POSTER: Virtual Scene as a Software Component

Radek Ošlejšek
Masaryk University
Botanicka 68a
602 00, Brno, Czech Republic
oslejsek@fi.muni.cz

ABSTRACT

Graphics systems use many advanced techniques that enable to model and visualize a virtual scene with varying level of realism. Unfortunately, rendering algorithms significantly differ in the way how they process a virtual scene. Concrete implementations therefore usually lead to monolithic solutions. In this paper we propose the concept of a component-based scene graph, i.e. an independent scene graph, which can be used by many rendering strategies simultaneously and, moreover, which can be easily replaced with another implementation.

Keywords: Computer graphics, scene graph, software component.

1 INTRODUCTION

Nowadays, computer graphics offers a huge collection of rendering algorithms. They differ in the speed, in the quality of produced images and, unfortunately, also in the way how they handle and process a virtual scene. This different nature of rendering algorithms poses great difficulty in developing a unified rendering architecture, i.e. the architecture, which is able to handle a wide variety of rendering techniques via just a single unified interface. In spite of the difficulties, there exist several experimental architectures, e.g. those in [Fel95, SS95, DH02, OS03]. These architectures attempt to integrate more illumination strategies into a single unified system.

All rendering algorithms share one common concept called *scene graph* – a tree-based container of virtual object. Well-designed scene graph is the basic building block of any generic rendering architecture. Precisely proposed scene graphs can be found in [Opea, Opeb, Rei02, OS05]. Many of them are based on the VISITOR design pattern [GHJV95, Bus96], which enables to manage the scene traversal and inspection comfortably.

Visitors represent an individual operations over the scene, e.g. ray-intersection detection, shading operation, etc., with high level of encapsulation. Once the visitor (operation) is applied to the root node of a scene graph then the operation is automatically applied to all subnodes and thus to the whole scene. On the other hand, this high level of encapsulation means that the visitors are very tightly interconnected with the scene graph. They have to know the implementation details of the scene graph structure as well as the implementation of individual graphical objects. Moreover, the concept of visitors makes it difficult to perform changes in existing scene graph, e.g. extending the scene graph with a new type of graphical objects, features, etc.

A rendering system, even a generic, using a visitor-based scene graph therefore has to adapt itself to concrete scene graph implementation and the scene graph becomes the inseparable part of the rendering system. Moreover, the scene graph usually has to run on the same computer as the rendering system just due to the high encapsulation and interconnection.

Our goal is to make a scene graph as an independent software component [CD04]. Behaviour of a software component is precisely defined by the interface. On the other hand, internal implementation of the behaviour is not limited. Having a scene graph as a software component could bring many advantages. It could be easy to employ many heterogeneous rendering algorithms on the same scene just because they can share the common interface. Internal implementation of the scene could be easily changed without the impact to the existing algorithms. A component-based scene could run on dedicated computer and could be shared by more applications, e.g. in collaborative environments of virtual reality.

Precise definition of a unified interface is the key for successful proposal of component-based scene graph. Simultaneously, it is very serious challenge and difficult task. Unfortunately, the high level of encapsulation of visitors disqualify them from their direct usage because components have to be more autonomous and independent of their implementation.

2 SCENE GRAPH COMPONENT

Any operation over the scene has to perform two basic tasks. It has to traverse the scene graph and inspect individual nodes. The visitor-based solution do both these tasks in a single step, as discussed above. To find interfaces suitable for software component we have to resign to the automatic application of scene operations inside the entire scene graph container. Instead, we have to separate the traversal and nodes inspection tasks and thus allow the invoker to gradually traverse the scene graph tree node by node and to inspect nodes individually.

Separation of the traversal from nodes inspection slightly breaks the strict encapsulation but enables us to manage scene operations outside of the scene. It allows to build different rendering strategies over the single unified scene graph, because the strategy can fully control behaviour of its operations.

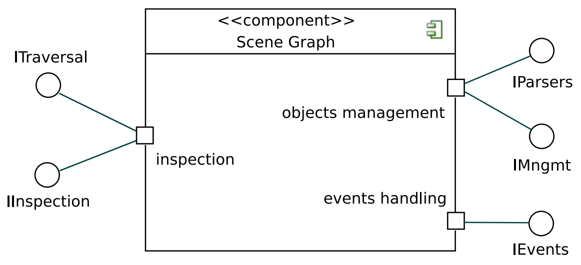

Figure 1: Scene graph component

Fig. 1 shows the basic component diagram of the scene graph. The *ITraversal* and *IInspection* interfaces are the key interfaces used for the scene traversal and nodes inspection respectively and they are discussed in the rest of this paper. However, a practically usable scene has to provide a lot of another useful operations, e.g. those related to the scene graph management, events management, etc. They are only suggested in the diagram for completeness.

