PREPRINT — the final version can be found in The Visual Computer.

Separating Semantics from Rendering: A Scene Graph based Architecture for Graphics Applications

Robert F. Tobler

Abstract A large number of rendering and graphics applications developed in research and industry are based on scene graphs. Traditionally scene graphs encapsulate the hierarchical structure of a complete 3D scene, and combine both semantic and rendering aspects. In this paper we propose a clean separation of the semantic and rendering parts of the scene graph. This leads to a generally applicable architecture for graphics applications that is loosely based on the well known Model-View-Controller (MVC) design pattern for separating the user interface and computation parts of an application. We explore the benefits of this new design for various rendering and modeling tasks, such as rendering dynamic scenes, out-of-core rendering of large scenes, generation of geometry for trees and vegetation, and multi-view rendering. Finally we show some of the implementation details that have been solved in the process of using this software architecture in a large framework for rapid development of visualization and rendering applications.

Keywords scene graph · systems architecture · semantics · rendering · design pattern

1 Introduction

Implementing complex applications using scene graphs—the standard approach for representing 3D scenes—requires a structured approach to the software architecture in order to deal with user input and simulation results that need to be rendered. Although traditional scene graph designs are well suited for abstracting away the details of the rendering backend, they only offer limited support for complex application logic: there is no established design pattern for dealing with scene graphs that constantly change. Thus, a number of applications that require dynamically changing scene graphs, such as out-of-core rendering, where scene-graph parts need to be loaded and discarded from memory due to the large size of the complete scene, procedural rendering, where scene graphs are generated on the fly, or semantically enhanced scene graphs, where the semantics of nodes are used to interactively change rendering styles, are difficult to realize with commonly used APIs. In this paper we will show how the separation of semantics from the rendering scene graph naturally leads to an architecture that is well suited not only for these tasks, but for most other applications of scene graphs as well.

Robert F. Tobler
VRVis Research Center, Vienna, Austria
E-mail: rft@vrvis.at

2 State of the art

With the advent of hardware accelerated graphics output a number of rendering APIs have been developed in order to provide a simple interface and hide the hardware specific details of rendering. Many of these APIs are based on scene graphs as a natural hierarchical representation of the structure of 3D scenes. In the following, we will highlight some of the more widely used scene graph systems, how they deal with changing state due to user input or simulation results, and dynamic scene graphs.

One of the first scene graph designs that saw widespread use was *Inventor* [SC92] and its successor *Open Inventor* [Wer93]. In addition to a wide variety of scene graph nodes for standard rendering purposes, Inventor also provides some support for handling state and user input: namely engines and event nodes. Engines encapsulate the concept of dynamic expressions that depend on various input fields, and can thus be used to

perform arbitrary calculations within the scene graph. Event nodes provide the mechanism to react to user input via call back functions. In order to realize dynamic scene graphs in Open Inventor, both new node types that represent changing scene graph parts, and custom actions, that actually expand the new types into scene graphs, need to be implemented. Reitmayr and Schmalstieg have presented such a dynamic extension of Open Inventor in order to create a context aware dynamical scene graph [RS05]. This is achieved by creating a new traversal which contains a context object or a context state. In addition, the scene graph can be annotated with context nodes. During the traversal, the context state of the traversal is combined with the annotated content in the graph. This combined data is then used to make decisions for the rendering path in the scene graph, or to modify properties of already existing nodes.

OpenSG is a scene graph system with a main focus on performance. OpenSG was developed by Reiners [VBRR02] as part of OpenSG Plus that is now continued as an open source project. The core mechanism for modifying the scene graph in OpenSG is the possibility to define and use different traversals of the scene graph that only visit the nodes needed for a certain action. Although this is a powerful and performant way to partially traverse the scene graph, the actual state and triggers for these traversals need to be maintained by the application. Actually creating additional parts of the scene graph during traversal is not directly supported in the current version of OpenSG. Implementing dynamic scene graphs requires special nodes and traversals, just as in Open Inventor.

Open Scene Graph [BO04] was inspired by IRIS Performer [RH94] with the emphasis on high rendering performance for 3D applications. It provides better encapsulation of the OpenGL state than OpenInventor and thus a better rendering performance. There is no comparable implementation to the graph operators from OpenSG. Dynamic content is again realized with callbacks, these are however limited to functions for update, cull and draw. In Open Scene Graph, changes to the structure of the graph can only be done during the update traversal. There is no concept for dynamic generation of graph structure, like in the two other scene graph systems, and thus the tools for dynamic scene graphs are somewhat restricted.

