A Flexible, Low-Level Scene Graph Traversal with Explorers

Radek Ošlejšek*
Masaryk University
HCI Laboratory
Botanicka 68a, Brno, 602 00, Czech Republic

Jiří Sochor†
Masaryk University
HCI Laboratory
Botanicka 68a, Brno, 602 00, Czech Republic

Abstract

We introduce a novel design architecture for scene graph based applications. A model is based on GoF design patterns with respect to reusability and maintenance. With integration of patterns into the construction of a scene graph, the scene's description can be easily extended by new features, as new types of scene graph attributes, spatial data structures and geometries. The proposed model supports efficient traversal of a scene graph based on unified interfaces of scene graph nodes. It offers generic yet sufficiently low-level framework usable by a wide range of various rendering strategies.

CR Categories: I.3.2 [Graphics Systems]: Stand–alone systems; I.3.8 [Applications]

Keywords: scene graph, design patterns

1 Introduction

The primary goal of this research is to find an object-oriented architecture allowing an implementation of various rendering techniques for 3D scenes which use a hierarchical description based on a scene graph. Because a scene graph forms the core of any rendering application it's architectural properties are crucial. This is noticeable particularly in the case of generic architectures that call for extensibility of a scene as well as for efficiency of objects search.

Scene graphs are heterogenous directed graphs, ussually acyclic. But heterogenity of nodes causes the maintenance of scene graphs very difficult. Heterogenity means that each node in the graph can be individual with its own nature, properties and interface. If we want to inspect a whole scene in an unified way we have to find a generic mechanism for a handling of nodes. Handling must include not only an inspection of the content of existing node but also the employment of new nodes which extend a scene graph by new objects and features. For example, photorealistic rendering techniques could require nodes describing participating media (e.g. fog, smoke, flame) whereas haptic applications could demand description of forces. A scene graph should be able to adopt these new features without radical changes to an existing source code.

In addition to extending a scene by new nodes and objects, various rendering architectures require special operations over the scene. For example, ray tracing architecture computes intersections with casted rays, haptic architecture must detect collisions. A scene graph should allow to append such global operations easily. Unfortunately extending the scene by new objects and extending the scene by new global operations seems to be antagonistic.

*e-mail: oslejsek@fi.muni.cz
†e-mail: sochor@fi.muni.cz

Another critical feature of a scene graph is the efficiency of objects search. In particular, the algorithms of global illumination search a scene many times computing the energy propagation.

To reduce the complexity of search in space, several spatial data structures were developed. However, most of existing graphics libraries and languages, e.g. Java 3D [Sowizral et al. 1997; Barrilleaux 2001], and VRML [Ames et al. 1997], are designed to use bounding volume hierarchies, only. The addition of a new spatial ordering to a scene graph is very difficult and the maintenance of a hybrid solution is complicated. We therefore aim to find the hierarchical description of a scene which is easily extensible with various spatial data structures.

In order to support the extensibility and flexibility of graphics solutions, we employ an object-oriented approach with patterns. The notation is based on the Unified Modeling Language (UML), which is the industrial standard (UML version 1.5 [Booch et al. 1998; Rumbaugh et al. 1998; Harmon and Watson 1997]) enabling software designers to specify, visualize and construct complex systems as well as to create their documentation. UML specification uses primarily diagrammatic models describing static and dynamic aspects of a system composed of the cooperating objects. UML is also the basic tool used for the specification of *OO analysis and design patterns*.

2 Previous Work

The definitions of class-type libraries and APIs are described in [Sowizral et al. 1997; Schroeder et al. 1996; Ames et al. 1997; Barrilleaux 2001]. Architectural decompositions are usually limited to the fragments of a complex architecture, e.g. to a scene graph representation, or to a monolithic solution of a rendering strategy. Object-oriented design and software patterns related to the objective of this article, are specified in [Buschmann 1996; Coad 1997; Gamma et al. 1995; Niemann 1981; Vlissides 1998].

In [Lee et al. 1999], a *Generic Graph Component Library* is described. GGCL is a generic programming framework for graph data structures and graph algorithms. OO decomposition includes DECORATOR pattern accomplishing inspection of nodes and VISITOR pattern, which defines strategy of searching in graph. Features of GGCL restrict its usage only to graphs with unified nodes which have a predefined structure. However, virtual objects stored in a scene do not posses unified interfaces. Therefore, GGCL is not directly applicable to a decomposition of a scene graph.

In [Döllner and Hinrichs 2002], generic rendering system uniformly handles the existing proprietary systems, as *OpenGL*, *POV-ray* or *RenderMan*. It also deals with a scene graph decomposition and it's maintenance. The analysis of the scene graph arrangement and of traversal strategies leads to an extensible solution of a rendering system. ITERATOR and VISITOR design patterns are explicitly mentioned in this context.

Since a skeleton of a scene graph often takes a form of a tree it can be implemented using a GoF pattern COMPOSITE. This pattern secures scene operations into the interface of a scene graph node. The whole scene then posseses a behaviour of a single unified object. Unfortunately, with this encapsulation it is difficult to define

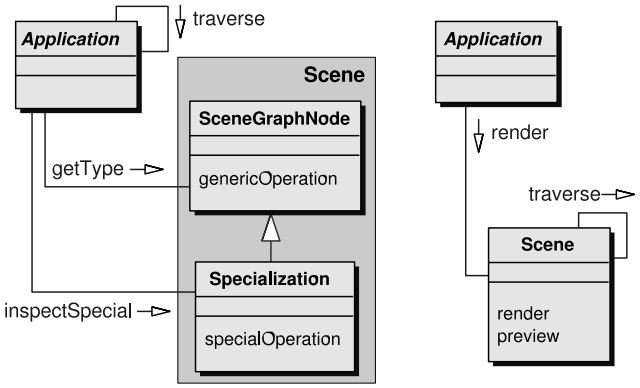

Figure 1: Minimal (left) and puristic (right) scene encapsulations

new operations. Such definition requires changes in the interface of the most of existing scene graph nodes, which is not practical.

