A Generalized Scene Graph

Jürgen Döllner Klaus Hinrichs

Institut für Informatik, Universität Münster
Einsteinstr. 62, 48149 Münster, Germany
Email: {dollner,khh}@uni-muenster.de

Abstract

Scene graphs are fundamental data structures for hierarchical scene modeling. The *generalized scene graph* overcomes various limitations of current scene graph architectures such as support for different 3D rendering systems, integration of multi-pass rendering, and declarative modeling of scenes. The main idea is to separate scene specification from scene evaluation. To specify scenes, scene graph nodes are arranged and equipped with rendering objects, e.g., shapes, attributes, and algorithms. To evaluate scenes, the contents of scene graphs nodes, the rendering objects, are evaluated by rendering engines, which use the algorithm objects to interpret shapes and attributes. Using generalized scene graphs, most real-time rendering techniques for OpenGL and several 3D rendering systems can be integrated in a single scene representation without loosing control over or limiting individual strengths of rendering systems.

1 Introduction

Scene graphs are ubiquitous: most high-level graphics toolkits provide a scene graph application programming interface (API) to model 3D scenes and to program 3D applications. An ideal scene graph API should simplify programming of 3D applications, optimize rendering performance, and provide abstraction by encapsulating rendering algorithms [11]. The generalized scene graph supports application development on top of different, independent rendering systems, integrates seamlessly rendering techniques that require multi-pass rendering, and facilitates declarative scene modeling.

The generalized scene graph is based on an object model with three main object categories (Figure 1). *Rendering objects* include 3D and 2D shapes, appearance and transformation attributes, handlers for rendering objects, and rendering techniques. *Scene graph nodes* hierarchically organize rendering objects and may generate and constrain rendering objects. *Rendering engines* traverse and interpret the contents of generalized scene graphs. During scene graph evaluation, the rendering engine manages a generic rendering context, associates rendering objects, and invokes handlers.

Handlers implement algorithms that map attributes and shapes to constructs of a target rendering system. We separate handlers from shapes and attributes to support different rendering systems. In addition, handlers do not restrict the kind of mapping and may support non-graphics rendering such as sound rendering as well.

Techniques implement multi-pass rendering algorithms, which need to traverse all or parts of the scene graphs multiple times; techniques trigger multiple traversals during scene graph evaluation.

Furthermore, generalized scene graphs support declarative scene modeling by a pre-evaluation phase which analyzes scene contents. This mechanism improves usability and compactness of scene specifications, and it can be optimized so that the computational overhead is reduced to a reasonable amount.

The increasing number of real-time rendering techniques possible due to significant advances in graphics hardware (e.g., per-pixel fragment

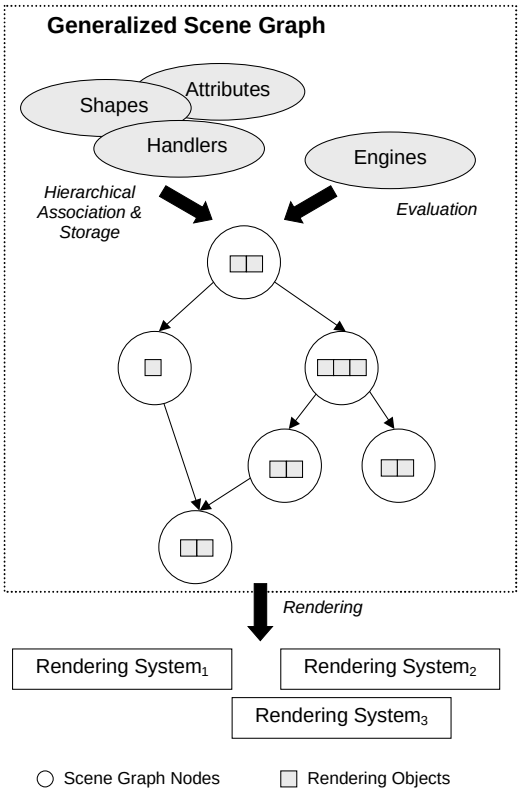

Figure 1: Components of generalized scene graphs.

coloring by register combiners [7]) motivated our work. These techniques are difficult to integrate in current scene graph systems due to architectural constraints. Furthermore, we have been looking for a uniform and open API for different rendering systems at the scene modeling level.