The *SceniX* scene graph system created by the graphics company nVidia is a multi-purpose scene graph which runs on different platforms and has been developed for fast rendering with high image quality [KM09]. The central design concept of the scene graph is to separate the data stored in the scene graph from the operations which are performed on them. These operations are implemented with so called *traverser* objects that can be programmed to perform specific actions on each node of the scene-graph. SceniX includes the basic nodes for dynamic data in the graph, such as animated transformations, level of detail, switch and billboard nodes. It has, however, no ready-made construct for creating scene graph data on demand or during runtime. The concept of the separation of data stored in the nodes, and the actions applied to the graph, as it is implemented in SceniX, provides a good extension point for dynamic generation, facilitating the creation of new scene graph data and hierarchies on demand in a simpler fashion than the previously introduced frameworks.

Although all of these scene graph systems (and many others that are based on similar principles, e.g. [Sea05], [Bos09]) provide some support for reacting to user input and the possibility to implement dynamic scene graphs, there is no well-defined design pattern for the actual implementation. In large and complex applications this can lead to multiple solutions for similar problems, and thereby significantly increase the maintenance overhead.

3 Separating semantic and rendering scene graph

In any non-trivial rendering application, it is necessary to modify the graphical output either according to user input, according to changes in some simulation state, according to a changed style based on the semantics, or any combination of these changes. If the graphical output of the application is represented by a scene graph, it is therefore necessary to change the scene graph in some fashion in order to reflect the changes. Additionally the state of the input or simulation needs to be stored, so that changes can be detected and applied to the scene graph. With standard scene graph designs, there are two possibilities to store this application state:

Application state outside the scene graph: In this approach, all state is stored in application specific data structures outside the scene graph, and the scene graph is only used as an internal representation of the rendered output. In this case, the application will maintain references to parts of the scene graph in order to modify these parts whenever changes in the input or state occur (see figure 1).

Application state inside the scene graph: Of course it is also possible to implement specialized nodes in the scene graph that store input and simulation state. In this case, the state in these specialized nodes is updated during traversal of the scene graph (see figure 2).

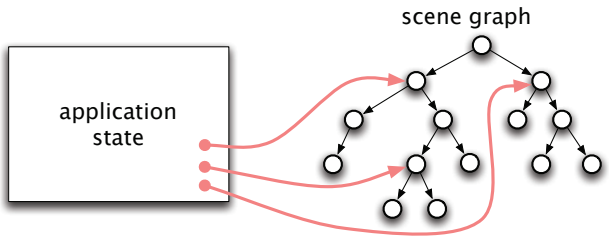

Fig. 1 In traditional designs, the application can maintain references to parts of the scene graph, in order to dynamically modify the scene graph.

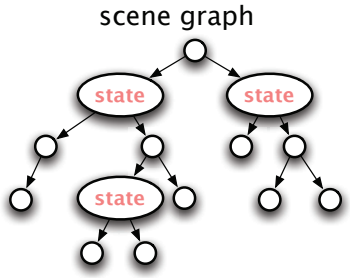

Fig. 2 In traditional designs, the state can also be directly stored in the scene graph, complicating traversal.

Both of these designs lead to draw-backs in large or complex applications. In the first case, where the state is separated from the scene graph, the hierarchical structure of the scene graph is not necessarily reflected in the way the state is stored in the application. In order to overcome this problem, the structure of the rendering scene graph can be partially implemented in parallel structures in the application, leading to a duplication of effort, or the differently structured state can be hard to maintain. Storing state in the scene graph, as in the second case, may lead to a significantly increased complexity if there are dependencies between different parts of the scene graph.

As an example, if complex updates that can be triggered multiple times during a traversal are implemented in a lazy fashion, oftentimes a "needs update" flag is introduced to various nodes in the hierarchy, that is used to fold multiple updates into one evaluation. If the updates of different aspects of the same node are handled with multiple such flags, the resulting application logic can become very cumbersome to implement.

The root of the complexity is the handling of both semantics and rendering specific operations in a combined fashion within the scene graph. A first step in the direction of separating the semantics from the rendering parts of the scene graph has been introduced by Mendez et al. [MSH*08] by using semantic tags as attributes in the scene graph. This leads to clear benefits if the rendering styles of nodes with the same semantic tag needs to be changed interactively.

We pursued a more complete solution for solving these problems by completely separating semantics from the rendering scene graph, and introducing a split scene graph architecture:

The semantic scene graph: This is a scene graph of semantic nodes, that embodies the scene as the user modeled it. In a pure rendering application this graph is never modified after its initial creation.

The rendering scene graph: This is the scene graph in the traditional sense, that generates the sequence of rendering operations that is necessary to display the scene. Its structure is influenced by the rendering backend that is used.

As an example consider a small table modeled by a user. The semantic nodes are the *legs* of the table, the *table top* each with an associated material as a parameter, and a *table* node that groups the other five nodes, each one a leg with a transformation. The rendering scene graph for such an object has a number of transformations and shader nodes that are specific to the rendering backend.

