OpenSG: A Scene Graph System for Flexible and Efficient Realtime Rendering for Virtual and Augmented Reality Applications

Vom Fachbereich Informatik
der Technischen Universität Darmstadt
genehmigte

DISSERTATION

zur Erlangung des akademischen Grades eines
Doktor-Ingenieur (Dr.-Ing.)

von
Dipl.-Inform. Dirk Reiners

aus Bergisch Gladbach

Referenten der Arbeit:	Prof. Dr. José L. Encarnação Prof. Dr. Stefan Müller
Tag der Einreichung:	5. Mai 2002
Tag der mündlichen Prüfung:	21. Juni 2002

D17
Darmstädter Dissertationen 2002

Acknowledgements

This work wouldn't have been possible without the continuous support and encouragement from a number of people.

I would like to thank Prof. Dr. José L. Encarnação for creating and maintaining an environment that encourages experimentation and free research without letting people drift too far off track, but keeping them focussed on what needs to be done.

My department head Stefan Müller was always there for me and kept believing in me even when I didn't. Which can be very annoying, but sometimes is just what's necessary.

Many ideas were formed and formalized in intensive discussions with the rest of the OpenSG dream team, Gerrit Voss and Johannes Behr, without whom many mistakes would have been made and many good aproaches ignored.

Many ideas were fine-tuned in discussion with other members of the graphics and later OpenSG community. Allen Bierbaum and David Blythe were instrumental in getting everything started.

All my colleagues from the A4 department were helpful in making even the hard times fun and enjoyable. Their influence on the overall work environment and general motivation can not be overestimated.

Dirk Reiners
Darmstadt, May 2002

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Virtual reality has been on the buzzword list for a long time. And it has actually started to live up to its expectations. It is spreading throughout the manufacturing industry, in Germany primarily the car industry, and is bringing in returns of invest. Applications in design review, simulation visualization and presentation are being used by more and more users, and the large car companies are building virtual reality labs every few hundred meters. The future looks bright for VR.

But hand in hand with the increased numbers of users come new expectations. It is no longer enough to just present some nice models, people want to actually work with the technology and want it integrated into their work-flow. They want their usual simulation tools integrated into the VR environment, and they want to be able to use it on their deskt, not only in the lab. The step to more users also means that it has to become cheaper. Not every company can afford to have a powerful graphics workstation from Sun, HP or sgi for every user. The smaller companies might not even be able to afford the one large machine that is needed to drive the one VR laboratory.

And they don't have to. Due to the increasing demand for graphics power in computer games, the graphics power of low-cost, low-end PC platforms has been growing at an amazing pace, by now easily overpowering high-end graphics workstations from a few years ago. Similar developments have happened in the general computing area. Todays gigahertz processors can reach impressive performance levels, dual-processors systems are becoming commonplace, and multi-core processors are appearing in the time-lines of the processor developers. The current systems can't solve every problem yet, the PC platform still has to catch up in some respects to the large workstation world, but in many cases they can easily compete. Thus the world for VR software has been getting ever more complex and heterogeneous.

But at the same time the development in one of the core areas of virtual reality software, the basic scenegraph layer that manages the geometric data and transforming it into images, has been stagnant. Many attempts at introducing a powerful new scenegraph standard have failed (Cosmo/Optimizer, OpenGL++, Fahrenheit, DirectModel, etc.), so now developers are basically left with the choices they had six years ago: using Performer or rolling their own. The only new contender is Java3D, but the performance penalties associated with Java diminish its viability for many applications. Inventor is still around, but it never really was a serious option for a high-performance VR application.

Performer has been evolving over the last several years, but careful evolution, being forced not to break compatibility too badly, can only carry so far. At some point there is a need to step back and take a broad look at the problem to judge if it is still being solved.

And the demands on a scenegraph for VR applications have changed significantly. Flexibility is ever more important, due to the growing number of different applications that are migrating to a VR environment. Flexibility is even more demanding from the technical side. Just rendering a bunch of polygons is not enough anymore. New qualities of rendering are needed, and the rapidly evolving graphics hardware can deliver them. But the rapid evolution also means that a system designed today can not foresee what the environment it will have to live in in a few years will look like. It has to be able to adapt significantly to still be useful and be suitable for future demands and possibilities.

Together with the growing push for flexibility comes the constant demand for more performance. The single-threaded program model hasn't been able to fulfill that in the workstation market for a long time, and hardware needing multiple execution threads to fully utilize it is becoming commonplace. A scenegraph system has to be able to support that, not only for rendering but also for other tasks that are to be executed on the geometric data that it ke.

Combining flexibility and performance is needed for the central task of a scenegraph, transforming the geometry into images. The rapid development in graphics hardware is by no means limited to better performance. New features have been built into the systems, and are going to be invented in the future, so that a large amount of flexibility is needed to drive the coming graphics systems at best performance while using them to their fullest extend.

1.2 Topic and Structure

The topic of this work is to define the demands for a scenegraph for the foreseeable future and solve the problems that these demands incur, which are not solved by currently available systems.

Chapter 2 gives an overview of the state of the art in computers, computer graphics hardware and scenegraphs, and analyzes them. It also tries to extrapolate the current trends into the future to define a set of goals that a scenegraph that should be used for long-term VR projects has to reach. Three areas are identified to be the key problems that are imperative for a useful and successful scenegraph system: extensibility, handling of parallel tasks and flexible and efficient handling of graphics hardware. These are detailed in the following chapters.

Chapter 3 analyzes and compares the different approaches to provide extensibility for a scenegraph. Extensibility includes the ability to extend the system in such a way that a new application or system extension can not only be used very easily, but is also able to extend already existing programs to benefit from new developments and extensions without having to be changed. The highest goal is to be able to create systems and tools that can not only use new features as a replacement for the old ones, but also use them and manipulate them natively. A solution to fulfill all these demands is developed.

Chapter 4 analyzes the different ways of executing parallel tasks in a scenegraph system. The big question is if it is necessary to replicate the scenegraph data for parallel tasks to work or not. An analysis of which kinds of tasks can be done without replicating data is conducted. Different alternatives for doing the replication and synchronization of data are developed and evaluated. As an especially important extension case the handling of a cluster of machines for rendering is detailed.

The central and most important task, while not the only one, of a scenegraph is the transformation of geometry into images. Chapter 5 handles the specific requirements that a scenegraph has to fulfill in order to flexibly and efficiently handle the graphics hardware that does the actual conversion.

One important aspect is the handling of the geometric data. It has to be flexible enough to support many different applications and adapt itself to be integrated into other systems, but at the same time it has to exploit the graphics hardware as much as possible.

The other main aspect of hardware is the management of the state of the system. Here the conflict is between giving the user an abstract, efficient interface to define surface properties, while at the same time giving him the flexibility and power to use new features as effectively as possible, and the need to manage the state in a way that reduces costly state changes, without itself costing too much time. These problems are split into state handling, which is concerned about being able to manage a possibly changing set of graphics state and reduce state changes, and material handling, whose task it is to abstract the internals of the graphics library and provide a useful interface to the rendering properties to the user that abstracts the specifics of the actual hardware and emulates features that are not supported natively as far as possible. An integrated concept solving these problems is developed. A specific set of problems appears as soon as multiple non-coplanar screens are used for display, e.g. in a CAVE environment. Consistency across screens becomes imperative and raises problems with assumptions of the graphics libraries. Efficient solutions to these problems are described.

Chapter 6 gives an overview of the applications that have been developed using OpenSG, the system that was developed in the course of this work.

Chapter 7 summarizes the results of this work in the area of extendibility, parallelism and flexible and efficient graphics handling, and goes on to show the areas that pose still open problems for future research.

1.3 Main Results

The analysis of the microprocessor state of the art in chapter 2 predicts that parallel processing of multiple independent threads will be ubiquitous soon, either as separate processors or in a single chip. On the graphics hardware front performance will continue to rise faster than processor performance, but more importantly programmability will continue to become more common and the need to differentiate themselves will drive the hardware vendors to keep adding unique features to their systems, demanding high flexibility and extendibility from the scenegraph systems. The analysis of currently available scenegraph systems shows that three areas are not adequately covered:

- extensibility
- handling of parallel tasks
- flexible and efficient handling of graphics hardware.

Extensibility A set of data structures is defined that can give information about themselves, coupled with methods to manipulate that data. Together with a simple to use interface for defining these structures interactively and creating them automatically, building a very generic and efficient system is made possible. The replacement of internal components by versions that are better suited to the task or hardware/software environment, even at runtime, is achieved through the *dynamic combined use of generative patterns*, namely the Factory and Prototype patterns.

The flexibility also extends into the specifics of the scenegraph, the nodes and leaves that define the graph, and the methods of traversing this graph. The developed node structure is able to combine the benefits of simple data sharing for efficient replication of scenegraph parts with the usability and consistent node identifiability of single-parent systems by the use of a *node-core split*. The flexibility designed into the graph structure demands equal flexibility in the active parts of the system, the actions that traverse the graph. The design developed in this work is able to efficiently handle the extensibility constraints that the abovementionened structures demand and furthermore supports flexible extension and replacement of node-specific actions itself.

Parallel Processing Some tasks can be usefully handled in parallel without replicating data, primarily parallel traversals that do significant work on a single node, but in general data replication is needed to allow multiple concurrent tasks to work together without interfering. The analysis shows that the *replicated field container structuring* with *change-list based synchronization* defines the best synthesis of ease of use and caching behavior. It has also been extended to handle the special case of a *distributed cluster for multi-screen or large screen rendering* in an easily integrated way.

Graphics Hardware Handling The *GeoProperty* abstraction to define the geometric data developed here is well suited to the OpenGL graphics library that is used to drive the hardware, as well as providing the flexibility to adapt to the different specifics of the different hardware systems and the specifics of the applications using them.

The state handling and state minimization complexity problems are solved by the definition of *state chunks* that cover a subset of the graphics state and allow efficient handling of the whole state. The *material as the rendering controller* concept allows the abstraction of techniques that go beyond the direct capabilities of the graphics library, like multi-pass techniques or techniques involving temporary images. bringing both of these concepts together in a flexible and efficient manner is done by the *draw tree*. It is a temporary graph that captures the information for the current frame and allows out-of order definition of subtasks as well as supplying the framework for efficient state change minimization.

A problem area that is gaining use and importance is the use of multiple non-coplanar projection screens. Some assumptions about coordinate systems in the graphics library conflict with the strict demand of cross-screen continuity that is imperative for using these systems. By splitting the usual two-step transformation pipeline of OpenGL into three st and varying the association of the third step it is possible to create *unified* as well as *split viewer coordinate systems*, which allow a finer adaption to the restrictions of the graphics library and solving the continuity problem.

OpenSG The results of this work have been realized in the OpenSG system. OpenSG is a freely available scenegraph that has been used in a number of projects and has proven that the concepts described here are viable and practically useful. These examples cover the range from simple applications that benefit from the simplicity of integrating extensions into the system, through medium-size systems that integrate external components to full-fledged Virtual Reality systems. The daily use of these systems demonstrates the viability of the concepts developed in this work.

Even though a number of solutions to important problems are proposed and described in this work and some have been realized in the form of OpenSG, the book on scenegraph systems has by no means been closed. On the contrary, the availability of the system described in this work opens new areas of research that will be outlined in section 7.

Chapter 2

State of the Art and Expected Future Trends

This chapter describes the state of the art and expected future trends in computing in general and especially graphics hardware, as far as it concerns Virtual Reality applications. It then takes a look at current scenegraph systems and analyzes how effectively they use the current hardware and how effective they will be on the anticipated hardware. A summary listing the shortcomings of the current systems and thus the motivation for this work closes the chapter.

2.1 Processors and Computer Systems

The main rule to judge the development of microprocessor systems, that has proven itself to be surprisingly reliable over the years, is Moore's law[126]. It predicts that the performance of microprocessors doubles every 18 months (24 months originally). This is mainly due to increases in clock frequency, but also increased parallelism. There are no signs that the validity of this law will cease for the next couple years, processors will increase in speed at a steady pace.

One of the major problems actually utilizing this power stems from the fact that the processor is not the only part that influences performance. The other significant part is the memory subsystem. Memory speeds, as compared to processor speeds, have grown comparatively slowly (see fig. 2.1). Thus the processors are often starved for instructions or data and have to wait for the memory, not being utilized to their fullest.

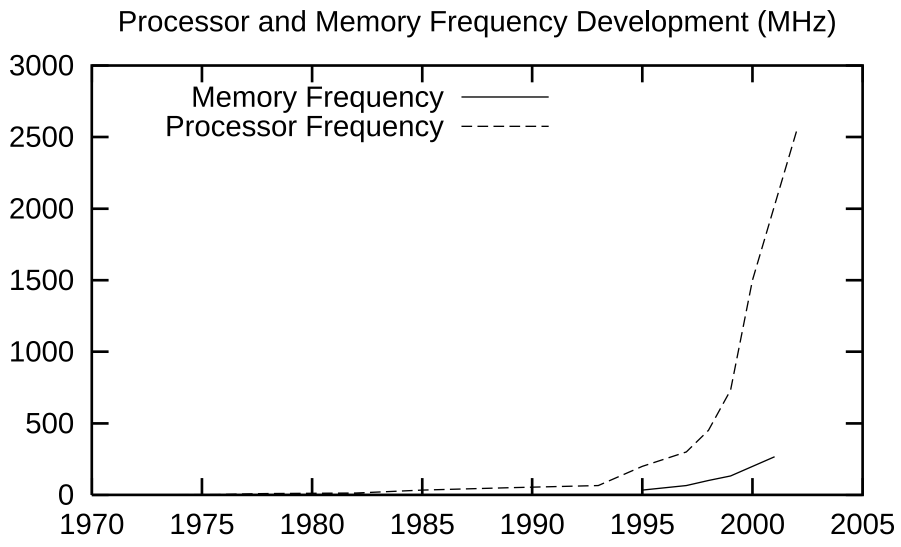


Figure 2.1: Processor Frequency vs. Memory Frequency

One way to alleviate that problem is to add caches to the processor, which keep the least recently used data and instructions on the chip and thus in fast reach. These caches have been growing steadily with the increase in processor frequency, to prevent memory stalls and processor underutilization. The main problem with caches is their size. Caches nowadays take a significant part of the chip space, thus forcing the processors to be bigger, more expensive and harder to produce.

An alternative approach that is getting increasing attention is supporting multiple threads of execution on the chip at the same time. This demands a duplication of all registers, program and status, and a fast way to switch between the sets. As current CPUs have very large register files for speculative execution anyway, adding duplicates of the actually visible registers is a small expense. Thus if a given data element is not in the, now smaller, cache and has to fetched from main memory, another thread whose memory access should hopefully be finished by now, can run. Thus the processor can use the waiting time for memory to do useful work. The strongest implementation of this idea has been realized by Cray in the form of their MTA (Multi-Threading Architecture) machine[4] (see fig. 2.2). It has space for 128 threads on chip, and to free space for all these registers, it doesn't have data caches, none at all. It depends on always running a large number of threads to hide all memory access latency, which for the applications the machine is designed for

Figure 2.2: Cray's MTA system

seems to work quite well.

Of course all these multi-threading approaches only work as long as there really are multiple threads to be executed. These don't necessarily have to be associated with the same process, but for the software system, e.g. the VR system, to fully utilize the processor, they'd have to. Very few existing scenegraph systems support this (see sec. 2.3).

2.2 Graphics Hardware

Processor speed increases according to Moore's law, doubling every 18 months. This is considered extremely fast in comparison with other industries. Graphics hardware speeds in the PC arena increases twice as fast.

When looking at the number of triangles that can be processed per second (see fig 2.3) they have risen from about 200,000 in 1996 to 50,000,000 today. This is a factor of 250 in 6 years, or an increase of 4 every 18 months. This speed was made possible due to the manufacturers using more and more up-to-date processes and bigger and bigger ch. The current 3D graphics flagship, nVidia's GeForce4, hosts more than 63 million transistors [132](see fig. 2.4). In comparison, the current processor flagship, the Pentium 4, only has 42 million transistors [81]. 3D graphics is becoming ever more pervasive. It's hard to find a graphics board nowadays outside the server market

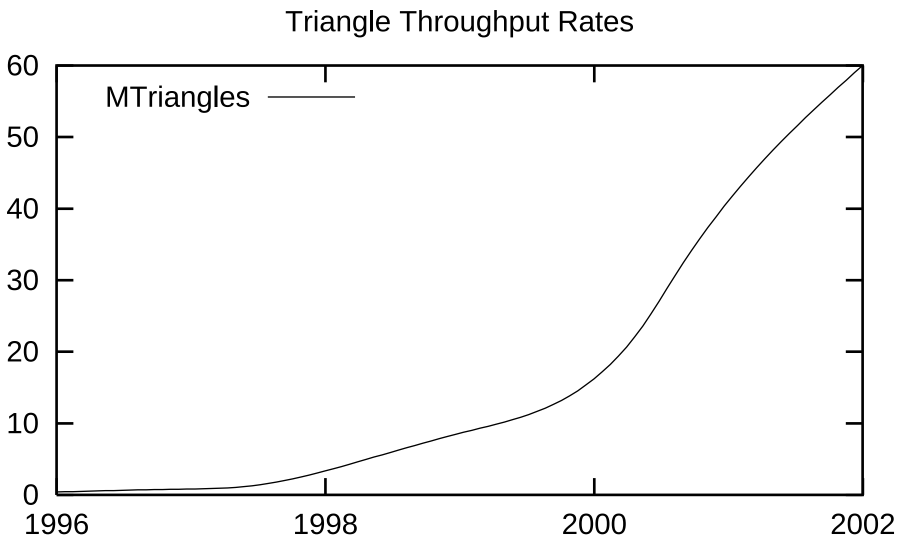


Figure 2.3: Triangle throughput rates

Figure 2.4: GeForce4 graphics processor

that doesn't have a 3D-capable chip on it. The economies of scale allow very high performance in and large development efforts for a cheap product.

Workstation manufacturers are trying to benefit from that trend, as their much lower unit counts put a lot of pressure on them to keep development costs down to not lift the product price into regions that no customer wants to accept. Thus systems based on the use of multiple commodity parts have been announced by sgi and other companies.

The incredible increase in power of the graphics ch will not continue indefinitely. The current ch already reach the top of the technological line, and thus from now on they will be linked to the general technological speed, which is slower than what the graphics companies are used to.

Additionally graphics ch are hitting the same bottleneck processors are hitting: memory speed. They still do have to do more work per memory access than processors for geometry, but texture and framebuffer accesses present a significant bottleneck. To sensibly use all the processing power they have the amount of work done per vertex or per pixel can be increased, resulting not in ever more polygons, pixel and texel, but in higher quality. In the vertex area that includes programmable vertex pipelines that allow user-defined programs to be run on the vertices and higher-order surface tessellation on chip. In the pixel area programmability is starting to appear, in a very limited range, but more flexible blending and texture combination modes are becoming commonplace. This is an area where the graphics chip companies leave the path that has been laid out by the workstation vendors like sgi before them [3, 1, 2, 76] and enter new territory. As a consequence, different manufacturers try different alternatives on which kinds of features to add to their ch to differentiate themselves from the others and to gain market share and support. For example current nVidia hardware supports 61 extensions, 17 of which are nVidia-specific [79]. Other companies have less, but still a sizeable amount. The playing field is widening, and it is not clear who, if anyone, is going to win.

As the primary task of a scenegraph is rendering the scene, these developments are very influential on the requirements of a scenegraph for VR applications. The increasingly different feature sets of current ch force it to be very flexible about extending itself to support a wide range of hardware to its fullest. Furthermore the use of multiple commodity ch in a single system forces the support of multiple independent rendering pipes, and different ways of their integration into a full system.

2.3 Scenegraph Systems

There have been a number of scenegraphs written for VR or VR-like applications over the years. The two oldest and best known are Open Inventor [117] and OpenGL Performer [99]. They were supposed to be succeeded by Cosmo3D [109] and it's high-level companion Optimizer [109]. A system a little younger than Performer is the Y system [89] used in IGD's VR system than is now marketed by vrcom under the name Virtual Design 2. A new contender with a somewhat different spin is Java3D [69]. They all have their pros and cons.

2.3.1 Inventor

The oldest but still in use in a significant number of applications. Inventor was one of the first C++ scenegraph systems and is designed with a strong emphasis on object-orientation. Its main strength is the set of tools to build an interactive graphics application that comes with it. It is very easy to write an application that has complex interactive elements like manipulators within hours (see fig. 2.5).

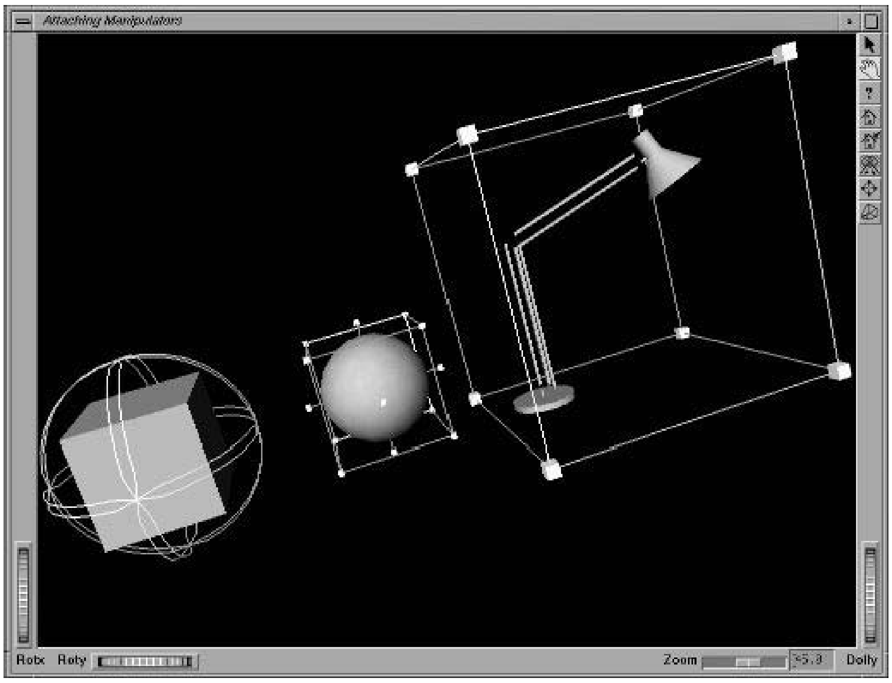


Figure 2.5: Open Inventor Manipulators

Its main weakness in the context of a VR system is its performance. The strongly traversal based attribute structure makes it very efficient memory-wise, but extremely hard to optimize. Furthermore it has no support for multi-threaded operation and data separation between threads. As a consequence of that it has no support for multiple rendering pipes (which would need multiple parallel processes to be used efficiently).

