Fourth Eurographics Workshop on Parallel Graphics and Visualization (2002)
D. Bartz, X. Pueyo, E. Reinhard (Editors)

A Multi-thread Safe Foundation for Scene Graphs and its Extension to Clusters

G. Voß, J. Behr, D. Reiners and M. Roth

Centre for Advanced Media Technology, Nanyang Technological University, Singapore, Singapore,
Computer Graphics Center, Darmstadt, Germany,
OpenSG Forum, Darmstadt, Germany,
Fraunhofer IGD, Darmstadt, Germany

Abstract

One of the main shortcomings of current scene graphs is their inability to support multi-thread safe data structures. This work describes the general framework used by the OpenSG scene graph system to enable multiple concurrent threads to independently manipulate the scene graph without interfering with each other. Furthermore the extensions of the presented mechanisms needed to support cluster systems are discussed.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Virtual reality I.3.2 [Computer Graphics]: Distributed/network graphics

1. Introduction

Scene graphs or in general graph structures are being used in computer graphics for some time now, as these kind of structures have proven their usability in a variety of application domains. Over the time different scene graph systems have been written, each with its own focus depending on its main application domain. Open Inventor[1] has been primarily used for highly interactive mouse-based applications. Whereas HP's DirectModel and SGI's OpenGL Optimizer [2] are targeted at applications in the CAD area due to their ability to handle free-form surfaces efficiently. OpenGL Performer [3] is currently the most widely used scene graph system in the Visual Simulation and Virtual Reality domain due to its strong focus on speed.

There are still some features that most of today's scene graphs do not provide, and one of the more important ones is multi-thread safe data handling. Performer was the first scene graph to introduce multi-threading, but only in a specialized sense. Multi-threading inside Performer is used to separate the application from the rendering part of the system and to further subdivide the rendering pipeline into culling and drawing tasks. Later versions of Performer introduced the concept of a database thread, used to execute loading operations concurrently and a compute thread to be used by slower simulations. But in general Performer's threading model is fixed to the APP-CULL-DRAW setup and as a result is not easily extendible, as for example only the scene graph structure itself is multi-thread safe, but data referenced by the nodes like vertex positions is not. Later versions of Performer introduced the concept of so called "pfFlux" buffers in order to provide a mechanism for asynchronous threads to change data stored inside the scene graph. Here the frame number of the writing thread is used to identify the different copies of the data stored inside a pfFlux buffer, and to implement a copy on write strategy based on frame numbers [4]. This results in threads sharing the same frame number also in sharing the same copy of the data. Other scene graphs like Optimizer only allow multiple threads on disjoint parts of the scene graph at the same time. Especially modern Virtual Reality applications can consist of several different threads which need to access a consistent database. Each of these threads has different time requirements, for example haptic simulations tend to require frame rates up to several kHz whereas rendering threads run at much lower rates usually between 10Hz and 60Hz, simulation threads tend to cover the whole range of update rates. Depending on the type of thread, data inconsistencies might affect one or more frames. Drawing threads usually regenerate the whole image each frame, so flawed data might corrupt one or a few

images but these errors will be overwritten as soon as the underlying information is valid again. Iterative algorithms on the other hand, like some of the simulation algorithms used, rely on the fact that the results of the previous frame are accurate in order to calculate the current one. In this case the effects of an inconsistency last longer than the inconsistency itself and in the worst case drive the whole system into an unpredictable and unrecoverable state. Another observable trend is the occurrence of Simultaneous Multi-Threading capable micro processors, here multiple ready-to-run threads are kept inside the processor and as soon as one stalls, for example while accessing main memory, a different one is scheduled immediately. Furthermore first multi-core processors like IBM's latest Power4 CPU[5], providing two CPUs packaged onto one chip, become commercially available.

Thus one of the prime design goals for the OpenSG[6] scene graph system was the ability to provide multi-thread safe data structures to the application. The approach taken is described in the following sections.

2. Multi Processor, Shared Memory Hosts

In order to allow multiple threads in a shared memory environment to work, including both read and write access, in parallel on the given scene graph without imposing any restrictions on the order in which these operations might occur, each of the different threads must have a unique copy of at least the data the thread is interested in. Otherwise a reading thread might for example encounter inconsistencies while a concurrent write operation on the same part has not finished. In general the amount of data stored inside a scene graph exceeds the threshold beyond which its treatment as a single entity, of which each thread has a full copy and a complete copy is made while synchronizing two threads, is feasible. In the following sections we describe the approach used by OpenSG to minimize data replication, the copy operations needed to synchronize different threads and its extension from single hosts to clusters.