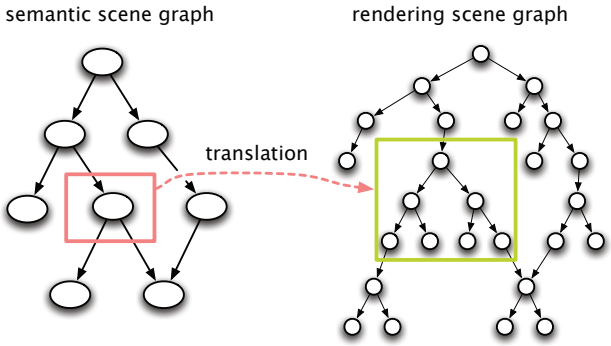

Fig. 3 A typical graphics application builds a rendering scene graph by translating the nodes of a real or implied semantic scene graph.

A typical graphics application acts like a compiler, that takes a real or implied semantic scene graph as an input and generates the rendering scene graph for 3D output. During this translation operation, a single node of the semantic scene graph is often translated into multiple connected nodes of the rendering scene graph (see figure 3).

This works well for applications with static output. However, if dynamic output is necessary the previously mentioned techniques for modifying the rendering scene graph are normally employed. In order to exploit the separation between semantic and rendering scene

graph, we can look at modern compiler technique: languages such as Java and C# translate the source code into an intermediate language and use a just-in-time compiler [DS84] to compile functions and methods as they are needed.

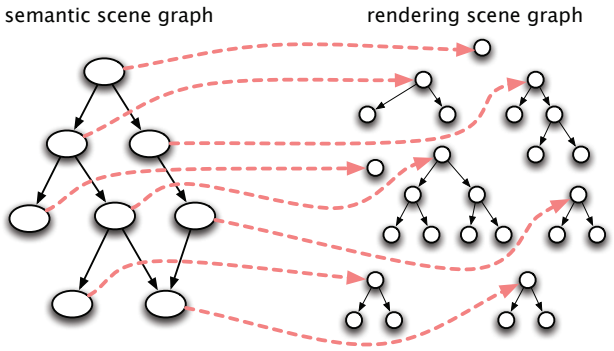

Fig. 4 During on-the-fly translation, the semantic scene graph is translated into a forest of rendering scene graph pieces.

By using the same technique for generating the rendering scene graph from the semantic scene graph, the rendering scene graph will become a forest of small scene graph pieces that are generated on the fly as they are needed when the semantic scene graph is traversed (see figure 4). In a way our separation of semantic and rendering scene graph is reminiscent of the Scene Graph as a Bus System [ZHC*00], however, the forest of rendering scene graph pieces can be completely reconstructed from the semantic scene graph, and represents an expanded or translated version.

In order to make the rendering scene graph truly dynamic we introduce state into the translation step, and allow the traversal of the scene graph to modify the existing rendering scene graph pieces to reflect the new state (see figure 5). This has been implemented by creating a dictionary of translation rules that contains creator functions for *rule objects* that contain the current state of the translation. Each constructor of such a rule object builds a rendering scene graph piece that corresponds to the translated semantic scene graph node, and stores a reference to it.

All the rule objects are stored in the so-called *traversal cache* of the render traversal, and on subsequent traversals of the semantic scene graph, a look-up in this cache is performed whenever a semantic scene graph node is traversed. By making all rule objects implement a single *action* method that is called before traversing the corresponding rendering scene gaph piece, each rule object can react to changes in the global state, and modify its rendering scene graph piece.

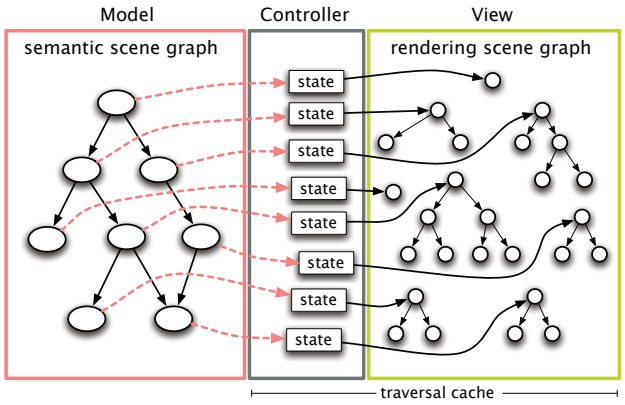

Fig. 5 On-the-fly translation of each semantic scene graph node creates a *rule object* that contains the current state of the translation. Each rule object contains a reference to its rendering scene graph piece. This structure can be viewed as a variant of the model-view-controller design pattern.

This architecture for scene graphs can be viewed as a variant of the well-known Model-View-Controller design pattern [Ree79]. Here the semantic scene graph represents the model, i.e. the data as the user conceived it, the rendering scene graph represents the view of this model, and the rule objects containing the current state of the translation represent the controller (see figure 5).

4 Implementation

The presented architecture has been implemented in a new rendering framework written in C#. Although it would have been possible to implement the architecture on top of one of the presented scene graph systems, the support for modern language features such as type-safe generics and functional programming simplified some details of the implementation.

