FI MU

Faculty of Informatics
Masaryk University Brno

Virtual Scene Designed as a Software Component

by

Radek Ošlejšek

FI MU Report Series **FIMU-RS-2008-11**

 December 2008

Publications in the FI MU Report Series are in general accessible via WWW:

`http://www.fi.muni.cz/reports/`

Further information can be obtained by contacting:

**Faculty of Informatics
Masaryk University
Botanická 68a
602 00 Brno
Czech Republic**

Virtual Scene Designed as a Software Component *

Radek Ošlejšek
Faculty of Informatics, Masaryk University
Botanická 68a, 602 00 Brno
Czech Republic
oslejsek@fi.muni.cz

December 16, 2008

Abstract

Graphics systems use many advanced techniques that enable to model and visualize a virtual scene with varying level of realism. Unfortunately, the huge collection of existing rendering algorithms significantly differ in the way how a virtual scene is processed. Concrete implementations therefore usually lead to monolithic solutions. In this paper we present a component-based virtual scene with unified interface exploitable by many rendering strategies. Moreover, proposed approach does not dictate internal implementation of the scene. One implementation can be therefore reused by many rendering algorithms and, vice versa, the scene can be easily replaced by another implementation, even at runtime.

1 Introduction

Nowadays, computer graphics offers a huge collection of rendering algorithms. They differ in the speed as well as in the quality of produced images. These algorithms come in useful in various domains. Fast but less realistic algorithms are used mainly for runtime visualization, e.g. in computer games and virtual reality, while slow photorealistic

*Short version of this paper has been published in the proceedings of the 16-th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision.

algorithms are required by movie industry for instance. Unfortunately, rendering algorithms differ not only in the quality and speed but also in the way how they handle and process a virtual scene. For example, a local illumination strategies process individual triangles vertex by vertex while photon mapping technique presents a two-stage algorithm shooting photons and rays in the 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 by means of a single unified interface.

Fortunately, all the rendering algorithms share one common concept, so called *scene graph*. A scene graph constitutes a tree-based skeleton of the virtual scene, which is traversed and inspected continually during the rendering process. Virtual scene is therefore a good candidate to be the basic building block of the generic rendering architecture with clear, general and unified interface suitable for various rendering strategies.

Our solution goes one step further. Our decomposition forms independent software component. A software component [2] is a unit of software with high degree of encapsulation. Behaviour of the component is precisely defined by the set of interfaces. Any client using the component depends only on these interfaces and thus it is possible to exchange the component with another compatible implementation even at runtime. Interfaces, their specification and usage therefore play the key role in the component-based software development.

Component-based scene graph has many advantages. As mentioned above, with the carefully selected interfaces it is possible to employ many different rendering strategies, all working with the same scene. Moreover, it is possible to replace one scene with another implementation without the impact on existing rendering algorithms. Component-based scene graph also forms extremely useful concept for distributed environments, e.g. when many graphics applications collaborate on a single shared virtual scene. Software components are highly encapsulated and thus they do not make any difference whether they are invoked directly or through the network. The scene can be therefore easily isolated on the separate remote computer.

To design a component-based scene graph it is necessary to define precise, yet general interfaces, which poses serious challenge and simultaneously difficult task.

2 Previous work

Dissimilarity of rendering algorithms usually leads to the monolithic solutions of rendering systems. In spite of that, there exist several experimental generic systems, e.g. those in [4, 12, 3, 9]. These systems attempt to integrate more illumination strategies into a single unified system. Scene graph forms integral part of these systems.

Well-designed implementations of a scene graph can be found in [7, 8, 11, 10]. Many of these sophisticated scene graph solutions use so called VISITOR design pattern [5, 1] to manage traversal comfortably. Each concrete visitor represents an individual operation, which is applicable to the scene, e.g. ray-intersection detection, shading operation, etc. It is possible to use more operations at the same time. Important advantage of visitors is that it is easy to define a new operation over a scene graph without the necessity to modify the scene graph. Second advantage is a high level of encapsulation, i.e. once the visitor is instantiated and applied to the root node of the scene graph tree, the operation is automatically propagated and applied to the whole scene. That is the visitor performes both the tree traversal and nodes inspection within the single invocation.