3 SCENE GRAPH TRAVERSAL

Tree structure of scene graphs enables us do define a simple but unified interface for the traversal. A client traversing the scene has available a pointer to the scene graph tree as well as a private stacks and pipes enabling to store and restore the pointer. The client can instantiate various stacks and pipes, use them to store and restore position in the tree and thus implement various traversal strategies, e.g. depth-first or breadth-first search. Because every client has its own pointer and the set of stacks and pipes, there can be many clients traversing the scene simultaneously.

4 INSPECTION OF NODES

Scene graph nodes present a wide range of graphical information, e.g. description of shapes, material, transformations in space, etc. To handle all these miscellaneous properties in a uniform way it is necessary to classify them in smaller groups and to define specific interfaces for these groups.

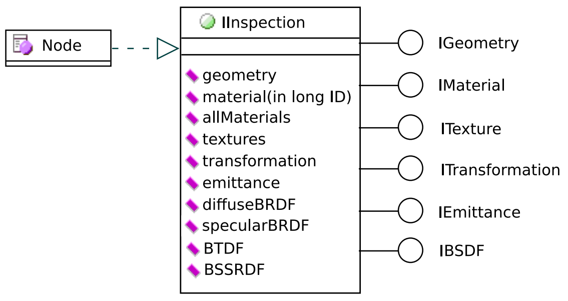

Figure 2: Inspection interface

Actually we have defined 6 groups of properties: (1) *geometry* defines shape of a virtual object in various ways, e.g. as a polygonal mesh, analytic surface, etc. (2) *material* determines coefficients for energy reflection and refraction, e.g. RGB color, transparency, etc. (3) *emittance* is necessary to model sources of energy. (4) *BSDFs, Bidirectional Scattering Distribution Functions*, determine what portion of incoming energy is reflected back to the scene with respect to incoming and outgoing directions and what portion is transmitted through translucent objects. (5) *transformations* are represented by 4x4 transformation matrices and allows manipulation with groups of objects in the space. Uniform management of matrices is clear and well-known and thus uninteresting. We therefore omit the 3D transformations from detail discussion. (6) *textures* are a common way to define color patterns on a surface. This property is also omitted from detail discussion.

Our complete inspection interface consists of two levels, as shown in Fig. 2. The *IInspection* presents the coarse-grained interface related to the actual node of a scene graph. Any scene graph node has to implement this interface. The *IInspection* itself is very simple. It just contains operations related to above discussed categories, each operation returning a relevant fine-grained interface of the category. An invoker selects required inspection category first, calls appropriate operation and then exploits returned fine-grained interface for final inspection. If some property is not present in the node, e.g. the node has no texture defined, then the fine-grained interface retrieval fails and the invoker continues with another inspections. In what follows we discuss individual fine-grained interfaces.

4.1 Geometry

Geometry represents shape of a surface. Computer graphics uses various kinds of geometry description, e.g. analytical surfaces, triangle meshes, etc. Our aim is to not restrict possible implementations of geometry. The unified interface in Fig. 3 therefore consists of only a general operations, which allow to “touch” the surface in a sense and to retrieve the necessary information about the shape. The set of operations cover a ray-intersection inspection, random sampling, basic volume and spatial information retrieval as well as collision detection.

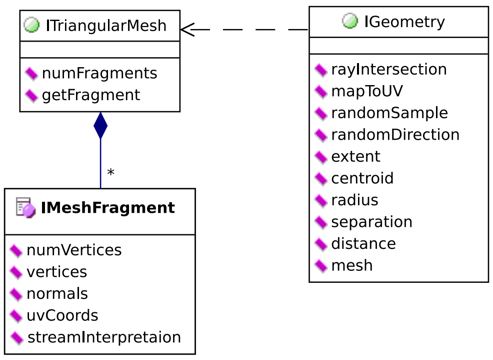

Figure 3: Geometry interface

Many existing graphical algorithms are based on polygonal surfaces. Any kind of geometry should be therefore transformable to this approximate description. The *mesh()* operation instantiates a triangular mesh, represented by the *ITringularMesh* interface in Fig. 3. Unified inspection of this mesh is ensured by using the mechanism known from the OpenGL [WNDS99], for instance. Externally, the mesh is composed of a stream of vertices, normals and mapping coordinates. Streams are interpreted as a *triangle strip*, *triangle fun*, etc. Thus, the concrete interpretation enables to reconstruct original triangles from the streams.