2 Related Work

OpenInventor [14] has introduced the classical concept of a scene graph API that has been adopted by other systems (e.g., Java3D and VRML). As a common characteristic, order and arrangement of rendering primitives in the scene graph reflect the underlying rendering pipeline. OpenInventor, Java3D, and VRML concentrate on real-time rendering and do not facilitate the usage of other rendering techniques (e.g., photorealistic and non-photorealistic rendering). The scene graph APIs differ in the way the underlying 3D rendering system can be accessed: OpenInventor provides full access to OpenGL whereas VRML and Java3D do not provide extensibility to that degree.

Graphics frameworks such as PREMO [3], BOOGA [1], and Generic3D [2] represent extensible, object-oriented computer graphics architectures but do not intend to generalize scene graphs as our work does. PREMO can represent scenes similar to OpenInventor [15] whereas Generic3D offers different low-level data structures to define the type of scene representation. However, in these systems the scene graph itself has not been improved substantially compared to OpenInventor.

GRAMS [4] is one of few graphics systems separating rendering algorithms from rendering objects. Shape rendering algorithms are selected using rendering efficiency as criterion [5]. In our generalized scene graph API, the separation has been extended towards algorithms evaluating attributes and multi-pass rendering algorithms.

The Vision architecture [13], focusing on graphics based on global illumination calculations, completely separates geometry objects and their attribute objects using object-oriented design even at a low level in the system architecture [12]. In our work, the separation between geometry primitives and attributes has been extended: attribute categories do not only include optical attributes and transformation attributes but also include rendering algorithms; a generic attribute management handles rendering-system dependent attribute types. Furthermore, we abstract and parameterize the evaluation process applied to scene graphs by rendering engines.

This paper outlines the rendering aspects of the generalized scene graph emphasizing the *generic and abstract specification of image contents* – interaction and animation represent complementary issues (e.g., event handling, constraint management) which we do not consider here.

3 Generalized Scene Graphs

A generalized scene graph is composed of scene graph nodes and rendering objects. Figure 2 outlines the object model.

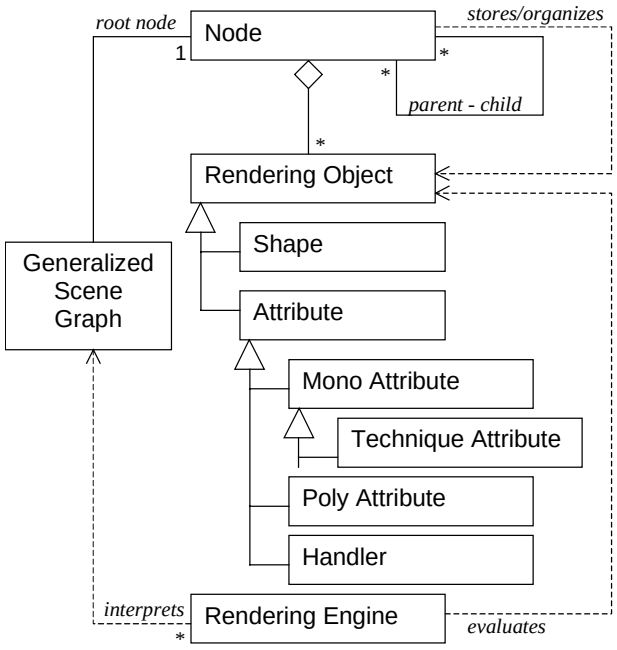

Figure 2: Object model of generalized scene graphs.