2.3.2 Performer

Figure 2.6: Performer Town

Nearly as old as Inventor, but with a very different mindset behind its design, is Performer. Where Inventor is meant to be object-oriented and easy to use, Performer's goal is much simpler: speed.

Performer was built to get the highest possible performance out of the sgi machines. The main target application were in the VisSim area (see fig. 2.6), and this is were Performer still has a very strong standing and special unique features like light-points and extremely large texture support. It was also the first system to employ multiple processes to divide the rendering task into a parallel pipeline, the famous APP-CULL-DRAW division. APP is the application process that handles user input and any kinds of simulation that need to go on in the world. CULL is the culling part that decides which parts of the (potentially very big) world are actually visible, and the DRAW part does all the graphics hardware access and the actual rendering. In later times the division was split a little further into separate processes for intersection and database access. It was also the first system to utilize multiple graphics pipes simultaneously to display different parts of a scene (for wide-screen projection systems).

Performer is a good system for fast rendering, one of the few that can handle multiple processes and multiple rendering pipes. But it is not flexible enough to support the current and future kinds of applications. The roles of the processes are pretty fixed, and it is hard to change them or to add new process types. Furthermore Performer has only limited facilities to add new graphics hardware features, it is possible to add callbacks to nodes that are called during the DRAW phase. But these callbacks disable the state sorting functionality, which is needed to get high performance from highly integrated graphics hardware, and as such are only useful for rare cases.

2.3.3 Y

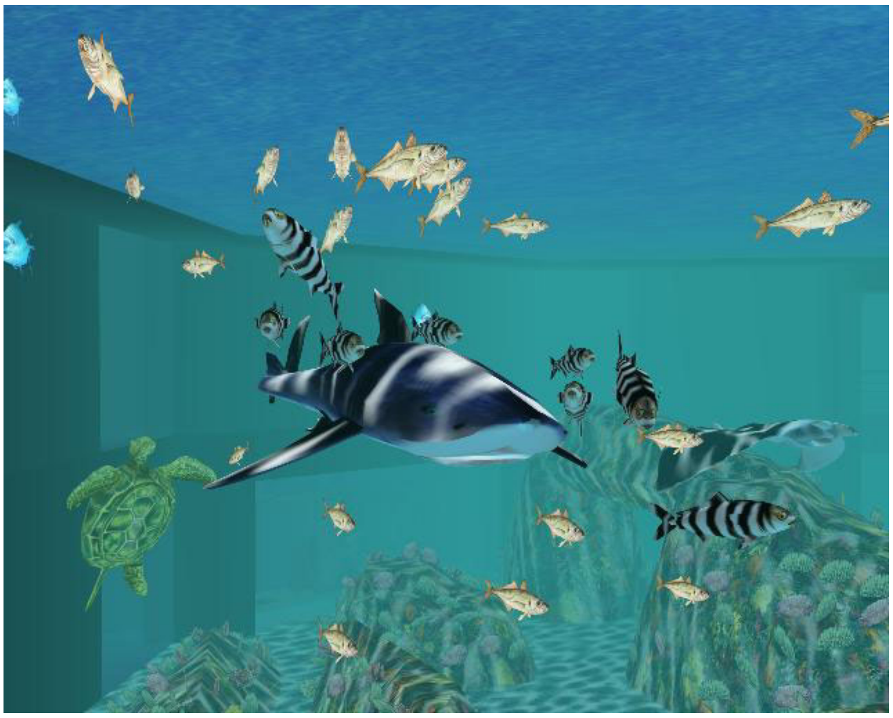


Figure 2.7: Virtual Oceanarium, using the Y system

A little younger and very similar to Performer, but less focused on VisSim applications is Y. It also uses multiple processes to support multi-pipe applications in an APP-CULL-DRAW division. It can use the multiple pipes to draw every kind of mono or stereo projection system and is fairly flexible, supporting a number of high-level effects like projected textures and shadows (see fig. 2.7).

Its main weakness is the difficulty to add new features to it and the C-only interface. It is also very rigid in its use of processes, anything beyond APP-CULL-DRAW is hard to do, in addition to resting the responsibility for multi-thread data safety quite strongly on the application.

2.3.4 Java3D

Java3D is the latest addition to the scenegraph arena. As the name implies it is a Java library. It features extensive support for projection systems, but no multi-pipe support. Furthermore it doesn't support multi-thread data safety beyond that of Java itself, which is just locking. Another weakness is the Java interface, which makes low-level access very costly and not applicable to real-time applications. It it also rather closed about extensibility for new graphics features.

2.4 Summary

This chapter analyzes the current state of the art and expected trends in computer hardware and especially graphics hardware. Given the growing gap between processor speed and memory speed, supporting multiple threads of execution on a single chip to hide memory access latencies will become commonplace.

Graphics hardware is evolving rapidly, even more so than processors, and devel new features and increased flexibility and programmability at an ever increasing pace. Envisioning what the hardware will be capable of in a few years is nearly impossible.

Most of the currently available scenegraph systems for VR applications don't support multiple independent threads working on the scenegraph and have limited extensibility in terms of adding new and replacing old structures and adjusting to new graphics hardware.

Thus there are three areas where existing systems fail to meet the demands of Virtual Reality applications: extensibility, parallel processing and flexible and efficient graphics hardware handling. The solutions for these problems will be described in the following chapters, beginning with extensibility.

Chapter 3

Extensibility

The computer field has always been highly dynamic. Processors double their performance every 18 months, memory sizes reach the sizes of harddiscs a couple years ago, while harddiscs reach never-imagined sizes easily. The rate of change in the computer graphics area is even faster. At the same time the realm of attackable problems expands, and existing problems grow in size to still exhaust all available capacities.

It's impossible to design a system now that includes all the possible features for the applications that it will have to handle in a couple years. There are different ways of handling this problem.

The easiest is to ignore it. If it's impossible to know what to do one can just ignore it and live with the fact that there will have to be changes to the system when the need arises. The inherent danger in this approach lies in the fact that even simple problems might demand changes to large parts of the system to be solved.

Thus it is better to be aware of the inability to support everything from the start and instead design for change. The best way to do that is to make sure that the system itself does not depend on special features that are internal and not accessible to externally built modules, i.e. except for the very core all modules of the system should only use generally accessible interfaces. This golden rule is the transfer of the micro-kernel idea from the operating system area into the scenegraph realm.

Apart from this basic design rule, design for change demands other abilities to be put into the system. To be able to efficiently extend a system it should be possible to write tools that can not only handle the data structures that were present when they were written, but also new structures added much later. This is especially important for generic components like file loaders and writers, but also for debugging tools.

The ability to write tools that can work with structure that are added to the system at a later time needs information about those structures. C++ does not have the ability to query structures about their contents. Thus the structures themselves have to be able to give this information, they need to know about their contents and be able to give this information to the outside. This aspect is handled in section 3.1.

Similarly useful is the ability to control the creation of new structures. This includes the creation of known structures and replacing them with new structures, e.g. to force the system to only use a new type of geometry that is more appropriate for the task at hand. It also includes the ability to create new structures only knowing their name and manipulate them using the methods described in 3.1. This creation handling is described in section 3.2.

It is not always necessary to really derive new structures from old ones. For many applications the ability to add information to existing structures that does not change their general behavior is enough. The way to handle this is described in section 3.3.

Central to a scenegraph is the design of its node structure, i.e. how children are handled and what relations exist in the graph between nodes. There are a number of alternatives here, each with their pros and cons, which are described in section 3.5.

So far this chapter only talks about the passive data structures and their abilities to be manipulated. Just as important is the flexibility and power of the methods to traverse the scenegraph and act upon it, which is described in section 3.6.

3.1 Basic Structure

A general prerequisite for a system that is to be extended and support tools that can work with it is the ability to access information about the structures used and the structures themselves. This capability has been termed reflection [62].

There are several parts needed in reaching that goal: information about which elements make up a structure, information about the different elements and the ability to change them.

Data structures like `struct`s and `class`es are an aggregation of primitive elements of different types. Sometimes not only a single element of the given type is needed but multiple elements. These can be handled as arrays, but if the number of elements needed is not known at compile time, a more flexible way of handling multiple elements is provided by STLs [116] `vector`s. The handling of these primitive elements is described in sec. 3.1.1.

These fields are then composed into larger units, comparable to classes, called field containers. They are the basic element of the scenegraph, all other structures are derived from them, and they provide the basic services the system needs from its structures. They are detailed in sec. 3.1.2.

3.1.1 Fields

The information in a scenegraph is naturally divided into fields, e.g. color of a material, position of a vertex etc. These come in different kinds, most basically single value and multi-value fields. Single value fields can keep a single instance of a specific kind of value and allow access to it. Multi-value fields keep a vector of values of a single type, allowing an arbitrary number of values to be organized together. The scenegraph will internally use a number of types of fields for different purposes, e.g. vector values for geometry normals, color values for vertex colors etc., but it should be possible to add new types of fields to the system in new modules, to allow consistent extensibility and to follow the golden rule of giving the same tools to users of the system as to the system itself given in section 3.

The fields need to be able to identify themselves and give access to the access methodologies defined below. Identification can be handled by the built-in C++ `typeof` construct. But this construct is closed, so it is not possible to extend it to add the access functions described below, thus a separate type needs to be added to the field. As this information only concerns the field class, i.e. every instance of the field in the same way, it doesn't have to be stored in the actual field, which would take up memory needlessly, but instead it can be stored inside field class as a static member.

Beside the programmatic parts that know the type of the field they access there are also tools that not necessarily know it. To facilitate access to the different kinds of fields for generic tools there should be a generic way to access their values. As an opaque access structure for field data there are two alternatives: as a string and as a block of binary data. Both make sense in their own context. The string representation is useful to present the data of arbitrary fields to a human observer and to read/write the field data to a human-readable file. The binary representation on the other hand is more efficient for storage and transfer in cases where human readability is not needed. Thus both need to be provided.

In C++ it is possible for classes to be parts of other classes. Of course a scenegraph also needs the ability to have fields that are not primitive types. The type system described so far doesn't prevent that, as long as the access functions can be provided arbitrarily complex types that can be put into fields.

The interesting question is whether these complex field types are described themselves, i.e. whether the description type system is recursive. This depends on how often these recursive structures are used and how important it is to be able to access and manipulate the individual elements of these structures, and what consequences this has for the rest of the system.

Large structures of data are rarely used in a scenegraph, as memory efficiency demands only storing the needed data. This leads to multiple, smaller structures, usually of primitive types. Thus the importance of complex structures as field data is limited and as such the parallel data handling methodology described in sec. 4.3 was designed inconsiderate of this. Consequently it is incompatible with it, and the typing system is not recursive, i.e. a field container containing a number of fields cannot be used as a field data type, but only as a referenced type.

3.1.2 Field Containers

The fields are organized in field containers. The quintessential type of field containers in a scenegraph system are the nodes of the graph, others include secondary structures like windows and cameras.

These field containers need to allow access to find out which fields they contain, and to facilitate reading and writing those fields. This is also a working point for extending the system, as higher level systems might need additional information to the one provided. One example would be information about which fields are 'in' and 'out' fields for a routing manager. It is also useful to have information about the inheritance relations of the field containers to be able to walk the inheritance tree and get information about the classes' ancestors, something that's not possible using standard C++ methods.

Similar to the fields, as this information is constant for every instance of the field container, the type information is stored only once in the class and not in every instance.

The field containers are also the basis for parallel data safety as described in sec. 4.3 and the type holds some information related to field container creation, as described in section 3.2.

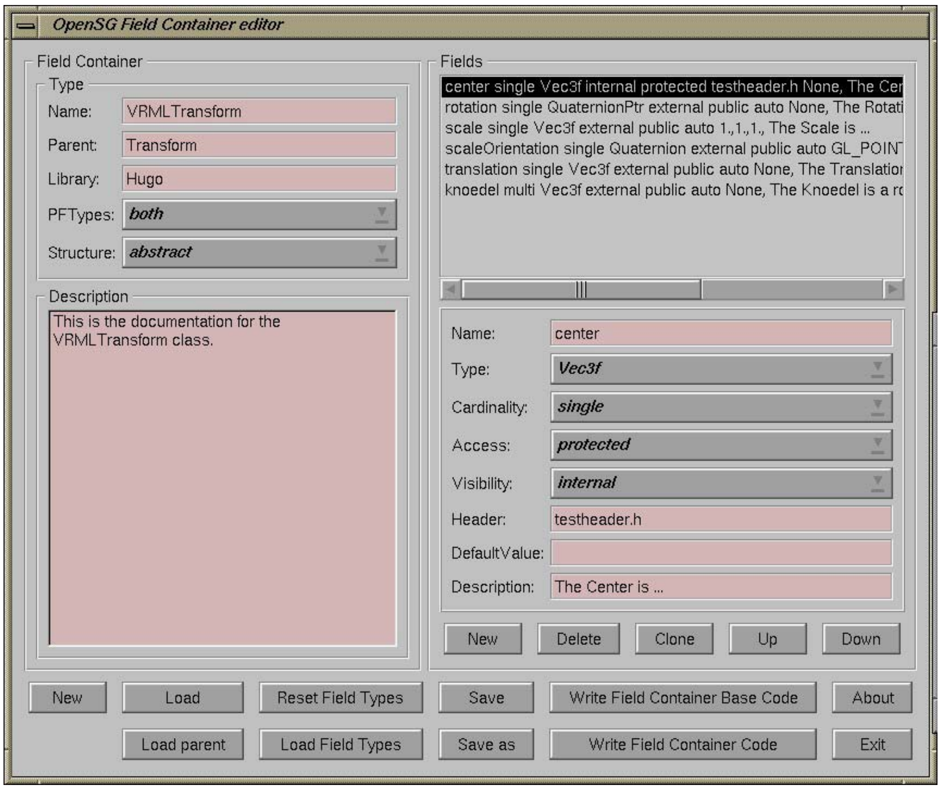
To make extending the system as simple as possible, these description mechanisms need to be usable in a very simple way. Especially in the context of dynamically loadable extensions as described in sec. 3.2 the initialization and registration of new types of fields and field containers should be as simple and automatic as possible. C++ allows this automatism via the initialization of `static` class instances. As these happen in an undefined order, not all initialization can be done within them. To make access to other statically initialized structures safe some actions have to be executed after all static inits are done. The easiest way to do that is keeping a list of init functions that are called during startup or after a new module has been loaded.

All the typing information in combination with some conventions for consistent field access, e.g. the availability of `getValue` and `setValue` methods, add quite a number of demands to the field container code. This code should be created automatically, relieving the author of field containers from having to take care of all the constraints that need to be satisfied to ensure a consistent and stable system.

This information that is used as meta-information to create the field-container service code should be kept in a way that allows automatic updates and that integrates itself gracefully into existing version and source code control systems. A very good way to handle this is using a separate text file that describes the meta-information. A separate text file can be used in dependency rules to automatically update the dependent code whenever the meta-information changes and can also be integrated in a standard source code management system.

To simplify the creation and interpretation of this file it should use a standard format. The best candidate is XML[128, 44], which allows arbitrary information to be formatted in a simple structure that be created, read and tested by standard tools. An example for such an XML file is given in fig. 3.1. To even further simplify the creation of new structures a graphical interface to generate the XML files has been developed (see fig. 3.2).

Figure 3.1: Field container meta-information description as a XML file


OpenSG Field Container editor

Field Container

Type

Name: VRMLTransform

Parent: Transform

Library: Hugo

PFTypes: both

Structure: abstract

Description

This is the documentation for the VRMLTransform class.

Fields

```
center single Vec3f internal protected testheader.h None, The Cer
rotation single QuaternionPtr external public auto None, The Rotati
scale single Vec3f external public auto 1.,1.,1., The Scale is ...
scaleOrientation single Quaternion external public auto GL_POINT
translation single Vec3f external public auto None, The Translatior
knoedel multi Vec3f external public auto None, The Knoedel is a rc
```

Name: center

Type: Vec3f

Cardinality: single

Access: protected

Visibility: internal

Header: testheader.h

DefaultValue:

Description: The Center is ...

Figure 3.2: The field container editor

3.2 Creation

In the context of extensibility and flexibility the creation of objects plays a central role. It should be possible to create objects that are not necessarily known at the compile time of the program, objects which are only known by their name and which are manipulated via the methods defined in sec 3.1. It should also be possible to replace the system's default objects by new types, which can be optimized to the specific application or hardware environment. This should include forcing the system to not use it's old types at all any more, thus allowing system modules like loaders to be seamlessly integrated into a new type structure.

Object creation in standard C++ is handled by the `new` operator. It can be overloaded, but the language possibilities here do now allow the needed flexibility. Another method for creating objects has to be used. There are numerous design patterns[33, 34] that handle creation of objects. The two that fit the abovementioned requirements best are the prototype and the factory patterns.

The prototype pattern uses a prototypical instance of the structure that is cloned to create new instances. This naturally facilitates replacement at runtime, as the prototype can easily be replaced by an instance of the new type. Creation from now on transparently creates instances of the new type and the new type only. By hiding the prototype in the base class and accessing it via a class method the pattern can be used nearly as efficiently as the native method, while at the same time giving the increased flexibility needed for a dynamic system.

The factory pattern hides the object creation in a specific factory object, which is asked to create a new object. This easily allows abstract object creation, especially in combination with the prototype pattern given above. The factory can keep a map of create functions, indexed by the name of the structure. Thus it is possible to create instances of a structure about which nothing is known but its name. This map can easily be extended at runtime to add new types of structures to the system. The factory can also be made to dynamically look for ways to create unknown structures in the form of dynamically loadable modules, thus allowing transparent extension of the system just by using new structure types. The map access makes this pattern less efficient than the direct creation via prototype, thus it makes sense to use both of them in parallel.

These dynamic creation patterns allow a system to automatically extend itself and adapt its inner workings to better support new applications and new hardware platforms optimally.

3.3 Attachments

The extension mechanisms described in sec. 3.1 and 3.2 allow very deep manipulation of the system by replacing existing components with new ones. In many cases this is a lot of work and not actually needed for applications.

Many applications don't need to replace existing structures and methods, they just need some additional data in the structure. Names are an example of such data. If an application wants to name a structure this can be attached to the structure, similarly for additional information about the origin of the data e.a. This information does not have to be interpreted by the system, it just needs to be kept around and be accessible on demand by the application. Many system feature a so-called 'user-data-pointer' for this purpose.

The disadvantage of a user-data-pointer is the limitation to a single pointer. As soon as two applications want to add data to the structure the concept breaks down. A logical consequence is an array of user-data-pointers. If the size of the array is limited the problem is only shifted but not solved. If the array is dynamically sized, the problem is replaced by another one: which index is assigned to which application? This has to be consistent, so that an application can access the data that belongs to it. To do that the indices have to be centrally managed and all applications have to use the central allocation method to get a private index. It still doesn't allow data hiding, every applications can easily access every other application's data by changing the index, which may happen even by accident.

A better and more general solution is to replace the array of pointers by a map. This map indexes the data with a name given by the application. When the application name is part of the name, uniqueness is automatically achieved. This also allows an arbitrary number of attachments per structure with little overhead.

For maximum generality every field container should have such an attachment map. But the map carries some overhead even if it is not used. The typical targets or the additional data are the nodes of the scenegraph, other structures don't often need to be extended. Thus a workable compromise is to allow attachments to the nodes of the scenegraph and other structures that are not created in large numbers and leave the other structures of the system alone.

3.4 Dynamic Fields

If the application that is built on the extensible scene graph is itself extensible, e.g. a fully compliant VRML system supporting dynamically created node types, some dynamic extensibility is needed. The creation of Field Containers, whose Fields are only determined at runtime, is one important aspect. As it is impossible to create new types and structures at runtime in C++, a dynamically extensible structure is needed, which can accomodate any number and types of fields.

The system described so far can be extended to support this. To do that it is necessary to change one of the basic assumptions given in sec. 3.1.2: the list of fields of the structure is not constant for all instances any more. As this list is only accessible via acessor methods from outside the class anyway, it is no big problem to have every instance of the class keep a private version of this data. This allows each instance to keep a different set of dynamic fields, which can be changed at runtime.

As the set of Fields is only known at runtime, no specific access methods can be used, only the generic access using the Fields' names is possible. This reduces efficiency when accessing data, but is unavoidable in a dynamic context like this.

3.5 Node Structure

The central structure in a scenegraph system is, of course, the scenegraph itself, i.e. the graph of nodes representing the scene.

The nodes are field containers as described in sec. 3.1.2, and as they need to define the hierarchical structure of the graph, they need to keep a list of the children of the node.

There are, however, a number of possible organization structures for other tree-relevant data.

3.5.1 Parent structure

If everything is done on the scenegraph as a traversal (see sec. 3.6), there is no need to keep more information that the children. Inventor [118] is an example for a system which ke no other information about the graph structure but the children (see fig. 3.3).

But as soon as information about a node depends on other nodes above it in the graph, e.g. the accumulated matrix at the node, or the bounding box of the node in world coordinates, information about the parent(s) of the node is needed. Thus many scenegraph systems include an additional field to reference the parent(s) of the node (see fig. 3.4).

But scenegraphs are not always structured as trees, they can just be acyclic graphs. In a typical scene some objects can appear multiple times, e.g. wheels of a car or trees in a forest. As the largest part of a scene's data is usually taken up by the geometry data, the memory consumption of a scene can be significantly reduced by reusing the same object multiple times (see fig. 3.5).

