Scene Graph Rendering

Dirk Reiners
OpenSG Forum
dirk@opensg.org

March 5, 2002

This part of the course will introduce you to scenegraph systems for rendering. Scene graphs can help simplifying application development and making optimal use of the available graphics hardware.

It is assumed that you have some basic knowledge about 3D computer graphics. If the words polygon, directional light source and texture mean nothing to you, you might have problems following the course. The ideal entry level would be having written some programs using OpenGL, or having read the "OpenGL Programming Guide"[1] or "Computer Graphics"[2].

The first section describes the basic structure of a scenegraph and the difference to a standard OpenGL program. As there are a large number of scene graphs around, Open Source and commercial, section 2 gives a short overview of the most commonly used ones. The remainder of this chapter will describe general concepts applying to most scene graphs and use OpenSG as a specific example. The next two sections describe the general node structure and how the graph is traversed. The most important leaf node, the Geometry, is described in section 5. The other of specifying the displayed geometry, the state of transformation and materials etc. is covered in section 6. Some more scene graph specific functionality is hidden inside other node types, as described in sec. 7. To minimize the memory footprint of a scene graph, data can be shared in different ways, sec. 8 gives details. Sec. 9 touches on the importance of multi-threading and what is done in scene graphs to support it. After it has been used as an example in all of this part, sec. 10 talks a little about OpenSG. The last section talks about strengths and weaknesses of scene graphs and in which situations it might not make sense to use one (hint: there are not too many).

1 What is a Scene Graph?

Low-level graphics libraries like OpenGL are immediate mode based. For every frame it is necessary to retransmit all the data to the graphics hardware by the application

program. The model of the communication is like “Take this polygon and this and this and please render them”.

Scene graphs are retained mode based. The data is passed to them once and only updated if needed. The communications model is more like “This is my data and now please render an image and another and another”.

The data the scene graph manages is structured as a graph. Nodes are linked together by directed links and a root node is used to define to start of the scene data. For rendering the graph is traversed starting at the root. Scene graphs are usually acyclic graphs, i.e. there are no links from a node to any of its predecessors in the graph on the way to the root. This would create loops and thus lead the traversal into an infinite loop.

One difference between scene graphs and mathematical graphs is that scene graphs are heterogeneous, i.e. the nodes have different types. The main distinction is between interior and leaf nodes. Leaf nodes carry the displayable geometry, interior nodes structure the graph into logical groups (see fig. 1). There are different kinds of in-

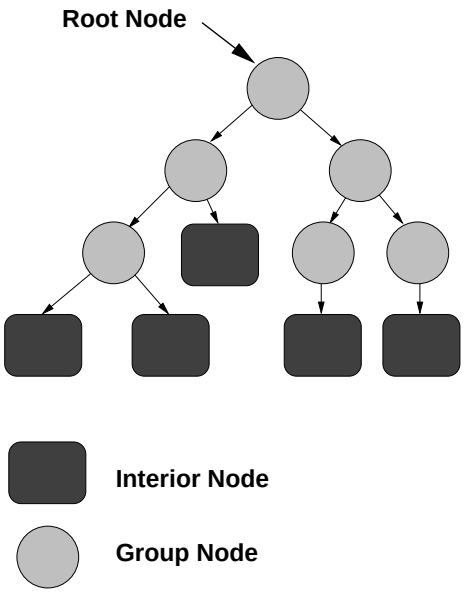

Figure 1: Scene graph structure

terior nodes, the most basic being the simple group, but other interior node types are described in sec. 6 and sec. 7, and, less often, different types of leaf nodes. The most important leaf node, however, is the geometry (see section 5).

2 Which Scene Graphs are there?

Scene graphs have been around for a long time. The following list is by no means complete, but lists most of the better known ones:

- OpenSG (http://www.opensg.org)[3]
- Open Inventor (http://oss.sgi.com/projects/inventor)[5]
- PLIB (http://plib.sf.net)
- SGL (http://sgl.sf.net)
- OpenRM (http://openrm.sf.net)
- Open Scene Graph (http://www.openscenegraph.org)
- Performer (http://www.sgi.com/products/performer)[4]

All of them are based on and written in C++, Inventor and Performer also have a C interface. Searching for "scene graph" on SourceForge (www.sf.net) will list a lot of other systems, but many of which are not actively developed any more.