In order to allow changes of operations in a scene graph node, some implementations of scene graph extend the COMPOSITE pattern with the concept of visitor allowing to append a new operation easily, without the necessity to change the current definition of a scene graph node. VISITOR pattern is used in implementation of the *Open Scene Graph* [Ope a].

OpenSG project [Ope b; Reiners 2002; Reiners et al. 2002] combines several design patterns in order to get extensible and easily maintainable scene. Similar to previous projects, the project is also focused on the solution of a local illumination rendering.

The remainder of this paper is organized as follows: Section 3 describes various levels of scene graphs encapsulation and explains relation to the rendering equation. Subsection 3.1 introduces a concept of explorers. Section 4 deals with methods of efficient scene graph traversal which are based on ITERATOR and VISITOR patterns. In Section 5 we discuss the maintenance of attributes in scene graph nodes. Overall architecture and experimental library is described in Section 6 and 7. We conclude with the discussion of future work in Section 8.

3 Levels of Encapsulation

Possible decompositions of scene may use different levels of encapsulation. Architectural designs which are easy to implement but difficult to maintain, are those offering minimal encapsulation. Nodes of a scene graph have interfaces that are individual for every different type of node. Applications have to implement a heterogenous traversal algorithm which takes into account various interfaces of node objects, Figure 1 left. The application then becomes very complicated and tightly interconnected with scene graph. Any change in the scene graph usually leads to the major changes in the code of the application.

On the other hand, puristic design fixes scene operations and traversal algorithms in the interface of scene. Application invokes requested operation and this operation is applied to the whole scene recursively, i.e. the scene behaves like single object, see Figure 1 right. Such behaviour can be found for example in scene graphs that use the COMPOSITE structural pattern.

Between these two extremes there is a wide space for decompositions more suitable for maintenance.

In the computer graphics the very basic operation is rendering, i.e. visualization of virtual objects stored in the scene. A support for rendering is also the basic function of a scene. The *rendering equation* [Kajiya 1986] is a general mathematical formula for rendering process. Concrete algorithms (including real-time local illumination) numerically solve this equation. In order to gain a general solution, we propose the decomposition based on a factorization of the rendering equation. The equation has the form:

$$L_o(x,\vec{\omega}_o) = L_e(x,\vec{\omega}_o) + \int_\Omega f_r(x,\vec{\omega}_o,\vec{\omega}_i) L_i(x,\vec{\omega}_i) \cos\theta_i d\vec{\omega}_i, \quad (1)$$

where L_o is outgoing radiance at point of interest x in direction $\vec{\omega}_o$, L_e is radiance emitted at the point x, L_i is irradiance from surrounding environment and f_r is BRDF.

Traditional structural decompositions of scene either suppose only a single rendering algorithm, mostly a local illumination of polygonal meshes, or they transfer the evaluation of the rendering equation to the process of traversal. For instance, they define object (often called *agent*, *action*, etc.) which act as render for the whole scene. This render traverses a scene graph, performs same operations on nodes and it evaluates an energy distribution in a scene, Figure 2 left.

Our approach is different. We release the strict encapsulation on the highest level in order to separate scene graph traversal from energy propagation. To the scene graph we delegate only those operations that are necessary. A recursive evaluation of the rendering equation is controlled by a graphics application, Figure 2 right.

3.1 Concept of Explorers

Irradiance L_i in equation (1) represents the energy reflected through a point x from environment. In practice, algorithm evaluating the equation must actively search for surrounding objects, i.e. concrete points at surfaces, neighbouring polygons. But different rendering algorithms can have different requirements to the search. A definition of a new rendering algorithm could therefore require a definition of a new search strategy. We focus on an efficient scene searching and a maintenance. For this, we define intermediating objects between a rendering algorithm and the scene.

Explorer defines an abstract interface mediating a scene traversal. Each explorer implements the specific searching strategy. For exapmle, explorer can return the first object that was impacted by a casted ray, other explorer can search for the light sources visible from the given space point, etc. This concept releases the dependence of a rendering system on the scene's internal representation.

While traditional agent-based class can be used to apply the operation on a scene directly, an explorer serves as a search engine only. It looks for required objects (nodes) of a scene graph. After the search, a graphics application can apply the operation to denoted primitive objects.

Since an explorer is responsible for the searching strategy, graphics application evaluating the rendering equation can use explorers to access the scene. During the traversal explorer controls the search by inspecting scene graph nodes. To inspect nodes in uniform manner explorers should maximally exploit the principles of behavioral design patterns.