Generic scene graph traversal

In order to provide all the functionality that is necessary in a typical rendering framework, multiple types of traversal need to be allowed. If every traversal T needs to be implemented for each and every node type N this is an $O(T \cdot N)$ implementation effort. Optimally we would like to limit the implementation effort to adding interface implementations to just those node types that need to perform operations that are different from a default traversal. As an example, only nodes containing geometry or shader information need to implement an interface that contains the actual rendering method.

This can be achieved by implementing the actual traversal function in a generic way with three type parameters: the type of the operation that needs to be

performed `TInterface`, the type of the traversal state object `TTraversal`, and the result type `TResult` type. In C# the signature of this generic traversal method looks like this:

```
public interface NodeInterface {
  TResult Traverse
    <TInterface, TTraversal, TResult>(
        TTraversal traversal,
        Func<TInterface, TTraversal,
             TResult> fun);
}
```

Implementations of this function on the various node types use run-time type information to determine if the supplied interface `TInterface` is implemented by the node type, and if it is, they use the supplied function `Func<TInterface,TTraversal,TResult> fun` to call a specific method of the interface. This function receives the node already cast to the specific interface and the traversal state object as a parameter, and returns the result of the traversal operation on the node. If the specified interface is not implemented by the node, the generic traversal function returns default values for leaf nodes, or traverses the children of the node for intermediate nodes, and aggregates the result by performing result-type specific aggregation.

Specific traversals such as the typical render traversal are then implemented by specifying an interface for all renderable nodes:

```
public interface IRenderable {
    RenderResult Render(
        RenderTraversal traversal);
}
```

This interface needs to be implemented for all node types that need to be rendered. An actual traversal calls the methods in this interface via the generic traversal method:

```
traversable.Traverse(
  renderTraversal,
  (IRenderable node, RenderTraversal t)
    => node.Render(t));
```

Traversal of semantic nodes

The generic traversal for a semantic scene graph node takes the type (or a type identifier) together with a unique identifier of the actual node as a key into the traversal cache. If a rule object is found in the cache, its `Action` method is called to allow the rule object to make changes in its render scene graph part. This action method returns the actual scene graph part that will then be traversed.

If no rule object is found in the traversal cache, the traversal state object's rule map, a dictionary of creator functions that uses the type as a key, is used to find a creator function for the type of the semantic node. This creator function is called to create the rule object and its associated render scene graph part. The rule object is stored in the traversal cache. Afterwards things proceed as if the rule object was found in the cache.

Attribute applicators

Attributes such as transformations and surfaces are applied to a scene graph with so call *attribute applicator* nodes. These applicator nodes are binary nodes with two sub-graphs: the left sub-graph specifies the scene graph on which to apply the attribute, and the right sub-graph specifies the actual attribute. The attribute value is obtained by performing a *GetAttribute* traversal on the right sub-graph. As an example, consider applying a transformation consisting of a rotation and a scale on a scene graph. Figure 6 shows the relevant piece of the scene graph.

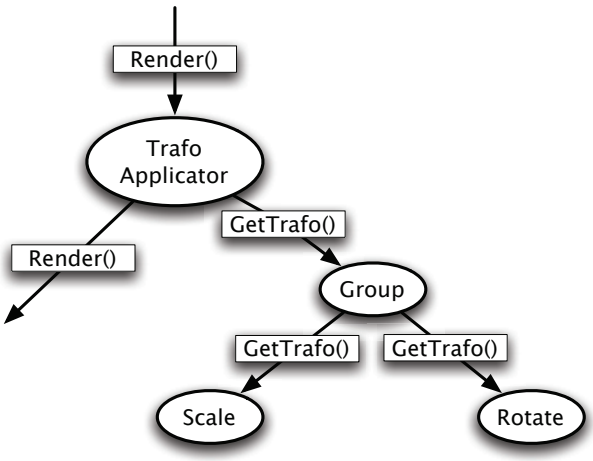

Fig. 6 A transformation applicator applies a transformation to its left sub-node by performing a *GetTrafo()* traversal on its right sub-node, and pushing the returned transformation onto the traversal state before traversing its left sub-node.

The result-type specific aggregation for transformations, which is performed by the generic traversal of the group node, is in this case the multiplication of the transformation matrices.

This logical separation of attributes in sub-trees allows the implementation of any type of attribute in a hierarchical semantic fashion, with the concrete interpre-

tation of the attribute constructed on the fly as a rendering scene graph. As an example, material properties can be implemented as semantic shading trees [Coo84] that are directly integrated in the scene graph system. In a current rendering system, the translation of such a shade tree will be a rendering scene graph that contains the vertex and pixel shaders necessary for achieving the desired semantic material properties specified in the semantic shading tree. By adding additional semantic attributes that modify the visual style created by the rendering scene graph, the semantic functionality that has been presented by Mendez et al. [MSH*08] can be directly implemented within the rule objects of the new scene graph system without resorting to application logic outside the scene graph.