4.2 Material

Material characteristics are in the computer graphics expressed as n-tuples of real numbers, e.g. RGB colors, transparency, roughness, etc. But because different clients and algorithms can use different names even for the same material properties, then our unified model is based on the name-service concept similar to the Domain Name Service (DNS) of the Internet. Client can register various materials under various names and aliases at runtime and thus share these names with another clients. Concrete instances of materials inside a scene graph are then easily identifiable by their common natural names in the unified way.

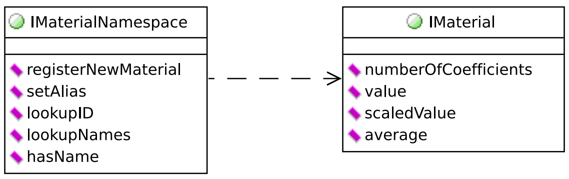

Figure 4: Material interface

IMaterial interface in Fig. 4 represents an n-tuple of concrete material coefficients, while the *IMaterialNamespace* interface provides the names registration.

4.3 Emittance of Energy

Light sources pose very important but very complicated part of virtual scene. A light source can by understood as an object emitting energy. Description of the emittance can vary, but we can find several common principles that enable us to define emission in a uniform way.

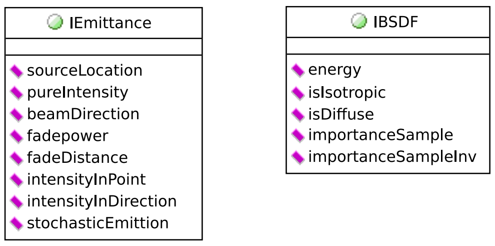

Figure 5: Emittance (left) and BSDF (right)

The basic characteristics of any emmitance consists of space location, beam direction, initial intensity (color) and attenuation. Unified *IEmittance* interface in Fig. 5 therefore contains appropriate inspection methods. Attenuation is controlled by two factors. Fade distance is used to specify the distance at which the full light intensity arrives, while fade power determines the falloff rate beyond the fade distance.

intensityInPoint() and *intensityInDirection()* methods computes concrete amount of energy in a space point. *stochasticEmission()* operation casts a ray stochastically with respect to the properties of the light source.

4.4 BSDFs

BSDF, Bidirectional Scattering Distribution Function, determines what portion of incoming energy is reflected back to the scene from a reflective surface or what portion is transmitted through a translucent material. There exist three variants of BSDF: BRDF, BSDF and BSSRDF. They differ only in details and thus the BSDF can be taken as their unified interface.

energy() method in Fig. 5 computes the portion of reflected or transmitted energy. *isIsotropic()* function distinguishes isotropic and anisotropic surfaces.

Some BRDFs do not depend on the outgoing direction but distributes the energy omnidirectionally. e.g. Lambertian function. They are called to be *perfectly diffuse*. Many real algorithms exploit this feature to accelerate energy distribution process. The *isDiffuse()* method informs an invoker about this property.

The rest of the *IBSDF* methods is used by stochastic algorithms of energy distribution.

Single virtual object can have assigned all three types of distribution functions. The basic coarse-grained inspection interface *IInspection* therefore contains inspection methods for these three variants of distribution functions. But all these variants share one fine-grained inspection interface *IBSDF*.

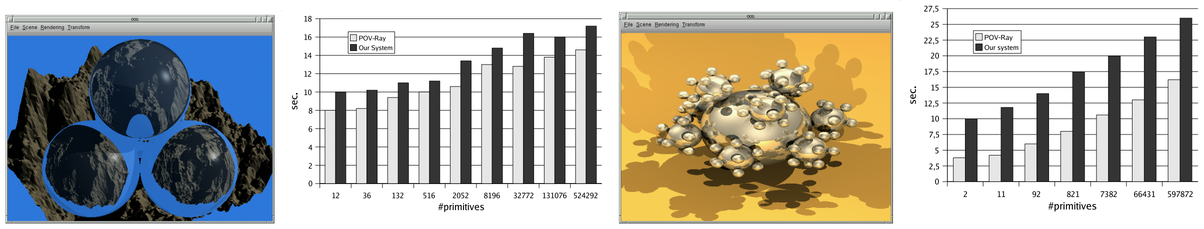

Figure 6: Fractal scenes used for efficiency tests: *Mountains* (left) and *Sphereflake* (right)

5 EXPERIMENTAL RENDERING ARCHITECTURE

To confirm the results of analysis we designed two experimental libraries.