4 Efficient Scene Graph Traversal

The search for an object can be done in two distinct ways. The first possibility is to somehow localize an object in space. Typically, a scene's sub-space is defined as a ray, half-space, or a volume, and we would like to find all objects colliding with the sub-space. Hence we need to find the candidates for intersection with a ray or the objects occupying a given region. For complex scenes with many primitives, it is difficult to traverse the whole scene graph

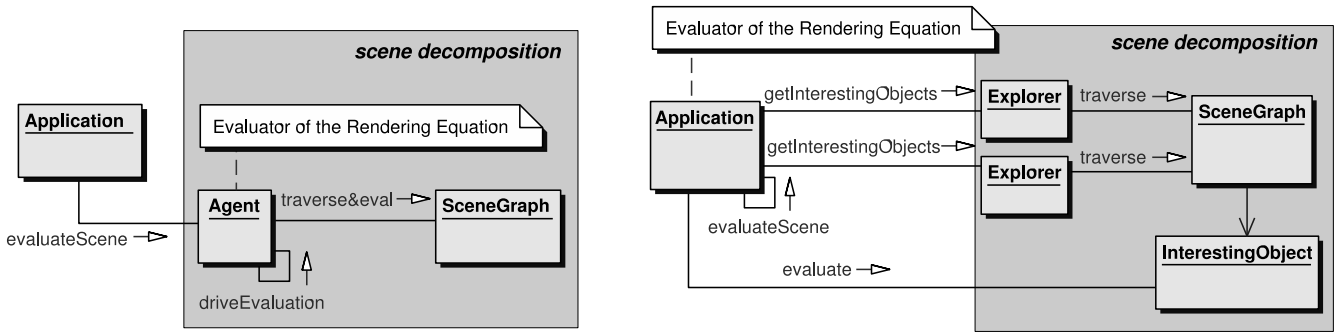

Figure 2: Scene decomposition with agents (left) and explorers (right)

and to check object by object for a given condition. A better way is to sort objects in a *space-ordering data structure* and reduce the number of candidates tested. In computer graphics, this kind of structure is usually referred to as a *spatial data structure, SDS*.

The second way of looking up an object is to use our knowledge of space coherency. For instance, if we know the nearest neighbour of an object in a scene then it is possible to take advantage of this knowledge and to move directly to the neighbour instead of traversing entire scene.

These search principles raise additional requirements on a scene graph. A space-ordering structure has to be implemented in a scene graph hierarchy and the space coherency knowledge must be preserved in a form of the relation allowing direct access from one node to another. There are two obvious choices how to store it: internally as a part of scene graph structures, or externally in the form of a static or dynamic coherency map.

The first alternative leads to solutions where specialized explorers know the coherency information and utilize it efficiently. The flexibility is limited to a narrow class of subspace travelling techniques. Coherency information is prepared in the preprocessing phase, stored as direct links between scene graph nodes, and in most cases, it remains static for the rest of the scene's life.

The second approach stores coherency information externally in a common or private *travel map* used by specialized explorers. The map could be used to store acquired knowledge about scene structure gained during earlier walks through the scene. Explorer knows what it is looking for and how to search for it efficiently. Explorer controls traversal and has the global information which is necessary to make appropriate decisions. This information is gained during the inspection of nodes and accumulated during the search. Hence, coherency knowledge can be easily hidden inside individual explorers with respect to their needs and it does not matter whether the coherency is included in the scene graph or described in a separate travel map carried by the explorer.

Since coherency does not significantly affect the structural models of scene with explorers, these techniques are not discussed in this paper.

4.1 Scene Graph Superimposed with BVH

When searching in a scene, the scene graph can be traversed node by node. However, acceleration techniques try to skip uninteresting branches of a tree. The only way how to leave out some branch under certain conditions is to use unified constraint rules and to implement this type of behaviour on explorers' side.

Usual solution combines two approaches. First approach uses specialization of generic grouping node and hides constraint rules into grouping nodes. As an example, we mention the *switch* node, which can be found in many scenes. This node changes a traversal strategy so that the search is directed to only selected children. Second approach uses constraints that are common for every node in a scene graph. Typically, any node may contain the information about *bounding volume*, which wraps the node and its children. The scene graph is superimposed with *Bounding Volume Hierarchy, BVH* (Figure 3) and the agent performing traversal can test the bounding volume before it moves to children.

This traditional encapsulation of a scene graph depends to the large extent on an internal organization of a scene. A concrete node decides which of its components, children, attributes etc., will be inspected. The decision is based on the information known to this node only. The node has no information about global context as well as about the operation realized by a visiting class. Therefore it is not easy to extend this concept to various SDSs or to coherency-based operations.