Dynamic scene graphs

The simplest form of a dynamic scene graph can be realized with the attribute applicator nodes shown in the previous section. In the example given in figure 6, the *GetTrafo()* traversal employs the described operations on the semantic *Rotate* and *Scale* nodes. It is thus possible to implement a *Rotator* node that can be used in place of the static *Rotate* node, that returns a time-dependent transformation (see figure 7).

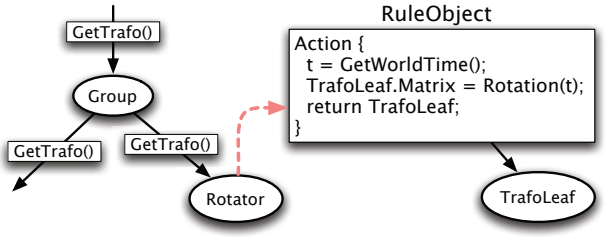

Fig. 7 A rotator is implemented by modifying its rendering scene graph—consisting of a single *TrafoLeaf* node that contains a concrete transformation matrix—in its *Action* method.

More complicated dynamic scene graphs have been implemented by creating new pieces of rendering scene graphs or performing more complex modifications on the existing rendering scene graph in the *Action* method of the various rule objects.

Out of core rendering

Out of core rendering represents a special case of a dynamic scene graph. In this case, we created semantic nodes that contain file references and usage data of the stored rendering scene graph. When the semantic node is encountered during scene graph traversal, the creator function loads the rendering scene graph from disk, if it is not already in the traversal cache. In order to remove rendering scene graph pieces from the traversal cache, some cache statistics need to be maintained, such as size and last access of nodes. If there is not enough space for loading a scene graph piece, other scene graph pieces that have not been used for a while can be deleted from the traversal cache.

Level-of-detail

A simple semantic level-of-detail node can be implemented by creating a rule object that maintains multiple rendering scene graph parts, each of which represents a single level-of-detail. The Action method of the rule object can then return the appropriate part, depending on the current position of the camera with respect to the nodes (see example in figure 8).

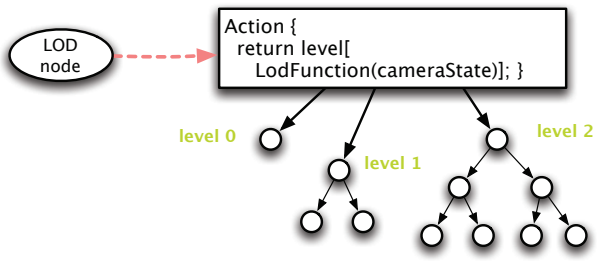

Fig. 8 A simple semantic LOD node is implemented by having its rule object select one of multiple rendering scene graph parts in the Action method.

Based on this simple design, we have implemented more complex level-of-detail nodes that load the data for the different levels onto the graphics card in the background, before actually switching to the optimal level of detail.

Procedural generation of geometry

Complex geometric objects such as trees and buildings are often created by using procedural generation of geometry. The presented system for creating rendering scene graphs on the fly has been extended to allow the creation of semantic nodes as well. In this case, the rendering scene graph part that is created by a rule object will also contain semantic nodes, that again trigger the creation of additional rule objects. Figure 9 shows an example of the semantic and rendering scene graph

parts that could be created during the procedural generation of a tree.

The fact that one type of semantic node (in the example the *tree* node) generates other semantic nodes (such as the *branch* and indirectly the *leaf* nodes) reflects the fact that the semantic type and parameters allowed for a sub-part of an object are dependent on the parent object (e.g. a maple only has maple leaves), nevertheless the sub-parts can exist as independent semantic objects (e.g. a maple leaf that has fallen to the ground).

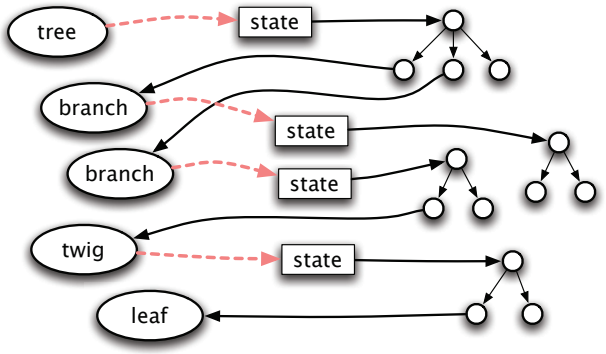

Fig. 9 Procedural generation of a tree through the creation of semantic nodes within the rendering scene graph parts created by rule objects: the *branch*, *twig*, and *leaf* nodes are created on the fly.

