Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE @ DIRECT•

Computers & Graphics 28 (2004) 63–66

COMPUTERS
&GRAPHICS

www.elsevier.com/locate/cag

Multi-threading and clustering for scene graph systems

Marcus Roth[a,*], Gerrit Voss[b], Dirk Reiners[a]

[a] Computer Graphics Center (ZGDV), Department A4, Fraunhoferstrasse 5, Darmstadt 64283, Germany
[b] Centre for Advanced Media Technology, Singapore

Abstract

The support for multi-threaded applications in current scene graphs is very limited, if it is supported at all. This work presents an approach for a very general multi-threading framework that allows total separation of threads without total replication of data. It also supports extensions to clusters, both for sort-first and sort-last rendering configurations. The described concepts have been implemented in the OpenSG scene graph and are widely and successfully used.
© 2003 Elsevier Ltd. All rights reserved.

Keywords: Multi-threading; Parallel processing; Clustering; Virtual reality

1. Introduction

Graph structures are ubiquitous in all areas of computer science. One group of graphs that has been successful in computer graphics are the scene graphs. Over the last couple of years a number of scene graphs have been developed to manage the scene contents for different kinds of computer graphics applications.

One area that has been inadequately handled by scene graph systems is the area of multi-thread safe data handling. OpenGL Performer [1] was the first system to support it at all, but its implementation is tightly coupled to the pure rendering process. Other relatively easy-to-realize alternatives are parallel work on disjunct parts of the graph, as supported by Optimizer [2].

For many applications, especially in the area of virtual reality (VR), these kinds of multi-threading approaches are not sufficient any more. VR applications have multiple independent tasks that need to access the scene data as stored in the scene graph. These can run independently, and need to, to achieve good performance. An added complication is the fact that they may run asynchronously, between a few Hz for simulations up to kHz for haptics. All these need access to consistent versions of the scene data, as many of the employed algorithms are iterative, and an inconsistency can have an effect for a large amount of time, up to and including making the results unpredictable and useless.

An extension of having multiple threads working on a shared scenegraph is having multiple machines working on a shared scenegraph, better known as clustering. The common off-the-shelf systems are affordable enough such that it is possible to use a cluster of them to drive a single display by splitting the scene graph over multiple machines and compositing the resulting image, which is also known as sort-last rendering [3]. But they lack multiple outputs, which are needed to drive common VR display systems like Powerwalls [4] or Caves [5]. This can be overcome by splitting the image across multiple systems, also known as sort-first rendering [3].

To support these kinds of applications a very general scheme for multi-thread safe data handling has been developed, which is the topic of this paper. Section 2 describes the basic approach used for shared memory multi-processor systems. Section 3 shows how to extend this approach to a cluster system. Results of the described approach as implemented in OpenSG are shown in Section 4, while Section 5 describes the future research directions.

*Corresponding author. Tel.: +49-6151-155-153; fax: +49-6151-155-196.
E-mail addresses: marcus.roth@zgdv.de (M. Roth), vossg@camtech.ntu.edu.sg (G. Voss), dirk@opensg.org (D. Reiners).
URLs: http://www.zgdv.de, http://www.camtech.ntu.edu.sg.

0097-8493/$ - see front matter © 2003 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cag.2003.10.004

2. Shared memory multi-processor systems

The general goal to support multi-threading is to give each thread a personal copy of the whole scene graph. The whole scene graph is necessary, as the thread might want to tear down and totally rebuild the graph, e.g. to initialize or destroy it. Thus, each thread needs to have the impression of having a private copy of the whole graph. The approach to provide this impression is described in Section 2.2. This approach is supported by some basic data structure concepts described in Section 2.1. To reduce the amount of memory needed not all the data is actually copied, as described in Section 2.3.

The final aspect of the multi-thread separation is the combination and synchronization of the separate partial results computed by the different threads, as described in Section 2.4.

2.1. Basic data structuring

A differentiation between two kinds of data structures, based on whether they hold single (SingleFields) or multiple values (MultiFields), is used. Fields provide some administrative information in addition to storing values, similar to the Field concept used in VRML97. MultiFields hold a one-dimensional array of values, similar to the STL vector concept, on which the MultiFields are based. These arrays are dynamically sized and can contain an arbitrary number of values.

Fields are grouped together into FieldContainers, which function as a base class for all scene graph classes like nodes, groups or geometries.

Another important concept is the aspect. An aspect is a separate version of the scene graph, as seen in the different threads. Every thread has a specific aspect associated with it. A 1:1 relation between aspects and threads is possible but not necessary, as there can be cases where different threads are known not to interfere. In these cases they can share an aspect to reduce the resource requirements.

2.2. Replication and access

As described in Section 2 every thread needs to have a private copy of the scene graph to ensure consistency. There are different levels at which this data duplication can take place.