On the other hand, the high level of visitors encapsulation means that they are very tightly interconnected with the scene graph. To be able to perform an operation in the whole scene, the visitor has to know details about the scene implementation. However, it usually leads to changes in existing visitors whenever the scene graph implementation is changed. Unfortunately, rendering strategies often requires special properties from the scene graph and thus demands changes in the scene. Therefore, even a generic rendering system that uses the visitor-based scene graph has to adapt itself instead of adapting scene graph to its requirements. The scene graph becomes an inseparable part of the rendering system and replacing the scene graph with another implementation is either impossible or requires nontrivial changes in the rendering system. Moreover, the scene graph usually has to run on the same computer just due to the high encapsulation and interconnection. These reasons disqualify the concept of visitors from the usage in the component-based approach.

3 Scene Graph Component

A scene graph can be understood as a container of virtual objects. Internally, scene graphs store objects in a tree structure. Tree structure is very practical, because it enables to organize the scene efficiently.

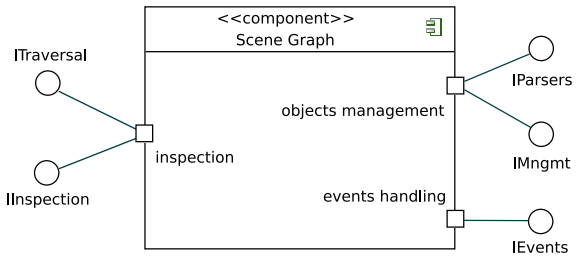

Figure 1: Scene graph component

Any operation working with the scene has to perform two basic tasks. It has to traverse the scene graph and inspect individual nodes. From the point of view of external invoker, the visitor-based solutions do both the tasks in a single step, as discussed above. However, if we want to propose 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 the nodes externally.

Separating these two tasks slightly breaks the strict encapsulation of the visitor-based solution. On the other hand, it provides the rendering strategies with sufficient control of scene operations.

A practically usable scene has to provide a lot of another useful operations, e.g. those related to the management of stored virtual objects, events management, etc. Because these additional functionalities do not affect the rendering directly, they are omitted in this paper. On the contrary, traversal and inspection interfaces are discussed in the rest of this paper in detail.

Component diagram in Figure 1 depicts a basic scene graph component in UML notation. It suggests discussed interfaces as well as several extra functionalities.

4 Unified Scene Graph Traversal

The tree structure of scene graphs enables us do define a simple yet unified interface for its traversal. A client traversing the scene stores a pointer to actual node in the tree. Traversal interface includes methods that move the pointer in the tree, i.e. to children, to parent etc. Moreover, the interface provides the management of stacks and pipes. 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.

In what follows, there is the exact definition of the traversal interface in IDL (Interface Description Language, [6]), as used in our project:

```
interface ITraversal {
    boolean goToRoot();
    boolean goToChild(in long child);
    boolean goToParent();
    boolean goToLeftSibling();
    boolean goToRightSibling();
    long    numChildren();
    long    actDepth();

    long    newStack();
    boolean deleteStack(in long stackID);
    boolean clearStack(in long stackID);
    boolean pushPosition(in long stackID);
    boolean popPosition(in long stackID);

    long    newPipe();
    boolean deletePipe(in long stackID);
    boolean clearPipe(in long stackID);
    boolean storePosition(in long pipeID);
    boolean restorePosition(in long pipeID);
};
```

Meaning of individual methods is clear. This IDL description shows concrete precise definition, which is required for implementation. In the rest of this paper we use a graphical UML representation, which hides unnecessary details and thus keeps clear meaning even a more complicated interfaces.

5 Unified Inspection of Scene Graph Nodes

Scene graph nodes represent a wide range of graphical information, i.e. 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. Actually we have defined 6 groups of properties.

1. *Geometrical properties* define shapes of virtual objects in the form of polygonal surfaces, analytic surface, etc. Geometry is required for example to determine the basic reflection directions of energy during the energy distribution process.