2.1. Basic Data Structures

On the lowest level data can divided into single value, so called SingleFields and multi value entities, called MultiField in the following. Fields itself are a well known concept and were used before in systems like VRML97 [8], where each field stores one or more values of the same type and provides additional administrative information. Furthermore MultiFields, like C++ Standard Template Library[7] (STL) vectors on top of which they are build, use dynamically sized one dimensional arrays to store their values, thus only references to the actual data are kept inside the MultiField object instead of the data itself. On the next level fields are grouped together in a structure called FieldContainer as shown in Figure 1. FieldContainers, in terms of C++, provide the base class from which the concrete scene graph elements like nodes, groups or geometries are derived.

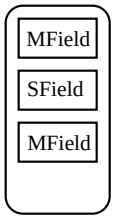

Figure 1: *FieldContainer*

In the following the term aspect is used to describe the copy of the scene graph a thread is working on. The technical reason for the distinction between thread and aspect is that two threads might share the same copy (eg. aspect) safely iff they are guaranteed to work on separate parts or otherwise synchronize each other. In general one thread is bound to exactly one aspect, but one aspect might be used by more than one thread at the same time.

2.2. Data Replication

On top of the data structures introduced in the previous section different mechanisms to replicate the data for each aspect are possible. One way would be to extend the fields. Using this approach each field is able to store multiple instances of its values, one set for each aspect, and, on access, retrieve the right set for the current one. The advantage of this approach is that it keeps the modifications needed local to the field implementation at the lowest possible abstraction level. The disadvantage is that consecutive memory locations are used for different aspects, as the values used by one aspect are interleaved with irrelevant values used by the remaining aspects. This kind of memory fragmentation would negatively impact cache hits, which, in the end would lead to a decrease in the overall system performance. An alternative approach would be to replicate the FieldContainers instead, so that multiple copies of one container are located consecutively in memory. As most FieldContainers exceed the typical cacheline in size, the negative impact of having multiple copies would be minimal compared to the common situation where only one copy is available. As a result of this approach a different pointer must be used to access the correct instance for each aspect . In order to make this approach usable and transparent to the surrounding parts of the system, in a sense that pointers can be passed around as usual and can be used within different threads without the burden of remapping them every time a FieldContainer is accessed through them, a new pointer type, named FieldContainerPointer or FCPtr for short, is introduced to wrap the required per aspect mapping into a single object which easily can be stored and passed around. Each FieldContainerPointer stores the address of the first copy, the base address, and the size of the FieldContainer referenced, as shown in Figure 2. To implement the actual mapping the pointer object needs to be

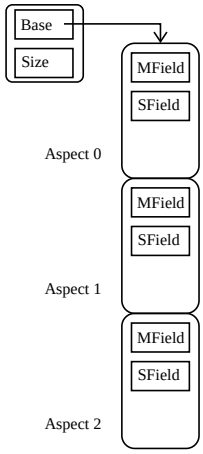

Figure 2: *FieldContainerPointer*

able to retrieve the correct copy for the current aspect. To achieve this an integer index, called AspectId, is associated with each aspect in a way that the correct FieldContainer instance can be returned using the following formula :

$$InstanceAddress(Aspect) = BaseAddress + Size * AspectId$$

The AspectId is the information by which each thread binds to its active aspect, furthermore the same value must be used to access every FieldContainer within a given thread. This opens up the possibility to make use of thread local storage to store this piece of information providing a kind of a per thread global variable which is accessible from everywhere when needed, especially from within the FieldContainerPointer class during the mapping process. Thus the AspectId must not be passed around by the application or surrounding software layers, again keeping implementation details local at the lowest possible abstraction level. Efficiently retrieving the AspectId is one of the key components of the custom pointer performance .

Additionally using the feature of C++ to provide custom implementations of the *pointer-to-member (->)* and *indirection (*)* operators for user defined classes FieldContainerPointers can be used much like built-in pointers, with two notable exceptions. First of all the built-in type conversion facilities of C++ (cast operators) might not be available and must be replaced by custom versions and secondly, in order to ensure that all of the copies are consistent and valid the standard creation (*operator new*) and deletion (*operator delete*) mechanisms are forbidden and must be replaced by customized versions. Within OpenSG we use the factory and prototype patterns to provide the new creation methods and references counting for lifetime determination, a different, simpler way would be to only overload the built-in *new* and *delete* operators. The factory and prototype patterns where chosen because they better fit the extensibility requirements of OpenSG. Creating FieldContainer instances through the cloning of a prototype object allows the application to change the actual object created at runtime by replacing or changing the corresponding prototype. In this way the application is able to change the default values of the objects created or to choose a specialized object, for example a geometry implemetation which is better adapted to the underlying hardware than the general purpose one, to be used throughout OpenSG without the need to recompile the libraries.

For OpenSG we decided to use a combination of both approaches, FieldContainers and the corresponding pointers are the main, high-level wrappers used to encapsulate the replication, but, in order to minimize the amount of data actually copied, a modified, copy-on-write capable MField implementation is integrated, as described in the following section.

