

A Flexible and Extensible Traversal Framework for Scenegraph Systems

Dirk Reiners
OpenSG Forum
dirk@opensg.org

15th February 2002

Abstract

Graph traversal is a very basic concept in computer science, as graphs are needed and used in lots of places. As scenegraphs are graphs too, the standard graph traversal methods should suffice to cover scenegraph traversals.

In practice, scenegraphs are a little different. They are heterogeneous, i.e. there are a number of different node types, which need to be treated differently. They are also used by a number of different types of traversals. Furthermore new node types as well as traversal types are added routinely to a scenegraph, often application-specific ones. This can be a problem when the source code of the library is not available.

This work describes a framework for scenegraph traversals that is flexible as well as extensible to allow adding node types as well as traversal types without having to change existing code.

The concept is based on three classes: Actors, Iterators and Actions. The Actors are the primitive active part and are used to be easily combinable to make up complex traversals. Actors are categorised according to different criteria to allow optimised traversals. Iterators are used to select the next node to traverse from the list of available nodes, allowing simple depth-first traversals as well as ordered traversals. The Action keeps all the parts together.

1 Introduction

Graphs are used in many different areas in computer science. Graphs allow turning many $O(n)$ problems into $O(\log(n))$ problems and as such have applications nearly everywhere. Thus every computer scientist is introduced to graphs and graph traversals very early in the education. As a scenegraph is also just a graph, traversals for scenegraphs should be rather trivial.

Scenegraphs are graphs, too. They are directed graphs, usually unweighted and sometimes without loops, i.e. they are trees. Thus traversing a scenegraph shouldn't be much different than traversing any other graph.

But that is not really the case. They differ from standard graphs in the sense that all non-trivial examples of scenegraphs have heterogeneous node types that need to be treated differently, something rarely found in other graphs. Scenegraphs also present their own problems, especially given the fact that scenegraph libraries for computer graphics research and development need to be flexible in terms of extensibility to allow quick extensions and testing of new approaches and algorithms. This includes adding new traversals as well as adding new node types. To facilitate application-specific traversals and to circumvent the need to recompile the whole library for additions, which might not be feasible in cases where the source is not available or the compilation times are prohibitive, this extensibility needs to be dynamic.

Section 2 describes the approaches that are described in the standard computer science literature and analyses the ones that are used by other scenegraph systems and their shortcomings.

A unified concept is described in section 3. It consists of three interacting parts, the actors (sec. 3.1) iterators (sec. 3.3) and the action to coordinate all these (sec. 3.4). Functors (sec. 3.2) are used to add dynamic extensibility to already existing Actors.

Section 4 summarises the contributions of this work and future directions.

2 Previous Work

The classical computer science graph traversals are characterised by the order in which the nodes are visited with respect to left and right subtrees.

Pre-order traversals visit the node before any subtree, post-order after any subtree and in-order visits the left subtree, then the node and finally the right subtree. A fourth ordering which is not used very often is level-order, which is a breadth-first traversal, in which all children of a node are visited before any of the children's children is visited.

Scenegraphs are in general no binary trees, and they don't split their children list into a left or right subtree, thus an in-order traversal can not be defined meaningfully. Most scenegraphs use a pre-order traversal, i.e. the node is visited before its children, in order to possibly select the suitable children to be traversed. Some systems provide a combination of pre- and post-order by using two visiting functions, one to be called before the children are traversed, one afterwards. No known scenegraph uses a level-order traversal.

One difference between standard graphs and scenegraphs lies in the types of graph nodes employed. Standard graphs are homogenous, i.e. all nodes have the same type. Except for trivial examples, that is not true for scenegraphs. As a consequence scenegraphs need to implement a double-dispatch strategy: the functions to be called depends on the type of the traversed node and on the type of traversal being done.

Dispatching on the type of traversed node is easiest to do by employing virtual methods or their equivalents in languages other than C++. The easiest approach to dispatching on the type of traversal is to use different methods for different traversals. The problem here lies in adding new traversal types. Adding new traversals demands adding a new method to the node base class and all affected classes. When the source is not available, this is not an option.