2. *Material* determines how an energy behaves when falls on a surface, i.e. how much of the energy is reflected, refracted, absorbed, etc. Various illumination models use various descriptions of material.

3. *Textures* represent a common way for the definition of color patterns on a surface. Textures often determine the basic appearance of surfaces.

4. *Emittance* is necessary to model sources of light in the scene. Several light source models exist, they differ mainly in the way how the light is spreaded out of the source.

5. *BSDFs - Bidirectional Scattering Distribution Functions* are used by photorealistic rendering techniques. These 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.

6. *Transformations* are represented by 4x4 transformation matrices and allows manipulation with groups of objects in the space, e.g. rotation, translation or scale. Uniform management of matrices is clear and well-known and thus uninteresting. It is therefore omitted from detail discussion.

Complete inspection interface consists of two levels, as shown in Figure 2. The *IInspection* represents the basic coarse-grained interface related to the actual node of the scene graph. Any scene graph node has to implement this interface. The *IInspection* itself is very simple. It just contains operations related to discussed categories, each operation returns a relevant fine-grained interface of the category. An invoker selects required inspection category first, then exploits a fine-grained interface for final inspection. If some property is not present in the node, e.g. the node has no texture defined,

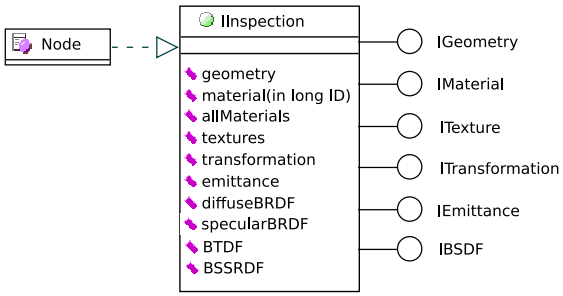

Figure 2: Inspection interface

then the fine-grained interface retrieval fails and the invoker continues with another inspections.

5.1 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 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. Complete class diagram is shown in Figure 3.

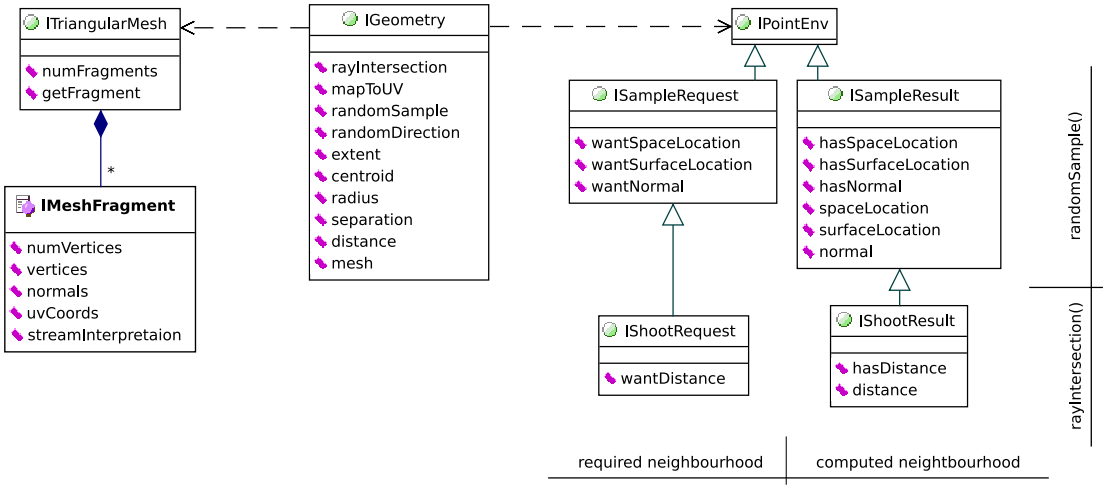

Figure 3: Geometry interface

Many geometric operations handle surface points. However, a surface point represented by a simple 3D vector is usually not sufficient for an invoker. The invoker

typically requires additional information about that point, e.g. normal vector, distance from the viewer, surface location, etc. The *IGeometry* interface therefore uses a general concept of a *surface point neighbourhood*. First, the invoker specifies which kind of information requires. However, it is not always possible to retrieve all the required information from concrete geometry. Geometric operations therefore compute as much as possible and inform the invoker which kind of required information has been really retrieved and which has been omitted. These requests and results form two branches of the *IPointEnv* class hierarchy in Figure 3. While the sample request and result interfaces work with the surface point and its normal vector, the "shooting" interfaces suppose that the surface point represents point of impact and then appends information about the distance from the "shooter".