Three of these (Performer, Open Scene Graph and OpenSG) have VR Juggler bindings and thus are good candidates for a VR application. Note that Performer is not Open Source but a commercial product.

3 Node Structure

All nodes in a scene graph have some common attributes.

One is the bounding volume. The bounding volume of a node is a simple volume, usually an axis-aligned box or a sphere, that encloses the contents of all the nodes below the current one. It is used by the scene graph to check the node for visibility. If the bounding volume of the node is outside the visible area, everything below it can't be visible and doesn't have to be passed to OpenGL at all. For large scenes this can have a significant impact on rendering speed.

Different scene graphs have slightly different organizations here. OpenSG keeps the list of children in every node, even if it is not used for leaf nodes, as it unifies the structures and simplifies traversals. It also keeps a pointer to the parent node, see sec. 8 for details on parents and their significance.

One OpenSG specialty is that the nodes are split in two parts, the Node itself and the Core. The Node keeps all the general info like the bounding volume and the pointers to the parent and to the children of the node. The Core is used to distinguish the different types of nodes (see fig. 2 for an example graph).

The simplest kind of Core is the Group, which has no further information, but is just used for structuring the graph.

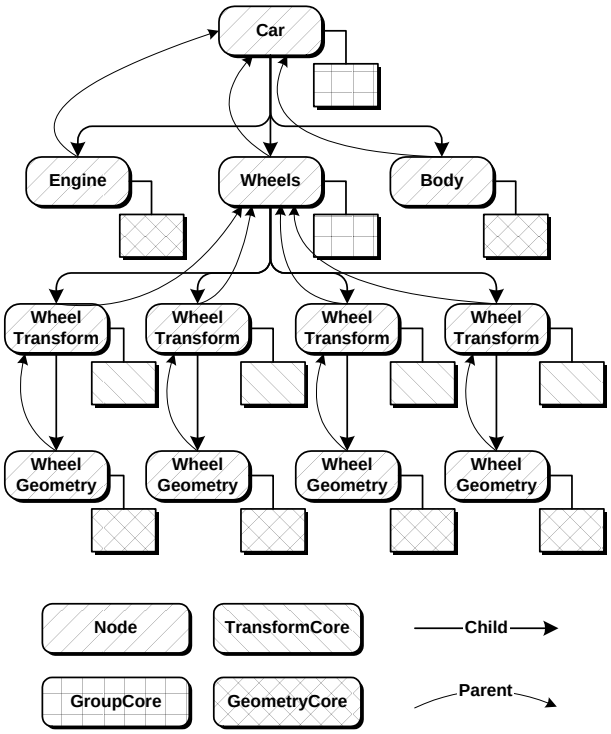

Figure 2: Node-Core splitting

4 Traversals

The basic operation on a scene graph or a graph in general is a traversal of the graph. Starting at a root node the links between nodes are followed until all nodes are visited. Most scene graph traversals are depth-first, i.e. before continuing with a brother of the node all its children are traversed (see fig. 3).

Some traversals are predefined by the scene graph system, but it is also possible for an application to traverse the scene graph. Some utility functions can significantly simplify that.

4.1 Actions

The classes encapsulating traversals are called Actions in OpenSG. It is an action object that is called with the root of the graph to traverse. Other systems use node methods, but the basic premise is always to start a traversal at a root node. Depending on the kind of node different operations are executed and different children are selected for or excluded from traversal.

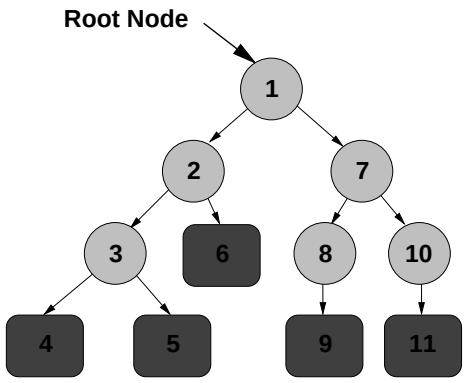

Figure 3: Depth-first traversal order

Different kinds of traversals are available in different systems. The most important being the Rendering traversal, but most system also have an Intersect traversal.

4.1.1 Render Action

The job of the RenderAction (also called Draw in other systems) is to turn the scene graph's data into low-level graphics library commands and thus turning the graph into an image.