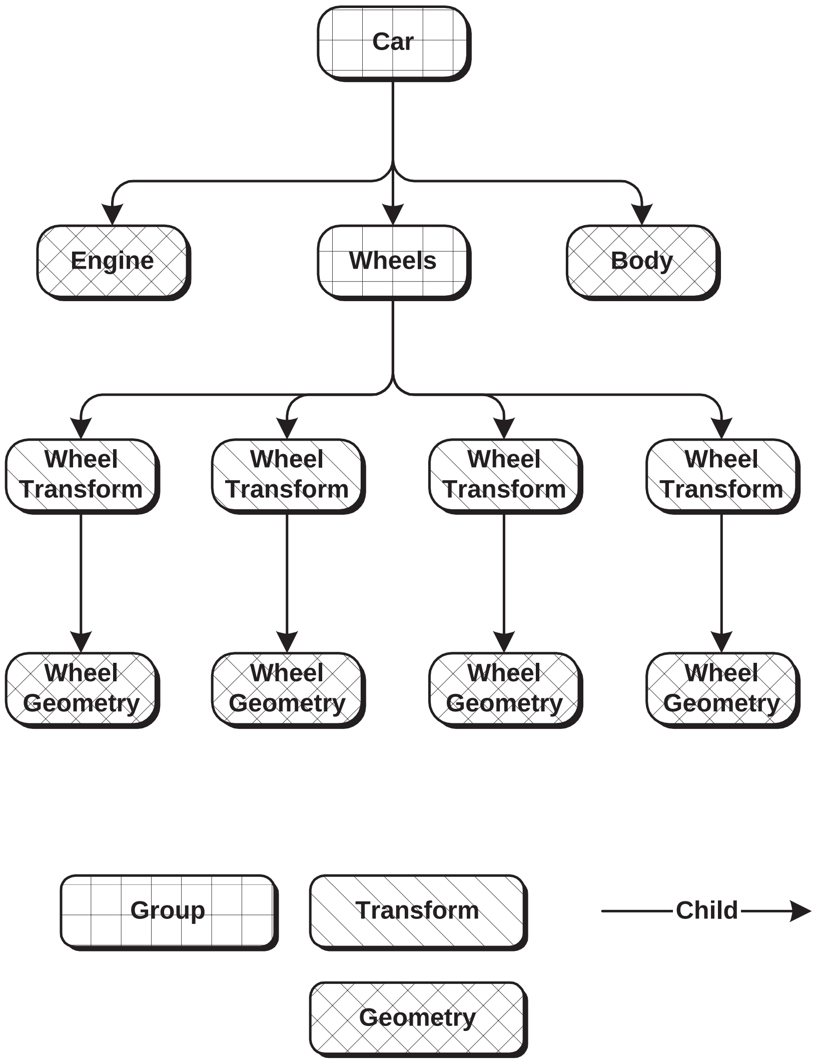


Figure 3.3: Tree structure without parents

But now an object can have multiple parents. That in itself is not a problem, the single-value parent field can be replaced by a multi-value parent field (see fig. 3.6).

This becomes a problem when combined with the node-relative information mentioned above. Given the bottommost node in figure 3.6, what is its `ToWorld` matrix? The answer is simple: there isn't just one, there are multiple. And the decision which one to choose depends on which copy of the object one is interested in. For this node the situation is rather simple, one only has to know the interesting parent index and follow the right path to root. In general the situation becomes non-trivial fast. For

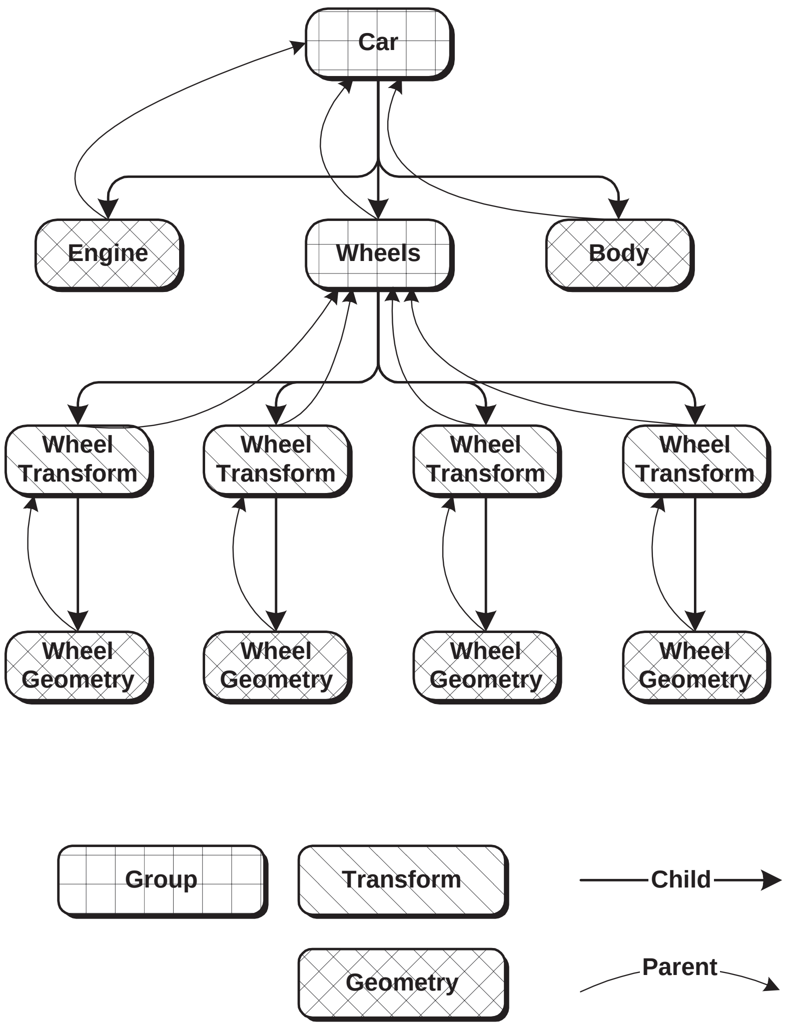


Figure 3.4: Tree structure with parent pointer

other nodes the situation looks simple: one parent, just go up. But further up the tree decisions can arise an arbitrary number of times. Thus to uniquely identify the node, in addition to the node pointer one needs an arbitrary number of parent indices for specific nodes on the way to the root, which constitutes a rather large data structure to just identify a node.

This workaround doesn't work anymore for another problem: names are no longer useful. If a node has a name, and the name is to be used to identify the node, in a multi-parent situation that is not possible. Thus names are not usable any more to identify nodes, which can be a significant problem for systems that need to identify

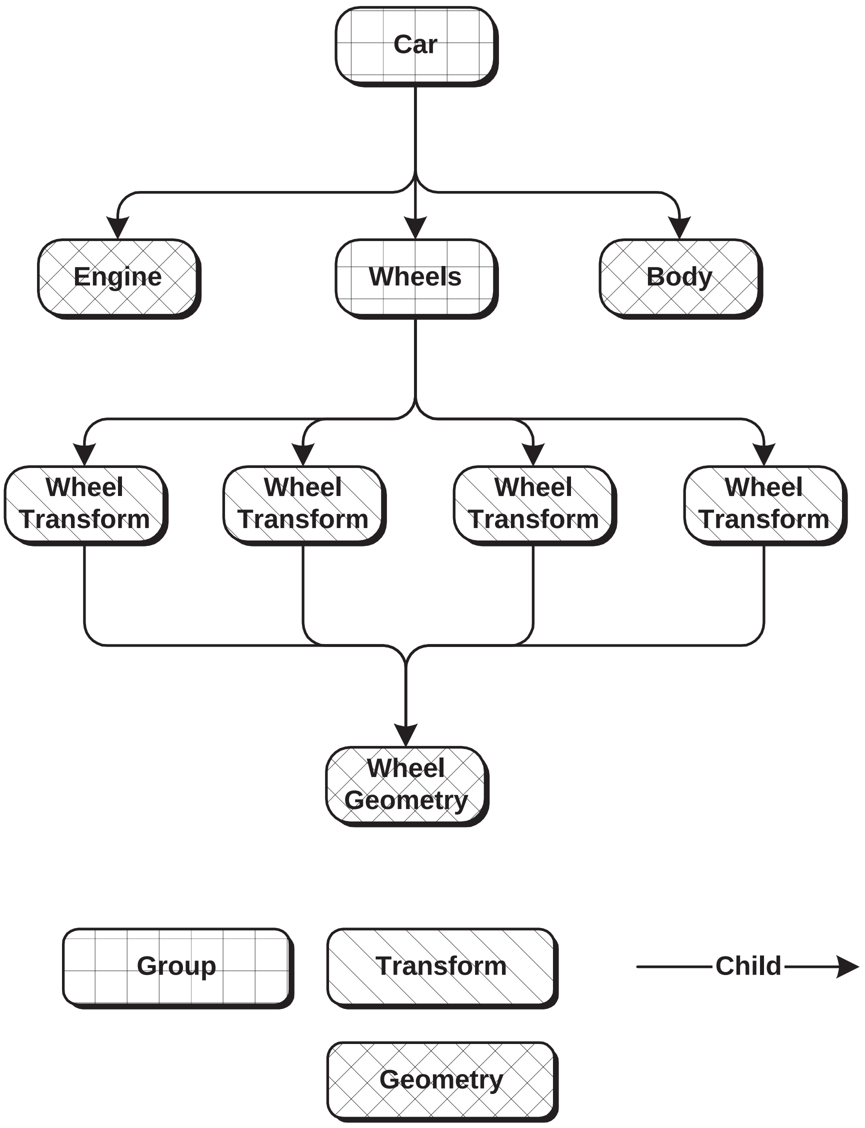


Figure 3.5: Object reuse

nodes from external sources.

Even if the non-unique names are not a problem, the node+index structure is rather complicated, especially in the context of dynamically changing scenegraphs. In general it is simpler to just keep a list of every node on the way to the root, which makes access to the parents field unnecessary and might even speed up the traversal. This can also be used when the parent field doesn't exist, and is indeed the way Inventor handles the problem. Some inquiries can only be done on a path that includes the

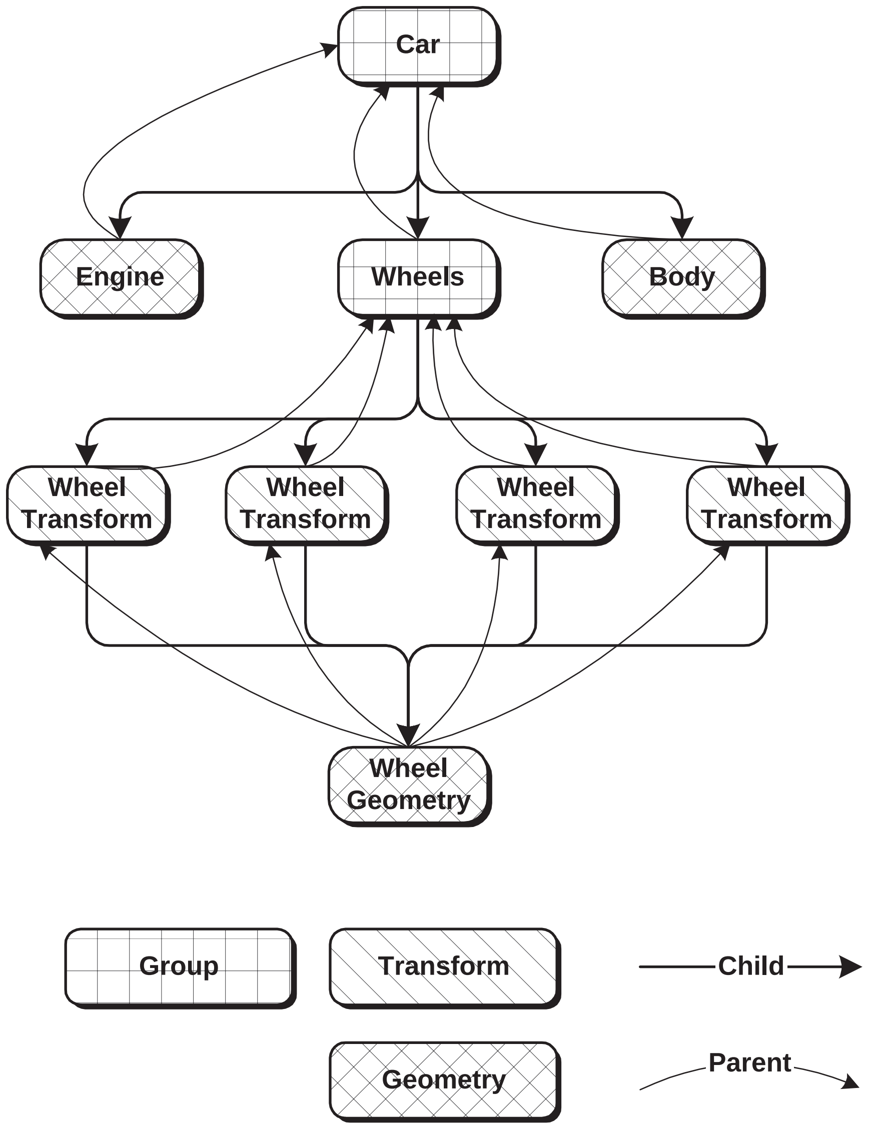


Figure 3.6: Multiple parents

complete parent hierarchy from a node to the root.

Other systems like Performer just ignore the problem. They use multiple parents and leave the responsibility to choose the right path to the root on the shoulders of the application programmer. For special cases that is acceptable, for a general system that is a significant burden which should be circumvented if possible.

Going back to the original goal of conserving the memory of multiply used objects, primarily of the geometry nodes, an alternative approach that reaches the same goal without adding the complication of multiple parents arises.

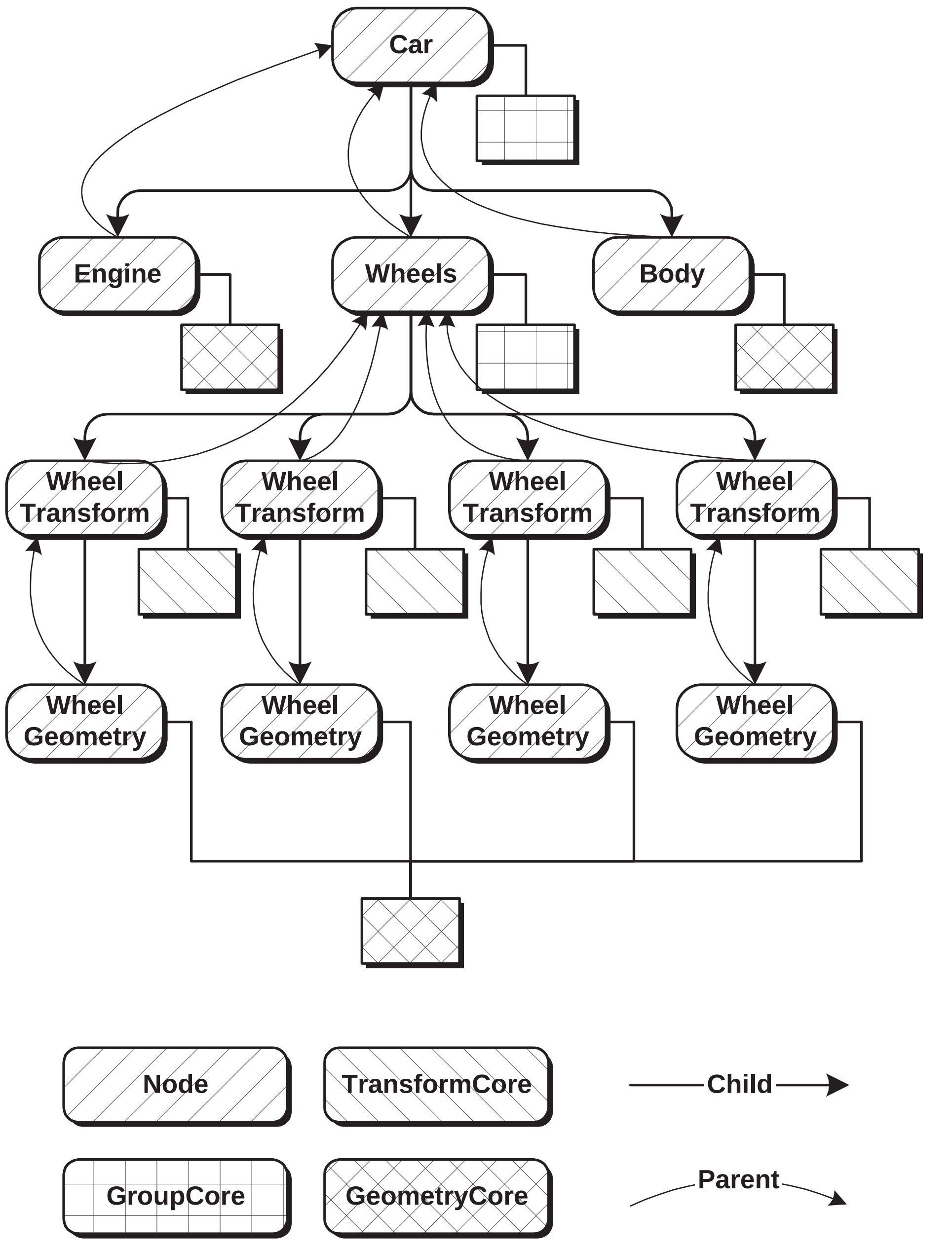


Figure 3.7: Split node: Node-Core division

The basic idea is to split the node into two parts (see fig. 3.7). One part ke the information that is needed to define the tree structure (the Node part), the other part ke the node-type-specific information, e.g. the geometry field data (the Core part). The node ke only one parent pointer and as such has a unique path to the root. The core can be used by multiple nodes and thus allows data sharing.

This node-core division has multiple benefits. Named nodes are unique, just as pointers are unique and can be used to identify a node. Furthermore not only geometry can be shared, any core can be used in multiple places in the graph, allowing very simple synchronization across different parts of the parts, e.g. a switch core can switch an arbitrary number of nodes at the same time.

The system has one drawback compared to the multi-parent situation, though: it is not possible to share full subgraphs easily, all the nodes have to be shared explicitly. Memory-wise that is not a big problem, as the node part is very slim and a couple replicated nodes don't need a significant amount of memory. It is a problem for changing scenegraphs, as the duplicated nodes do not automatically reflect changes to the original, e.g. adding a new child or removing a child from the master is not automatically done for all the copies. This might actually be the intended behavior, but not always. It could be alleviated by using special nodes for the copies and the master to reflect these changes. That would be quite a bit of work and cost some amount of performance, but for situations where this behavior is needed it would be a solution.

3.6 Actions

So far only passive extensions have been discussed. But also the active part of the system has to be taken into account. It has to be able to cope with the dynamics that the passive part creates, and it should also be extensible itself.

The active part of a scenegraph system revolves around traversals of the tree. There are different names for these traversals, in this context the name "action" was chosen. An action traverses the graph it's applied to and at every node of the graph acts according to the type of the node respectively the node's core before it traverses the children. How and when the children are traversed is also an interesting question.

3.6.1 Action structure

There are different ways of realizing this. The simplest and the one that first comes to mind is having a method of the node that is called for every kind of action in the system. This method acts for the node and also initiates the traversal of the children, if any.

This approach, while simple and easily understood, has several shortcomings in the context of an extensible system.

Action extensibility

A scenegraph system has a number of predefined actions that are needed for it to actually do work. The most prominent ones are drawing, the process of transforming the abstract data in the tree into an image, and intersection, testing the geometry in the tree against a ray or cone for picking objects or for simple collision detection.

But a dynamic, extensible system has to be able to be added new actions. For the simple method given above, this is not possible, as it's impossible to add new methods to existing structures without changing the source code.

While it is still desirable to be able to use object methods for actions, as they have direct access to all the private data, it is not enough. There has to be a more flexible way of selecting the action to take for a node.

Overridability

In many situations, especially during development of an application, it is desirable to be able to override the action method for a specific node type with with a different one. Either to log the fact of a node being traversed, or to ignore the given type of node, or to try a new node action without having to change the source of the library and recompile.

The static method system of course cannot handle this situation, as the method for each node is hard-coded.

Node extendibility

The dynamic extension system described in sec. 3.1.2 and 3.5 can extend the set of nodes that are known to the system dynamically, even at runtime by loading new modules.

The action system has to be able to handle this situation. The simple system described above can actually do that, as every node is responsible for traversing itself, but for a new system that handles the other constraints given above this has to be taken into account, too.

Solution: Functors

A system that satisfies all the constraints given above is based on a vector of functors [116].

A functor is a function object that enculates a method or function and that can be called similar to a real function or method. These functors can be made flexible enough to fulfill the requirements: a simple function, a method of a given instance of a class or a method for an object given as a parameter can be wrapped in functors that present the same interface to the outside and as such are interchangeable (see fig. 3.8).

The action can keep a vector of those functors, indexed by the node type, and call the appropriate functor for each node. New types can register their functor with the action and thus are handled correctly. New actions can be easily defined, they just need their own vector of functors.

Local adaption is also possible, i.e. overriding a special node type's functor for a specific instance of an action, by keeping the actually used functor vector inside the action instance. This vector is copied from the default vector that is kept inside the action class at instantiation time. Thus changing the functor for a specific action instance does not influence the functor used by other instances of the same action. A diagrammatic description of the action data structure is given in fig. 3.9.