In such procedural geometry generation scenarios, all of the semantic nodes of an object (the *tree*, *branch*, *twig*, and *leaf*, in the example in figure 9) are parametrized, and may need to access some parameters of their parent nodes. In order to facilitate this behavior, the generic traversal of the semantic scene graph has been extended to maintain a so-called *scope stack*. This is actually a dictionary of stacks—one for each type of semantic node—that maintains the last traversed rule object of each type. Whenever the traversal enters a semantic node, its rule object is pushed onto the appropriate stack. As the traversal leaves, its rule object is removed from this stack. With this stack in place, each rule object has access to all its parents, i.e. the rule objects that are based on semantic nodes higher up in the semantic scene graph, and can thus access any parameters that are made available to it via public fields. In our example, the *leaf* can be made dependent on various fields of the *twig*, *branch*, and *trunk* rule objects. The reason for this scoping feature to be implemented with a stack, is that even a recursion of rule objects of the same type is made possible, and thus typical L-system type procedural vegetation that generates a sequence of segments of a branch or stem can be realized.

Although L-systems and other types of grammars for procedural generation of vegetation geometry have been successfully implemented on top of scene graphs or similar systems [GT96], [SG97], the clear separation between semantic and rendering nodes in the new design leads to a significantly simpler implementation.

Note that the rules in figure 9 represent only one possible way of using the separation between the semantic and rendering scene graphs for procedural generation of geometry. By implementing special traversals that retrieve the geometry from a complete subgraph, and adding semantic nodes for processing such geometry, it is also possible to implement other approaches, such as the one introduced by Lintermann and Deussen [LD98] within the presented architecture.

Multi-view rendering

In a number of application scenarios the same scene must be displayed from multiple view-points at the same time. Since the semantic scene graph is only accessed in reading fashion during all traversal operations, the simplest implementation provides separate traversal state objects and separate traversal caches. Although this is possible, it leads to a replication of a possibly large number of nodes. A more sophisticated approach splits the traversal cache of each traversal state object into a shared part, containing rendering scene graph parts that can be shared with other views, and a private part that is not shared (see figure 10). Note that only the shared cache needs to employ locking mechanisms if the actual traversals happen in parallel. In our implementation the decision on where to put a specific rule object and rendering scene graph, is based on the type of the semantic scene graph node.

This type of parallel interpretation of the same scene graph for multiple views highlights the advantages of the separation between semantic and rendering scene graphs: any design that involves fixed references from the semantic to the rendering scene graph (in most of the previous designs various rendering caches can be viewed as such references), needs a much more complex and elaborate locking scheme for the traversal of a dynamic scene graph, if actual parallel execution is involved.

Editing of semantic nodes

In modeling or editing applications, the one-way information flow from semantic to rendering node needs to

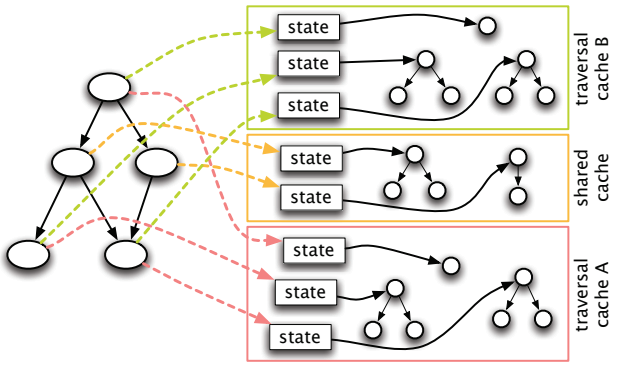

Fig. 10 When a semantic graph is traversed with two traversal state objects, each builds its own traversal cache (A, B). For some semantic node types it can be decided, that they reside in a single shared cache.

be augmented with a way to actually modify the semantic scene graph, since this graph represents the model as it is stored on disk. This can be accomplished by having each rule object maintain a reference back to the semantic node from which it was created. In typical modeling applications, objects that are edited are selected and rendered with a set of handles for manipulating the object. This can be easily realized in the new architecture by having the rule object maintain two rendering scene graphs: one for rendering, that just contains the visual representation of the semantic node, and another one for editing, that contains the visual representation of the handles for editing (see figure 11).

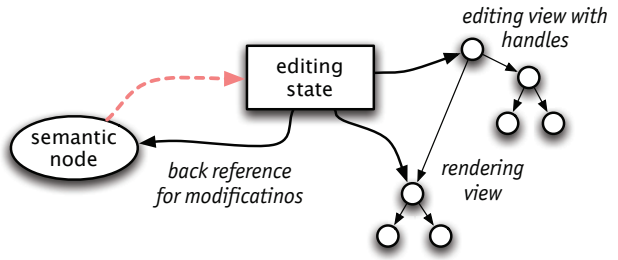

Fig. 11 The rule object of a semantic node in a modeling application, contains the editing state of the semantic object and maintains references to the semantic node, and two rendering scene graphs, one for rendering and one for editing.

Allowing every type of operation on semantic nodes via the back reference could potentially undo some of the advantages of the clean separation of semantic and rendering scene graphs. In order to avoid this design problem, we only allow a access via a special synchronized interface, with the additional benefit that the intermediate layer that implements this interface acts as a multi-level undo manager.