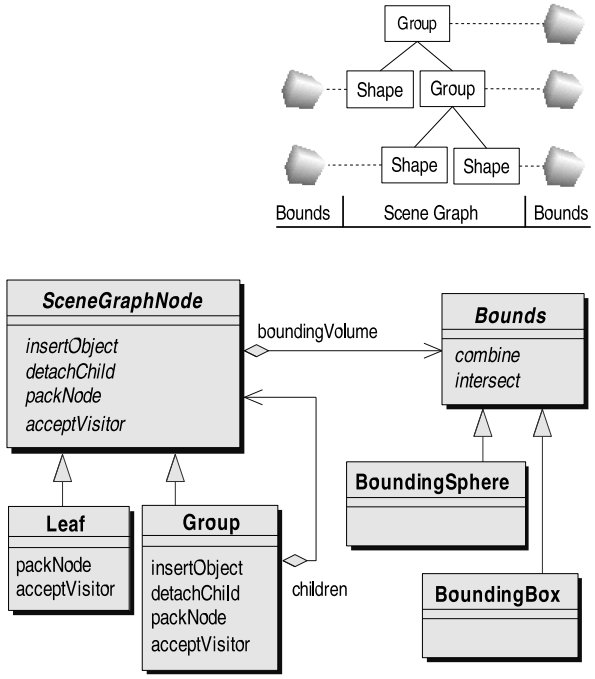

Figure 3: Scene graph superimposed with bounding volume hierarchy

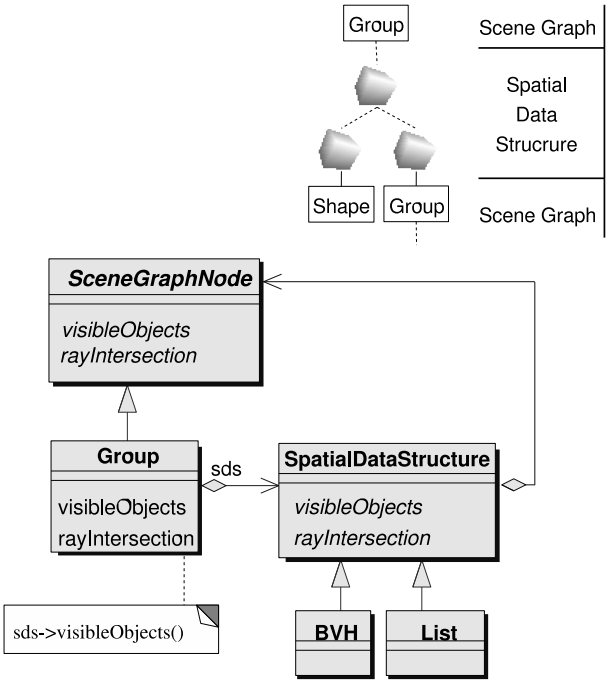

Figure 4: Spatial data structure hidden inside the group node

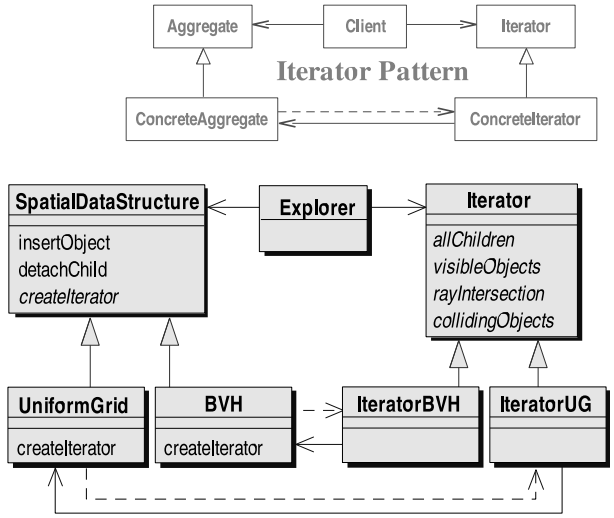

Figure 5: Traversal operations detached from scene graph by ITERATOR pattern

4.2 Generic Model for Spatial Data Structures

When we want to use an arbitrary spatial data structure together with a scene graph, we must detach SDS from the scene graph skeleton. Rather than associating the individual nodes of a scene graph with spatial ordering information, we can include the general SDS with the ordering into the branching nodes, as can be seen in Figure 4. In this case, the group node can be understood as a node storing its subnodes in a specific spatial data structure.

The advantage of this solution is that any spatial data structure can be employed inside a group node. On the other hand, a simple narrow interface of bounds-driven traversal is lost. This is because the diverse structures require different interfaces for traversals. For instance, a sequential list of subnodes provides the native functions like *firstChild()* and *nextChild()* for children inspection. Tree-based structures (volume hierarchy, octree, BSP tree, k-d tree etc.) typically implement back-tracking, i.e they offer functions like *goToIthNode()* and *backtrack()*. The uniform 3D grid (voxels) implements functions returning the closest neighbouring cells. To cover this diversity by a uniform interface of the group node, any function over the scene would have to be fixed in the scene graph interface. Hence operations like *objects intersected by ray*, *object visible from a point in space*, *colliding objects* etc. must become part of the group node interface. These operations should return only the subnodes that have the opportunity to succeed.

The group node behaves as a simple mediator between client (explorer) and concrete spatial data structure. The entire structure of a scene graph is significantly simplified and the spatial structure can be reused in other tasks, for example to create complex geometries. A general group node can be specialized without the necessity to implement an individual subclasses for every existing spatial data structure. For instance, a special group node can transform the whole subtree in space by overriding the general group node and associating an arbitrary SDS to it. This concept therefore significantly reduces subclassing.

Having a SDS associated with a group node brings another advantage. It is possible to share a spatial data structure among several group nodes and to share the whole subtree of a scene graph. Sharing individual objects, especially geometries, is important feature that reduces memory requirements and simplifies a scene maintenance.

This decomposition allows to use many SDSs inside one scene graph, preserving the uniformity of nodes. A whole presorted scene can be stored in other spatial data structure, since the class hierarchy of SDS is defined separately from the class hierarchy of scene graph nodes. A graphics application can ignore existing scene graph ordering and spatially restructure the scene to improve it's efficiency.

The proposed model has a serious drawback. A group node and a spatial data structure have all the operations over the scene fixed in their interfaces. Whenever it is neccessary to modify the interface, usually adding a new operation, we have to redefine all derived classes. The definition of a new operation over a scene could therefore lead to extensive changes in the class hierarchy. This problem is disscussed in following sections.

4.3 Iterator-based Traversal

In order to simplify adding of a new scene operation, we use the ITERATOR pattern. Operations that might be added or changed in the future, are moved out of the original class into **Iterator**. In Figure 5, four operations were selected as the subjects of potencial modifications. *allChildren()* provides the access to all subnodes sequentially, *visibleObjects()* selects actually visible objects, *rayIntersection()* returns the candidates for intersection with the ray and *collidingObjects()* finds the candidates for collision. Specialized group nodes instantiate their own iterators. Since the iterator knows the inner structure of its instantiator, it can efficiently implement the desired functions.