2.3. Copying

So far we only presented a method to replicate data nearly transparent to the application, but we did not address the initial problem of avoiding unnecessary copies. The baseline observation on which the copy or better the copy prevention strategies are build is that the data stored inside a scene graph is usually unevenly distributed between the different parts, especially between Single- and MultiFields, for example a typical scene graph for a model of medium complexity might consist of 2,000 nodes (inner and leave nodes). Assuming that each node on average takes up 200 bytes this adds up to roughly 400kbyte of data. The typical number of triangles used for this kind of model would be around 200,000, at 40 bytes per triangle the geometric data needs 8.0 mbyte of memory, 20 times as much. For different examples this ratio will vary, but on average the amount of geometric data will be 10 to 20 times larger than the structural data of the scene graph. As most of the geometric data like points or texture coordinates is stored inside MultiFields this observation could also be used to estimate the ratio between data stored inside SingleFields and MultiFields. Additionally, due to the chosen replication mechanism, only the pointer to a FieldContainer could be stored inside a SingleField not the FieldContainer itself, further reducing the probability of larger SingleFields, as most of the compound structures must be based on FieldContainers to achieve the desired multi-thread safeness. Based on these observations Single- and MultiFields will be treated differently. Of SingleFields full copies, the field and the value stored, will be replicated, as they are assumed to be small enough not to negatively impact the overall memory consumption.

In case of MultiFields advantage of the underlying STL vector implementation is taken to integrate the data-sharing and copy-on-write strategies. STL vector objects do not directly contain the stored values, instead most of the implementations only store references, most of them three, to an external memory location (data block), a one dimensional,

dynamic C++ array containing the actual values. For Multifields replication takes place only for these references not for the whole external data block. As these external data blocks are shared between different aspects, they must be copied if necessary, eg. as soon as a thread starts a write operation to one of these locations it has to receive his private copy. Figure 3 shows the situation where aspects zero and one share the same data block whereas aspect two has its own private copy, caused by a write operation to this field from one of the threads using aspect one. The decision when a write operation, requiring a private copy, is about to happen can either be hidden inside the FieldContainer or left to the application. The disadvantage of trying to hide this decision inside the FieldContainer is that checks have to be added and executed for every function that either changes or returns a changeable reference to a MultiField inside the FieldContainer. In case of changeable reference things get even worse, as a copy has to be made even if the particular reference is only used for reading, as, from the perspective of the FieldContainer, it is undecidable what kind of operations might be executed on the returned reference, reintroducing unnecessary copies. For these reasons it is more efficient to force the application to notify a container explicitly in case a write operation is about to happen, this is how this issue was resolved for OpenSG.

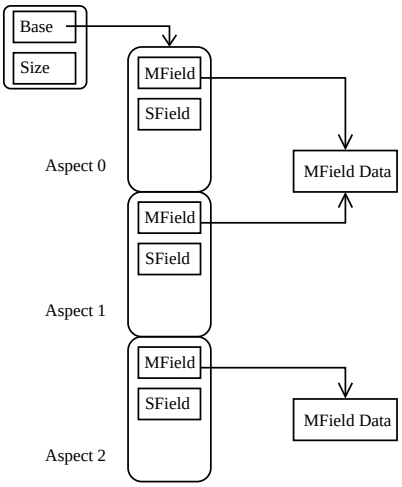

Figure 3: *MultiField data sharing*

During the synchronization of two threads containing private copies, as shown in the next section, one of these copies is thrown away and both aspects will reference the same storage location until one of the threads involved starts writing to this particular field again.

2.4. Synchronisation

Providing separate copies to different threads as they write to their aspects solves only half of the problem. The other half consists of synchronising them so that the results of one thread, stored inside his aspect, are able to influence the remaining thread(s) within the system. During this process data from the source aspect must be copied to the destination aspect. Copying all the data is not suitable at all, as even with the exclusion of the large MultiField data blocks the memory footprint would be enormous since two instances of every single FieldContainer must be touched to achieve a full copy. In practice most of the FieldContainers will not change for every frame. For most applications there will be a static part that does not change at all and some dynamic elements that might change some of their attributes over time. In general only a small part of the scene graph will change, thus it is more efficient to keep track of what was changed and just synchronise these parts. To achieve this the requirement to notify a FieldContainer in case a write operation to a MultiField is about to happen, as introduced in the previous section, is extended in the following ways. First the application is also required to inform the FieldContainer as the write operations have finished. At this point the changed FieldContainer and the information which field was changed is stored in a thread-specific change list. Furthermore write operations to SingleFields must be indicated too. At the point of synchronisation the change list is traversed and only the recorded fields are updated. In case of SingleFields the stored value is copied from the source to the destination aspect, in case of MultiFields the external data block referenced by the destination aspect is discarded and the corresponding references are changed to point to the external data block of the source aspect.