5 Practical considerations

The clean separation between semantics and rendering specific scene graph makes it very easy to build reusable components that do not affect other parts of the scene graph. Within the framework that was implemented based on this architecture, more than 100 such components, each represented by semantic nodes, have been created. This ranges from various level-of-detail nodes, over a number of editing nodes for a modeling application, animation nodes for displaying animated 3D content, to nodes for procedural generation of vegetation (see figure 12 for some examples).

One of our decisions when implementing this architecture, was to opt for the more flexible implementation as opposed to the optimized implementation in a number of cases. As an example, all nodes are traversed with the generic traversal presented in the implementation section, even if they are nodes of the rendering scene graph.

In practice this results in a limitation of about 1000 independently transformed and animated simple primitives (with dynamic transformations as shown in the previous section) for an interactive frame-rate of about 30 frames per second (all performance figures are given for a state of the art desktop PC with a Quad-Core Intel i7 processor at 2.7GHz, 12 GB of RAM and an nVidia GTX 295 graphics card).

Although a specialized rendering traversal for these nodes could potentially boost the traversal performance of the framework, we avoided specialized implementations and the associated maintenance overhead and opted for a different, more global optimization strategy by offering generic tools for optimizing the scene graph to obtain a smaller number of nodes, each with a larger amount of geometry. This has been implemented with a *GetGeometry()* traversal to extract geometry of various types (e.g. static geometry) from a scene graph and store it near the root node. Using such a traversal (that additionally identifies identical primitives with different transformations) and optimizing a scene of 1000 independently transformed and animated simple objects to use hardware instantiation results in a frame-rate of more than 1200 frames per second.

In a number of application scenarios, such as terrain rendering and rendering of laser-range point clouds, we use a hierarchical scene structure with level-of-detail nodes and leaf nodes containing a fairly large number of triangles or points (10,000 to 100,000) in order to achieve frame rates of more than 60 frames per second while displaying around 10+ million textured triangles or points.

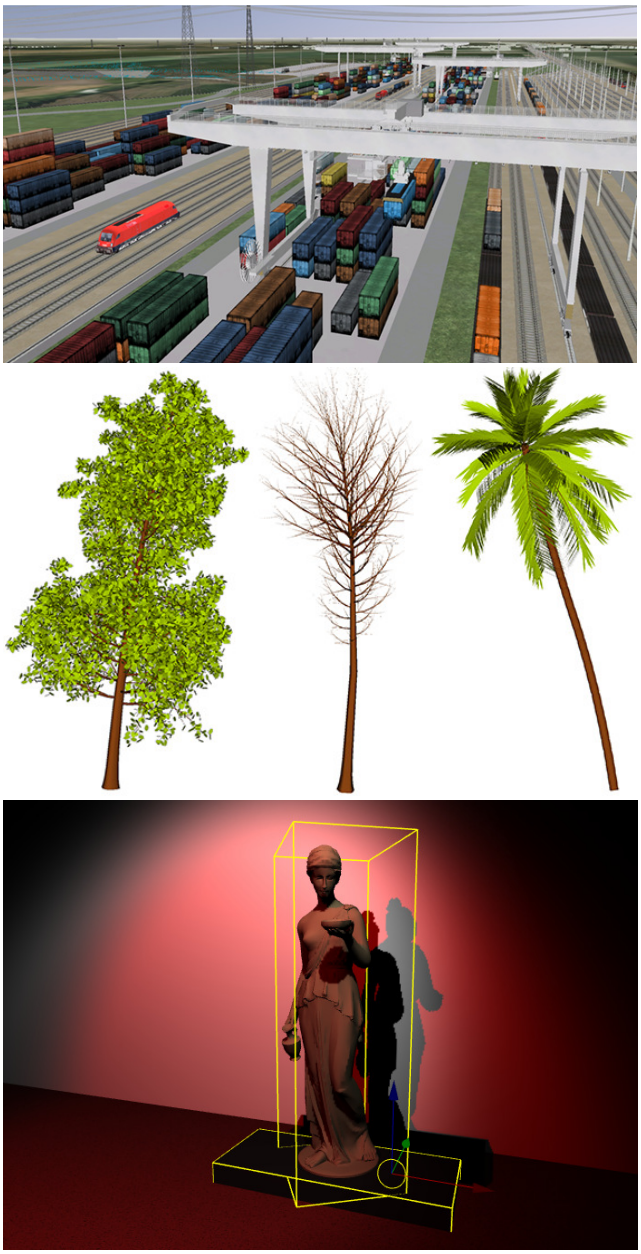

Fig. 12 Examples of the use of semantic nodes in practice (from top to bottom): (1) A terrain rendered with LOD nodes and animated trains in a previsualization of a large construction project. (2) Vegetation geometry generated with semantic nodes as described in the previous section. (3) A selected object with handles and transformation UI elements displayed via a semantic scene graph rule with editing view.