Having a ray, *rayIntersection()* operation computes point of intersection with the surface. This operation accepts the *IShootRequest* as the input parameter and returns the *IShootResult*.

mapToUV() operation transfers a surface point from the global 3D coordinates into the local 2D coordinates of the surface. This operation is useful mainly for texture mapping. This operation can be implemented in many ways, i.e. direct mapping as well as the mappings through an auxiliary surface.

randomSample() and *randomDirection()* return random point at the surface and random direction from given space point towards the surface respectively. These operations accept the *ISampleRequest* on input and return the *ISampleResult*. Sampling operations are necessary for stochastic algorithms of energy distribution.

radius(), *extent()* and *centroid()* determine approximate shape, size and location of the surface. They return radius of minimal bounding sphere, min-max extent in given direction and point of gravity. These operations are useful mainly for spatial data structures and space searching algorithms.

separation() and *distance()* are used for collision detection of two surfaces. While the *separation()* only checks whether two surfaces penetrate or not, the *distance()* method computes their approximate distance.

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 Figure 3. *ITringularMesh* interface describes the mesh as a set of fragments, each fragment composed of the stream of vertices, stream of their normals, stream of their

local UV coordinates (e.g. mapping coordinates) and the interpretation of that streams. Stream interpretation determines how to build triangles from these individual vertices. Possible interpretations are individual triangles, triangle strip and triangle fan. This unified mechanism is well-known from the OpenGL [14] for instance and allows to reconstruct original polygons.

5.2 Material

Material characteristics are in the computer graphics expressed as a sets of real coefficients. Material coefficients are usually defined by triples, which represent properties of individual color elements, i.e. red, green and blue (RGB) colors. But also single coefficients or general n-tuples can be used. Material coefficient directly affects shading and visualization models.

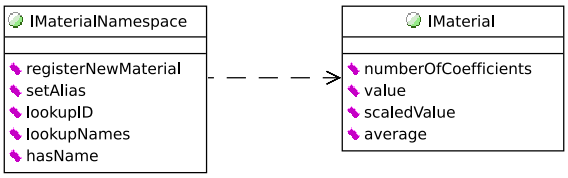

Figure 4: Material interface

Our unified interface handles materials as a named n-tuples of real numbers. *IMaterial* interface in Figure 4 represents an n-tuple of coefficients and enables to retrieve individual values as real numbers via the *value()* operation. Values scaled to given range are returned by the *scaledValue()*, average value of the n-tuple by the *average()* operation.

Because different clients and algorithms can use different names even for the same material properties, we use the name-service concept similar to the Domain Name Service (DNS), the well-known technology of the Internet. The *IMaterialNamespace* interface provides registration of material coefficients at runtime. User can register various n-tuples under various names and aliases. The interface translates that names into unique IDs and vice versa. All users share this database of material names and IDs. Common names, e.g. specular, diffuse and ambient coefficients, reflectance, transparency, etc., could be predefined by the implementation of the software component.

5.3 Textures

Textures can represent various data, e.g. color texture, bump texture, mip-mapping, etc. An invoker should have available the row data of the texture as well as the interpretation information of the data. The *ITexture* interface in Figure 5 therefore informs the invoker about the size, mapping properties, etc. We omit technical details here and limit oneself to the statement that the data available to the invoker are very similar to those available through the *glGetTexParameter()* and *glGetTexLevelParameter()* OpenGL functions.

5.4 Emittance of Energy

Light sources pose very important but very complicated part of virtual scene. 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.