The interesting part of this are the optimizations that the scene graph can do because of its higher-level knowledge about the whole scene. It can test nodes or whole subtrees for visibility, in the simplest case just for inclusion in the field of view of the viewer, but more complex tests are possible. Even the simple test however will already remove a large part of the data from consideration for most scenes.

The scene graph can also optimize and minimize the changes that need to be made to the low-level library's state to render the whole scene. This can significantly improve rendering performance for pipelined systems.

The rendering is where much of the effort of a scene graph system is put, including a lot of experience on how to make the most efficient use of graphics hardware and thus reaching high rendering performance.

4.1.2 Intersect Action

Another common action is the IntersectAction. It tests the geometry of the graph against a ray (sometimes a set of rays) and thus allows intersection tests with geometric objects in the scene. These can be used for picking and selecting objects or for simple collision detection tests, e.g. to keep a constant distance from a ground geometry.

Again, the high-level data embedded in the scene graph allows optimizations by testing larger parts of the scene before having to bother with the geometric details. Thus the

scene graph can be more efficient than a hand-made test, not to talk about relieving the application writer from to program it.

A scene graph will usually not be useful to replace a ray-tracing renderer, though, as these tend to have a lot more specialized and thus more efficient data structures for ray-tracing.

4.2 Simple Traversals

Applications sometimes need to traverse the scene graph themselves to implement specific functionality that the scene graph doesn't supply itself, e.g. to find geometric objects that do (or do not) satisfy some conditions.

Of course the application can implement a full traversal itself or derive from one of the basic actions implemented by the scene graph. But most of the application's traversals will be rather simple and don't need all the infrastructure needed by the complex traversals.

For these cases some systems, including OpenSG, feature a simple way to just traverse the graph and pass every encountered node to a user-supplied function or method to work on it and decide whether to continue or abort the traversal. In OpenSG this function is called `traverse()` and is available in some variants for different situations. The details go beyond the scope of this text, see the OpenSG traverse tutorial for details (see sec. 10 on how to get the tutorials).

5 Geometry

The most important node type in a scene graph is the geometry, as it holds the geometric data that is finally rendered. Relating it to OpenGL, the geometry keeps everything that happens between the first call to `glBegin()` and the last call to `glEnd()` for an object. There are many different potential ways of specifying that data, as OpenGL is very flexible.

A common interface style has proven to be useful and is used with little variation by all current scene graphs. It is based on splitting the data into separate arrays and is very close to the OpenGL `VertexArray` interface.

5.1 Vertex Properties

Each vertex of a primitive in OpenGL can have a number of attributes. The most important, of course, is the position of the vertex. The other common ones are the normal, which is used for lighting calculations, the color and the texture coordinates.

Scene graphs use separate arrays for these attributes. How these arrays are managed differs from scene graph to scene graph. In the simplest case the scene graph itself

only stores pointers to the data and the application is responsible for managing the data itself. Another option is storing the data in the geometries themselves, typically in the form of STL vectors or similar dynamically sized arrays.

The disadvantage of this approach is that the types of the data are fixed. OpenGL is very flexible in which kinds of data it can accept and different applications have different needs. It would be possible to create separate types of geometry for different parameter types, but the combinations of different dimensionality (1 to 4) and different data types (unsigned/signed, byte/short/int/float/double) for the different attribute types would lead to hundreds of classes for completeness, which is not practical.

OpenSG solves that problem by putting the attributes into their own data structures, called Properties. The Geometry Core itself keeps only references to these Properties. There are abstract base Properties for every type of attribute, and differently typed concrete versions. This way the single Geometry Core types can accept all variants of data types.

5.2 Primitives

Vertices themselves are not enough to define geometry, they need to be connected in the right way. OpenGL has a number of different ways to connect vertices. From drawing them as simple points, lines or connected line via triangles, quads and polygons to connected primitives like triangle strips and fans. These need to be mapped by the scene graph, too.

This can be done in different ways. One is to have different geometries for different primitive types and only allow one primitive type per geometry. Some systems try to lessen the impact by splitting the Geometry into a number of micro-geometries, all of which can have their own primitive type.

A geometry must keep more than one primitive instance, even if they are all the same type. For some types (e.g. points, lines, triangles) a single vertex count is enough, as they already contain multiple geometric primitives defined by a fixed number of vertices. Other types (e.g. polygons, strips, fans) need a separate length per primitive instance. This length/these lengths are usually also stored as an attribute of the geometry just like the vertex attributes.