6 Conclusions and future work

The presented separation of semantic and rendering scene graph constitutes a architecture that can be applied to all kinds of rendering applications. As an example, rendering an HTML file in a web browser can be easily expressed with this architecture: the parsed HTML tree represents the semantic scene graph, the HTML-page is rendered from a rendering scene graph that is generated from this parsed HTML tree on the fly with a set of rules that expands each semantic HTML node into a rule object with a reference to a rendering scene graph part describing its visual representation.

Implementing the architecture in a large rendering framework called Aardvark, that is used as a basis for numerous projects at the VRVis Research Center made it possible to explore its implications and has proven its applicability in a variety of scenarios, some of which have been presented in this paper.

Since the architecture works similar to a just-in time compiler, a number of optimizations from compiler technology can be used on the semantic scene graph. As an example, the equivalent to constant folding in compiler optimization is the combination of the static parts of a scene graph into a small number of optimized rendering scene graph nodes that can be quickly rendered without state changes. Performing these optimizations automatically, will be explored in the future, in order to improve rendering performance.

Another research avenue that has not been followed up yet, is parallelized rendering within this architecture. Due to the clean separation of semantic and rendering data this will be fairly straightforward to integrate.

Acknowledgements I would like to thank all my colleagues at the VRVis Research Center who made this work possible. Especially Stefan Maierhofer, Matthias Buchetics, Harald Steinlechner, Michael Schwärzler and Christian Luksch provided invaluable help in validating the approach presented in this paper.

References

[BO04] Burns D., Osfield R.: Open scene graph a: Introduction, b: Examples and applications. In *VR '04: Proceedings of the IEEE Virtual Reality 2004* (Washington, DC, USA, 2004), IEEE Computer Society, p. 265.

[Bos09] Bosi M.: Visualization library. First official version, July 2009. available from: http://www.visualizationlibrary.com/ (accessed February 11, 2011).

[Coo84] Cook R.: Shade trees. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques* (1984), ACM, pp. 223–231.

[DS84] Deutsch L. P., Schiffman A. M.: Efficient implementation of the smalltalk-80 system. In *POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (New York, NY, USA, 1984), ACM, pp. 297–302.

[GT96] Gervautz M., Traxler C.: Representation and realistic rendering of natural phenomena with cyclic csg graphs. *The Visual Computer 12*, 2 (1996), 62–74.

[KM09] Kunz H., Miller P.: NVIDIA® SceniX™ scene management engine. First official version, 4 Aug 2009. available from: http://developer.nvidia.com/object/scenix-home.html (accessed February 11, 2011).

[LD98] Lintermann B., Deussen O.: A modelling method and user interface for creating plants. *Computer Graphics Forum 17*, 1 (1998), 73–??

[MSH*08] Mendez E., Schall G., Havemann S., Junghanns S., Fellner D., Schmalstieg D.: Generating Semantic 3D Models of Underground Infrastructure. *IEEE Computer Graphics and Applications* (2008), 48–57.

[Ree79] Reenskaug T.: Models - views - controllers. XEROX PARC tech note, 19 Dec 1979. available from: http://folk.uio.no/trygver/themes/mvc/mvc-index.html (accessed February 11, 2011).

[RH94] Rohlf J., Helman J.: Iris performer: a high performance multiprocessing toolkit for real-time 3d graphics. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1994), ACM, pp. 381–394.

[RS05] Reitmayr G., Schmalstieg D.: Flexible parametrization of scene graphs. *Virtual Reality Conference, IEEE 0* (2005), 51–58.

[SC92] Strauss P. S., Carey R.: An object-oriented 3d graphics toolkit. In *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1992), ACM, pp. 341–349.

[Sea05] Streeting S., e. a.: Ogre3d: Object-oriented graphics rendering engine. First official version, Feb 2005. available from: http://www.ogre3d.org/wiki/index.php/ (accessed February 11, 2011).

[SG97] Schmalstieg D., Gervautz M.: Modeling and rendering of outdoor scenes for distributed virtual environments. In *VRST '97: Proceedings of the ACM symposium on Virtual reality software and technology* (New York, NY, USA, 1997), ACM, pp. 209–215.

[VBRR02] Voss G., Behr J., Reiners D., Roth M.: A multi-thread safe foundation for scene graphs and its extension to clusters. In *EGPGV '02: Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization* (Aire-la-Ville, Switzerland, Switzerland, 2002), Eurographics Association, pp. 33–37.

[Wer93] Wernecke J.: *The Inventor Mentor: Programming Object-Oriented 3d Graphics with Open Inventor, Release 2.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.

[ZHC*00] Zeleznik B., Holden L., Capps M., Abrams H., Miller T.: Scene-graph-as-bus: Collaboration between heterogeneous stand-alone 3-d graphical applications. In *In Proceedings of Eurographics 2000* (2000), pp. 200–0.