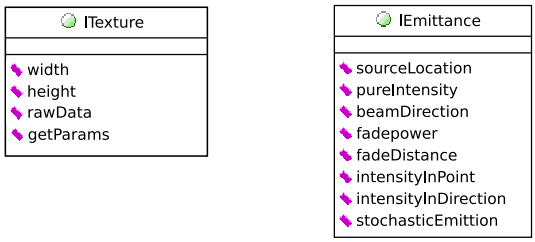

Figure 5: Simplified texture (left) and emittance (right) interfaces

Emittance typically consists of exact location, beam direction, and initial intensity (color). *IEmittance* interface in Figure 4 contains operations for inspection of these basic properties. The *sourceLocation()* operation returns 3D location of the light source, the *pureIntensity()* returns pure color of the energy and the *beamDirection()* returns main direction of the beam. Source location and beam direction can be undefined in some types of light sources. For example, a point light source radiates omnidirectionally and thus has no beam direction while a parallel energy simulating light of the Sun has no exact location.

Another important property of a light is attenuation with respect to distance from the source. We model attenuation by two factors. Fade distance is used to specify the distance at which the full light intensity arrives. Attenuation beyond the fade distance

is described by fade power, which determines the falloff rate. For example, linear or quadratic falloff can be used by setting fade power to 1 or 2 respectively. This attenuation formula seems to be sufficiently general to model usual shapes of attenuation curves. Operations *fadePower()* and *fadeDistance()* are used to inspect these parameters.

While previous operations describe the basic characteristics of a light source, the *intensityInPoint()* and *intensityInDirection()* methods computes concrete amount of energy transmitted from the light source into given space point. These two operations differ only in the way how the space point is defined. The first version works with target 3D point while the second version with direction and distance from the energy source. Energy is attenuated with respect to distance and attenuation characteristics of the light source.

The *stochasticEmission()* operation casts a ray stochastically with respect to the properties of the light source. Stochastic emission is often used by the algorithms of photo-realistic rendering, e.g. photon mapping.

Emission interface defined in this way supports many types of light sources including point lights, parallel lights and spot lights. All these lights remain uniformly manageable by clients.

5.5 Bidirectional Scattering Distribution Functions

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.

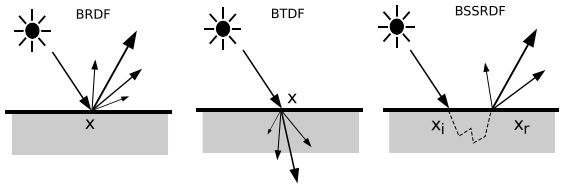

Figure 6: Difference between distribution functions

There exist three variants of BSDF. First variant is referred to as a BRDF, Bidirectional Reflectance Distribution Function. It describes reflectance of glossy materials. Second variant is BTDF, Bidirectional Transmittance Distribution Function, which is similar but describes transmittance through translucent materials. Third variant is BSSRDF, Bidirectional Surface Scattering Reflectance Distribution Function. While the BRDF and

BTDF work with a single point at a surface, i.e. incoming and outgoing energy are related to the same point, BSSRDF is able to handle more natural behaviour. It takes into consideration the fact than the energy impacts the surface at one point but radiates at another. This model enables to handle subsurface scattering for surfaces like milk, skin or marble. Figure 6 depicts difference between these variants.

Single virtual object can have assigned all three distribution functions. For example, a glass object has to have defined BRDF for reflectivity as well as BTDF for transparency. Moreover, it could have defined BSSRDF, which can be used instead of BRDF for more precise reflection calculations. The basic coarse-grained *IInspection* interface therefore contains inspection methods for these three variants of distribution functions. Nevertheless, all these variants share one fine-grained inspection interface *IBSDF* as shown in Figure 7.

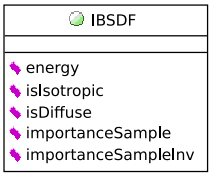

Figure 7: Unified interface for BSDF

The most important operation of the *IBSDF* interface is the *energy()*. It takes incoming energy direction, viewer direction, point of impact and outgoing point and returns portion of reflected/transmitted energy. BRDF and BTDF just ignore the outgoing point.