OpenSG allows mixing primitive types by also adding a types property keeping a list of primitive types to use. Every entry in the types property corresponds to an entry in the lengths property, assigning a length to every primitive type instance and thus defining it (see fig. 4). This offers maximum flexibility by allowing any number of primitives, homogeneous as well as heterogeneous.

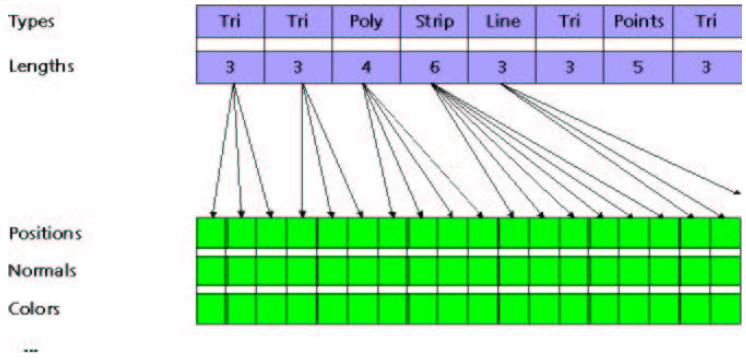

Figure 4: Geometry Structure

5.3 Indexing

5.3.1 Single Index

In a closed surface, vertices are often used multiple times. In a simple grid made out of quads, nearly every vertex will be used by four primitives. Having to store these vertices four times would significantly increase the memory needed by the system and thus is not acceptable.

Thus an indexing layer is inserted between the primitives and the vertices. Instead of using the vertices in the order they are given in the property, indices are used in order to indicate the vertices to be used, which are not duplicated. As an index, which is 2 or 4 byte, uses a lot less data than a vertex, which typically uses 32 and up to 120 byte, this reduces the memory consumption significantly (see fig. 5).

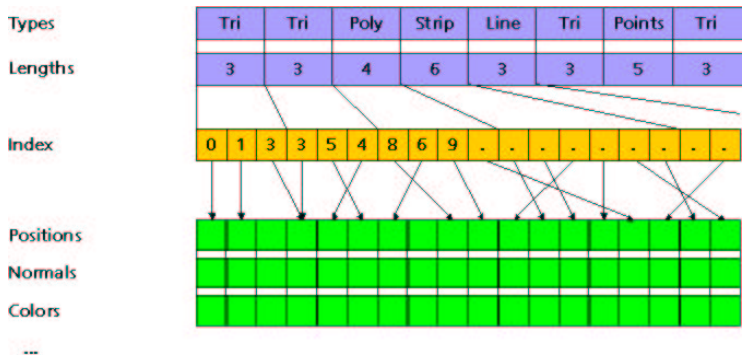

Figure 5: Indexed Geometry Structure

The indexing does incur an additional overhead, though. Thus it should only be used when necessary. Some scene graphs make the distinction in the form of different geometry node types. OpenSG just detects the presence or absence of an `IndexProperty`.

5.3.2 Multi Index

So far the attributes were all linked together, i.e. if there were colors and/or normals at all, there was a color and normal for each vertex position. This can be too much and not enough at the same time.

If colors are only taken from a limited subset, only a couple of different colors may be needed. Thus having a separate color for every vertex can eat up a lot more memory than necessary. On the other hand a single vertex position can need different normals. Usually faces are smooth-shaded across the face to give the impression of per-pixel lighting calculation, even if it's only done on a per-vertex basis. If a hard edge is desired, the single vertex has to use different normals.

The simplest form of multi-indexing is a distinction between per-vertex and per-face attributes. Per-face attributes are not connected to the vertices, but to the faces instead. They are usually not indexed. They solve the normals problem given above in most cases, but can't help with the inverse problem given in the color example. A more flexible solution is having independent indices for every attribute.

OpenSG uses a different approach: interleaved indices. Instead of taking a single value from the index property per vertex multiple values can be used. An index mapping is used to define which index is used for which attribute. Every entry in the index map is a bitfield that can indicate any combination of attributes. This makes it possible to share indices for different attributes, or have a separate index for each (see fig. 6).