Rendering Objects

Rendering object are categorized as follows:

- *Shapes* include 2D and 3D geometric objects as well as volume data. Shape objects do not provide any rendering functionality; they contain only the parameters that describe their geometry and state. This ensures that their interfaces are not constrained to a specific kind of rendering technique.
- *Attributes* include appearance attributes (e.g., color, material, texture), transformation attributes (e.g., rotation, look-at, billboarding), properties (e.g., level-of-detail, picking and filtering identifiers), and lighting attributes (e.g., light sources, lighting switches).
- *Mono attributes* are attributes stored on stacks whose top elements represent the active attributes (e.g., color, material, drawing style).
- *Poly attributes* are attributes stored as collections, i.e., more than one attribute of one type can be simultaneously active (e.g., light sources, picking identifiers).
- *Techniques* are specialized mono attributes that encapsulate multi-pass rendering algorithms.
- *Handlers* are specialized mono attributes that map shapes and attributes to constructs of a specific rendering system, perform calculations, or convert rendering objects into a collection of rendering objects of less complexity. Handlers include, for example, *shape painters*, *shape simplifiers*, *ray intersectors*, and *attribute painters*.
- *Rendering engines* evaluate shapes, manage attribute stacks and collections, and install (respectively de-install) handlers. A rendering engine represents a generic rendering context.

Scene Graph Nodes

Scene graph nodes organize rendering objects in a hierarchical manner; they can also generate and constrain rendering objects and thus automate scene modeling.

Generalized scene graphs distinguish between two types of scene graph traversals: evaluation and inspection. The *evaluation* interprets and maps scene node contents (e.g., image synthesis); rendering engines perform the evaluation. The *inspection*, in contrast, only explores the scene graph contents and graph structure (e.g., scene graph storage). Both are implemented based on the visitor design pattern.

Evaluation

During evaluation, scene graph nodes communicate shapes and attributes to a rendering engine. The scene graph nodes formulate a "rendering micro program" which processes rendering objects using rendering engines (Figure 3). In a scene graph node, rendering objects and subgraphs can be stored in an inhomogeneous list. The pseudo code below outlines the node class as well as the *apply* and *unapply* procedures:

```
class Node {
private: List<Object> children;
public:
 void eval(E : Engine) {
  unapply(apply(children,E),E);
 }
 void inspect(V : Visitor) {
  for each child c do {
   V.explore(c)
   if(c is a node) { c.inspect(V) }
  }
 }
};
```

```
proc apply(C:List, E:Engine):Stack {
 S:Stack = {}
 iterate c through C {
  if(c is a shape) E.eval(c)
  else {
   S.push(c)
   if(c is a mono attribute) E.push(c)
   else if(c is a poly attribute) E.add(c)
   else if(c is a handler) E.install(c)
  }
 }
 return S
}

proc unapply(S:Stack,E:Engine) {
 while S not empty {
  c ← S.pop()
  if(c is a mono attribute) E.pop(c)
  else if(c is a poly attribute) E.remove(c)
  else if(c is a handler) E.deinstall(c)
 }
}
```

Mono attributes are pushed to (popped from) the engine's rendering context. Poly attributes are included in (excluded from) the collection they belong to. Handlers are installed (deinstalled) in the handler table of the context. In any case, appropriate attribute handlers or shape handlers are searched and invoked that map or apply the rendering object to the underlying rendering system. In a scene graph node, child nodes can be arbitrarily mixed with rendering objects in the list of children – child nodes start a nested evaluation. In any case, the attributes of a node affect only its children and never its sibling nodes.

Rendering Engines

A rendering engine is responsible for associating shapes and attributes as well as for invoking handlers. For mono attributes, the engine maintains a generic list of attribute stacks. The first time an attribute of a specific category is actually used, a new stack for that category is created; the top element of the stack represents the active attribute. For poly attributes, generic collections are used; their elements are not ordered.

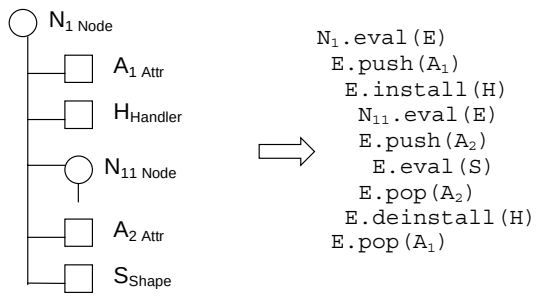

Figure 3. Scene graph structure and resulting engine microprogram.