Important property of any BSDF is also an *isotropy*. Isotropic BSDFs are independent on rotation of the surface around its normal, while anisotropic functions change during the rotation, even though the incoming and outgoing vectors keep unchanged. Anisotropic surfaces are brushed metal or compact disc, for instance. The method *isIsotropic()* informs an invoker about this BSDF behaviour.

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 of the interface informs an invoker about this property. Separation of BRDF into its diffuse and specular part is quite common and useful for many global illumination algorithms. The *IInspection* interface therefore contains two separate *diffuseBRDF()* and *specularBRDF()* inspection operations.

The *importanceSample()* and *importanceSampleInv()* operations are required by stochastic algorithms of energy distribution. The *importanceSample()* takes preferred outgoing direction and two generating random numbers from the range $[0, 1)$ and returns direction with respect to the scattering characteristics of the BSDF. The *importanceSampleInv()* is the inverse function, i.e. it takes outgoing direction and returns its generating "random" numbers.

6 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. 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. This library is used to check that proposed inspection interface of a scene graph is sufficient for various types of rendering algorithms. Actually implemented rendering strategies include a few variants of local illumination, Whitted ray tracing, Monte Carlo ray tracing and photon mapping. Examples are show in Figure 8. These strategies cover the whole range of rendering approaches and thus confirm validity of proposed interface.

The most important result of this project is 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 efficiency tests with actual standalone implementation.

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

The ray tracing algorithm was compared with the POV-Ray system [13]. Tests were performed on fractal scenes that enables to change the number of primitives in the scene easily. Both the systems had set a similar parameters of the algorithm. Figure 9 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

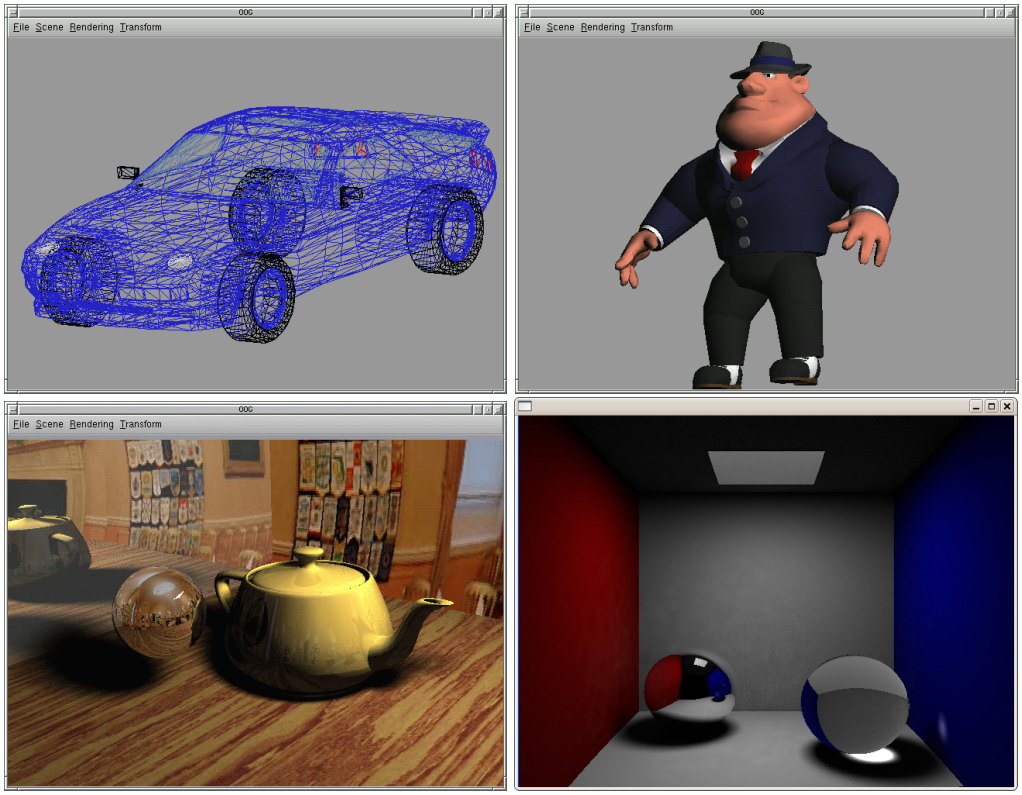

Figure 8: Wire-frame model (upper-left), local illumination (upper-right), Monte Carlo ray tracing (lower-left) and Photon mapping (lower-right)