Performer Performer [5] is a pretty close match to the simple approach given above. Performer defines four types of traversals (ISECT, APP, CULL, DRAW) which cover the most common tasks for Visual Simulation, Performer's main domain. The traversal functions are virtual methods of the nodes and are hard coded, i.e. there is no way to add traversals to the system.

To allow users to manipulate the traversals every node can have traversal callbacks associated with it. These can be specified independently for every traversal type and for pre- and post-traversal calling. This gives quite a bit of flexibility to the user, at the cost of potential user intervention everywhere. As the user intervention can happen at any place in the traversal, care has to be taken in terms of optimisation to not interfere with this behaviour.

Open Scenegraph / Visitor Pattern The usage of design patterns [1] has become commonplace and accepted good practice in software engineering. The standard book [2] includes a pattern that is applicable to the scenegraph traversal problem, the visitor pattern.

The visitor pattern delegates the mapping from node to function to a separate visitor object, which holds the traversal functions for every type of node in the system. This pattern is used in the Open Scenegraph system [4].

The disadvantage of the visitor pattern is the difficulty in adding new node types. These have to be added to the visitor definition, as the visitor needs to have an accept() method for every node type. Sometimes it might be enough to add a dummy method to the root of the visitor inheritance tree, but not always. In any case, changes to the base library source code are required.

Open Inventor To circumvent the extensibility restriction associated with having to manipulate the library code a more dynamic approach has to be taken. The main point lies in the need to detach the traversal function from both the traversed node class as well as the traverser class. They cannot be members of either, as that would imply having to change one of them for adding a node or traversal.

To solve this problem Open Inventor uses a list of functions per traversal type, indexed by a suitable mapping from node type to index into the list. This allows dynamic extension of the list for new node types, as well as adding new traversals.

In general the invoked functions should be node methods, to allow access to the private members of the objects. This is a problem for traversal mechanisms, as pointers to members of different classes are not compatible. Open Inventor circumvents the problem by wrapping every member function into a static member function, which is compatible. The drawback is the need to add an additional function for every member that will be used in a traversal.

Jupiter / DirectModel The Jupiter [3] toolkit (formerly DirectModel) takes a very different approach to solving the decoupling problem, and adds some new ideas.

The traversal is split into two parts: acting on the current node and selecting the children that should be traversed, and selecting the next node to traverse. These tasks are delegated to separate objects.

The first task is handled by the Agent classes. An Agent is a class that has an apply method, which is called for every traversed node. It's the responsibility of the Agent to switch on the type of passed node, and to select the children of the node that should be considered for further traversal. A traversal consists of not one but a number of these Agents. This allows a more fine-grained and focused implementation of the Agents, i.e. there are separate Agents to do view frustum culling, LOD selection and finally, rendering. Thus specific tasks can be exchanged quite easily. The dispatch on node type is left to the individual Agent, every Node is passed through the same function.

The second is done by the Iterator classes. These keep the not yet visited but active nodes and select the next node to visit. Two different kinds of iterators are available: Stack and Heap Iterators. The Stack Iterator realizes a standard depth-first traversal by keeping a stack of active nodes and just pushing and popping the top element. The Heap iterator realizes a priority queue. Every node that is accepted for traversal has an associated value describing its priority. The node with the largest (or smallest) value is traversed next.

Conclusions A flexible and extensible traversal concept for a scenegraph needs to fulfil a number of goals.

Extensions and additions need to be possible without having to change the code of the main library. This code might not be available at all, or even if it is available for an Open Source system like OpenSG [?], having to adapt additions and changes every time a new version is used can create a significant amount of work. This precludes the use of the simple virtual method approach as well as the Visitor pattern, as both require base code changes for one of the two additions.

The monolithic actions most systems use make it difficult to replace parts of them, as the whole action has to be replaced. Jupiter's Agent concept is a good alternative. Furthermore the idea of splitting off the node selection into its own object allows different traversal orders including priority-based traversals.