The lowest possible level is the Field. It would be possible to just change the Field to store multiple copies of the data. The big advantage of this approach is the ability to hide all the multi-buffering inside the Fields and present an interface just like standard classes to the user. The disadvantage of this approach is the memory layout. The different copies of the data, which are only accessed by separate threads, lie in memory consecutively. This interferes very destructively with the caching done by modern processors. Only a small part of every cacheline is actually used for productive data of the thread. Even worse, the different parts of the cacheline are accessed by different processors in a multi-processor system, necessitating expensive cache consistency negotiation between the processors.

The next higher level where the replication is possible is the FieldContainer level. This alleviates the caching problems of the Field replication, as general FieldContainers will have sizes close to or exceeding that of a cacheline. The disadvantage in this approach is the necessity for the application to be aware of the multi-buffering, as it can no longer be hidden in the Fields.

This awareness can be reduced by introducing a replacement for pointers to FieldContainers called FCPtrs. The mapping of the data associated with the current thread can be hidden inside the FCPtr, in C++ by overloading `operator*` and `operator->`. This allows an application to use the replicated FieldContainers nearly exactly like built-in pointers.

The operation needed to realize this mapping from FCPtr to the pointer for the actual instance associated with the Thread needs to be very efficient, as it is to be executed a large number of times, for every data access via the pointer. Using a linear data layout, where all the FieldContainer copies are just located consecutively in memory, the necessary operation can be reduced to a single multiplication and addition.

To simplify the mapping operation this far the FCPtr needs to store the base pointer of the FieldContainer block and the size of a single FieldContainer. This allows formulating a mapping as: $InstanceAddress_{FC}(AspectId) = BaseAddress_{FC} + Size_{FC} * AspectId$.

The *AspectId* identifies the aspect associated with the thread. Thus, it has to be the same for every data access in the thread, but differs between threads. This access pattern makes the *AspectId* a perfect candidate for storing it in thread-local memory (TLS), which is supported in a very efficient manner by all modern operating systems. This relieves the user from the responsibility of keeping and passing around the *AspectId* and automatically ensures consistent use of a thread's aspect.

2.3. Reuse

The concepts described so far allow the replication of data and the separation of the different threads. But they replicate all the data and thus incur an unacceptable memory penalty on the system.

An analysis of the memory distribution for typical scenes shows that the most significant part of the data is concentrated in the geometry data fields and textures. These vertices, normals and texture coordinates, together with the texture images used consume about 90% of the memory, with the rest divided between the nodes and additional structures like materials. An interesting property of these large data fields is that they are all MultiFields.

As described in Section 2.1 MultiFields store their data in one-dimensional arrays of arbitrary length. To ensure that the actual data is not stored in the MultiField, but rather used as a pointer to the data, allows a very efficient optimization.

Instead of replicating the MultiField data for every aspect, it is shared between them by having all of them point to the same data. Only when a thread starts actually changing the data a copy is created for this thread. This way only the data needed by the thread is replicated, significantly reducing the memory footprint of the system.

The difficulty in this approach is the detection when a thread is about to change the data. This could be hidden in the access functions, hiding it from the user. But by doing that the overhead of testing whether a thread already has a private copy of the data and creating it if necessary is incurred on every access. For the usual MultiField operations with a large number of elements being changed this overhead is too expensive. A more efficient approach is putting the burden of declaring the intention to change data on the application, which it can do exactly once and just before the change is done.

2.4. Synchronization

Splitting the data is only one part of the problem. Sooner or later the different copies of the scene graph need to be synchronized to have all threads work on the same data again, otherwise they could not communicate and interact with each other.

Even if the sharing of MultiFields reduces the amount of data to be copied, just blindly copying everything that is not shared is going to be inefficient. In most cases a thread will only change a rather small part of the scene graph. Thus it is only necessary to copy the changed parts, reducing the amount of work to be done to what is needed.

Finding out what has changed can be done with an extension of the application change announcement for MultiFields described in Section 2.3. In addition to announcing the MultiFields to be changed, also the SingleFields that are going to be changed are mentioned in the system. After the changes are done the application informs the system of that fact, again stating the Single- and MultiFields changed.

At this point the changes are recorded in a ChangeList, in the form of the changed FieldContainer and a mask listing the changed Fields. When a synchronization is triggered by the application, the ChangeList is traversed and the recorded Fields are copied to the aspect that should be updated. For SingleFields this is done as a simple copy. For MultiFields the copy just copies the pointers to the actual data and thus is very efficient. If the MultiField had a private copy of the data this copy is deallocated first.

The ChangeList approach also allows operations on the change information like merging and compression, which allow very flexible change propagation and management.