3. Extension to Clusters

As the common off-the-shelf PC based system is getting more and more powerful, one observable trend is to replace the old single large systems by clusters of these common PCs. From the low-level technical point of view covered so far ,cluster systems are similar to the single host setup within the proposed framework, in the sense that data between different locations must be synchronised and with the notion that it is not intended to offer access to data located on a remote computer through the local pointers using their aspect mapping mechanism during dereference operations. As before several copies of the scene graph exist, this time on different computers, and these copies must be synchronized as each node in the cluster needs access to the same data in order to generate a consistent final output image. To fit clusters into the given framework the aspect concept is extended to cover both local as well as remote aspects, whereby remote aspect denominates the copy of the scene graph located on a remote, network connected machine. In order to keep remote aspects up-to-date the local synchronization mechanism is changed in the following way. During synchronization the change list is traversed as usual but instead of executing local copy operations the field content and the information to which field it belongs is packed and send over the network

to the remote machine. On arrival the transmitted information is unpacked and applied to the remote copy of the scene graph. In a similar way changes made to the remote aspect are packed, send, received and integrated into the local copy. Thus allowing distributed copies of the scene graph to synchronise each other.

4. Results

Within this work we presented a general data replication framework to overcome one of the main shortcomings of current scene graph systems, their inability to provide multi-thread safe data access. In order to provide a solution to this important problem a two level approach was proposed. The replication of store data, nearly transparent for the application, was introduced on the FieldContainer level whereas a copy-on-write strategy on the field level was used to minimize the amount of data copied. This copy-on-write strategy was based on the observation that the data stored inside a scene graph is unevenly distributed between Single- and MultiFields. To efficiently synchronise different aspects, a per thread change list in combination with an explicit notification mechanism was chosen. Furthermore the applicability of the chosen synchronisation algorithm in a cluster environment was discussed.

Build on top of the concepts described herein and successfully proving their applicability within in the scene graph domain, the OpenSG scene graph system has been implemented, for additional information see www.opensg.org. Figure 4 shows the cluster version of OpenSG driving a four by four tiled display at the NSCA.

Figure 4: *VW Beetle on a tiled screen driven by a cluster (Model courtesy Volkswagen, Image taken at NCSA using their Display Wall-In-A-Box implementation)*

5. Future Work

One remaining problem of the synchronisation method is that every thread, in principle, receives a full copy of the data. As some threads will only work on a well-defined subset of the scene graph unnecessary copies might be avoided if a per thread filter system is used, allowing each thread to explicitly state in which part it is interested in. This feature is especially useful for cluster environments as transferring data over the network is a quite expensive operation. Furthermore measuring the performance impact of the custom pointer implementation is still work in progress. So far the work done mainly focused on the scene graph domain, but the framework in general might not necessarily be restricted to it. The ideas presented seem to be transferable to other application areas with the need of multi-thread safe data structures, as long as they feature a similar data distribution and allow the use of Single-, MultiFields and FieldContainers.

Acknowledgements

We would like to thank Prof. Dr. Ing. J. L. Encarnação for providing the environment in which this work was possible and our colleagues for providing valuable feedback during the design and implementation phases. Parts of the work presented herein were funded by the German Ministry for Research and Education as part of the OpenSGPlus joint research project in the field of virtual and augmented realities.

References

1. P. S. Strauss and R. Carey. An object-oriented 3D graphics toolkit. *ACM Computer Graphics (Proc. of SIGGRAPH '92)*, 341–349, 1992.
2. sgi. Unleashing the power of sgi's next generation visualization technology. *http://www.sgi.com/software/optimizer/whitepaper.html*, 2001.
3. J. Rohlf and J. Helman. IRIS Performer: A high performance toolkit for real-time 3D graphics. *ACM Computer Graphics (Proc. of SIGGRAPH '94)*, 381–395, 1994.
4. sgi. OpenGL Performer Programmer's Guide. *www.cineca.it/manuali/Performer/ProgGuide24/html/Perf_PG-18.html*, 2000.
5. K, Diefendorff. Power4 Focuses on Memory Bandwith. *Microprocessor Report*, **13**(14), 1999.
6. D.Reiners, G. Voß, J.Behr. *OpenSG: Basic Concepts*. *www.opensg.org/OpenSGPLUS/symposium/Papers2002*, 2002.
7. sgi. Standard Template Library Programmer's Guide *www.sgi.com/tech/stl*, 2002
8. International Standard ISO/IEC 14772-1:1997. Information technology – Computer graphics and image processing – The Virtual Reality Modeling Language (VRML) – Part 1: Functional specification and UTF-8 encoding, 1997.