With the iterator-based solution, adding a new scene operation leads to a reconstruction of iterator interfaces, but it is not necessary to modify the scene graph itself. Moreover, number of spatial data structures and thus number of iterators is limited and the interface modifications would be effortless. Adding a new spatial data structure can be easily done by defining a specialized SDS and it's iterator implementing traversal algorithms.

The interface of a group node associated with the spatial data structure, is also significantly simplified. Instead of rewriting all

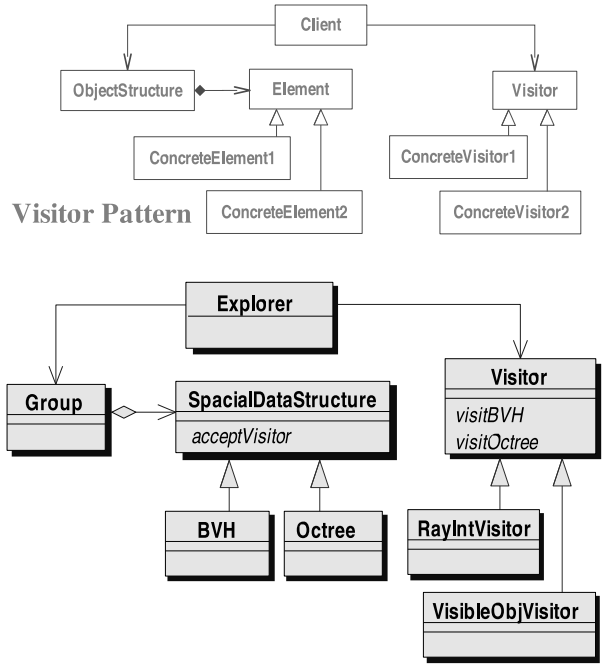

Figure 6: Ramification using VISITOR pattern

scene operations it is sufficient to have a single method only, *traverseSubnodes()*, that instantiates an iterator of included SDS and returns it to the client's (explorer's) disposal. The separation of traversal methods from scene graph nodes therefore keeps a narrow interface of scene graph classes and significantly improves the scene graph maintenance.

4.4 Visitor-based Traversal

Another design pattern applicable to the inspection of a set of stored objects is VISITOR. In general, visitor performs a specific operation on stored object based on data it has inspected. In traditional agent-based encapsulation of scene, a visitor could implement the operation over a scene graph and combine traversing strategy with the operations over virtual objects (e.g. during visualization). In case of explorer-based architecture visitor is more specialized. An explorer is responsible for the entire strategy of traversal, but it can use a specialized visitor to traverse concrete spatial data structure hidden inside group nodes. In this case, the visitor controlled by explorer realizes local searching strategy for a single SDS.

A group node contains a set of spatial data structures (mostly just a single structure), see Figure 6. If a specific visitor is applied to a spatial data structure the visitor searches the structure and remembers those objects that are suitable for a performed operation. These objects are then available for the visitor's instantiator, typically explorer, which can continue the traversal of those nodes only that have a chance to succeed.

The disadvantage of this solution is that adding a new spatial data structure enforces the definition of a new method in the **Visitor** class. The method has to implement a traversal algorithm for specific spatial ordering and it must be re-implemented in all existing visitors. Appending a new data structure implies changes to all visitors. Hence, creating new ordering structures is complicated.

On the other hand, adding a new operation is simple: a new visitor is defined and then an efficient searching strategy for all existing spatial data structures can be performed.

	ITERATOR	VISITOR
New spatial data structure	easy	difficult
New operation over scene	difficult	easy

Table 1: Properties of design patterns in context of scene graph

4.5 Visitor vs. Iterator Discussion

Both the ITERATOR and the VISITOR patterns can be used for inspection of stored components. The question of which solution is better, remains opened. In order to compare them, we select the criteria related to their usage.

Th first criterion is the speed of traversal. Because both visitor and iterator can exploit the knowledge of the internal representation of a specific spatial data structure, they should be equally fast.

Another important issue is the extensibility of proposed models and their maintenance. Table 1 summarizes the most frequent requirements on scene graph with respect to applicable design patterns. In the case of iterator-based traversal, an implementation of a new spatial data structure requires only the implementation of a new iterator. However, the definition of a new operation induces changes in (not many) existing iterators. On the contrary, visitor-based traversal allows new operation to be defined easily, as it requires a new visitor only. In this case, the implementation of a new spatial data structure leads to the changes in all existing visitors. Neither concept is therefore ideal and concrete selection depends on concrete preferences.

5 Scene Graph Attributes

So far we have discussed the scene graph traversal and the relationship to spatial ordering structures. Now we focus on the properties of virtual objects. The motivation is this: Virtual scene contains a set of virtual objects with a variable characteristic. For example, the description of a drawable shape consists of a geometry and material characteristics. A light source can be described by directionality, intensity and color. Each property could have more variants. For instance, geometry can be defined analytically or as a polygonal mesh. Material can be described by color, texture and coefficients of a shading model. The implementation of virtual objects has to be flexible enough to allow the maintenance of a varying number of different properties with variable interfaces. We refer to these properties as *scene graph attributes*. The definition of attributes is required not only for the leaves of a scene graph, but also for the group and inner nodes. The example of an attribute, which is common to any node, is a transformation. It can be included in a group node, as well as in a leave.