The first library implements scene graph using inspection interface as discussed in this paper. Actually, the library does not represent real software component running under some kind of component system, e.g. CORBA, but it is a standalone C++ library implementing discussed interfaces. This library was developed in order to ensure that proposed concept is practical and functional. Translation of this standalone library into the real CORBA component is in progress.

The second library implements various rendering strategies, e.g. few variants of local illumination, Whitted ray tracing, Monte Carlo ray tracing and photon mapping. This library is used to check that proposed inspection interface of a scene graph is sufficiently general.

The most important result of this project is the existence of the unified component-based scene graph. But practical usage depends mainly on the rendering speed. Although our scene graph is not yet implemented as a real CORBA software component, we performed several preliminary efficiency tests.

Implementation of the local illumination is based on the OpenGL and thus we did not measure any significant decrease of performance in comparison with native OpenGL applications.

Ray tracing algorithm was compared with the POV-Ray system [Tea91]. Fig. 6 shows an overview of tested scenes and the rendering times. Tests were performed on Pentium 4 3.0GHz, 1GB RAM, image resolution 800x600. Results of the tests show a less efficiency of our system. The reason is that the unified inspection interface does not allow direct access into the scene graph and then forbids implementation of various acceleration tricks. Memory requirements are very similar for both the systems.

6 CONCLUSION AND FUTURE WORK

We discussed a unified interface of the scene traversal and inspection. This interface does not restrict implementation of the scene graph, just prescribe necessary inspection operations. On the other hand, proposed inspection operations are sufficiently general for wide range of rendering strategies, from real-time local illumination to photorealistic image synthesis. The interfaces therefore enable to develop a scene as an independent software component, which is very useful mainly in distributed and collaborative environments.

Inspection and traversal interfaces present only a fragment of all required functionality. Another unified scene graph interfaces, e.g. scene creation and maintenance, event handling, etc., have to be proposed.

7 ACKNOWLEDGMENTS

This work has been supported by the Czech Science Foundation under Contract No. GA 201/06/P247 and the Ministry of Education, Youth and Sports of the Czech Republic under the research program LC-06008.

REFERENCES

[Bus96] F. Buschmann. *Pattern-Oriented Software Architecture: A System of Patterns.* John Wiley & Sons, 1996.

[CD04] John Cheesman and John Daniels. *UML Components.* Addison-Wesly, 2004.

[DH02] J. Döllner and K. Hinrichs. A generic rendering system. *IEEE Trans. Visualization & CG*, 8(2):99–118, 2002.

[Fel95] D. W. Fellner. Mrt - an extensible platform for 3d image synthesis. Computer Graphics Lab., Dept. of Computer Science, University of Bonn, Germany, December 1995.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[Opea] Open scene graph. http://openscenegraph.sourceforge.net/.

[Opeb] Opensg. http://www.opensg.org/.

[OS03] Radek Ošlejšek and Jiří Sochor. Generic graphics architecture. In *Theory and Practice of Computer Graphics*, pages 105–112. IEEE Computer Society, June 2003.

[OS05] Radek Ošlejšek and Jiří Sochor. A flexible, low-level scene graph traversal with explorers. In *Spring Conference on Computer Graphics*, pages 194–201. Bratislava : Comenius University, May 2005.

[Rei02] Dirk Reiners. A flexible and extensible traversal framework for scenegraph systems. In *OpenSG Symposium*, 2002.

[SS95] P. Slusallek and H.-P. Seidel. Vision - an architecture for global illumination calculations. *IEEE Trans. Visualization & Computer Graphics*, 1(1), 1995.

[Tea91] POV Team. Persistency of vision ray tracer (pov-ray), version 1.0. Technical report, 1991.

[WNDS99] M. Woo, J. Neider, T. Davis, and D. Shreiner. *OpenGL Programming Guide.* Addison-Wesley, 1999.