The functor vector system can also support a positive side-effect of the simple method: inheritance. If for a given node type no functor has been defined, the action can walk the inheritance tree which is stored in the field container type described in sec. 3.1.2

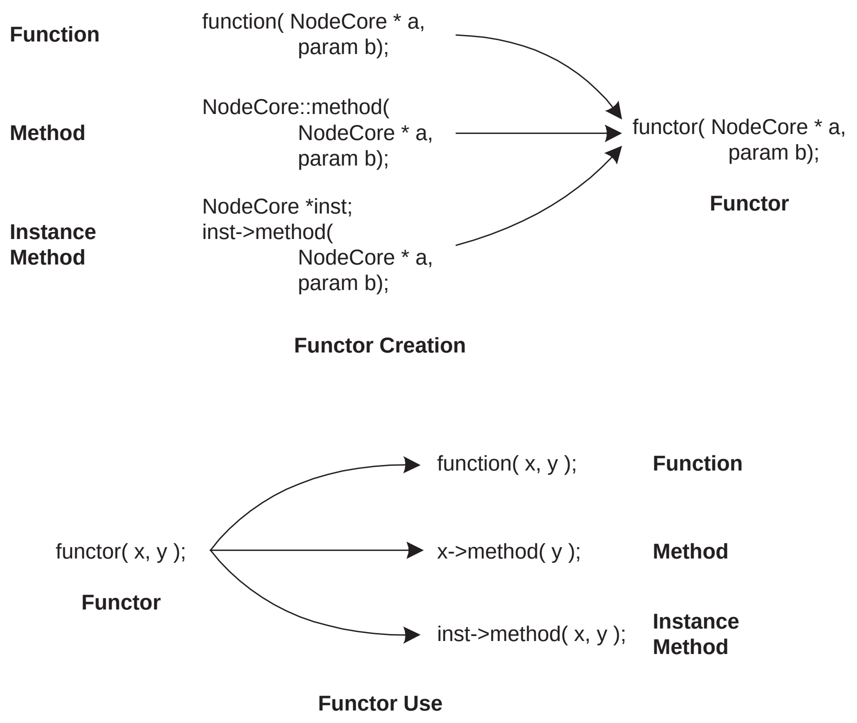


Figure 3.8: Functors as wrappers

until it finds a defined functor or hits the root, in which case a default action will be taken.

So far it was still assumed that the method itself actively traverses the children. This is not necessarily the best way of handling the recursive traversal.

3.6.2 Recursion Control

Selecting the next node to traverse can have a significant influence on the usability and performance of an action.

The standard ordering for traversals is depth first, i.e. all children of a node are traversed before any siblings are traversed. This is consistent with inheriting state downwards in the tree, e.g. transformations. When entering a node the state is changed, when all children (and their children) are processed the state change can be reversed.

For some actions the depth-first traversal is not optimal. Ray intersection for example

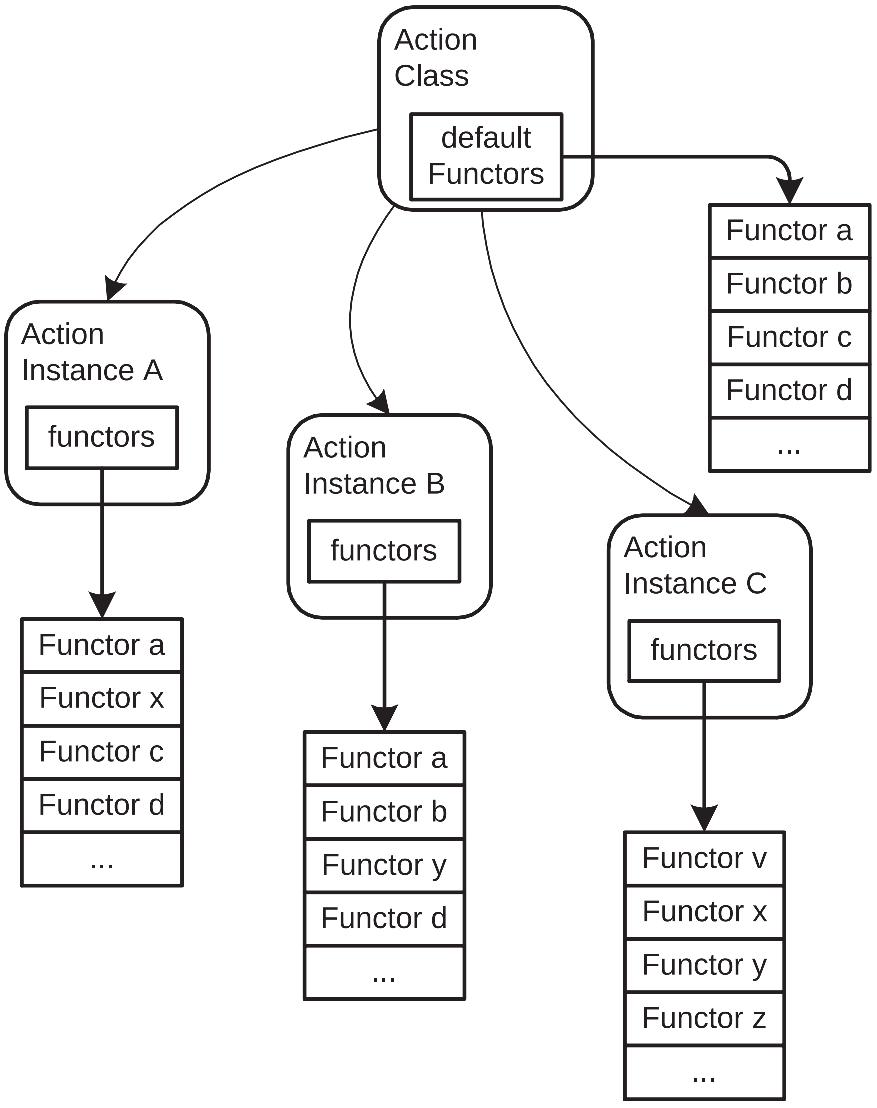


Figure 3.9: Action structure

works more efficient if nodes are traversed in front-to-back order, as the goal is to find the first intersection, which is more likely to happen with closer objects. It is also independent of the constraint that all children have to be traversed before siblings, it is only concerned with the next node, wherever it may originate in the graph. Similar reasoning applies to ordered rendering traversals, e.g. front-to-back traversal which is needed for occlusion culling to be effective, or back-to-front traversal, which is needed for rendering without Z-buffer.

Thus it makes sense to open up the recursion control and put it into a separate iterator object that gets passed all the children of the currently active node that should be traversed, optionally with a priority of each child, and which decides which node should be traversed next.

The depth-first iterator is a simple stack, which always pushes new candidates on top of the stack. The pqueue iterator ke a priority queue in which the candidates are sorted based on their priority, and the lowest (or highest) priority node is selected for traversal next.

This allows a free combination of action and traversal order and gives maximum flexibility to the user of the system.

3.6.3 Micro-Structure

Traditionally actions are rather closed entities. An action traverses the tree, does what it has to do and delivers a result. For a static system that is acceptable, but for a system that is used as a development basis, for experimentation and trying new algorithms it has significant shortcomings.

It is not possible to combine separate actions, so the monolithic action has to do all the work itself. If only a small part of the work is to be replaced, the whole action has to be replaced and most of the code has to be replicated. One example for that is the combination of culling and rendering. There can be many different kinds of culling that make sense for a scenegraph. The most obvious is view-volume culling to reject objects that are not visible from being rendered. But importance culling, which rejects objects that are not important in the current context also makes sense, as well as higher-level culling methods like portal or occlusion culling. And of course, after everything is culled, the remaining objects have to actually be rendered.

In a monolithic action system, all the different culling methods have to be put into one big action. Alternatively there could be a combinatorial set of actions that offer all possible combinations of actions. Neither is a good solution.

There are different ways of combining multiple simple actions into one compound action. One is called *scene graph rewriting* [15]. In scene graph rewriting an action creates a new scenegraph that only contains the nodes that are actually traversed by the action and that can then be used by another action (see fig. 3.10). This is a very flexible and consistent way of handling cascaded traversals, as the traversals can even

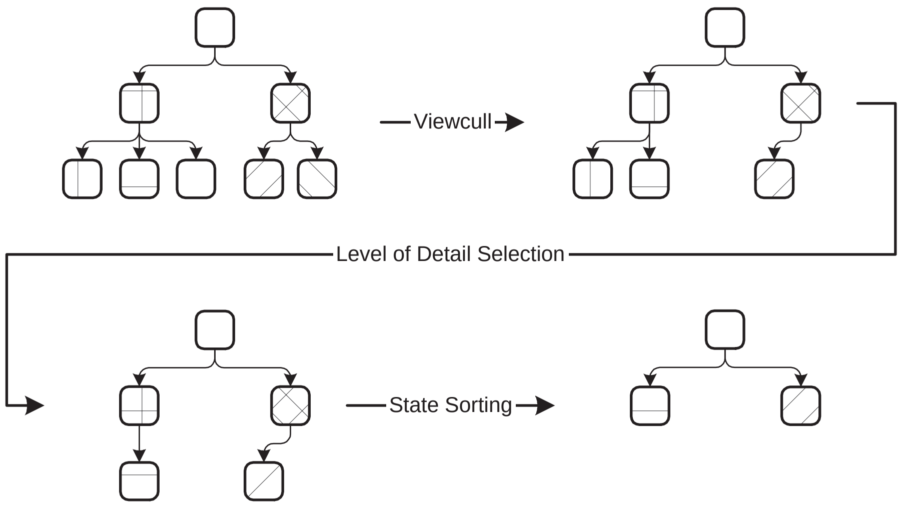


Figure 3.10: Scenegraph rewriting

be spread across multiple threads. The big disadvantage of scene graph rewriting lies in the difficulty to making it efficient. It creates a full scenegraph for every step of the cascade. A scenegraph, especially a multi-thread safe scenegraph as described in sec. 4, takes some amount of effort to be created. Doing that multiple times for every frame is going to be hard to optimize and get efficient.

An alternative approach is to split the Action into smaller parts that can be combined freely. These Actors [91] can all deselect some of the children of the currently traversed node from further traversal and thus can be combined relatively freely. Thus the view volume culling can be replaced without having to touch other parts of the rendering traversal, or a semantic culling Actor can be added easily. The flexibility and overridability described in sec. 3.6.1 can be added to each Actor individually, if it makes sense.

This approach is quite a bit simpler than the scenegraph rewriting. No temporary structures are being built, only the child list is worked on in multiple st. The same node is worked on multiple times, so after the data is loaded into the cache first it can be accessed very efficiently.

For some cases it is even possible to distribute the actions across multiple threads. This is described in sec. 4.2.

3.7 Summary

This chapter describes different aspects of extendibility that a scenegraph should have. The concepts described here support a system that can be extended easily by an application due to the simple creation of new structures from a meta-description. The structuring of the fields and the reflective nature of the system also allow very generic tools to be built that can directly work with these extended components.

By carefully choosing the creation patterns employed the system can be extended at runtime, even for applications that do not know about the new extensions, without having to recompile. As not all applications need the full flexibility of deriving new classes, an efficient manner to add data to existing instances is described, as well as a way of having structures with dynamically added Fields.

A central part of every scenegraph is the graph itself. A new way of structuring the nodes of the graph and dividing the graph-structural data from the node-specific data is developed. This approach combines the strength and ability of efficiently sharing data in the graph with the important feature of uniquely identifying a node in the graph by its pointer.

Finally an equally flexible approach to structuring the active parts of the scenegraph system, the actions, is described. This approach supports the extensible nature of the system by allowing the addition of new components at runtime even for existing action instances. It also allows the efficient cascading of actors, thus encouraging a flexible and extensible action micro-structure.

Chapter 4

Parallel Processing

Performance is of very high importance for Virtual Reality applications and software systems. The need to keep the system latency under 100 ms [50] in the face of ever growing complexity expectations demands very high and stable throughput. In general there are two ways to improve performance. One is to get a faster system. Faster processors are developed all the time. The development is speeding up at a phenomenal rate, already out-pacing Moore's original law stating a performance doubling in 24 months by cutting the time to 18 months.

But the cost of the latest generation of processors may be prohibitive due to the immense development costs of new processors and production lines, which lie easily in the billions of dollars. Or the waiting time until the needed performance is available in a single processor might be too long, so that the second performance-increasing method has to be used: using multiple processors. Parallel processing units are used in many places in a standard computer system, using multiple processors is only the last step. But there is only one current application thread running, so the parallel nature is not directly visible for the application. This is changing, however. The next generation of processors [17, 29] will feature multiple execution threads directly on chip. However, even systems available today support multiple simultaneous application threads, by using multiple processors.

Multi-processor systems have been in use for a long time, with processor counts ranging for two to several thousand. Systems with very large numbers of processors are very rare, but systems with smaller numbers are becoming quite common. To ease writing applications for them, in many if not most cases these systems present themselves as a single system with a single address space, independent of the actual system organization. These kinds of single-image systems are the main topic of this chapter. Loosely coupled distributed systems or clusters are touched in sec. 4.5.

Servers with processor counts between 4 and 32 are available from a wide variety of vendors [68, 71, 70, 82, 41, 111, 112] and dual-processor systems are becoming very common even for desktop machines. The exponential pricing policy of large processor companies makes it possible for two slightly slower processors to be available at a lower price than a single fast one. Due to the increasing demand, dual-processor motherboards are only slightly more expensive than single-processor ones, so cheap dual-processor systems are spreading.

To take advantage of multiple parallel execution threads, be it on a single or on multiple processors, there need to be tasks that can be executed independently, or that are big enough to be spread over multiple threads. In a typical VR system there are both types. But parallel running threads add new complexity to data arrangement and protection. Two threads that change the same data element at the same time will create unpredictable results. This has to be prevented either by structuring the tasks so that its impossible to happen, or by designing the data structures to prevent the unpredictability.

4.1 Tasks in a VR system

Description In a VR system there is a large number of tasks, many of which can or should work independently/asynchronously. One possible set of tasks is depicted in 4.1.

The tasks are centered around the user of the system. In a multi-user system most of these tasks would be replicated for every user, while some would be shared.

Input tasks measure her actions and transfer them into the system, so that it can react to them and let the user interact. These tasks are not concerned with the representation of the virtual world, as they are only involved in the real world.

The reactions to these actions however change the virtual world to reflect the actions. These changes of state are often reflected in a visual way and thus have to manipulate the data stored in the scenegraph and create a new image. This image creation can be split up into different tasks. Visibility determination to restrict the amount of data

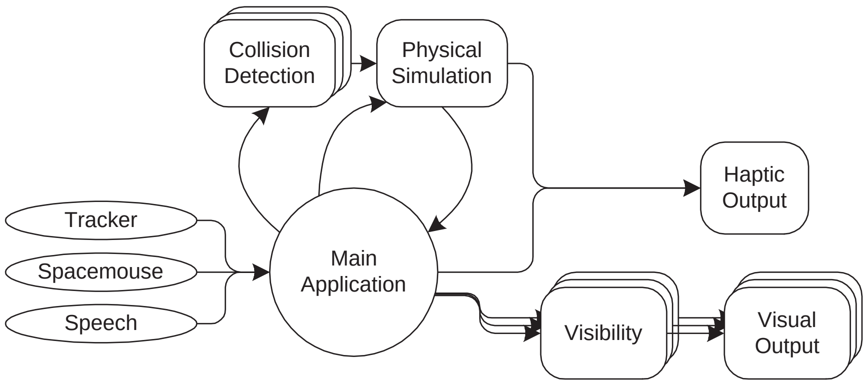


Figure 4.1: Tasks in a VR system

needed to be fed to the graphics pipeline is done. This can include simple view frustum culling [8], Level of Detail selection [20] or occlusion culling [131, 10, 22]. The remaining geometry is then optimized to minimize state changes and make optimal use of the graphics system, before it is sent to the pipeline. Even if the reactions to the user's actions are not reflected visually, other reflections may need access to the geometric data. Auditory feedback [9] might need the geometric situation to calculate attenuation and echos. Haptic feedback [28, 74, 64] needs the geometric data to check for collisions and to calculate appropriate forces.

Collision detection [21, 42, 45, 75, 35, 130] is needed not only for haptics, but also for a number of other tasks, e.g. user interactions with the virtual world in the form of virtual objects like buttons, or as a part of the physical simulation of the virtual world.

This physical simulation [121, 86] is another large task that, depending on the sophistication of the simulation, can take a significant amount of processing power.

Analysis These tasks have different interrelations and dependencies, necessitating different processing models to fulfill their needs.

Some of the mentioned tasks are very computing-intensive and as such can benefit from classical parallelization techniques that have been developed by the high-performance computing community. Tasks that work on a large number of independent objects are promising candidates. Physical simulation and collision detection are obvious choices, hierarchical visibility techniques are another.

One central problem in parallelizing is data consistency. As long as all threads are only reading the data consistency is ensured, which is possible if the tasks return their results as separate data structures. The above mentioned tasks can do that quite easily, the result of the collision detection is typically a list of colliding objects, physical simulation generates a list of new transformations (as long as only rigid bodies are involved) and visibility creates a list of visible objects. It is also possible for the threads to change the data, as long as the design of the algorithm ensures that the data is only read by the same thread. As this division has to be absolute, that can prove to be difficult. Great care has to be taken to ensure access to data that can not be changed at the same time by another thread. Some attributes that are stored in a scenegraph can depend on other parts of the graph, which can be higher or lower in the hierarchy. The concatenated transformation to the root, better known as `ToWorld`-matrix, depends on all the transformations on the way up to the root. The bounding box of a node depends on all the bounding boxes of nodes below it. The bounding box of a node in world coordinate space depends on both of these and thus on all the nodes below and above. As these structures are usually managed using lazy evaluation care has to be taken to make sure that inconsistent data is not returned for multiple threads working on the graph.

One option would be to lock access to these structures to prevent multiple threads from accessing them at the same time. This can become inefficient very fast, as even very simple accesses, which happen a lot, need to be locked. Locking in a large multiprocessor system is an expensive operation and should be avoided as far as possible. For some cases it is possible to structure a parallel scenegraph traversal in such a way that locking is not needed (see sec. 4.2), but in general an application has to be very careful to avoid race conditions and inconsistent data.

It gets even worse for independently running threads, which are a common occurrence in a VR rendering system.

The classic example for parallel processing in a rendering system is the App/Cull/Draw division as defined by Performer[98]. It starts with the application, which contains everything needed to set up the scenegraph for the next frame, moving through the culling stages, which try to extract the minimal amount of data needed to create the current image, and ends in the drawing stages, which feed the graphics hardware. These tasks form a pipeline, which is usually frame-clocked, i.e. the data for a whole frame is passed between tasks. This pipeline can have more st, if different culling approaches are used together. Typical combinations are a cell or portal based visibility pass[120] followed by standard view volume culling and/or occlusion culling [10, 22, 131]. After the visible objects are thus identified, an appropriate level of detail is chosen (or generated, in the case of a multi-resolution representation), which is converted into a stream of graphics commands and optimized.

Asynchronous operation is another important area. Different tasks may run at different speeds and might need different update rates. For visual simulation update rates between 10 and 60 Hz are a common and adequate goal. Haptic calculations need to run at much higher rates, in the 1000s of Hertz', due to the high sensitivity of the human touch and motion sensors. Physical simulations on the other hand can take a lot more time to calculate and thus might only be able to run at single-digit rates.

These tasks should be able to run independently of each other, each at its own pace. At the same time, they all need to have a consistent view of the system and not be confronted with partial results from other tasks running in parallel. To maintain consistency throughout the system the different views also need to be synchronized and can not be kept apart forever.

The following sections describe the different problems that were raised in this analysis and their solutions. Section 4.2 describes what can be done without replicating the scenegraph data. Tasks that can not live with the restrictions to allow that or that have to run asynchronously will have to have private copies of the data to ensure consistency. Section 4.3 describes different ways of storing these private copies and their pros and cons.

A significant trend in virtual reality applications is the step away from head-mounted systems to immersive projection systems. These usually use multiple projectors to create a large, not necessarily planar, display surface. To drive these displays multiple graphics subsystems are needed. These are usually joined in a single computer. Thus a scenegraph used to drive these displays has to be able to handle multiple graphics cards. To drive all those cards at full speed a single thread can not be used, multiple driving threads, possibly running on different processors, are needed. When these multiple graphics pipelines are put into one system this is called multi-pipe handling and covered in sec. 5.5. There are also cases where they are not part of the same system but distributed in a cluster architecture, where each node is a separate low-cost system. This is a special case of a distributed system. Distributed systems in general are outside the scope of this work, the important special case of a clustered renderer however is handled in sec. 4.5.

4.2 Non-replicated Data / Actions

Large models can consume large amounts of memory. Increasing problem sizes and user expectations have made memory a scarce commodity. Thus it would benefit the system if parallelization was possible without needing to replicate data, which eats up even more memory. Furthermore the gap between cache and main memory speed is widening fast with the strong increase in processor and cache frequencies, compared to the small increase in main memory frequencies. Thus compact data that has a higher chance to stay in the cache can sped up program execution significantly.

General Parallel operations on a shared data structure are only safe if any piece of data can only be changed by one process at a time. There are different ways of ensuring that no two processes write the same data item, which would result in race conditions and undefined results.

The simplest is to just disallow changing the data. Many tasks, especially analysis and statistical tasks, only read the data, which is safe to do with multiple concurrent threads. A possible complication arises from the use of lazy evaluation, though.

Some information in the scenegraph is expensive to compute, e.g. the bounding box of a geometry node. Thus it is not updated as soon as it changed, but rather flagged as invalid. Only when it is needed again will it be calculated and updated, and is set valid again. This complicates the simple parallel traversal, as just accessing data can result in changes. The problem can be alleviated if knowledge of the task to be performed is available. In this case the needed data can be updated before the parallel threads are run, which can then resort to just reading the data. The preprocessing diminishes the benefits of parallelization, as it increases the sequential fraction of the task. Added difficulty lies in determining which parts of the data have to be updated and the inconsistent results if something was forgotten.

Tasks that that only do very little work per single node, e.g. view volume culling, the overhead for distributing work might not pay off. But there are tasks that have to do a lot more work per node, especially in preprocessing. Examples include tessellation, sorting, striping, creation of a multi-resolution hierarchy and others. These single-node tasks can be safely run in parallel, as they only influence the one node they are working on. Many other tasks, however, have to access and depend on data of several nodes, which is hard to guarantee consistency of.

4.3 Replicated Data

Asynchronous processes working on the scenegraph, including changes to contents and structure, cannot work on the same set of data. To ensure consistency on structural changes the graph would have to be locked for a long time, making the operations nearly sequential. Furthermore, the inability to depend on the graph staying the same for certain amount of time makes it very difficult to work with.

Thus asynchronous processes should each have their own copy of the data, so that they can work independently. Some questions have to be raised in this context: what to replicate, how to replicate it and how to synchronize the different copies.

4.3.1 Granularity

The spectrum of replicated data is wide. It reaches from replicating everything to replicating just specific fields. All of the options have different restrictions and costs associated.