Common types of engines include:

- *Synthesis Engines* use painters and simplifiers to map rendering objects to a rendering system.
- *Geometry Engines* analytically calculate intersections between rays and objects, objects and objects, or objects and environment, e.g., used for 3D interaction.
- *Analysis Engines* gather information about frequency of rendering objects and attributes that apply to a given shape, e.g., used for scene editing.

Handler Table

The handler table, part of the rendering context of an engine, is a two-dimensional array of stacks storing handlers. The stacks are sorted according to their *service* they provide (e.g., painting, intersection calculation, and simplification) and their *target* rendering objects they process (e.g., spheres, boxes, materials). For each service-target pair, there is a stack whose top element represents the active handler. This way, multiple dispatching is implemented encapsulating functionality into objects.

For each supported rendering system, a specialized rendering engine has to be implemented. At construction time, the engine configures its handler table with native shape and attribute painters. In a scene graph, handlers can be overwritten like attributes, that is, the handling of shapes and attributes can be reconfigured in each node or subgraph.

Specialized rendering engines can provide optimized implementations of engine methods to exploit special functionality of the underlying rendering system. For example, the OpenGL engine overloads all methods concerned with geometric transformations because OpenGL handles them directly.

Design Paradigms

The following design principles characterize the generalized scene graph:

- *Strict separation of shapes and attributes* similar to Vision [12]. Furthermore, there is no restriction for the types of attributes –

engines provide a generic rendering context.
- *Strict separation of declaration and implementation*. Handlers encapsulate all kinds of algorithms applied to rendering objects. Therefore, shape and attribute classes are kept lightweight, and algorithms can be partially substituted in order to optimize or introduce application-specific implementations.
- *Separation of structure and content*. Scene graphs provide structure; rendering objects provide contents. Both can be extended independently.
- *Scene graphs are understood as parameterized scene content specifications.* A scene graph can only be evaluated for given a rendering engine. The contents of a generalized scene graph can be interpreted for different purposes by different rendering systems.
- *Shapes and attributes are context-dependent.* For example, a billboarding attribute, which orients things towards the viewing direction, is defined relative to the current model-view transformation. This allows for intelligent, automated attributes.

4 Interfacing Renderers

The generalized scene graph supports multiple rendering systems. For each system, we implement the handlers for built-in shapes and attributes, the attributes for system-specific features, and the specialized rendering engine.

Handling of Shapes

Two categories of handlers are responsible for shapes: *Shape painters* map shapes to native constructs of rendering systems, and *shape simplifiers* decompose shapes into collections of lower-level rendering objects (including attributes). If no painter is provided for a shape, the engine searches for an appropriate simplifier. If no simplifier is found, the shape is ignored.

Rendering engines install suitable handler at construction time. If handlers should be substituted, they can be included in generalized scene graphs. Typically, we substitute handlers to use optimized rendering algorithms. For example, an application might want to switch between shape painters for OpenGL 1.1 and OpenGL 1.2.

In addition, generalized scene graphs can reuse existing graphics codings. For example, consider OpenGL parsing and display functions for objects in the Wavefront "obj" format: To include this coding, we need
- a shape class representing a Wavefront object;
- an OpenGL shape painter class that wraps the C function for parsing the "obj" file and calls the C function to draw the object; and
- a shape simplifier class that converts the "obj" object into a sequence of rendering objects consisting of material attributes and polygon shapes – this way, renderers other than OpenGL are immediately supported.

Therefore, the generalized scene graph API does not suppress existing, possibly very efficient graphics code. Figure 4 illustrates a sample scene graph using the "obj" shape classes.

