 3D graphics library. In *Proc. ACM SIGGRAPH '98*, pages 361–370, Orlando, Florida, USA, July 19–24 1998.

[10] Marc A. Najork and Marc H. Brown. Obliq-3D: a high-level, fast-turnaround 3D animation system. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):175–193, June 1995.

[11] Dirk Reiners. A flexible and extensible traversal framework for scenegraph systems. In *Proc. 1st OpenSG Symposium*, 2002.

[12] Dirk Reiners, Gerrit Vo, and Johannes Behr. OpenSG: Basic concepts. In *Proc. 1st OpenSG Symposium*, 2002.

[13] Gerhard Reitmayr and Dieter Schmalstieg. Location based applications for mobile augmented reality. In Robert Biddle and Bruce Thomas, editors, *Proc. AUIC 2003*, volume 25 (3) of *Australian Computer Science Communications*, pages 65 – 73, Adelaide, Australia, February 4 – 7 2003. ACS.

[14] Gerhard Reitmayr and Dieter Schmalstieg. Collaborative augmented reality for outdoor navigation and information browsing. In *Proc. Symposium Location Based Services and TeleCartography - Geowissenschaftliche Mitteilungen*, volume 66, pages 53–62, Vienna, Austria, January 28–29 2004. Wiley.

[15] John Rohlf and Jim Helman. IRIS performer: A high performance multiprocessing toolkit for real-time 3D graphics. In *Computer Graphics (SIGGRAPH'94 Proc.)*, pages 381–394. ACM, July 1994.

[16] Dieter Schmalstieg, Anton Fuhrmann, Gerd Hesina, Zsolt Szalavari, L. Miguel Encarnao, Michael Gervautz, and Werner Purgathofer. The Studierstube augmented reality project. *PRESENCE - Teleoperators and Virtual Environments*, 11(1), 2002.

[17] Dieter Schmalstieg and Michael Gervautz. Modeling and rendering of outdoor scenes for distributed virtual environments. In *Proc. VRST'97*, pages 209–216, Lausanne, Switzerland, Sep. 15–17 1997. ACM.

[18] P. Strauss and R. Carey. An object oriented 3D graphics toolkit. In *Proc, ACM SIGGRAPH'92*. ACM, 1992.

[19] Systems in Motion. Coin 3d library. http://www.coin3d.org/, April 5th 2004.

[20] H. Tramberend. Avocado: A distributed virtual reality framework. In *Proc. IEEE Virtual Reality'99*. IEEE, IEEE Press, 1999.

[21] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, June 2000.

[22] Josie Wernecke. *The Inventor Toolmaker: Extending Open Inventor*. Addison-Wesley, 2nd edition, April 1994.

[23] Bob Zeleznik, Loring Holden, Michael Capps, Howard Abrams, and Tim Miller. Scene-graph-as-bus: Collaboration between heterogeneous stand-alone 3-D graphical applications. In *Proc. EUROGRAPHICS 2000*, volume 19(3), 2000.