Everything

Memory has become cheap, but increasing problem sizes have made it a scarce commodity nonetheless. Thus it is not feasible to have a private copy of all the data for every thread (see figure 4.2). A typical scene contains 30 Megabyte of data (10000 nodes at 250 byte + 500000 triangles at 56 byte). It is not uncommon to have 10 or more parallel threads (3 graphics pipes with CULL and DRAW each, 1 APP, 2 for collision detection, 1 for physical simulation), which would result in 300 Megabyte of memory, which is not huge but not insignificant either.

Replicating everything also becomes a big problem for synchronization, see sec. 4.3.3.

Everything but the geometry

Looking at the data distribution for the abovementioned typical scene it becomes apparent that the largest part of the data is needed for the geometric data of the rendered objects.

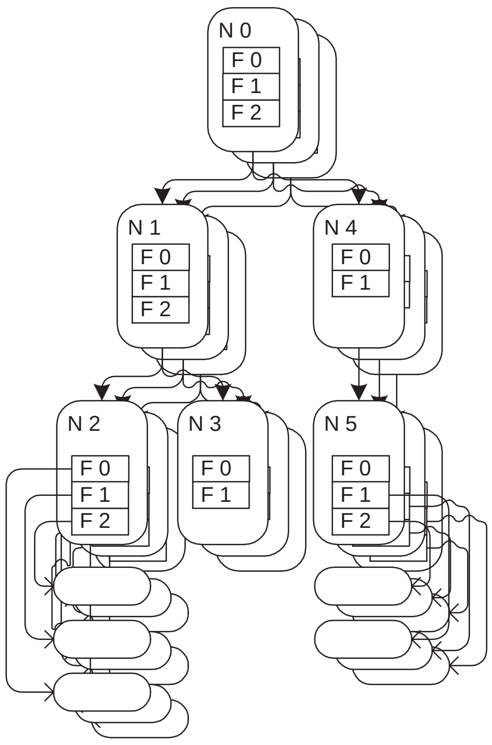


Figure 4.2: Complete replication

As the geometric data is static in most scenes, there is no point in replicating it for every thread. This leads to the approach of replicating everything but the geometric data, leading to a different result for the abovementioned situation. For 10 threads the replicated scene would take up 53 Megabyte, a number that can easily be handled by a standard workstation. This is the approach taken by Performer.

The drawback comes as soon as the geometry is going to be changed. In this situation the application has to take care of allocating and handling the needed replicated buffers. For special cases with strictly defined data flow and strong synchronization like the APP-CULL-DRAW pipeline this is possible without too much work, for more general situations with asynchronous processes this can become quite difficult.

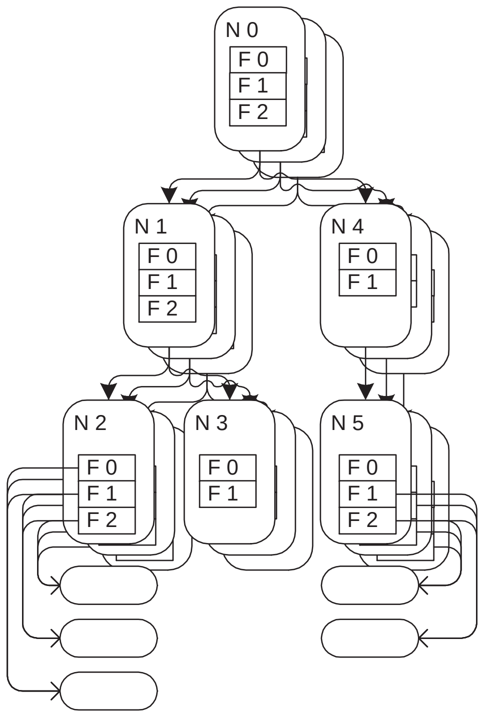


Figure 4.3: Structure replication

Just specific fields

In cases where parallel processing is only needed for specific tasks (e.g. CULL) it might be enough to just replicate the fields that are needed for these tasks. This further reduces the amount of data that needs to be replicated, but also limits the versatility of the approach, as only the replicated fields can be accessed.

Conclusion

It is possible to reduce the amount of data significantly, if the parallel threads are only used for specific tasks. As a scenegraph is no longer only used for rendering, these restrictions can significantly reduce the usefulness of the system. The goal is to create a system that combines the good aspects of full replication without the enormous overhead of geometry replication.

One way to do this is to use virtual replication. This has been used in operating systems successfully to reuse the memory taken by identical executables. It employs

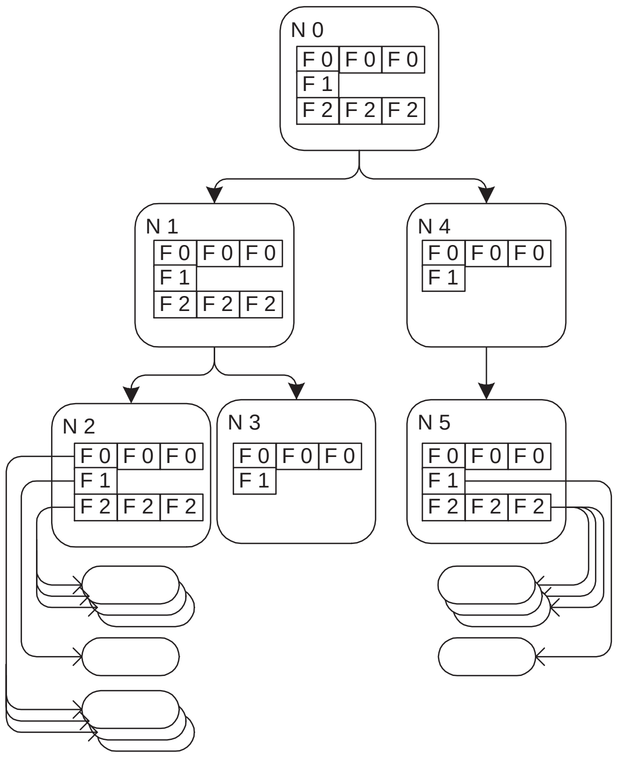


Figure 4.4: Field replication

a copy-on-write paradigm to replicate the data only when it's needed.

The operating system only has to deal with course-grained replication, if a page is written, it is replicated. For the scenegraph a page is too coarse, a finer grained parallelization is needed. Additional complications arise when the access modality is taken into account. As the fine-grained replication is not directly supported by the operating system, it has to be done by the scenegraph system. Thus the scenegraph also has to handle the access to the replicated data and make it as transparent as possible for the running application, so that functions that take a lot of time can easily and transparently be put into their own threads.

4.3.2 Organization and Access

The organization of the replicated data is an important factor defining the simplicity and transparency of access and also influencing the performance of the system due to it's memory and cache usage.

Parallel Trees

A simple method to organize the replicated data is to use completely separate trees (see figure 4.5). The data structures don't have to be changed from a single-thread case and threads can easily and directly access their private copy. It can also have good cache performance, as data for different threads can be held apart. Performer uses this method.

Parallel trees are problematic, however, when information has to be passed between threads. A pointer to a data element in one thread holds no information about the related information in another thread. Thus they have to be mapped from one space to the other when passed between threads. Using a standard map for this mapping can be expensive as soon as a large number of pointers has to be translated, for example when synchronizing the data (see sec. 4.3.3). One solution taken by Performer is to add a numeric identifier to the structures that can be used as a simple index into a list, avoiding the expensive map search. The problem still remains for thread-agnostic access methods.

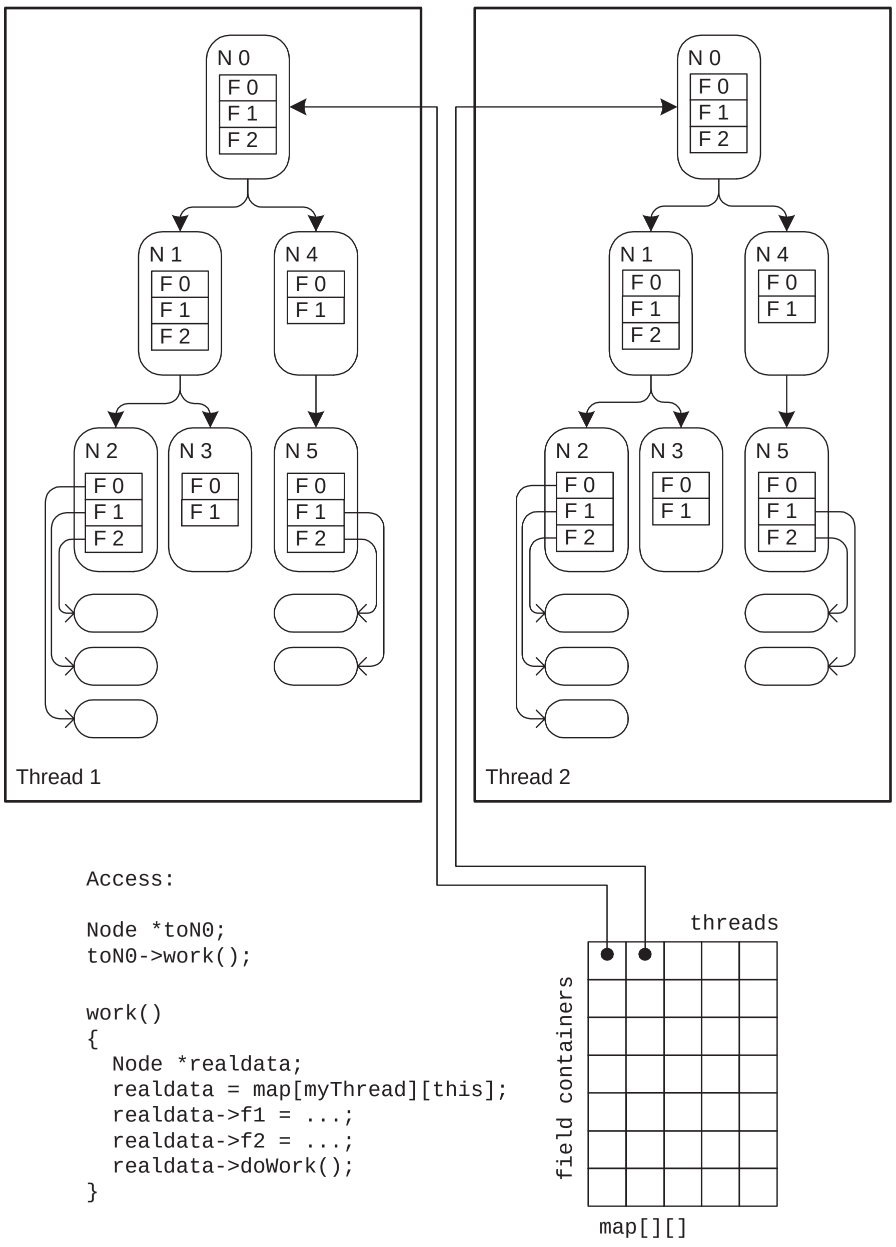
If a function can not be specialized to a specific thread but should rather be able to run in every thread a simple pointer to the data element needs to be mapped for every access, which can become a significant burden soon. Alternatively it can use the numeric index instead of the pointer, still necessitating a table lookup for every access.

These costs for very low-level actions like every data access decrease the attraction of parallel trees significantly.

Replicated Fields

The problem of parallel trees is the mapping from a pointer to the thread's instance. This mapping can be easily avoided by replication on a lower level: at the separate field (see fig. 4.6). As the principle of data hiding votes against direct access to the separate field, the access goes via access methods. These methods can use the thread id as an index into the actual data field for the thread's data. Thus it is possible to transparently use the data pointer in every thread and no global mapping needs to be done.

The problem with this approach is the spatial organization of the data. The data for the different threads is very close together and will probably end up in the same



```
Access:

Node *toN0;
toN0->work();

work()
{
  Node *realdata;
  realdata = map[myThread][this];
  realdata->f1 = ...;
  realdata->f2 = ...;
  realdata->doWork();
}
```

Figure 4.5: Parallel Trees

cache-line. In a multiprocessor system, where every thread runs on its own processor, this will lead to separate processors changing the same cache-line at the same time, necessitating expensive synchronization. Furthermore the size of the data structure

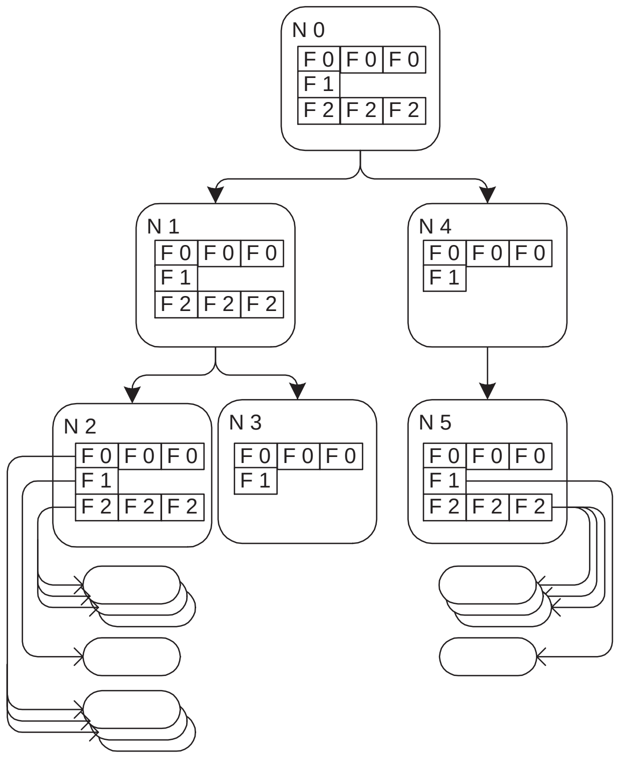
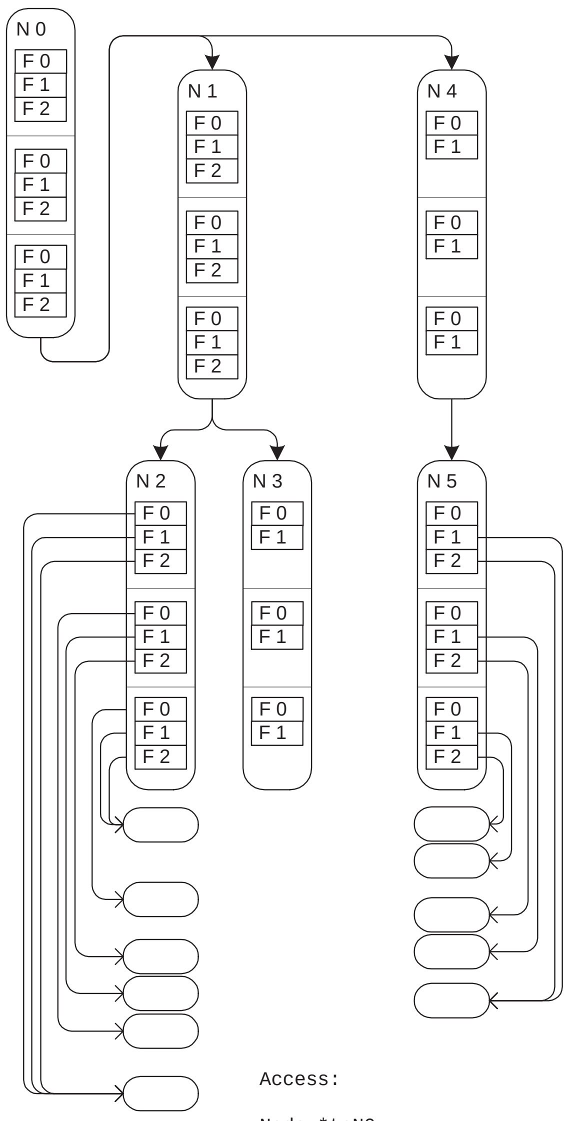


Figure 4.6: Replicated Fields

increases significantly, so that it might not fit into a cache-line as easily or not at all, diminishing cache efficiency significantly.

Replicated Containers

A synergy between the two previous approaches, that captures the advantages of both, is using replicated containers (see fig. 4.7). Stepping one level up from the replicated fields alleviates the cache problems while at the same time keeping the positive aspect of the pointer being valid in every thread. To access the correct version of the data the replicated containers approach needs access to a global thread identifier, similarly to the parallel trees approach, but it does not need the map or table lookup, as knowledge about the base address of the first version of the container and of the size of the container allows directly accessing the thread's copy.



```
Access:

Node *toN0;
toN0->work();

work()
{
  realdata = this + myThread * size;
  realdata->f1 = ...;
  realdata->f2 = ...;
  realdata->doWork();
}
```

Figure 4.7: Replicated Containers

4.3.3 Synchronization

Replicated data allows free asynchronous thread operation. But the threads have to work together to form a single application, thus they have to synchronize their different structures at some points.

Synchronization is between at least two partners. In general synchronization between more partners is split into separate pairs. But even with only two partners there needs to be a consensus whose data has priority. This has to be defined by the different roles the threads take, a democracy doesn't work in the context of a scenegraph.

Complete copy

The easiest method of synchronization is a complete copy (see fig. 4.8). As the

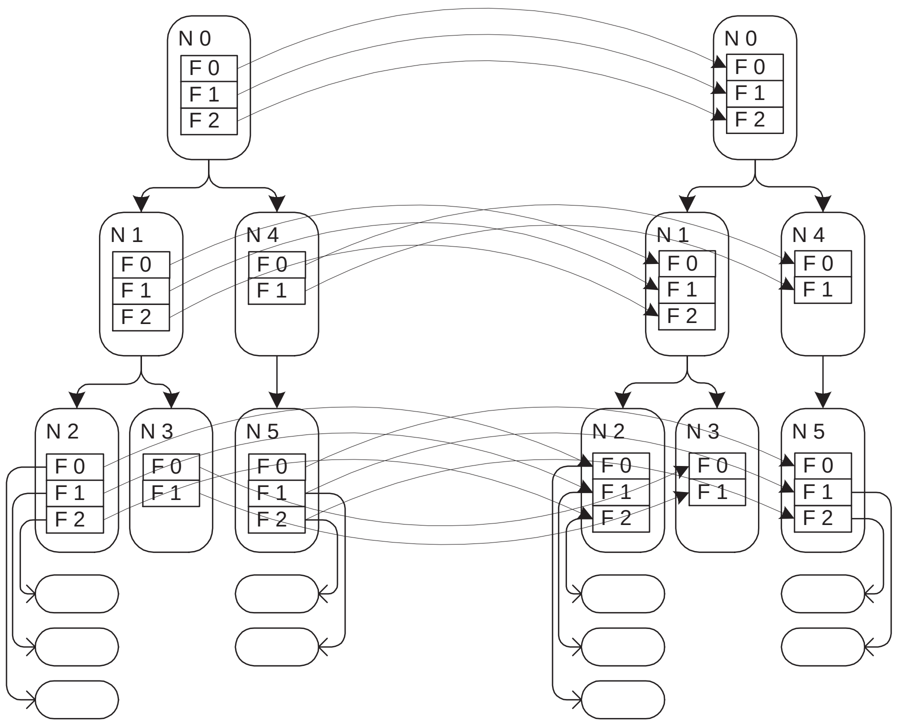


Figure 4.8: Complete copy

complete copy is only able to propagate the changes made in one thread to another, it can only be used for simple pipelined parallelization like an APP-CULL-DRAW pipeline. Asynchronous worker threads like an intersect or collision thread are not possible, as they have to consolidate changes made in both threads into a consistent whole.

The complete copy is very simple to do and needs no additional data structures. Its biggest problem is its inefficiency. Not all of the scenegraph changes for every frame. The structure changes very little, in the worst case the whole geometry is regenerated for every frame, e.g. for particle tracing for flow field visualization. But in general only very small parts of the scenegraph change, little more than the camera position and orientation. In these cases copying all the data is extremely inefficient.

Change Flags

If not all of the data is to be copied, there has to be a way to find out which data has to be copied. One way to decide which data to copy is to keep flags indicating changes inside the containers (see fig. 4.9). These flags are set as soon as the data is

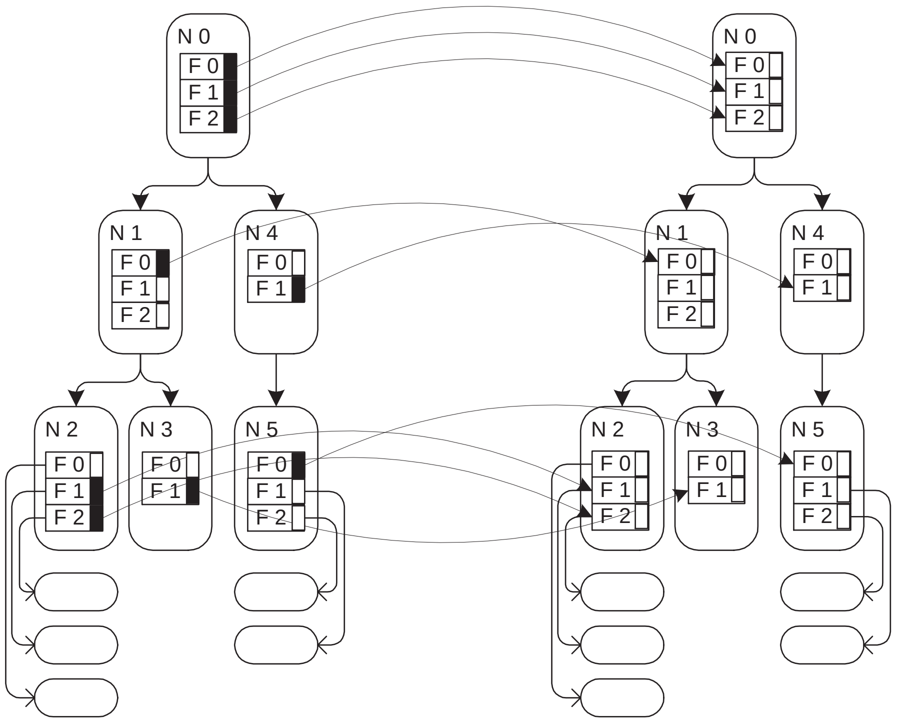


Figure 4.9: Change-flags

changed. For synchronization a traversal of the tree identifies the changes and copies the changed parts to the other thread's data. Keeping the flags in the container reduces the danger of leaving the cache when accessing the data.