The different nature of scene graph attributes and the need to manage and inspect them in a uniform manner, lead to contradictory requirements. One possible solution is to employ the VISITOR pattern again, Figure 7.

The behaviour of any scene graph node can be specified by a set of scene graph attributes. An internal representation of the set (in the figure referred to as the **ObjStructure**) is not important. Since most of the nodes contain only a few attributes, they can be stored in a list.

Attributes are defined in a separate class hierarchy. Concrete specialization describes a specific object's property and it usually has a special interface which is different from other classes in the hierarchy. Concrete attributes correspond to **ConcreteElements** of the original VISITOR pattern, Figure 6.

Visitors are defined for every attribute. A client, typically explorer, instantiates specific visitor, for example a visitor inspecting the geometry of shape. Then it tries to apply the visitor to

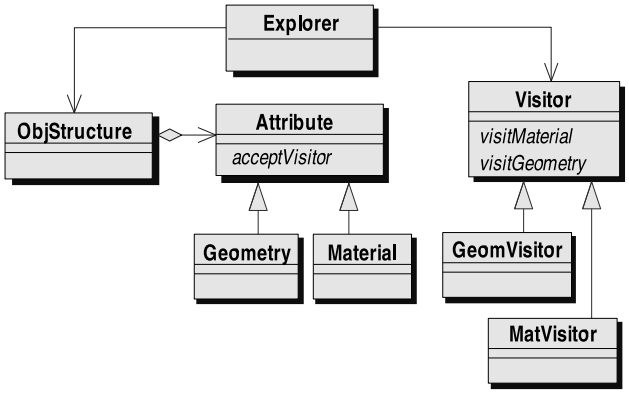

Figure 7: Inspection of attributes using VISITOR pattern

all attributes of the scene graph node, invoking the *acceptVisitor()* method with the visitor as argument. If the visitor succeeds in accessing of the attribute, it changes it's state accordingly; otherwise the retrieval fails. Visitor succeeds with compatible attributes only. Because a client instantiates the visitor, the client also knows which methods of the visitor it has to use for the state inspection. Client does not need to know the interface of attributes, but only that of the employed visitor.

In general case, the VISITOR pattern has one important restriction already discussed in visitor-based ramification: It is quite complicated to append a new specific element. For a scene graph, it would be complex to add a new attribute. This is because the new specific element leads to the definition of a new abstract method at the top-level **Visitor** class. The new abstract method should be properly implemented in all specialized visitors. Fortunately, in the concept of explorers, visitor-based access to the scene graph attributes is needed only in special cases. The visitor serves as a tool for getting the state of specific attribute after the searching stage, rather than to accomplish the operation over attributes. When creating a new attribute we can define the respective *visitNewAttribute()* method in a generic **Visitor** class which is empty by default. The method can be implemented only in the newly defined visitor. Whenever the visitor with such an empty function is applied to the unfamiliar attribute, nothing happens and the inner state of the visitor remains unchanged. This is correct behaviour because the client requires only the state of those attributes that are compatible and understandable to the visitor. An empty result means that this kind of property is not present in the scene graph object. We can therefore keep the interface of old visitors unchanged, which makes the definition of new attributes easy.

While it is possible, for the scene graph ramification, to use both visitor and iterator, for node attributes the concept of iterators is not applicable. The reason is that scene graph attributes have variable interfaces, but an iterator requires a uniform interface of stored objects.

6 Overall Decomposition of Scene

Figure 8 shows the overall decomposition of scene graph based on the principles discussed in previous paragraphs. The software architecture is composed of three major blocks highlighted by hatching. Each rectangle contains one coherent subpart of the decomposition.

The upper right rectangle borders the basic scene graph with scene graph objects. Attributes of scene graph nodes are stored in a list and inspected using visitors (the left rectangle). Branching in group nodes can use various spatial data structures. A specific spatial data structure is appended to the group node by prototyping. Traversal operations are moved from spatial data structures to their iterators for simple maintenance (the bottom right rectangle). We use the iterator-based solution for a scene graph ramification.

As discussed above, access to scene is dealt with specialized explorers. During a traversal, the explorer invokes two kinds of operations:

1. If a current node is a group node designated for the tree ramification then the explorer gets an iterator from the node, sets traversal conditions invoking proper operation, e.g. intersection with ray, and finally it retrieves the list of children satisfying the conditions. These children can be further inspected or traversed down.

2. At an arbitrary level of the traversal, having an arbitrary scene graph node, the explorer can inspect its attributes (transformation matrix, geometry, material, etc.). To do that, explorer instantiates visitor, which gains access to the attributes of the actual scene graph node. If the node object contains the required attribute then the state of the visitor is properly changed and the explorer can read the state.

7 Experimental Library

Extensible Scene Graph, ESG, has been proposed as the part of the *Generic Rendering Architecture, GRA* [Ošlejšek and Sochor 2003] which allows implementation and easy switch between several rendering systems working with shared scene. The systems include various illumination methods, from local illumination techniques to global illumination algorithms. Examples in Figure 9 show three scenes rendered with different techniques: wire-frame model, classical ray tracing (mountains behind crystalic spheres) and teapot scene (soft shadows, various materials, BRDFs and textures).

Although the figures show different scenes, all techniques operate on the same scene decomposition, the extensible scene graph. ESG supports description of virtual objects and their attributes that are required by these very different rendering algorithms. Implementation of ESG corresponds to the diagram in Figure 8. Scene graph attributes are maintained using visitors. Scene graph ramification uses a variant with iterators. Spatial data structures are associated with group nodes. Access to the scene is mediated by various explorers. Some basic shapes and lights are directly implemented in the library (cube, sphere, triangular mesh, point and spot light, etc.) as well. The spatial data structures actually used are the bounding volume hierarchies employing either bounding spheres, axis-aligned bounding boxes or k-DOPs (discrete-orientation polytopes) [Zikan and Konecny 1997; Klosowski et al. 1998], namely 14-DOP (Figure 10).