of our system in comparison with the POV-Ray. The reason is that the proposed 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.

We did not perform a serious tests for photon mapping algorithm so far, but the preliminary experiments show an acceptable performance for this algorithm too.

7 Conclusion and Future Work

We discussed a unified interfaces of the scene traversal and inspection. These interfaces do 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

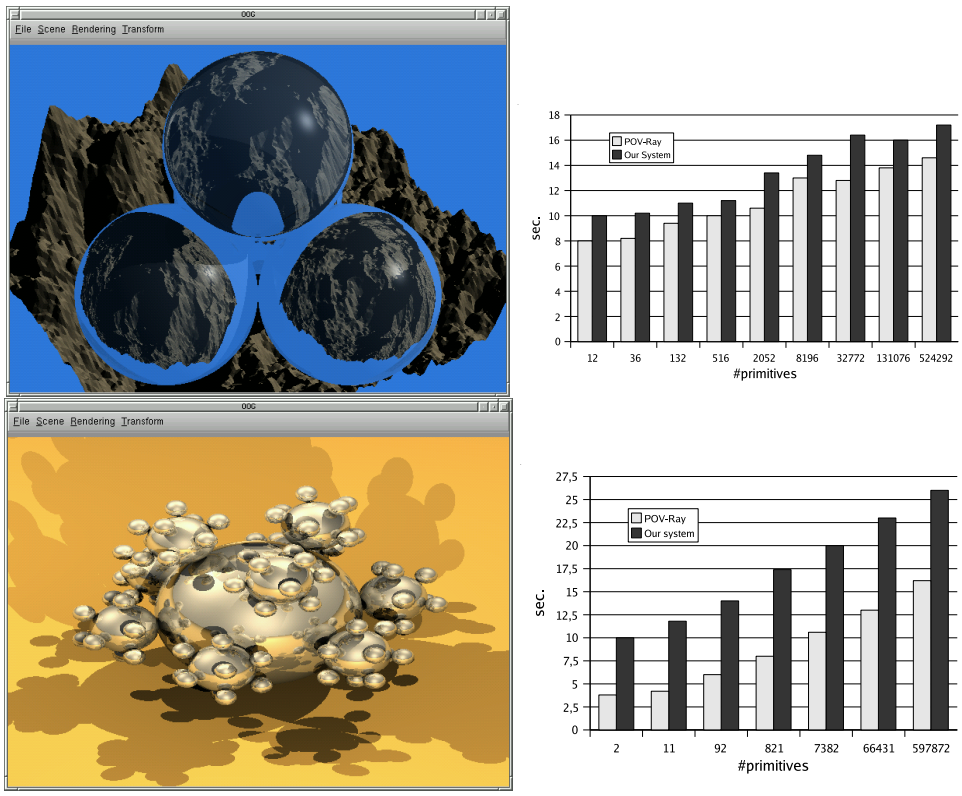

Figure 9: Fractal scenes used for efficiency tests: *Mountains* (left) and *Sphereflake* (right)

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.

Another interesting research area include development of a generic rendering architecture, i.e. architecture enabling to handle and change wide variety of rendering algorithms at runtime.

Although we performed some preliminary tests, is it necessary to check seriously how much the component technology decrease performance of the rendering applications. Acceptable loss of performance have to be balanced with advantages of component technology.

8 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

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

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

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

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

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

[6] Martin Gudgin. *Essential IDL: Interface Design for COM*. Addison-Wesley Professional, 2000.

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

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

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

[10] 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.

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

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

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

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