Handling of Attributes

Attribute types differ to a high degree among rendering systems. Therefore, generalized scene graphs permit to store any attribute type. Attributes are not evaluated unless a handler is

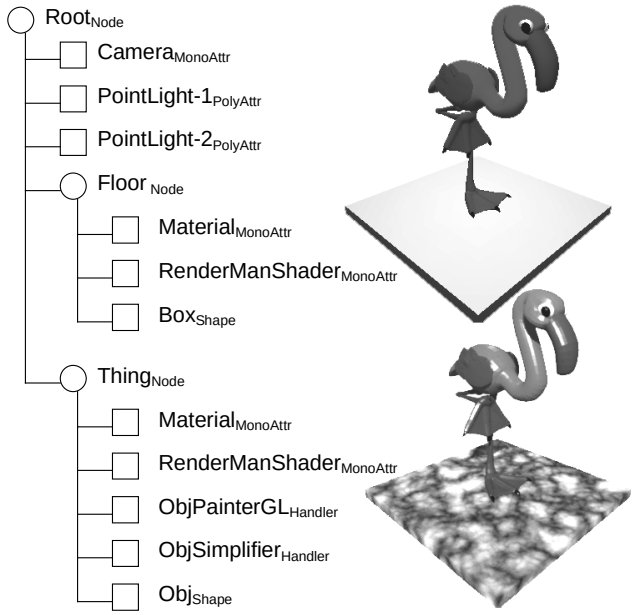

Figure 4: Flamingo scene graph evaluated with an OpenGL engine and a RenderMan engine.

installed. This way, attributes not applicable to a rendering system are ignored.

The generalized scene graph API defines a small collection of standard attributes (e.g., appearance and transformation attributes), and provides specialized attributes for each supported rendering system. For example, OpenGL-specific attributes cover most of OpenGL's functionality. For RenderMan a shader attribute interfaces compiled RenderMan shader files (Figure 4).

In particular, renderer-specific attributes, which are included as regular attributes in scene graphs, facilitate the production of high-quality animations: manual post-processing of exported scene descriptions is no longer necessary because all details of the target rendering system can be expressed in generalized scene graphs. For all built-in attributes, default conversions are available to map the attributes reasonably (e.g., materials and light sources).

In analogy to shapes, attributes can be directly evaluated (attribute painter) or further decomposed into a collection of lower-level rendering objects (attribute simplifier), which permits to model complex aggregated attributes. For example, an "importance" attribute might be decomposed in a certain line style and material for OpenGL. In comparison, a nonphotorealistic renderer might interpret the attribute directly.

5 Multi-Pass Rendering

Multi-pass rendering is one characteristic element of hardware-accelerated shading, lighting, and modeling techniques [6][7][10][16] that implement, for example, high quality illumination models, bump-mapping, and image-space CSG modeling. In generalized scene graphs, *techniques* implement multi-pass rendering. A technique defines one or more rendering passes, and it takes control over the evaluation of shapes and attributes. Techniques may be nested, triggering nested multiple passes. The technique interface is defined as follows:

```
class Technique : MonoAttribute {
public:
  //multipass control
  void start(Engine);
  void stop(Engine);
  void nextPass(Engine);
  bool needsPass(Engine);

  // redirection of evaluation
  void push(MonoAttribute,Engine);
  void pop(MonoAttribute,Engine);
  void add(PolyAttribute,Engine);
  void remove(PolyAttribute,Engine);
  void install(Handler,Engine);
  void deinstall(Handler,Engine);
};
```

To activate a technique, we push the technique as a regular mono attribute; it re-directs to itself all evaluation methods of the engines for attributes and shapes. For example, the technique might decide in one pass to evaluate only shapes and attributes of certain categories. Then, rendering objects and subgraphs are processed as many times as required. Finally, the technique is popped and redirection ends. The modified *apply* procedure is given below:

```
proc apply(C:List, E:Engine):Stack {
 P : List = C    // objects to be processed
 S : Stack = {}  // objects to be unapplied
 while P not empty do {
  c ← first element of P
  P.remove(c)
  if(c is a shape) E.eval(c)
  else {
   S.push(c)
   if(c is a mono attribute) {
    E.push(c)
    if(c is a technique applicable to E) {
     t.start(E);
     while(t.needsPass(E)) {
      unapply(apply(P,E),E)
     }
     t.stop(E)
     P ← ∅
    }
   }
   else if(c is a poly attribute) E.add(c)
   else if(c is a handler) E.install(c)
  }
 }
 return S
}
```