8 Conclusions and Future Work

In this paper, we described an extensible decomposition of the object-based implementation of a 3D scene. The decomposition introduces the concept of explorers, which is a bit lower-level then usual solutions using visitors. Design patterns principles were used in order to achieve the required features, such as the flexibility and reusability of graphics applications. We have discussed the applicability of several models under different constraints. We focused on an efficient scene graph traversal and on the inspection of scene graph attributes. We omitted other aspects of rendering and manipulation within a scene such as the deposition of light energy on surfaces when solving global illumination, or the dynamic changes

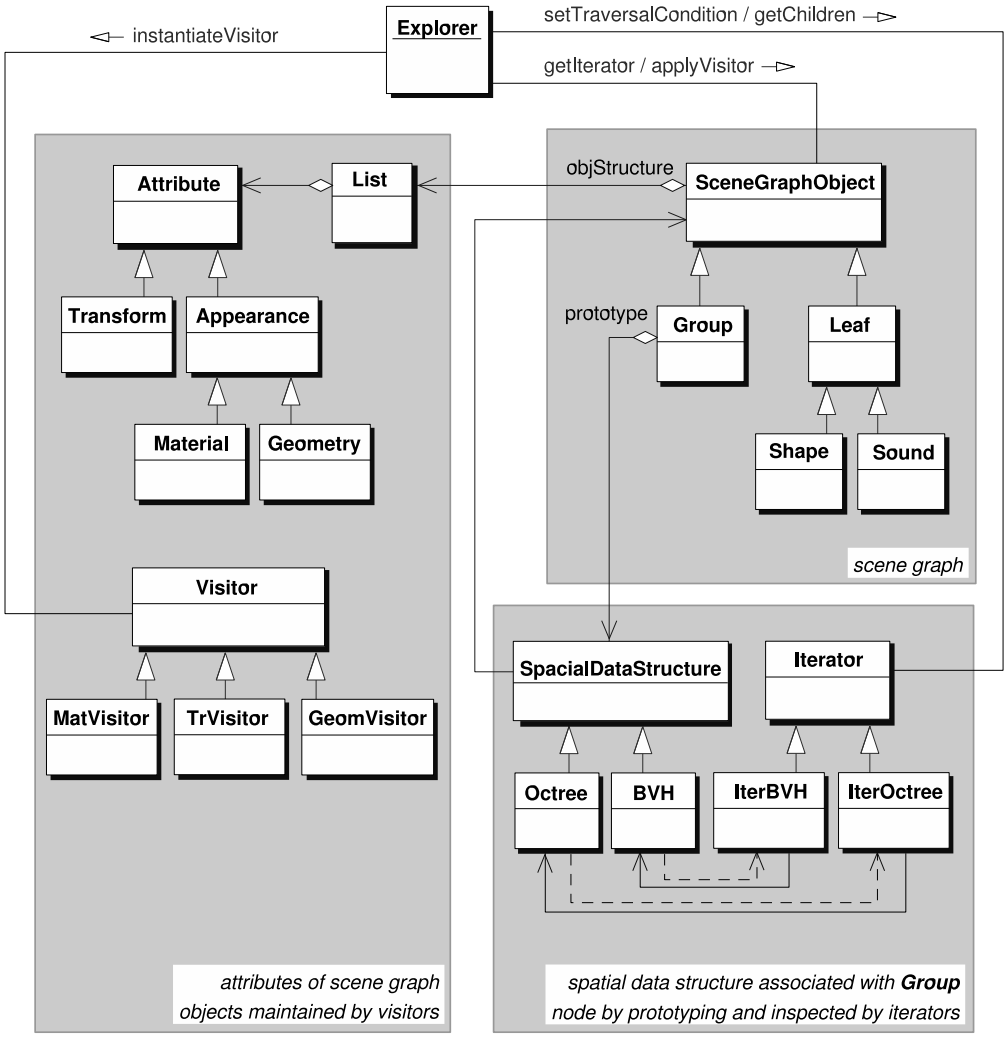

Figure 8: Scene graph based on patterns

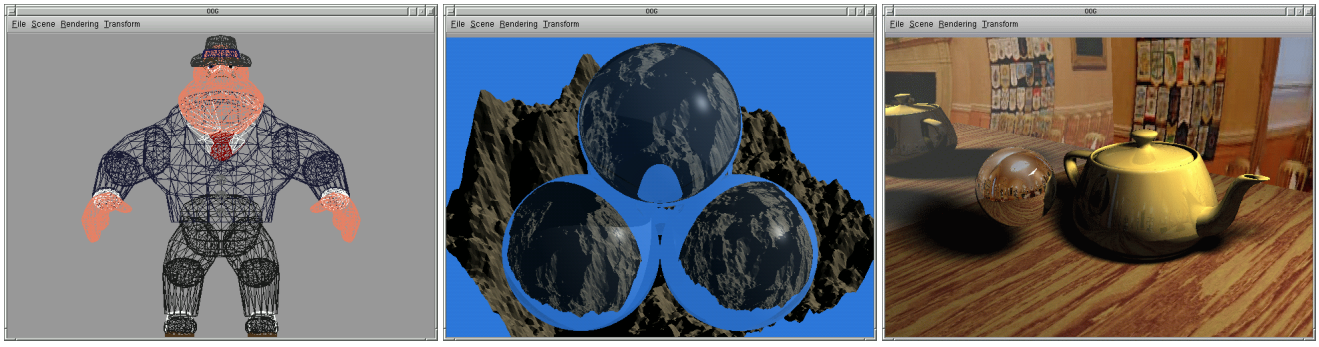

Figure 9: GRA: Wire-frame model (left), Whitted ray tracing (middle) and Monte Carlo ray tracing (right)