The granularity of the change flags can be twofold.

The simple case is a single flag for a whole container, i.e. any change to one of the container's fields sets the same flag. This reduces the amount of memory needed for flags to a minimum. However, for big containers it might be too coarse to be efficient, as copying a large number of fields can be expensive.

A finer approach uses a flag for every field. This reduces the copying to the relevant parts of the scenegraph, while the additional memory cost of one bit per field is not significant enough to cause problems.

The problem this approach has is the traversal. The whole tree needs to be traversed for the synchronization. As the tree for a complex scene can get big, tens of thousands of nodes, a traversal is not necessarily a cheap undertaking. Even in dynamic scenes not all of the nodes are going to change for every frame, so the traversal cost might not pay off. Keeping the flags in the container reduces the danger of cache misses, though, so a clean argument cannot be made.

For relatively static scenes a different approach promises more efficiency, though.

Change List

Instead of reserving change flags in the container itself, the changes and only the changes can be recorded into a global list (see fig. 4.10). The change list records the changed containers and their fields. For synchronization the change list is traversed and the changed fields are copied. In a typical rather static scene the change list will be short, much shorter than the number of nodes.

The change list has the further advantage of being a separate object, not integrated into the scenegraph. This allows copying of the change list independent of the scenegraph, so it can be recorded for later use. One such use is the synchronization of multiple asynchronous worker threads to a central application thread (see sec. 4.4). This advantage is also a disadvantage, as the change list will be in a different place in memory from the container itself, thus increasing the possibility of cache misses.

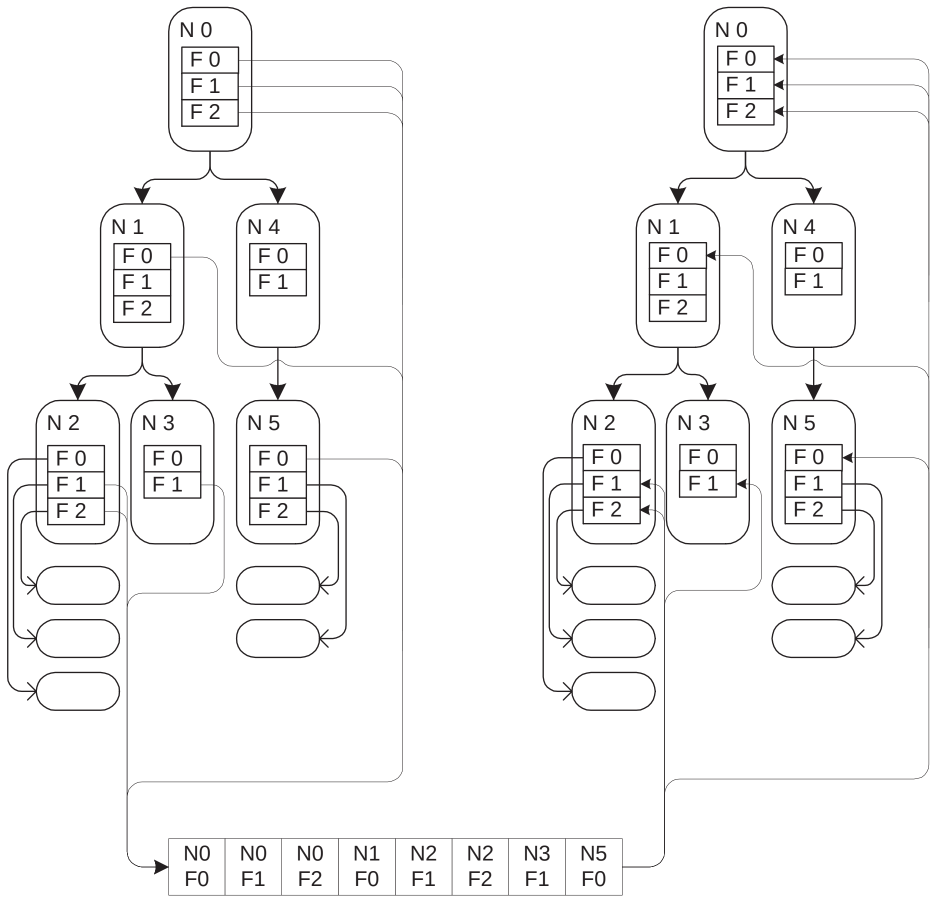


Figure 4.10: Change List

Conclusion

Complete copy is a rather theoretical option, due to its inefficiency and restrictions. The choice between change flags and change lists is not quite as clearly cut. For highly dynamic systems with a simple parallel structure change fields can be very efficient. In general though, change lists allow more flexible threading models and will be more efficient for the rather static scenarios that dominate most current VR/AR applications.

The OpenSG implementation targets a very flexible system to allow multiple asynchronous threads working on the scenegraph. It also tries to keep the thread structure as open as possible.

Therefore it uses a replicated data model, based on the duplicated container concept and change lists.

4.4 Threading Models

Being able to synchronize two threads is just the beginning of a parallel system. In general there will be more than two threads which need to be synchronized. Synchronizing multiple threads can be subdivided into multiple two-thread synchronizations in different ways, which gives rise to different threading models. Additional aspects in the thread models are concerned with the use of shared or separate data. This is not a hard distinction. Even given replicated data, multiple threads can work on the same data if it makes sense.

Pool

The pool is just a group of threads that are used as a pool of workers. Possible uses are a general producer-consumer pool or as a group of workers for a parallel traversal (see sec. 4.2). All members of the pool work on the same aspect, as such care has to be taken that their work doesn't interfere with each other.

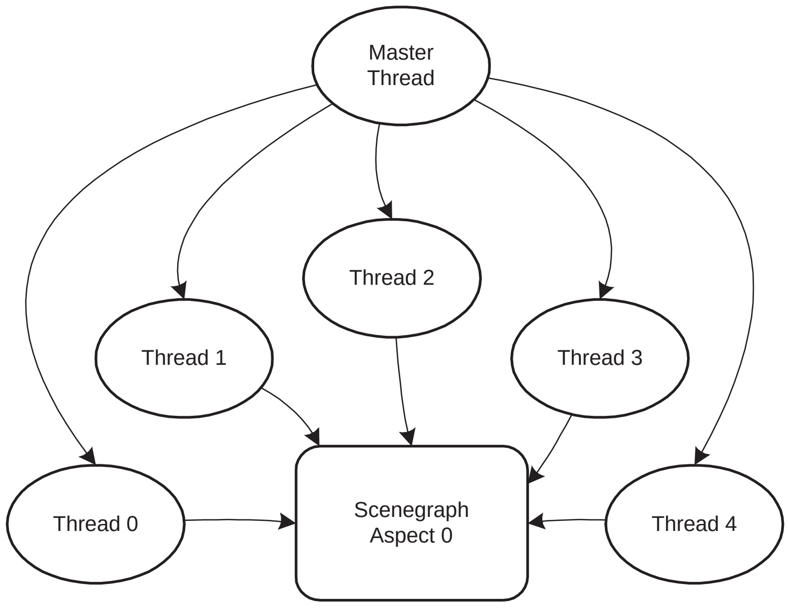


Figure 4.11: Pool model: a group of threads working on the same aspect

As all the threads work on the same aspect, there is no need for data synchronization. The thread's coordinator has to ensure integrity.

Pipeline

The simplest threading model with replicated data is the pipeline (see fig. 4.12). Here the data flows linearly from one thread to the next. By synchronizing to the next thread in the pipeline before being synchronized by the previous thread data will only be passed on after it has been worked on. As the data at the end of the pipeline

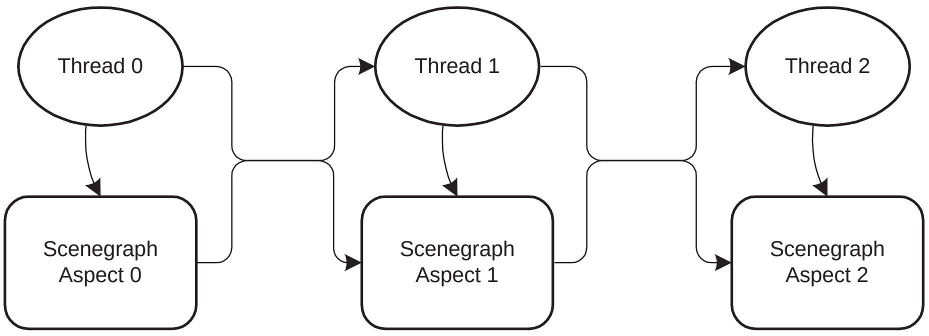


Figure 4.12: Pipeline model: direct flow of data in one direction

disappears from the system the last element of the pipeline has to be concerned with output, e.g. to the screen or to any other output medium. This is the principal use of the pipeline: distribute the work needed for output across several threads.

The utility is limited by the added latency that every step of the pipeline incurs. In the most common case of visual output, which is where the pipeline was used first by Performer, the latency associated with every stage is typically a frame. At 60 Hz that means 16.67 ms, which is not an insignificant factor in the overall latency budget. Thus there are rarely more than two pipelines stages (e.g. cull and draw) in a visual output pipeline. Haptic pipelines are even worse due to their more stringent latency requirements.

Fork

The fork is a specialized form of a pipeline that splits the incoming data into two or more directions (see fig. 4.13). As in the general pipeline, data only flows in one direction, but the changes are not applied to a single next step in line. Instead it is used to update multiple dependent stages.

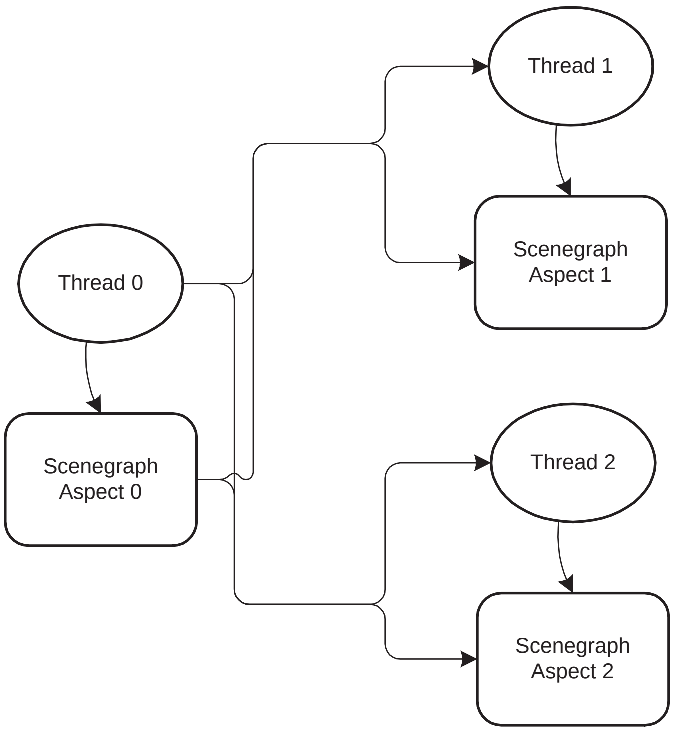


Figure 4.13: Fork model: split the execution into separate directions

As the fork is similar to the pipeline its uses are in a similar area. Forks are usually used to split up a pipeline for multiple output streams, e.g. the rendering threads for a CAVE[23] or Powerwall[129].

Star

The star is turntable-like core organization model. It coordinates, merges and distributes the data and activity of several independent workers (see fig. 4.14). The master thread holds the master copy of the data, all other threads should, after synchronization, have the same data it has. To do that the master thread has to carefully manage its change lists.

As the thread's change list only stores the changes that happened since the last synchronization it is not enough for the star master, which has to synchronize itself with

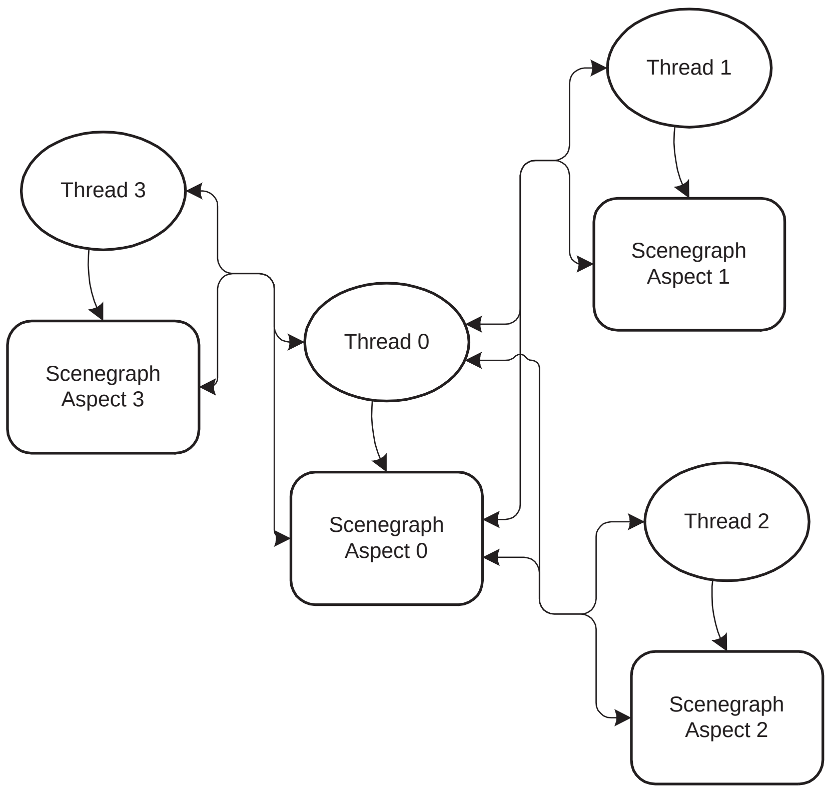


Figure 4.14: Star model: one central master thread

a number of different, independent threads, to keep a single change list. The star master ke a number of change lists, one for each dependent thread. Whenever it synchronizes itself with another thread, the current contents of its change list are appended to the change lists of all threads. Only then is the synchronization done, using the change list assigned to the thread to be synchronized with. This ensures that every thread receives all the changes that happened since it was last synchronized, and allows synchronization of different threads independently of each other, even at different rates.

As a special case a slave thread can also be used without an associated change list, if it is only going to be used as an input server thread. As these are independent of the system's state they don't need to be kept up-to-date with respect to the system's data.

The star is the central model for a coordinating master in the system. Thus there is usually only one star in a system used to coordinate all other threads.

4.4.1 Examples

Fig. 4.15 shows an example of a simple multi-threaded system. It uses a star as the

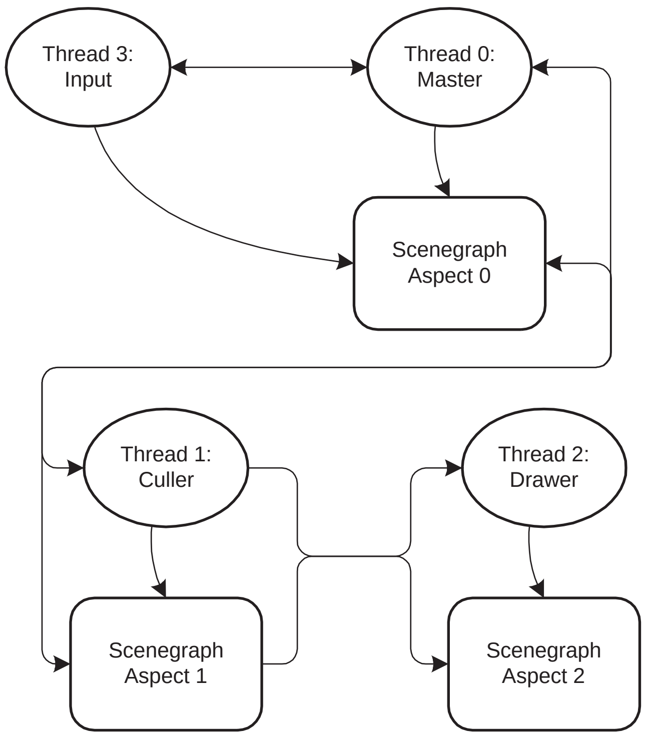


Figure 4.15: Simple multi-threaded application

central coordinator. An input thread that reads the input devices feeds into the star, which outputs into a short pipeline that drives a display. This setup can even make sense in a single-processor environment. The input thread only needs to execute whenever new input data arrives and as such does not need a lot of processor time. At the same time the master thread only needs to actually act whenever new input arrives and update the rendering-relevant data. The busiest thread will be the drawing thread, but even it may not have to run the whole time. Modern graphics hardware can handle some time-consuming operations like filling large pixel areas or rendering simple geometry stored on the graphics subsystem autonomously. It will stall the drawing thread and allow other threads to run until the time-consuming work is done. Thus it is possible that there might still be processing time left over, even though there are three threads in a single-processor system.

A more complex example is given in fig. 4.16.

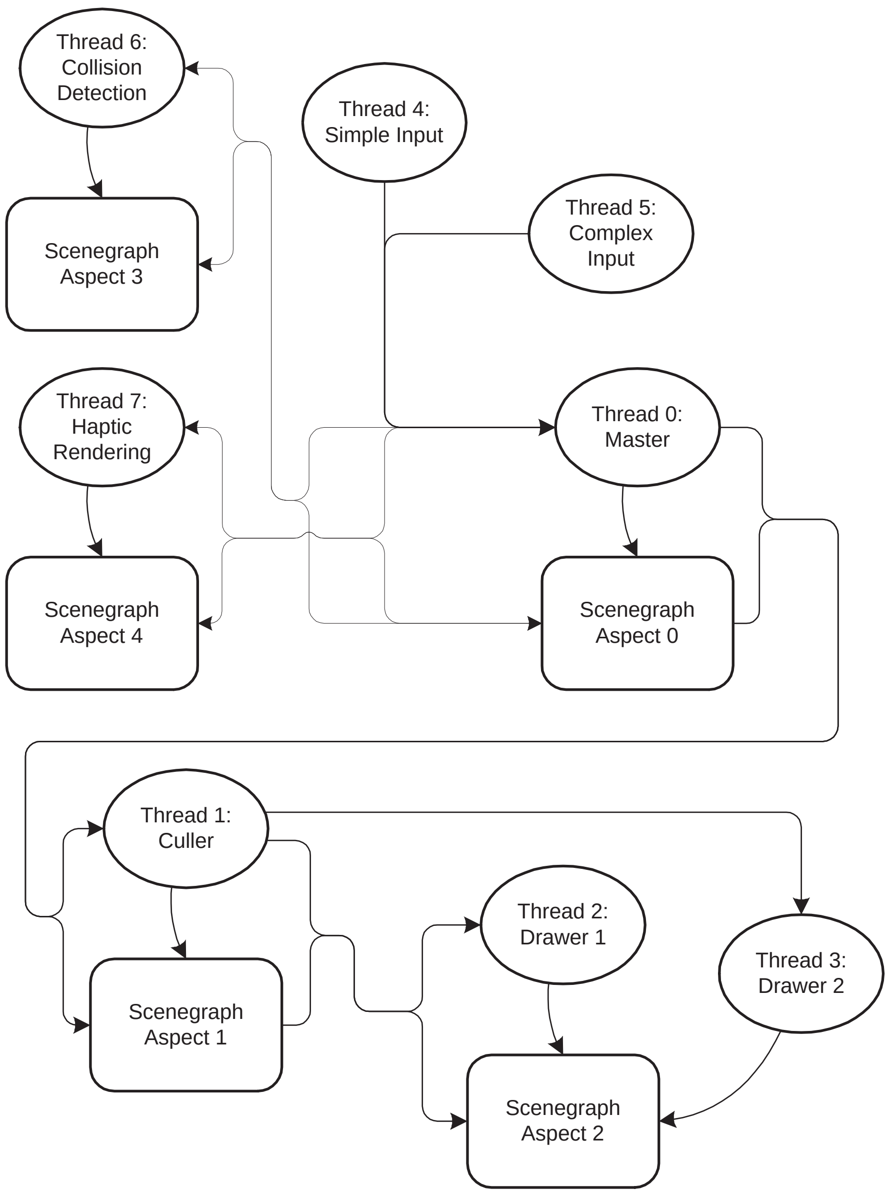


Figure 4.16: Complex multi-threaded example

It also uses a star as a central coordinator. But everything else is heavier. It uses two threads for input handling, one for the low-impact devices that just need reading a serial port like magnetic trackers, and another for processor-demanding input like an optical tracking system or speech recognition. Output is handled by a single culling thread that drives two independent drawing threads feeding two separate graphics pipes. Separate threads are used for collision detection and haptic rendering.

4.5 Distributed Systems

The discussion so far has only talked about shared memory systems, as it assumes possible access to all data by all threads. Another class of parallelization is using several independent systems that do not share memory. These systems have been in use in the scientific computing community for a while now, as they provide high performance at a low cost due to the use of COTS parts.

Distributing tasks in a VR system across a distributed system is rarely done, due to the impact on latency that a loosely coupled system has. There are distributed VR systems for multi-user applications that use different machines for different users, but they use specialized protocols that can handle the low bandwidth and large latency of a network connection.

One area that is gaining significant interest lately is using a cluster of PC systems for the rendering task in the VR system. Commodity graphics hardware has significantly improved in performance lately (see sec. 2.2), approaching formerly high-end graphics workstations very closely. One area they're lacking in however is the availability of multiple high-performance output channels, due to the availability of only one AGP port in a system. Thus to use PC systems for a multi-screen or very high-res application multiple independent systems have to be combined in a cluster.

How to distribute the work in cluster rendering system is a research topic in itself and is not discussed in this work. The interested reader is referred to [102, 103, 104]. The topic here is just the integration of a distributed system into a scenegraph. Other alternatives to including it into the scenegraph is the distribution of the low-level OpenGL commands [18, 47, 46, 83] or the distribution of the full appliction.

The former has the disadvantage of a possibly very large data volume, as it can not benefit from information about shring and other higher-level information inherent in the scenegraph. The latter is the typical approach taken by long-range distributed systems like distributed simulation [113] or internet-capable games [32, 12]. It has by far the lowest data volume, but the demands on the application are significant, as it has to do the whole distribution work.