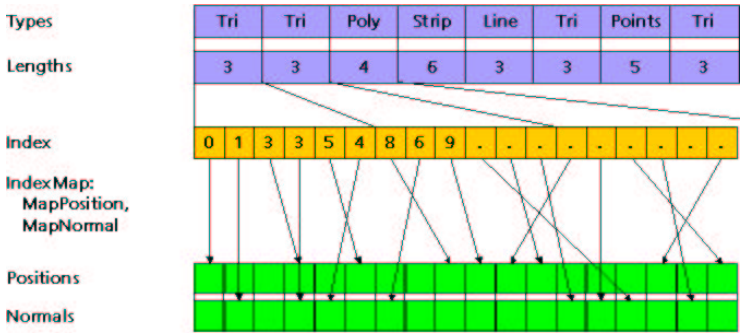

Figure 6: Multi-Indexed Geometry

One problem with multi-indexing is the performance overhead it causes. It can not be mapped directly to OpenGL's vertex arrays, it has to be realized using explicit calls, which costs performance, especially on the PC platform. For static models this cost can be offset into an initialization phase by putting them into a display list. For dynamic models it has to be payed every time the model is rendered, so care is advised when tempted to use multi-indexing.

5.4 Access

All the different variants of geometry data can make access complicated. Especially for OpenSG, which has just a single node type that supports all the variants of primitives and data types. There are two mechanisms to simplify accessing the geometry data.

The properties have a generic interface that hides the different data types. To do that every type of property has a generic data type that all types are converted to and from on access, e.g. `Pnt3f` for positions or `Vec3f` for normals. This conversion can loose some data, e.g. converting from 4D coordinates will drop the fourth coordinate. It also incurs a performance penalty, as the access is via a virtual function, and the data might have to be copied/converted. Thus if the types of the data are known and a large number of elements needs to be accessed, it is recommended to use the type-specific access instead.

The second mechanism helps getting around having to worry about the different primitive types and indexing: the iterators.

OpenSG has three different kinds of geometry iterators: primitive, face and triangle. The Geometry has `begin` and `end` methods to create iterators suitable for iterating through itself, just like STL containers. They all take care of all index dereferencing and allow direct access to the values of the primitive. They allow access to the Geometry as if it was composed of specific data structures for the separate primitives instead of having all the data separated in different arrays.

The difference between the different iterator types lies in what they consider an element that they iterate over. The `PrimitiveIterator` just steps through the types property and thus considers for example a triangle strip a single iteration element. The `FaceIterator` is more selective and ignores all points and lines. It splits all the remaining primitives into 3 or 4 vertex polygons for iteration. The `TriangleIterator` is similar, but only returns triangles.

The iterators make walking arbitrary geometry very easy, no matter which of the many possible variants of types and indices are used.

6 State Handling

Defining geometry is one half of creating a graphical scene, defining the surface, lighting and transformation attributes is the other half.

6.1 Transformations

Transformations in this context are limited to the geometric transformations that are used to move objects in the scene, transformations as far as camera manipulations are concerned are covered in other chapters of these course notes.

Transformations are applied to complete objects and cannot be used on parts of objects. Transformations are also usefully applied to whole subtrees of the scene graph, to move around a composite object or multiple objects and thus are the prime example for state inheritance in the graph.

Transformations are realized as a separate node type that carries a matrix or the elements needed to define the transformation (e.g. scale, orientation and translation). The transformation influences everything that's below it in the tree, including other transformations. Transformations accumulate, i.e. a new transformation does not replace the ones on top of it, but is added to them. This allows defining transformations in local coordinate systems while still being able to transform the whole system, and they allow defining hierarchical transformation chains. The prime examples for use of hierarchical transformations is a planetary system with planets and moons.

In a planetary system moons rotate around their planets, while the planets rotate around the sun. Having to specify the motion of the moons in absolute coordinates would be a complicated process. Specifying them in relation to their planet, while the planet-moon conglomerate moves around the sun is very simple.

6.2 Light Sources

Light sources pose interesting problems in the context of a scene graph. The position of a light source node in a scene graph can define two things. One is the position and orientation of the light source. This is useful for attaching light sources to other objects, e.g. headlights attached to a car. Due to the transformation inheritance the light source would automatically follow the movements of the car. The other meaning can be to define what's influenced by the light. This would be consistent with the state inheritance, i.e. everything below the light source in the graph is lit by it.

These two meanings of the position of a light source in the graph contradict each other and can not be solved with a single position in the tree. If the position of a light source in the graph would have both meanings at once the headlight scenario would be impossible, as the lights could only light what's below them in the graph, i.e. parts of the car but not the road, which is not typical headlight behavior. Nonetheless Inventor uses this method.