If a technique is not applicable to a rendering engine, it is ignored and no multiple passes result. If techniques are nested, they can come into conflict with framebuffer resources such as stencil buffer, depth buffer, or color buffer. OpenGL implementations do not provide an efficient solution to that problem because the reading and restoring operations for framebuffers use application memory. The implementation of nested techniques could be improved if OpenGL would permit to restrict framebuffer

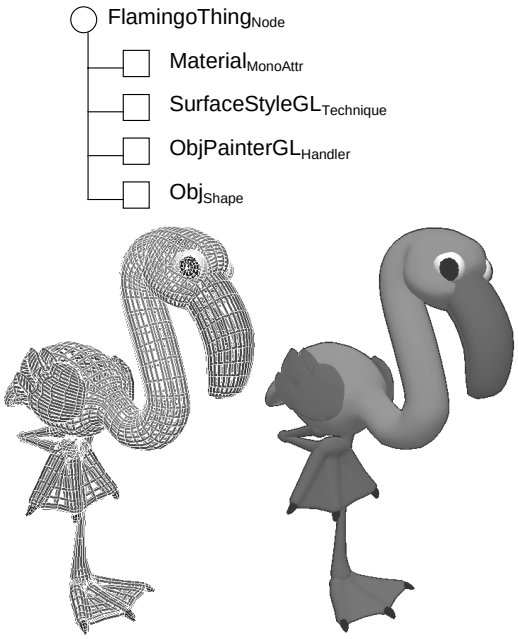

Figure 5. (left) Line drawing with haloed wires. (right) Model with additional silhouette.

operations to a selected (e.g., rectangular) region and would provide a kind of framebuffer stack implemented on the graphics hardware.

Examples

We have implemented various line drawing styles such as haloed wire-frames or silhouette drawings for OpenGL [9]. The corresponding technique uses multiple rendering passes and stenciling (Figure 5).

In addition, algorithms for shadows and reflection [8] have been encapsulated this way. In Figure 6, the shadow-volume technique provides interactive shadows, and the mirror technique calculates reflections of scene objects.

6 Declarative Scene Modeling

Scene graphs such as in OpenInventor evaluate scene nodes in a depth-first order. Therefore, global information, for example about light sources and cameras, is not available – scene contents has to be arranged in such a way that the depth-first order is consistent with the rendering pipeline. Generalized scene graphs can pre-evaluate the scene gathering global information. Candidates among the pre-evaluated rendering objects are:

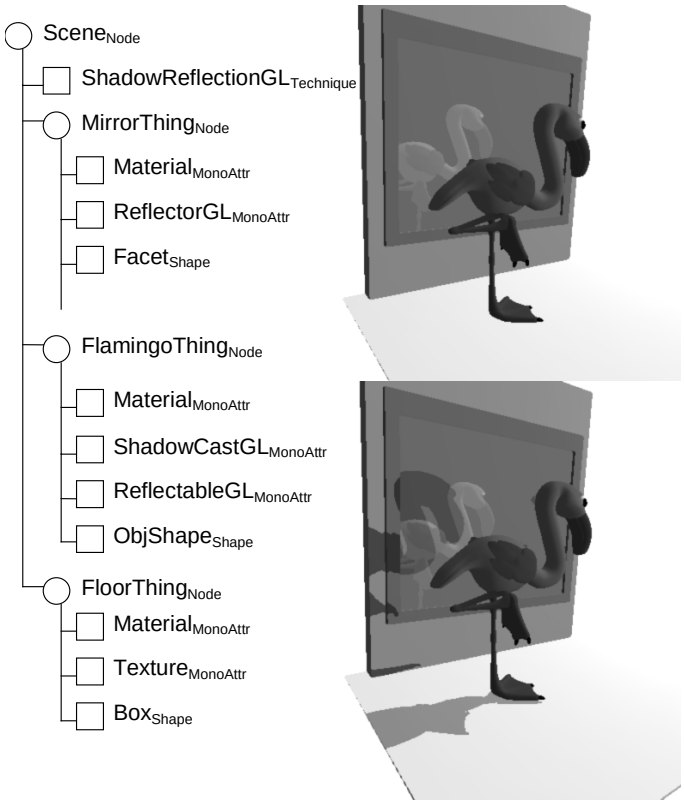

Figure 6. Generalized scene graph with shadow-reflection technique.