A relatively generic way of distributing full applications is distributing the user input to multiple indepedently running applications [13, 88]. It works with little or no application changes, but only as long as the application really only depends on the input to define the output. If randomness or view-dependent calculation are used, the synchronisation is more difficult. It is also problematic to access external data sources, as multiple applications will need consistent data. Thus the utility of this approach is limited, especially when the application needs to do any aignificant amount of computation, which would have to be replicated on all cluster machines.

A middle ground is distributing the scenegraph changes.

4.5.1 Scenegraph synchronization

The synchronization system described in sec. 4.3.3 is designed for a shared memory system. However, the information that is stored in a change list can be used as a base for distribution (see fig. 4.17). To do that for every change list item an identification for the associated field container together with the new data has to be transfered over the network. On the other side the change list integration has to copy the new data from the network into the field container's field, very similarly to the standard synchronization.

The only problem apart from necessary endianness conversion between architectures is the field container identification. The parallel data system described in sec. 4.3 has been specifically designed to use pointers as the prime mean of identifying field containers. These will not be valid on another machine, thus across the network another identifying means has to be used.

It would be possible to add a numeric identifier to the field container that could be used to look up the corresponding pointer on the other side of the net. This id would increase the size of the field container, which is not desirable. Alternatively the pointer itself can be used as an identifier. it can not as easily be used as an index to look up the pointer, but using a hash map or similar mechanism the lookup can be made fairly efficient. As the network is most probably not capable of transferring large amounts of data for a large number of field containers in the allocated time, the lookup shouldn't be a significant burden on the processing time.

This approach has been implemeted [100] and proves to be working very efficiently. Static scenes (see fig. 4.18) create nearly no network traffic, dynamic scenes can be handled even over standard networks. Fig.4.19 shows 15000 particles being calculated on a host system and being transfered and rendered at interactive frame rates over a 100 MBit Ethernet.

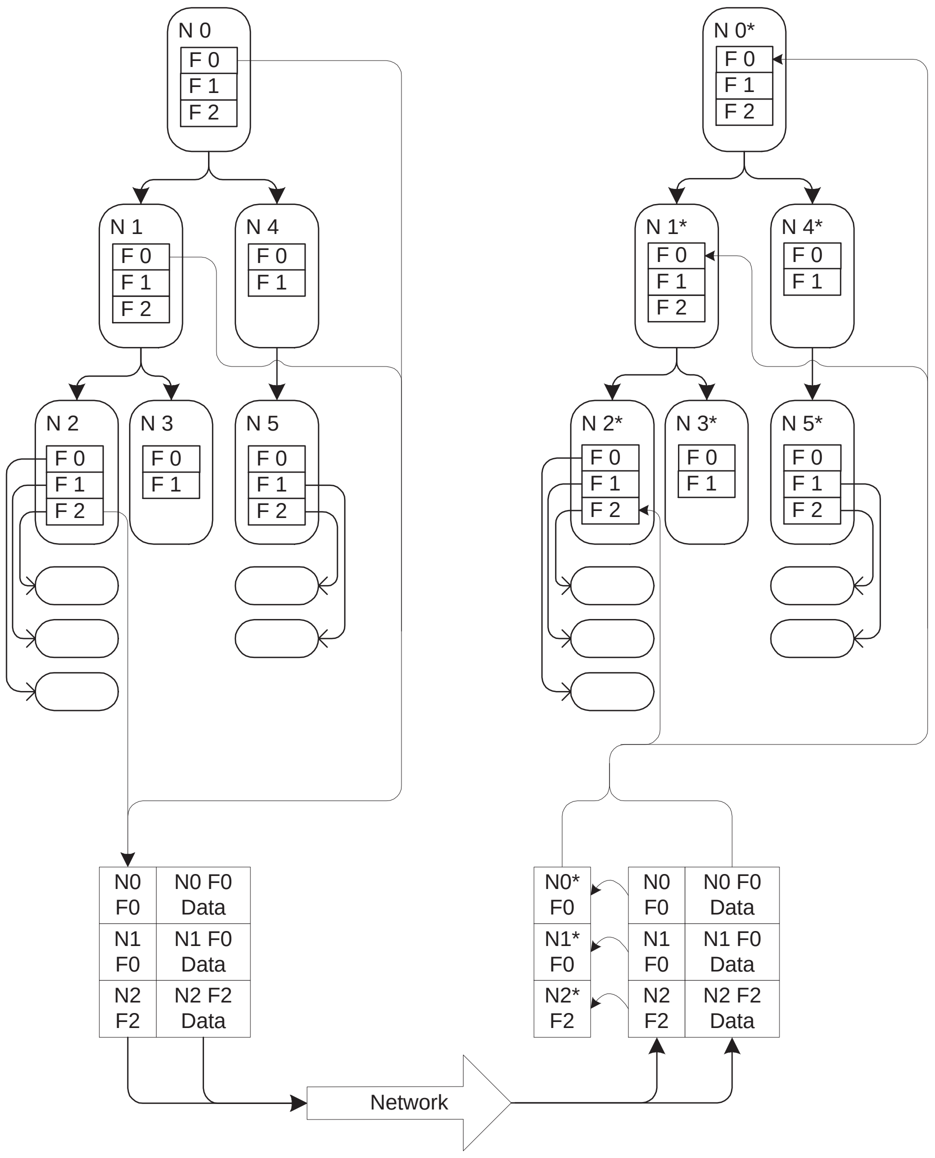


Figure 4.17: Change data marshaling

If a frame of latency can be tolerated, the data transfer can become insignificant timewise, as it can happen in parallel to rendering the last frame.

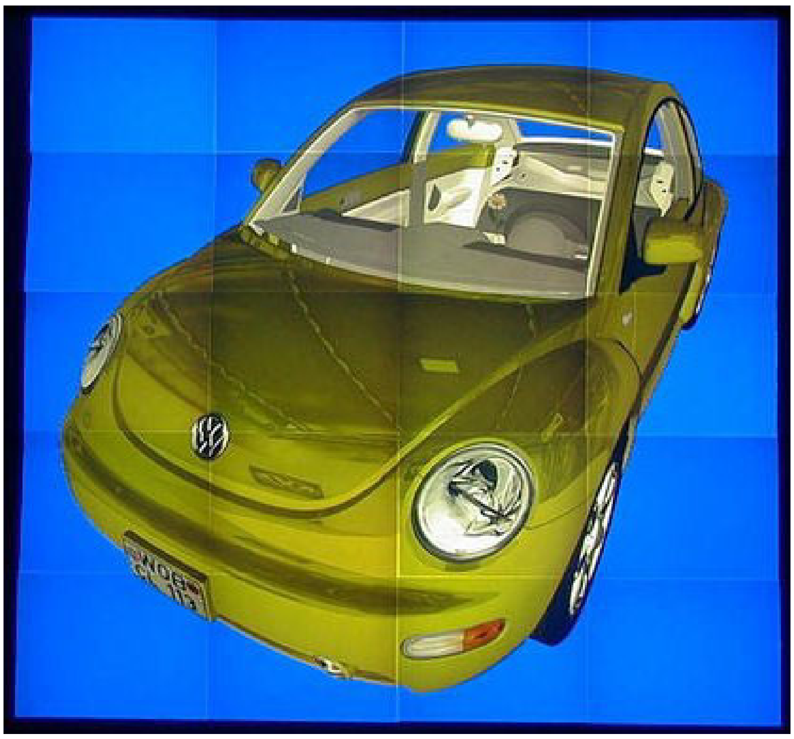


Figure 4.18: VW beetle on the NCSA Wall (Model courtesy of Volkswagen)

4.5.2 Swapping Synchronization

Synchronizing the scenegraph ensures that the distributed system works on consistent data. If the distributed system is going to be used to drive a multi-screen projection system like a Powerwall or CAVE, another synchronization becomes significant: synchronizing the buffer sw of the different graphics boards.

When multiple independent systems drive a multi-screen display, all of them have to swap between front and back buffer at the same time, otherwise the edges between the domains of the different machines will show discontinuities and destroy the illusion of a seamless display.

The specific demands of the swapping synchronization lie in the low latency needed. High-end machines have specific hardware to synchronize independent graphics pipes, but the low-end PC clusters hardware doesn't have that. A full frame at 60 Hz is only 16.67 ms. A buffering TCP network stack can easily eat up that time before it sends out data to the network and thus stall the whole system. Lower level protocols like UDP don't suffer quite as bad, but the different protocol layers still take up measur-

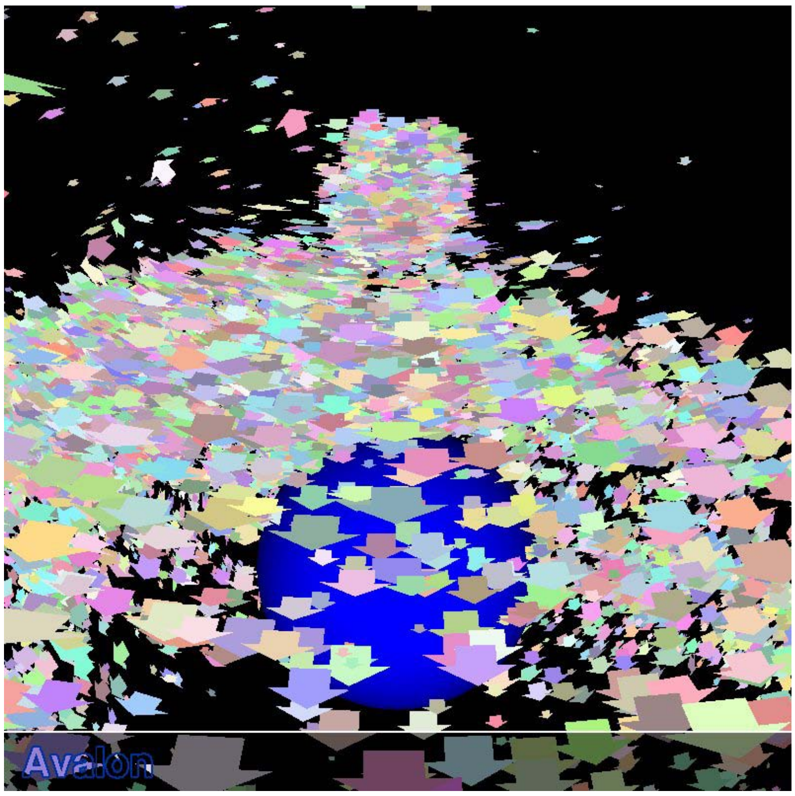


Figure 4.19: Distributed Particle System

able time.

The network approaches have the advantage of not needing additional hardware, as the network is used anyway. If the goal is not very high framerates the latency and speed is acceptable. Even the variance can be accepted when doing passive stereo systems. For active stereo or high frame rates a faster solution needs to be found.

Hardware Synchronisation As the machines to drive the display will be located close to each other, an alternative way of synchronizing them is to use a direct connection.

A standard PC has a number of low-latency I/O ports, namely the serial and parallel ports. These are very directly driven by hardware which can be manipulated by the application. There is no software overhead involved, which reduces the latency significantly.

To use these ports to synchronise multiple machines additional hardware is needed.

The same problem has been approached by the high-performance computing community a couple years ago, and the PAPERS[77] networking system has been used with great success. The same system and the available hardware can be used here.

4.6 Summary

This chapter analyzes applications of parallelism in a VR system and different ways of exploiting this parallelism.

An important but nonetheless limited subset of actions can be applied to a scenegraph in parallel without replicating data. In general however replicating data is needed to shield the effects different processes have on the scenegraph from each other. The different alternatives to organize this data lead to the concept of replicated field containers as the most generally useful solution, which has also been implemented in OpenSG. Different threading models that use the replicated data concepts are described together with their applications.

As parallelism is not necessarily restricted to a single machine an extension of the concept to a distributed cluster system for rendering is developed and described. An important special problem of a distributed renderer is the synchronization for double buffer swapping.

The concepts developed in this chapter allow the efficient use of multiple processors or threads for a wide variety of tasks. The different approaches make it possible to use the optimal setup for every task, in a single integrated system.

Chapter 5

Graphics Hardware Handling

Computer systems develop fast. Computer graphics systems develop faster. While the processors have been following Moore's law (doubling performance every 18 months), the performance of graphics ch has been doubling every 9 months, or about twice as fast. The trend is slowing somewhat due to a much slower increase in available memory bandwidth, but it will keep increasing steadily for the foreseeable future. By now low-end PC graphics accelerators easily outperform systems that had cost a hundred thousands dollars or more just 5 years ago.

Coupled with the increase in performance is an increase in features, and an increase in diversity. In an attempt to gain exclusivity hardware developers add unique new features to their systems that distinguish their products from the competition. These new features either open up new possibilities or simplify methods that were difficult to achieve before. To take advantage of the latest hardware a scenegraph system has to be able to use these new extensions. At the same time there is always a demand to support older hardware, of course at a reduced speed, but hopefully with the same features. It is not possible to satisfy both demands completely, but both are valid and have to be considered.

To access all the features of the graphics hardware an appropriate low-level API has to be used. As portability is one one goal of a widely usable scenegraph system the only available low-level API is OpenGL[16]. OpenGL is the most widely used graphics API and is available from the smallest platforms like the hand-held Palm Pilot[110] via all current PC graphics hardware to workstations and graphics supercomputers like sgi machines. There is no alternative and no successor in sight, so OpenGL is the right and only choice for a portable high-performance scenegraph.

OpenGL also has the advantage of being open, so that all hardware manufacturers can add private extensions to support their new features, thus OpenGL is usually able to support all the latest features of a given hardware. It's rather strict definition [107] also encourages manufacturers to write conforming drivers and to also specify their own extensions very exactly, reducing ambiguities and surprises when using them.

But just being able to consistently use a feature or a system is not the same as using it to its fullest potential. Every hardware has a sweet spot, where it performs optimally and all resources are used, and in general has a limited number of fast paths, which are carefully optimized. When using a feature set that is not on a fast path, performance can suffer significantly, thus some core parts of a scenegraph system have to be able to adapt to the given hardware.

Conceptually OpenGL is a big state machine. Commands set some part of the state, other commands use the current state to render geometry. Changing this state can be costly, and efficiently managing it to use the graphics hardware for maximum efficiency is a central task of a scenegraph system, as described in sec. 5.2.

State handling is only one part of the task. The OpenGL state is rather hardware-oriented and low-level. Applications don't necessarily want to work on that level. They are rather interested in describing the wanted effects, not the exact way to get them. This is especially important in the context of heterogeneous hardware, which will necessitate different ways of achieving a specific result. These higher-level descriptions of the surface attributes and characteristics are described in sec. 5.3.

Realizing the needed flexibility to handle different kinds of hardware platforms is hard to do in the confines of the scenegraph itself. To efficiently combine multi-pass algorithms, temporary images and other methods needed to create sophisticated rendering effects needs a secondary structure specialized for this, the draw tree described in sec. 5.4

As described in sec. 4.1, a scenegraph's uses are not limited to rendering, it is the general container for geometric data used by a VR system. Thus the geometry structure is in a difficult position. On the one hand it has to conform to constraints that allow it to be rendered fast and efficiently. On the other hand it should flexible enough to satisfy the possible needs of the application. It should be memory-efficient and easy to access. It's not possible to satisfy every need equally, there's always going to be a compromise. One possible compromise and its motivation is explained in sec. 5.1.

5.1 Geometry Structure

The geometry node of a scenegraph has to be a connector between the realm of rendering and the rest of the tasks that a VR system has to fulfill.

The structure has to be memory-efficient, as a large part if not the largest part of the memory an application consumes will be used by the geometry data. Memory has become cheap and large, but problem sizes have been outgrowing the memory growth easily. Even worse, memory access is a very expensive commodity, due to the much faster growth in processor speed compared to memory access speed. Thus it is important that the data that is needed and only the data that is needed can be stored efficiently. OpenGL is very flexible in that respect, as it allows the geometry data to have a large variety of formats, e.g. vertices can be stored as 3 or 4 component vectors of shorts, integers, floats or doubles, colors can be stored as 3 or 4 component vectors of signed or unsigned bytes, shorts, integers, floats or double and so on. These formats use different amounts of memory, but are also an important factor to decide if the rendering falls on a fast path or not, thus they can have a large impact on the performance of the system. As these fast paths can vary from graphics system to graphics system, different choices will have to be made for different systems.

Besides the data types provided by OpenGL, specialized representations that consume even less memory are possible. A number of researchers have developed compressed geometry representations [25, 37, 19, 84], which store geometry data in a way makes it possible to handle larger models. Vertices can be represented in a quantized manner relative to the bounding box of the given object and normals can be stored as an index to a quantized normal map that divides the unit sphere into a number of equivalent areas. Connectivity and index information can be compressed down to little more than one bit per vertex. All these compressed representations can not directly be rendered by OpenGL and thus have to be decompressed when rendering, but when compared to having to page the memory for the decompressed model from disk, the result can be significantly faster. Thus it is desirable to be able to support these compressed representations.

To further reduce the amount of used memory, data can also be shared to a large degree, to prevent unnecessary copies lying around. Parts of the geometry data can be used by different objects, e.g. for a filled and a wireframe representation of a dataset.

The whole data also has to be easy to use and access, as a number of different tasks need access to the geometry. Rendering-related tasks like striping or the calculation of normals, as well as other system tasks like collision detection and haptic simulation.

5.1.1 Geometry Properties

A geometry object can have a number of properties that define how it's going to be rendered. The only mandatory part are the vertices, everything else is optional. The optional parts include normals, colors and texture coordinates (several sets for multi-texturing). Other data that needs to be stored with the geometry concerns the types of primitives that the geometry uses (polygons, triangles, triangle str etc.) as well as the lengths and indices used by the primitives. As this data can just as well be stored as a simple array of values, it makes sense to define similar geometry properties for it.

This structure is oriented towards OpenGL, which supports a similar approach to efficiently define geometry data known as vertex arrays. It can also be used to efficiently render the geometry manually, if the higher level OpenGL vertex array functions don't support the selected set of attributes.

In order to facilitate the flexibility needed to support the different data-types mentioned in sec. 5.1 and to support sharing of attributes between different geometries it makes sense to wrap a property into a separate data structure.

This property data structure also defining a generic interface to allow simple access to the geometries data, no matter how the data is actually stored. This greatly simplifies writing tools that work on the geometry, but at the cost of some performance if the data has to be converted to the generic format.

This setup also allows keeping the geometry data in specific memory areas where it can be directly accessed by the graphics hardware, which can make a significant difference in performance [54, 114].

Furthermore it allows the use of data that is not actually stored in the scenegraph itself. By defining a structure that supports the properties' interface the data itself can be kept inside another library or application to prevent replication.

Of course not all of these uses are availabel at the same time (if the data is kept inside the application, it cannot be moved into graphics memory), but the flexibility of the concept allows adaption to the different needs of different applications.

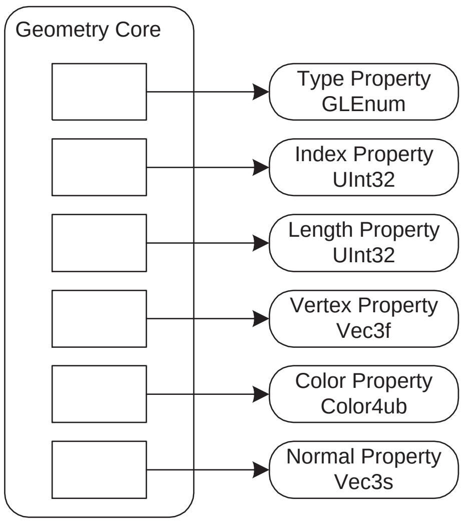


Figure 5.1: Geometry structure

Using this property scheme the geometry itself is little more than a container for the different properties (see fig. 5.1). It knows how to render the data is ke (see sec. 5.1.3) and allows access to the data.

5.1.2 Geometry Iterators

As mentioned, the structure given above is targeted towards OpenGL and efficient rendering. For other tools it can prove to be somewhat tedious to access, though. It does not explicitly keep triangles or faces at all, these are hidden inside the type, length and index properties.

To simplify access to the geometry's polygonal data other ways need to be defined. For consistency reasons it makes sense to use the STL vector interface as a model. Thus iterators should be defined that can iterate through the primitive structure of the geometry. The geometry itself supplies the to create iterators for the first and last primitive while the iterator provides the increment operator, which facilitates interpreting the the geometry as a vector of primitives.

The iterators can also provide a generic interface to the geometry's data, relieving the user from having to access the properties explicitly. Thus an iterator can be used to mimic a structure that uses explicit primitive structures, even though it doesn't.

Even more access simplification can be handled by different kinds of iterators. Up to now only primitives have been mentioned without specifying what these primitives are.

In the most basic case they are the OpenGL primitives that are stored in the geometry. But as the iterators are an abstraction already, they can be more abstract and thus break down the OpenGL primitives into more useful units.

Many algorithms are defined to work on triangles or quads. A single OpenGL primitive like a triangle strip or a quad set can define a large number of of these. Specialized iterators can thus be used to split the OpenGL primitives into the kinds of data the application's algorithms can directly use (see fig 5.2) , greatly simplifying

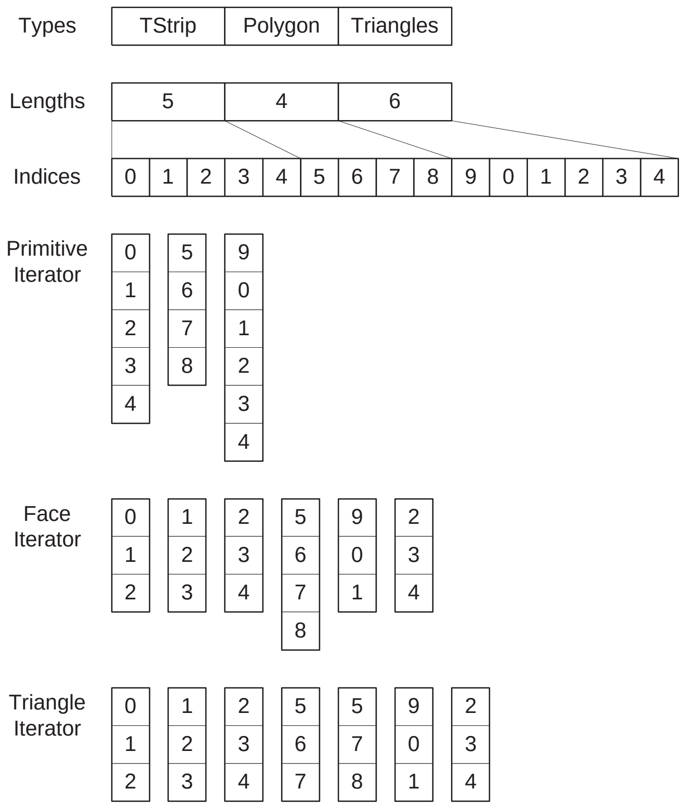


Figure 5.2: Geometry Iterators

application development without endangering the data integrity and the rendering performance.