Different systems use different solutions to solve this. One solution is to define all light sources as global, i.e. everything is lit by every light source. In this case the position in the graph can be used for positioning the light only. This solution is simple but restricts the expressiveness of the system. Performer uses a workaround by allowing each geometry to explicitly define which light sources its influenced by.

OpenSG uses another solution. The position of the light source in the graph defines the parts of the scene that are influenced by it, in accordance with the state inheritance principle. The position and orientation of the light source are defined by a different node in the graph, the so called beacon, which is only referenced by the light source.

6.3 Materials

Materials are used to define the surface properties of the geometry and in OpenSG indeed are an attribute of the `Geometry` nodes. The attributes of the materials are mostly direct mappings from the OpenGL material states, i.e. emissive, ambient, diffuse and specular color and shininess.

Another innocent looking material attribute that can not directly be mapped to OpenGL but instead forces significant effort to be correctly implemented is transparency. Transparency and OpenGL don't match well. Transparent objects need to be rendered with the correct blending function after all other objects are rendered, and they need to be rendered back to front. To be fully correct actually they have be rendered back to front on a face by face basis, and for intersecting geometry the intersecting faces would have to be split and sorted.

Full and perfect support for transparency is immensely expensive, especially for dynamic geometry, thus no scene graph supports that. What most of them do is support for rendering transparent objects last and sorting them back to front, on an object to object basis, i.e. the objects will be rendered back to front (usually sorted by a reference point, e.g. the center of the bounding box), but sorting inside objects is not done.

OpenSG uses the `SimpleMaterial` and `SimpleTexturedMaterial` classes for this. The `SimpleTexturedMaterial` also keeps an image to be used as a texture for the material together with parameters to define the texture filtering.

7 Other Node Types

Different scene graphs offer different additional functionality in the form of other node types. Some of these are pretty much ubiquitous, others are specific to a few systems. The following are some typical examples.

7.1 Switch

The Switch node is a simple but useful variation on the Group node. It selects zero or one of its children for traversal instead of all of them. This is an easy way to switch off subgraphs or cycle through a static set of models.

As a variant some systems allow selecting a set of children using a bitmask.

7.2 Level Of Detail

Level of Detail (or LOD for short) is a rather simple but efficient way of optimizing rendering for large scenes. The basic idea is that objects that are far away don't have to be rendered as detailed as close objects.

To do that the scene graph needs to be able to reduce the complexity of the rendered objects. The best and easiest way to do that is for the application to supply a number of versions of an object that have different complexities. Each of these is also assigned a range, usually a distance from the viewer, for which is should be used. The ranges are stored in the LOD node, with the different versions as the children of the node.

The task of the LOD node is to select the correct child, e.g. for the given viewer distance. Generally it will select only one child, and the one most appropriate for the current situation, thus only displaying as much complexity as necessary.

7.3 Particles

Sometimes a large number of simple dynamically moving objects is needed. One example are viewer-aligned quads that can be used for rotationally symmetric objects, e.g. balls in a ball-and-stick molecule model, or to simulate smoke, using a transparent texture. Another example would be arrows to show the flow of a flow field.

Creating a Geometry node for each of the thousands of these very simple objects would incur a lot of overhead. Turning all of them to be viewer-aligned can be impossible for an application, if multiple different views on the scene are active, and thus should be left to the scene graph.

OpenSG supports these situations with a specific `Particles` node, which takes positions, colors and sizes as input and turns them into different kinds of geometry for every frame.

8 Sharing

8.1 Node-Based

The scene graph needs to have all the data to render the scene. In many scenes however there are objects that are just copies of other objects. A simple example are the wheels of a car. Four exact copies of the same object, just in different positions. For the scene graph to use memory efficiently there has to be a way to share the data used by these objects.

One way to do that is allowing the addition of objects to multiple parents. Using transformations the different copies can be positioned independently while sharing the same data. This is a very simple method that is used by many scene graphs.

It has a couple problems, though. Objects can not be identified by a pointer any more. As all the copies are the same object, a pointer doesn't know which copy is meant. Thus when walking up the graph, e.g. to accumulate the transformations on the way up to get the global position of the object, it's not possible to know which way to go. Another problem is the inability to associate instance-specific data to the node like a name. Again, the copies are all the same object, there is no distinction.