- *Light sources*: During the pre-evaluation, all encountered light sources are installed and enabled. They can be embedded in a local modeling coordinate system (e.g., the lights of an automobile). During the main evaluation, light switch attributes control which light source to turn on or to turn off in each subgraph.
- *Cameras*: The camera, modeled as an attribute containing projection and orientation transformations, can be arbitrarily positioned in the scene graph, for example, in a leaf node. Therefore cameras can be moved and positioned like geometric objects.

7 Conclusions

The well-known scene graph concept can overcome its current limitations if we distinguish between structure and contents of a scene specification. The scene graph nodes and

their hierarchical organization constitute the structure; the rendering objects, which are categorized in shapes, attributes, handlers, and techniques, represent the contents.

The evaluation of the contents is left to rendering engines, which delegate the interpretation of the contents to handlers and techniques that encapsulate mapping and multi-pass algorithms. The splitting of specification (shapes, attributes) and implementation (handlers, techniques) also permits to bypass abstraction when performance (e.g., integrating native, optimized code) or compatibility (e.g., code reuse) become important.

A single generalized scene graph can map its contents to different rendering systems without suppressing the individual strengths of the rendering system. In addition, generalized scene graphs cope with advanced real-time rendering techniques because they can express multi-pass algorithms.

References

[1] S. Amann, C. Streit, H. Bieri, "BOOGA – A Component-Oriented Framework for Computer Graphics", *GraphiCon '97 Proceedings*, 193-200, 1997.

[2] E. Beier, "Issues on Hierarchical Graphical Scenes", *Programming Paradigms in Graphics*, Springer, 3-12, 1995.

[3] D. J. Duke, I. Herman, "Programming Paradigms in an Object-Oriented Multimedia Standard", *Computer Graphics forum*, 17(4):249-261, 1998.

[4] P. K. Egbert, W. J. Kubitz, "Application graphics modeling support through object-orientation.", *IEEE Computer* 25(10):84-91, 1992.

[5] P. K. Egbert, "Utilizing Renderer Efficiencies in an Object-Oriented Graphics System", *Programming Paradigms in Graphics*, Springer, 13-22, 1995.

[6] W. Heidrich, H.-P. Seidel, "Realistic, Hardware-accelerated Shading and Lighting", *Computer Graphics* (SIGGRAPH '99 Proceedings), 1999.

[7] M.J. Kilgard, "A Practical and Robust Bump-Mapping Technique for Today's GPUs", GDC 2000 - Advanced OpenGL Game Development, 2000.

[8] M.J. Kilgard, "Improving Shadows and Reflections via the Stencil Buffer", NVIDIA White Paper, 2000.

[9] T. McReynolds, D. Blythe, B. Grantham, "Advanced Graphics Programming Techniques Using OpenGL", SIGGRAPH 99 Course Notes, 1999.

[10] M.S. Peercy, M. Olano, J. Airey, J. Ungar, "Interactive Multi-Pass Programmable Shading", *Computer Graphics (Proceedings SIGGRAPH 2000)*, 425-432, 2000.

[11] H. Sowizral, "Scene Graphs in the New Millennium", *IEEE Computer Graphics and Applications*, 20(1):56-57, 2000.

[12] P. Slusallek, H.-P. Seidel, "Object-Oriented Design for Image Synthesis", *Programming Paradigms in Computer Graphics*, Springer, 23-34, 1995.

[13] P. Slusallek, H.-P. Seidel, "VISION – An Architecture for Global Illumination Calculations", *IEEE Transactions on Visualization and Computer Graphics*, 1(1):77-96, 1995.

[14] P. Strauss, R. Carey, "An object-oriented 3D graphics toolkit", *Computer Graphics (SIGGRAPH '92)*, 341-449, 1992.

[15] D. Wang, I. Herman, G. J. Reynolds, "The Open Inventor Toolkit and the PREMO Standard", *Computer Graphics forum*, 16(4):159-175, 1997.

[16] T.F. Wiegand, "Interactive Rendering of CSG Models", *Computer Graphics forum* 15(4):249-261, 1996.