3 Action Approach

The approach developed in this work uses three types of objects: Actors, Iterators and Actions.

3.1 Actors

Several Actors are combined to make up an Action, they are similar to Jupiter's Agents. But there are some specialisations to simplify optimisations and Actor development.

The main task of the Actors is selecting which if any out of the current list of active nodes are to be traversed. They have access to the currently traversed node and a list of currently active nodes from which they can deactivate any number of nodes, which will not be considered for traversal. In most cases the current; active list will be the list of active children of the current node, but not necessarily.

In addition to deactivating a node from the active list the Actors can also assign a key value to each node that is used by the priority queue Iterator.

There are several aspects that differentiate classes of actors.

3.1.1 Enter and Visit Actors

To allow both of the standard traversal order (pre- and post-order) and the combination of both the Actors have an enter and a leave method.

Enter is called whenever a node is starting to be traversed, leave is called when all the children of this node have been traversed. Enter Actors only use their enter method, they are the most widely used Actor. Visit Actors use both of their methods and are mainly used for keeping traversal-dependent state up-to-date (see sec. 3.1.2). It is also guaranteed that between the enter and leave call of the Visit Actor only descendants of the current node are called, so that the state is valid for all called nodes.

The choice of Actor can reduce the options the Iterators and Actions have in the traversal.

3.1.2 State

One significant distinction between different actor types is whether they manipulate their state.

StaticStateActors StaticStateActors do not change their state during traversal. The state can be used to parameterise the Actor, but it cannot depend on the traversed nodes.

Node-based optimisations like striping are a typical example for StaticStateActors.

The main advantage of StaticStateActors is that they can be used on the nodes in any order, allowing to combine nodes from several levels in the tree to cut down on the numbers of calls that need to be made to the actor. They can even be called in parallel threads, as they only read their state synchronisation problems cannot arise.

UnorderedStateActors UnorderedStateActors have a state that does not depend on the traversal order. They can keep state, e.g. for parametrisation, and they can change the state, but the changes do not depend on the traversal order.

A typical UnorderedStateActor is the Statistics actor. It works on nodes independent of any other node. It does change its state, but the changes are independent of the traversal order.

The main advantage of UnorderedStateActors is that they allow combining nodes from all over the graph and thus allow reducing the number of calls to the Actor significantly, similar to the StaticStateActor.

As they can change their state, parallel execution is not directly possible. It is possible if the state changing is serialised. This can be done in the Actor if necessary.

OrderedStateActors OrderedStateActors keep a state that is changed during the traversal and depends on the traversal order. Thus they either need the nodes to be visited in the right order, or keep a copy of their state around for every visited node.

One example for an OrderedStateActor is the rendering actor. It needs to keep an up-to-date state concerning the active matrices and OpenGL state during traversal. Objects are added to this state when entering nodes and removed when leaving them.

To allow OrderedStateActors to be used in traversals other than depth-first they provide methods to create a copy of the active state. The controlling Action is notified when the actor changes data and thus can create a state copy if needed.

3.1.3 Dispatch

Another factor distinguishing classes of actors is their handling of different node types.

UniformActors A number of actors are only concerned with the Node itself, not with its type. These Actors handle all nodes similarly and thus don't need any dispatch based on node type, a single function suffices. The most prominent example is the FrustumCullActor, which checks nodes for visibility, regardless of node type.

SingleActors Other Actors are only interested in a specific kind of node. These have a single entry point like the uniform actors, but only have to do work on a small number of nodes, e.g. Level of Detail nodes.

DispatchActors Other Actors, e.g. the Render Actor, have to execute completely different functions for different kinds of nodes.

To simplify supporting these the Dispatch Actor base class contains a mechanism to register a functor per node type and automatically dispatch to the registered functor for every active node. Due to the functor use the operation to be executed can originate from very different sources.

3.2 Functors

Functors generalise the callback functions used by Open Inventor or Performer. They define a unified interface to different ways of specifying a call to a function.