This is the main reason for the Node/Core division that OpenSG employs. Nodes are unique, they can not be shared, every Node just has a single parent pointer. Cores can be shared, they have multiple users (see fig. 7).

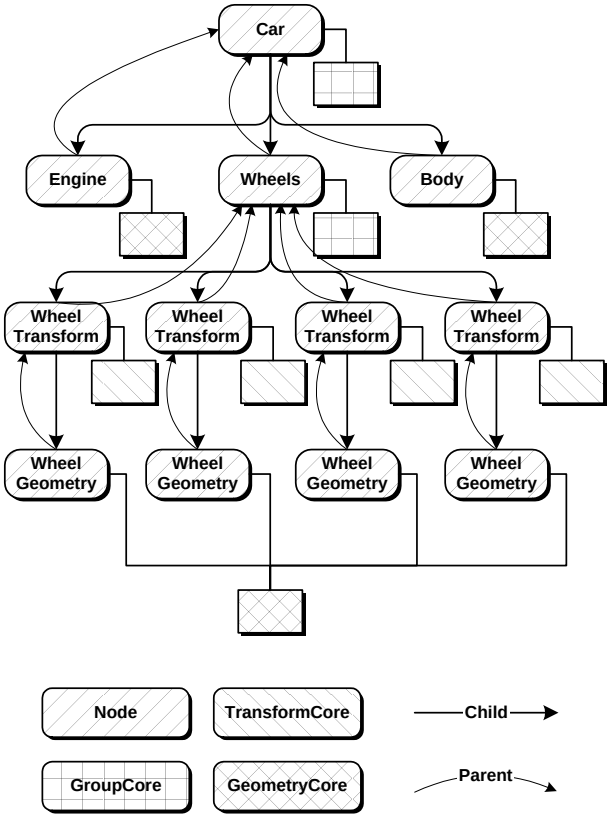

Figure 7: Shared Cores

As Nodes are pretty small, the memory penalty incurred by duplicating them is not a problem. The main amount of data contained in the Cores can be shared. Splitting the Graph structure from the node contents also allows sharing control. Sharing a `Switch` Core for example allows setting the value on the single Core to affect any number of Nodes in the scene. The same goes for `LOD`s, `Transformation`s or any other kind of Core.

8.2 Geometry

The bulk of the scene graph's data is contained in the geometry. Sharing complete geometries is possible with the node-based sharing described above. There are, however, situations where it makes sense to only share parts of the geometry, e.g. when drawing

a wireframe on top of a shaded model or to highlight parts of an object by drawing them in a different color.

As the data of the geometry is split into different properties anyway, it is a small step to sharing the separate properties separately. Most scene graph systems allow that in one way or another, either because the data is managed by the application anyway, or, like OpenSG, by splitting the data off into their own data structures.

9 Multi-Threading

Multi-threading is an aspect that is growing in importance, given the current trend in processor designs to go to simultaneous multi-threading (SMT).

A number of scene graphs provide simple multi-threading support, which allows simultaneous reading or rendering of the scene graph. For many applications this is quite adequate, but for more demanding applications that use asynchronous simulation threads needing consistent data and/or manipulate the graph it is not enough.

OpenSG supports these kinds of applications by allowing each structure to have multiple aspects, i.e. multiple copies of itself. This is done efficiently by only copying the bulk of the scene graph's data, the geometry data, when it is needed.

Different aspects can be efficiently synchronized under application control, making them contain the same data again. To support this in a general and easily extensible way, OpenSG adds reflectivity to the system. Reflectivity is the ability of the system's structures to give out information about themselves, e.g. which fields they contain and how to access those fields. This ability is used to automatically add synchronization capabilities to each structure of the system.

It can also be used to write a generic OpenSG loader/writer that can automatically handle new structures and extensions in the system without having to write any specific code for it. An extension of this system also allows synchronizing scene graphs across a cluster of machines. Again, no cluster-specific code needs to be written, the data is automatically distributed and synchronized.

The difference between this kind of clustering and the kind of clustering the already introduced systems clusterJuggler and netJuggler lies in the point on the pipeline the distribution is done. Both netJuggler as well as clusterJuggler distribute the inputs to all nodes. This approach minimized the amount of data to be sent over the network, but the full application is running on all nodes and needs to process the input. If that input processing takes a lot of calculation or depends on large external data this can be a problem. OpenSG distribution on the other hand distributes after the inputs have been processed but before OpenGL commands are generated.