3. Clusters

Another advantage of the ChangeList approach is the fact that it extends quite naturally to clusters of machines.

The ChangeList contains all the necessary information about what changed in the scene graph. To propagate these changes across a network to a different machine all that needs to be done is to replace the pointers in the list with a network-transparent id and to add the actual data that has changed, which is pretty easy to get from the changed Fields.

On the other end of the network the ids are replaced by pointers again and the data are copied into the Fields. After this is done the scene graphs on both ends of the network are identical and the different machines can work in a synchronized manner.

If all the cluster machines need to have a common, complete copy of the scene graph, the data distribution can be done using a Multicast protocol, making the amount of data to be sent independent of the cluster size and increasing scalability.

The described communication method is rather general and allows a wide variety of data distribution structures. Two of the most common ones have been implemented to support two different scenarios: split and single display rendering.

3.1. Split display clusters

For a split display rendering, the resulting image is divided into parts, one of which is assigned to each cluster machine. This splitting approach is also known as sort-first rendering. In many cases the division is regulated by the physical constraints of the used display, for example the tiled display in Fig. 1 assigns one projector to each cluster node, with a separate part of the image per machine.

This is a typical case, where the calculated partial images are combined on a projection screen, removing the need to read back the partial images to combine them to a complete image, significantly reducing the network bandwidth needed.

This is not possible for single display clusters.

Fig. 1. OpenSG driving the 48 node HEyeWall tiled display.

3.2. *Single display clusters*

In single display clusters the partial images are combined to a single result image that is displayed on a single display (e.g. monitor). This actually necessitates concentrating the parts in a central location. If a special video combiner network is not available, this has to be done using the standard communications network, putting a significant bandwidth load on it.

Single display cluster can be driven sort-first (i.e. split the image into disjunct parts), but in general it is more efficient to instead split the model into disjunct parts and combine the resulting images of the partial models. In general, these partial images have to contain a depth buffer to make sure that they can be combined in a visibility-consistent manner. Another advantage of this approach is the ability to handle models that are larger than a single-machine memory, as every machine needs only a part of the model's data.

The described system supports this kind of distribution as well. This can be done through smart handling of the ChangeLists, splitting them and sending only the parts that are necessary for every specific cluster machine. The problem with this approach is the necessity to keep all the data on the master machine, limiting the possible model size.

An alternative is the use of a ProxyGroup node in the scene graph, which contains a URL, identifying the file that should be loaded to the Proxy's position in the scene graph. Only when rendering is the actual data loaded and displayed. This allows the master machine to keep just the lightweight ProxyGroups, while the actual data is only loaded on the servers.

4. Results

We have presented a framework for adding very general thread-safe data structures to a scene graph system. It uses selective replication of data to give every thread the illusion of owning a full copy of the whole scenegraph without the memory penalty of actually doing so. Synchronization is handled via ChangeLists, which can also be used to synchronize scene graphs on remote machines, supporting cluster-based rendering in sort-first or sort-last fashion.

The described concepts have been implemented in the OpenSG Open Source scene graph system [6] and are used in a number of sites to run cluster systems as small as three up to 48 machines.

5. Future work

The presented system assumes that every thread has an interest in the whole graph. This is not always the case, resulting in parts of the graph being updated that are never accessed by the thread. Some research into ChangeList filters should be able to make this more efficient.

Another area is the sort-last clustering, which currently is severely restricted by the available network bandwidth. Some significant optimization opportunities exist in a deeper analysis of which parts of the image need to be transfered, whether the depth buffer is really needed or not.

Acknowledgements

This work has been supported by the OpenSG PLUS project funded by the German Federal Ministry of Education and Research (bmb + f).

References

[1] Rohlf J, Helman J. IRIS perfomer: a high performance multiprocessor toolkit for real-time 3D graphics. Computer Graphics, SIGGRAPH'94 Proceedings; Orlando, 1994.

[2] Silicon Graphics, Inc. Unleashing the power of SGI's next generation visualization technology; http://www.sgi.com/software/optimizer/whitepaper.html.

[3] Mueller C. The sort-first rendering architecture for high-performance graphics. Proceedings of the Symposium on Interactive 3D Graphics; Monterey, 1995. p. 75 ff.

[4] Woodward P. The Powerwall; http://www.lcse.umn.edu/research/powerwall/powerwall.html.

[5] Cruz-Neira C, Sandin DJ, DeFanti TA. Surround-screen projection-based virtual reality: the design and implementation of the CAVE Computer Graphics, SIGGRAPH'93 Proceedings; Chicago, 1993. p. 135–42.

[6] Reiners D, VoB G, Behr J. OpenSG—basic concepts. Proceedings of the OpenSG Symposium 2002; Darmstadt, 2002.