The simplest case is that of a real function. In this case the arguments are simply passed to the function and the functor behaves like a standard callback.

The second variation allows calling a specific object instance's method. This is especially useful for passing information to the called function. In most systems the callback function can take a single user data argument to parameterise the callback. This leads either to temporary structures to keep the data, or to a lot of static methods that can be called with the object as the user data. The method functor simplifies that by allowing a direct call to the object's method.

The third variation calls a method of an object that is not specified beforehand, but is specified as a parameter instead. This allows the functor to be used in a very generic way, as the called function can be virtual and thus switch on object type.

Thus the functor allows hiding very different ways of calling functions and specifying what should happen, giving maximum flexibility to the user.

3.3 Iterators

Iterators are used to order the traversal. They manage which nodes still have to be traversed and select the next set of nodes to be passed to the actors. They will mostly choose a node to traverse and use its child list as the active list, but for specific sets of Actors other choices are possible.

3.3.1 Depth-first

The most basic iterator is a stack-based depth-first iterator. It always selects the the node from the top of the stack and uses its' children as the active list. Nodes still active after all Actors have been called are added to the stack.

The depth-first traversal automatically traverses all children of a node before any other nodes are traversed, and thus simplifies the leave handling.

3.3.2 Priority Queue

The priority queue iterator also selects a single node to act upon and its children as the active list. But the single node is selected according to the priority assigned to the node by the Actors that selected it for inclusion. This allows directed traversals for specific purposes, like back-to-front rendering etc.

The disadvantage of this approach is the effect that it has on the traversal state. As the nodes can be traversed in an order that is completely separate from the tree order, Actors that maintain ordered state need to have a separate copy of the state for every node, so that they can continue the traversal based on the correct state for the node. This copy is kept by the Iterator. In practice not every node needs a separate state, as the state doesn't necessarily change with every node. Thus the states can be shared between nodes, and only when the state is changed a new copy needs to be made.

The power of the priority queue Iterator is severely limited when using Visit Actors. As the Visit Actor depends on the fact that only the children of the active node are visited between begin and end, only these can be sorted and traversed.

3.3.3 Breadth First

The breadth first iterator is used to reduce the number of actor calls by coalescing the children of multiple nodes into the active list before calling the Actors.

It suffers from similar limitations like the priority queue iterator. As the order the nodes are traversed in is decoupled from the tree, Visit Actors and Ordered State Actors cannot be used. As the nodes in the active list don't even have to be children of a single node, all Actors that use the active node cannot be used either.

The main applications of the breadth first iterator are Actions that only act on leaves like stripping or static statistics.

3.4 Action

The Action does little more than keep the Iterator and the set of Actors together and call them in the right order.

It asks the Iterator for the next node and the list of active nodes, applies all Actors to them and returns the results to the Iterator. This is repeated until the Iterator runs out of nodes.

4 Summary and Future Work

The traversal concept described in this work supports extensibility and flexibility by allowing free combination of different Actors and the use of Functors to specify the operations in the DispatchActors. Thus new traversals as well as new node types can be added easily.

By characterising the different Actors different optimisations become possible in connection with different Iterators.

An open problem is the parallelisation of the traversals. Breadth first traversals are relatively simple, as only writing access to the state has to be synchronised. Depth first and priority queue traversals also need to ensure the ordering constraints, which limit the freedom of parallel traversals.

Another untouched aspect are hybrid iterators. It can make sense to switch iterators while traversing, e.g. start with a priority queue iterator to render large blocks front-to-back, but switch to a simpler depth first traversal once a certain level in the tree has been reached.

References

- [1] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In Oscar M. Nierstrasz, editor, *ECOOP '93 - Object-Oriented Programming 7th European Conference, Germany, July 1993. Proceedings*, number 707 in Lecture Notes in Computer Science, pages 406–431. Springer-Verlag, New York, NY, 1993.
- [2] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [3] HP. DirectModel, 2000.
- [4] Robert Osfield. Open Scenegraph, 2001.
- [5] Dirk Reiners. Open SG, 2000.
- [6] 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.