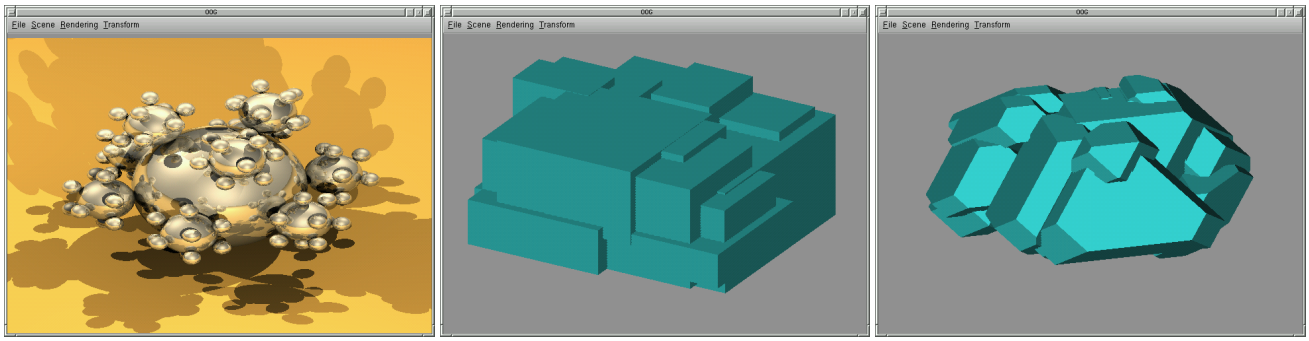

Figure 10: ESG: Spatial data structures: Original ”sphereflake” scene (left), axis-aligned bounding boxes (middle) and 14-DOPs (right)

of a scene invoked by user interactions. The applicability of various design patterns in this context will be verified. The coherency-based traversal of a scene and the traversal with explorers using maps is investigated as well. Our goal is to accelerate and unify the development of complex distributed solutions, using a thorough analysis of computer graphics rendering architectures and a precise definition of interfaces. .

9 Acknowledgements

This work has been supported by the Ministry of Education, Czech Republic, Contract No. MSM143300003 and by Faculty of Informatics Masaryk University Brno, Czech Republic.

References

AMES, A. L., NADEAU, D. R., AND MORELAND, J. L. 1997. *VRML 2.0 Sourcebook*. John Wiley & Sons.

BARRILLEAUX, J. 2001. *3D User Interfaces with Java 3D*. Manning.

BOOCH, G., JACOBSON, I., RUMBAUGH, J., AND RUMBAUGH, J. 1998. *The Unified Modeling Language User Guide*. Addison-Wesley, September.

BUSCHMANN, F. 1996. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons.

COAD, P. 1997. *Object models : strategies, patterns and applications*. Yourdon Press, Upper Saddle River.

DÖLLNER, J., AND HINRICHS, K. 2002. A generic rendering system. *IEEE Trans. Visualization & CG 8*, 2, 99–118.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley.

HARMON, P., AND WATSON, M. 1997. *Understanding UML: The Developer's Guide*. Morgan Kaufmann, October.

KAJIYA, J. T. 1986. The rendering equation. In *ACM Computer Graphics (SIGGRAPH'86)*, 143–150.

KLOSOWSKI, J. T., HELD, M., MITCHELL, J. S. B., SOWIZRAL, H., AND ZIKAN, K. 1998. Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Trans. Vis. Comput. Graph. 4*, 1, 21–36.

LEE, L.-Q., SIEK, J. G., AND LUMSDAINE, A. 1999. The generic graph component library. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 399–414.

NIEMANN, H. 1981. *Pattern Analysis*. Springer-Verlag, Berlin.

Open scene graph. http://openscenegraph.sourceforge.net/.

Opensg. http://www.opensg.org/.

OŠLEJŠEK, R., AND SOCHOR, J. 2003. Generic graphics architecture. In *Theory and Practice of Computer Graphics*, IEEE Computer Society, 105–112.

REINERS, D., VOSS, G., AND BEHR, J. 2002. Opensg - basic concepts. In *OpenSG Symposium*.

REINERS, D. 2002. A flexible and extensible traversal framework for scenegraph systems. In *OpenSG Symposium*.

RUMBAUGH, J., JACOBSON, I., AND BOOCH, G. 1998. *The Unified Modeling Language Reference Manual*. Addison-Wesley, December.

SCHROEDER, W., MARTIN, K., AND LORENSEN, B. 1996. *The Visualization Toolkit : An Object-Oriented Approach to 3D Graphics*. Prentice-Hall.

SOWIZRAL, H., RUSHFORTH, K., AND DEERING, M. 1997. *The Java 3D API Specification*. Addison-Wesley.

VLISSIDES, J. 1998. *Pattern hatching : design pattens applied*. Addison-Wesley.

ZIKAN, K., AND KONECNY, P. 1997. Lower bound of distance in 3d. Tech. Rep. FIMU-RS-97-01, Faculty of Informatics, Masaryk University Brno, Czech Republic, January.