This section can only give an idea of what's done in OpenSG for multi-threading, see [3] for more information.

10 Getting to know OpenSG

OpenSG is one example of the current breed of modern scene graphs. It is developed and distributed under the Open Source principle and is available including source code from www.opensg.org.

It is designed with multi-threading (see sec. 9 for details) and application extensibility in mind, i.e. changes to internal data structures are possible without having to change the library's source code (and thus without having to reapply changes whenever the library changes).

The development is supported by a number of companies and research institutes (see www.opensg.org/forum for details), guaranteeing sustained development. Further development is also sponsored by the German Government by a research project called OpenSG PLUS (see www.opensg.org/OpenSGPLUS) which funds continuous development of the core as well as a lot of high-level functionality (NURBS rendering, subdivision surfaces, Clustering, Occlusion Culling, Volume Rendering, High-Level Shading to name but a few).

It is developed on Linux, Irix and Windows, with a MacOS X port pretty much working and an HP port in the works. It is hosted on SourceForge (www.sf.net) and uses the SourceForge infrastructure to build a developer community, mainly based on mailing lists and IRC.

The 1.0 version was released in October 2001, the 1.1 developer version will be released in April 2002. But an automatic dailybuild system makes sure that the current CVS always compiles on every platform and allows access to daily snapshots if needed.

Some automated documentation is created from the source code using the doxygen tool, but more useful as an introduction is the Starter Guide (www.opensg.org/starter) and the tutorial examples (available in the tutorials/ directory in the source and distributions).

11 When does it make sense to use a scene graph?

This part of the course talked about scene graphs and what they can do for a graphics application. Of course they cannot solve all the problems, and they are not always the best solution to every problem.

Their main strength and their main weakness is that they are a retained mode structure. Having access to all the data making up the scene allows them to optimize the rendering by only rendering visible objects and sorting them by state to make effective use of the graphics hardware.

The weakness is that the data structure needs to be consistent with the application's data. If that data is very dynamic, especially if the node structure changes for every frame, keeping the scene graph up-to-date can eat up the profit that it brings. If only

the contents of the objects change it's not as bad, especially if the maximum size of the dynamic areas is known beforehand, but there is still some overhead involved.

Scene graphs really shine in complex scenes, complex in the sense that they consist of a large environment that uses many different materials. For situations where there is a small number of objects, all using the same rendering parameters, and all visible all the time, a simple application obeying some basic rules (only change state if it is really different, use vertex arrays with native data types) can reach a very good performance. Note however that even these applications can get a taste of complexity quickly, as soon as interaction gets involved and different kinds of markers, information displays and higher level rendering features are needed, in which case a scene graph becomes a more viable solution.

Besides being highly efficient rendering engines, scene graphs also tend to accumulate useful utility functions for handling scene data. The most prominent examples are loaders for models and images. Writing loaders is a job that nobody likes to do, so being able to use somebody else's work can already make using a scene graph worth it. Similar reasoning applies to the intersection test (see sec. 4.1.2).

A special case is the clustering support given by OpenSG, as it simplifies bringing an application to a cluster significantly, something which is becoming more and more important and can turn into a lot of work quickly.

When faced with an already fully developed application that displays a single object always centered on the screen, switching to a scene graph will only make sense in very few situations. However, when starting a new project it can never hurt to start using a scene graph. If the application turns out to be too simple for a scene graph going back won't be a big problem. But the amount of useful functionality that a scene graph provides, especially in it's domain of loading, processing and rendering scenes, can become addictive pretty quickly. Try it, you might enjoy it!

References

[1] Mason Woo et.al. *OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. Addison-Wesley Pub Co, 1999.

[2] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics, Principles and Practice, Second Edition*. Addison-Wesley, Reading, Massachusetts, 1990. Overview of research to date.

[3] Dirk Reiners, Gerrit Voss, and Johannes Behr. OpenSG Basic Concepts. In *OpenSG 2002 Symposium*, Darmstadt, 01 2002.

[4] John Rohlf and James Helman. Iris performer: A high performance multiprocessing toolkit for real-time 3d graphics. *Proceedings of SIGGRAPH 94*, pages 381–395, July 1994. ISBN 0-89791-667-0. Held in Orlando, Florida.

[5] Paul S. Strauss and Rikk Carey. An object-oriented 3D graphics toolkit. In Edwin E. Catmull, editor, *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 341–349, July 1992.