5.1.3 Geometry Pumps

The structure described above allows a large amount of flexibility. Data can be defined in a variety of different formats, many of which can be directly used by OpenGL, but some of which cannot. It also allows a number of attribute combinations and indirection that are not directly supported by the OpenGL vertex array mechanism.

In order to render all these different combinations a single rendering loop is not adequate. As OpenGL is a very low-level API potentially a very large number of function calls is needed to define all the data of a geometry object and render it. Deciding for every vertex which attributes are to be used is going to be much too expensive. It is even going to be rather expensive to select the right rendering function, which might be vertex array based or not, for every frame.

Thus the decision which one out of the available rendering functions is to be used for the given geometry has to be made beforehand. As this decision only depends on the kinds of data that are used, not on their actual values, the decision can be made whenever the kind of data used by the geometry object changes, which is going to be rather rarely. Thus it makes sense to keep a functor or a simple function pointer to the selected function in the geometry that is used to render it.

Additional complications arise due to the fact that these rendering functions can depend on special OpenGL extensions, e.g. [73, 72, 122, 123]. In a heterogeneous multi-pipe environment, one of the main targets of a flexible scenegraph system, these extensions might differ between rendering pipelines. Thus it is not possible to actually decide on the specific function used to render the geometry, as it depends on the graphics pipe that is going to render it.

Instead the selection functions that is called when the geometry changes can only analyze the geometries attributes and condense them into an index that is used to select the actual rendering function.

The mapping from index to function can be specialized for every used rendering pipe, thus allowing the use of the optimal rendering function for every pipe in the system.

This is also an area where extendibility is needed. As specialized geometry representations can be loaded at runtime as new geometry property implementations, the function map has to be extendible as well as the indexing system. This can easily be done by using a function pointer for the analysis function that can be replaced by newly loaded modules, which then can call the previous analysis function if they can't handle the situation themselves. To make the index unique the maximum used index is stored in a central geometry pump factory and will be incremented by newly loaded modules to include their available functions.

The described structure allows a very flexible and extensible geometry setup which facilitates simple and generic access to the data for the application and system tools, while at the same time allowing specialization to the given graphics hardware to reach optimal rendering performance.

5.2 State Handling

Graphics hardware as abstracted by OpenGL has a large internal state that defines how primitives are being drawn. This state includes parameters for lighting like diffuse, specular, ambient and emissive material colors, but also things like texturing, transformations, blending and many other variables. Changing this state can be very expensive for highly integrated hardware.

The rendering pipeline that transforms defined geometry into an image needs to be put in hardware to achieve real-time performance for complex models. But to obtain good performance it not only has to be put into hardware, it also has to be pipelined, i.e. different parts of the rendering pipeline work independently on different geometric primitives. It also can be parallelized, i.e. multiple independent units working in parallel on different parts of the geometry. The benefit can be immense, as some important operations can be 'embarrassingly parallel', especially pixel filling can be parallelized to hundreds of independent fill units.

To be able to keep these many parallel units and the pipelines in hardware they have to be rather simple. As a number of different pieces of geometry may be in operation at an given time, changing the state of the hardware has to be synchronized with the data flow through the pipeline. Simple hardware cannot do that. Thus the pipeline has to empty before state changes can be applied. Afterwards it has to fill up again before the nominal performance can be reached. If state changes happen often it might not be able to fill up completely, and in the worst case the whole effect of the pipeline is lost.

Another effect of the simple hardware is that it might not be able to directly take the parameters defined by OpenGL, but instead demand them to be preprocessed. A typical example would be the inability to directly use the specular exponent for lighting calculations, but instead needing to get a table of pre-calculated values. Preprocessing these values can be very expensive, so when it has to be done often it can go up to the point where it completely negates the beneficial effect of the hardware.

Not all benefits of state change minimization can be attributed to the simplicity of the hardware, though. Some just stem from the limited resources that have to be managed, like texture memory and texture caches. Textures are usually stored in special, high-performance memory on the graphics board, sometimes the active parts of the texture are even stored inside the graphics chip in a special cache [48, 26]. Transferring data into this cache is going to be a lot slower than using whatever is inside. Therefore it can be a significant benefit to reduce the number of texture changes to the minimum and thus use the texture cache optimally.

Thus a scenegraph system has to take care of managing the state of the graphics hardware in a flexible and efficient manner, to allow the use as much flexibility as possible while at he same time using the graphics hardware as well as possible.

5.2.1 State Changes In A Scenegraph

A simple example of a scenegraph with its states is given in fig. 5.3. The state in this case consists of three types A, B and C which can have one of two values, lower or upper case. All the nodes in the scenegraph have to be rendered using their associated state. They can also be seen as independent nodes of a graph (see fig. 5.4). Every path between two nodes is possible, and every path has an associated cost measured in the time it takes to get from one state to the other. The task of the scenegraph system is to find a path through this graph that reaches all nodes and incurs the lowest possible cost. This problem is well known in computer science as the traveling salesman problem[67, 6]. It is known to be NP complete.

The NP-completeness implies that there is no simple efficient algorithm to find an optimal solution. There are incremental algorithms that can approximate a solution, but they all take significant time. As this problem has to be solved for every frame, of which there are 20 or more per second, an expensive algorithm is not going to be useful. It might find an optimal path through the graph, but it will take longer than just rendering the visible nodes in any order and thus will not be effective. But the NP-completeness also implies that is not possible to find an optimal graph in the available time frame, thus simplifications are acceptable and necessary.

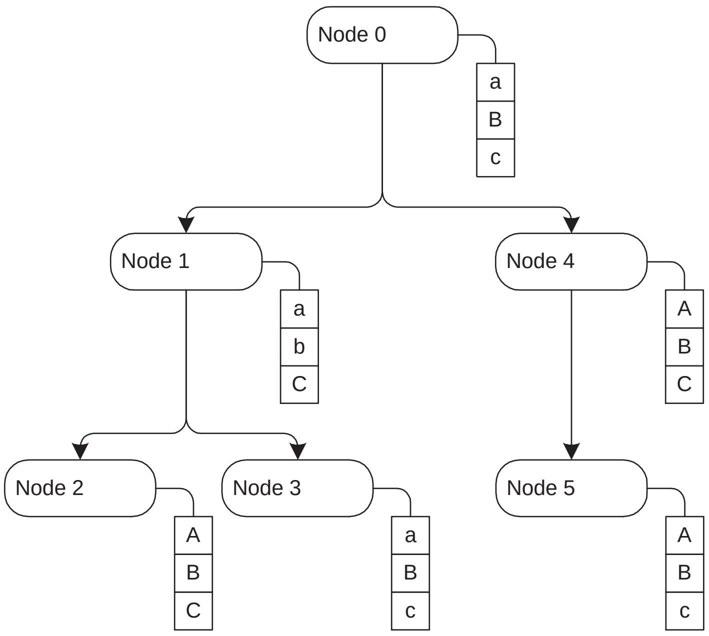


Figure 5.3: Scenegraph with attached states

5.2.2 State Chunks

The first simplification is to reduce the problem space. OpenGL has a rather large state. Using every element of the state independently creates so many possible paths through the state graph that it's impossible to efficiently analyze it.

But not all state variables are truly independent. Many of them are usually changed together, like the material or the texture parameters, or the parameters for a single light source. Others are only rarely used and changed together, e.g. the parameters for lines and points, as only very few geometry uses lines and points at the same time.

Thus it makes sense to group parameters that are usually changed together into a larger chunk and to create different chunks for parameter sets that are rather mutually exclusive. The OpenGL specification [107] gives a first idea about a sensible separation. Further analysis leads to the following set of chunks:

- Transformation: a transformation to be applied to the modelview matrix in addition to the camera transformation.
- Material: the material parameters for lighting calculations.

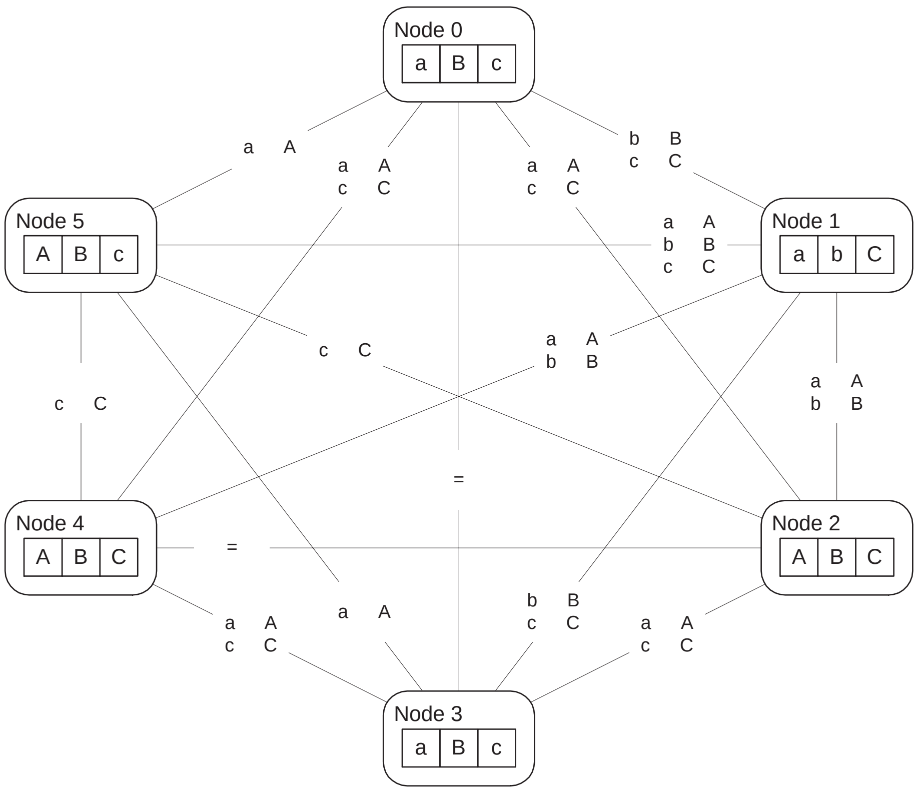


Figure 5.4: State graph

- LightSource: a single light source's parameters.
- Texture: a single texture's parameters including texture environment mode.
- TexGen: automatic calculation of texture coordinates
- TexMatrix: texture coordinate transformation matrix
- LightModel: the global lighting parameters.
- Blending: parameters that define how incoming fragments are mixed with the frame buffer
- Polygon: the parameters that only concern polygons, e.g. stipple pattern.
- Line: the parameters that only concern lines, e.g. line width.
- Point: the parameters that only concern points, e.g. point size.

- Fog: the parameters for defining the gradual blend to a fog color

These chunks cover the OpenGL 1.1 specification. There are other parts of the state that not relevant in the context of a scenegraph, e.g. the pixel pack and unpack definition, these are not handled by the scenegraph's state handling mechanisms. Using these chunks as a basis for state change optimization the OpenGL state has been reduced to twelve kinds of variables.

As this is an area where new hardware adds new features it's especially important to be able to expand it [52, 53, 55, 122, 123]. Again following the golden rule given in sec. 3 this is made easy. New kinds of chunks can be defined at any time. As the chunks are field containers themselves, their creation is managed by the same extendible mechanisms that are described in sec. 3.2, thus existing chunks can be replaced by new ones easily.

The chunks are the basis for state change minimization. But to have a quantitative basis for sorting the states more information is needed. To be able to decide which state changes are more costly and thus should be reduced the cost of a state change needs to be known. As state changes are delegated to the chunks now, they have to know the cost of changing.

At the same time chunks can use the knowledge about themselves to optimize changing between different instances of a specific chunk type, possibly avoiding setting a specific value of the state to the same value it already has. As drivers are optimized for full performance, they don't necessarily check this and leave it to the application. Thus ignoring changes that would set the already active value can have a positive effect and reduce state change costs, too.

The chunks themselves are only the first part. They have to be organized into a whole that represents a full state.

5.2.3 State

The state is the organizing structure for a set of chunks. It's main purpose is to keep all the chunks that form the state for a given object together.

This includes handling of situations where multiple instances of one type of chunk can be used. In the standard OpenGL state there are two kinds of chunks where this is relevant: textures and light sources. These have to be handled differently.

Light sources are unordered. There can be a limited number of them active at any given time, but all of them are equivalent, i.e. it doesn't matter if a specific chunk is assigned to one hardware light source or another, they're all handled equally.

Textures on the other hand are ordered. The operations that combine the different textures with each other are not necessarily commutative and thus textures have to be assigned to specific slots.

To be able to efficiently add chunks to a state and to be able to check if there's already an instance of that chunk in the state every chunk type is assigned a numerical id that can be used as an index into a chunk vector.

As states are used to bundle chunks they are the low-level structure that contains everything needed to render a piece of geometry. They can activate and deactivate themselves and can efficiently switch between each other, activating, changing or deactivating the necessary chunks.

But the states are still a low-level concept intimately linked to the OpenGL state machine. The users of a scenegraph don't necessarily want to know or care about this level and rather work on a higher level, the level of the material.

5.3 Material Handling

Applications do not necessarily want to be concerned with the logics of OpenGL. They need an abstract and logical interface to define surface properties to define how a geometry object is going to be rendered. They also don't want to care about the capabilities of the currently used hardware, the material interface should abstract that away, if necessary use different ways to realize the same goal of rendering the geometry in the specified way.

5.3.1 Simple Materials

In the simplest case a material is just a front for one or several chunks. A standard material that supports the features provided by most scenegraph systems and similar systems like VRML would contain a Material chunk for the lighting equation parameters. In addition to that the standard materials contain a transparency parameter. Transparency can not be handled by OpenGL directly in the most general way. The transparency value has to be assigned as alpha to the diffuse material color or the blend color extension has to be used and the correct blending function has to be set. Thus the transparency has to influence how the Material chunk is specified and also needs a Blending chunk. On top of that it also needs a specific rendering order. This is handled in sec. 5.4.

Simple materials are a useful basis to work with, but their uses are limited. One easy way to make the more useful is to open them up and allow the user to add arbitrary chunks to them. Thus he can depend on the basic material to take care of the basics while at the same time being able to add specific options to it.

5.3.2 Abstraction

One primary use of materials is the abstraction of the OpenGL needs from the wanted effects.

OpenGL gets a lot of its power from the fact that vendors can add arbitrary extensions to it. After these extensions have proved themselves to be useful and are adopted by a number of different vendors they become official and can be supported by everyone [7]. But they don't have to be supported.

For many of them it is still possible to simulate their effects in other ways. Multiple rendering passes can go a long way to simulate more complex effects [66]. The abstraction that the material allows he to shield the application from having to know and to care about that.

On the highest level materials can be defined by a shading language [65, 43, 40, 80, 85, 5, 87, 39]. It is not really a different class of problem, just the final consequence of the material abstraction.

One problem that the abstract material faces is number of state changes. As the material is only concerned with one geometry at a time, having to do multiple passes will result in a lot of state changes. If that has to be done for a number of objects that use the same material, a lot of unnecessary state changes will be used. To alleviate that problem a secondary structure is needed that he minimize state changes on a global scale: the draw tree.

5.4 Draw Tree

The demands of current users are rising. The advent of 3d graphics hardware in the consumer space quickly increased the developer community significantly and at the same time increased the pressure to create new and stunning visuals. This could not be done strictly within the confines of the available graphics APIs like OpenGL, new methods had to be developed.

Current rendering algorithms are very dynamic. They use multiple textures per polygon and multiple passes over the same geometry with different materials that are blended together to create new results. Other features require temporary images to be created, e.g. projected shadows [108, 27].

Supporting all these cases in a generic scenegraph puts a high demand on the system. Doing it efficiently demands a global approach that can decouple the sequence in which geometry is rendered from the order it is specified in the scenegraph and split and combine multiple different rendering passes in an efficient way to minimize state changes.

It would be possible to do that within the confines of the scenegraph using scenegraph rewriting [15]. Scenegraph rewriting creates temporary manipulated versions of the current scenegraph that are used by specific rendering or optimization actions. But scenegraph manipulations are not lightweight, due to the parallel process handling as described in sec. 4.

Thus it makes sense to create a specialized secondary structure that is used for organizing the commands that will be executed to create a picture. Some systems used linear lists of commands to do that [89, 99], but for the current and coming demands that is not flexible enough. A hierarchical structure is better suited for that: the draw tree.

5.4.1 Structure

The draw tree is a specialized version of a scenegraph, only to be used for the final rendering pass and for state change optimization. Thus there is no need for it to be fully featured, instead it has to be fast to create and to traverse.

The draw tree will be recreated for every frame, as it only stores the visible objects. Thus there is no need for it to thread-safe, as it will only be used in one thread. There is also no need for a complicated type system.

For fast creation the draw tree should be created with the least possible number of memory allocations. If possible these should all have the same size, or at least a small number of different sizes, so that they can be satisfied from a small number of pools. This also demands storing the child relation without dynamically sized arrays, instead the children will be linked together using a linked list. To simplify child insertion, the list itself is singly linked but the node ke pointers to both the start and the end of the list, see fig. 5.5. As a consequence nodes cannot be shared, which would be a very desirable property, as for multi-pass sections the exact same nodes are needed multiple times. To still support the sharing without giving up the linked list a special node is used which does not use the linked list for its children but ke just a single pointer to its single child, which can then be a reused tree segment (see fig. 5.5 on the right).

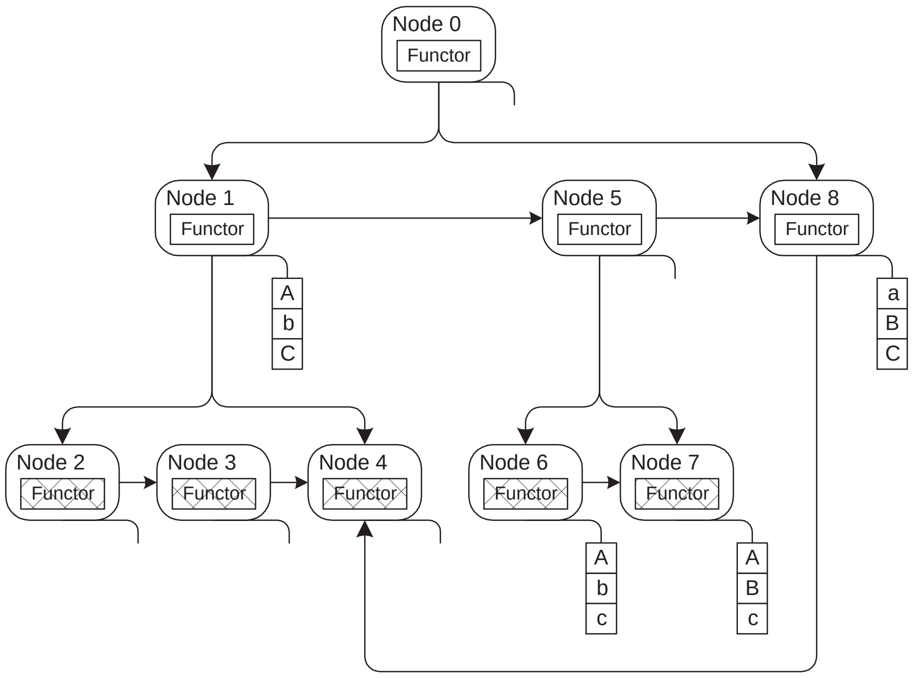


Figure 5.5: Draw Tree structure

Finally, the active parts of the tree are the leaves. They carry a reference to a state, used to render themselves, and a functor that is called during rendering traversal. This functor is the one actually calling the drawing functions. The flexibility given by the functor allows arbitrary extensions without any change to the system.

5.4.2 Reordering Constraints

One of the main purposes of the draw tree is the efficient serialization of multi-pass situations and the support of temporary images. At the same time it is used for state change minimizations.

These two tasks contradict each other, as the serialization demands a fixed rendering order, while the state change minimization needs to change the rendering order to actually do something.

A solution to solve this contradiction is the creation of reordering constraints on the nodes, or the introduction of different types of nodes. There are four configurations that are of interest.

One is a node whose children are completely free to be reordered in any way. This is the node that ke the normal rendered scenegraph nodes. It is called a soft node, as it can be changed arbitrarily. It can even be deleted and its children redistributed.

Closely related but not quite as unselfish, is the squishy node. It can not be deleted, but its children can be reordered arbitrarily. It is mainly used as the first level after hard nodes.

The second variant is the hard node. It can not be manipulated at all by the optimizing process, all its children are rendered in exactly the same order they are given and there can be no other nodes added. These nodes are primarily used close to the top of the draw tree, they are used to keep the different setup, rendering and image grabbing nodes in their predefined order.

The last type is the brittle node. They are in-between hard and soft nodes in the sense that their children will be rendered in the given order but it possible to add other nodes in-between those. These allow the mixing of multiple objects with the same multi-pass settings to significantly reduce state changes. The name brittle was chosen because the node can be broken apart, but it can only be put together again in the same order.

5.4.3 Construction

The draw tree itself is just a structuring framework. It is constructed by a rendering action and this action has to take care how to structure it. One goal of the draw tree concept is to split the work needed to actually render a scene from organizing it, to simplify extending the system and adapting it to new applications. The following is just an example that should satisfy a somewhat sophisticated application. For simple cases the tree can be much simpler.

The root of the tree is a hard node (see fig. 5.6).The top level consists of the temporary images needed for e.g. active environment m or transparent mirrors, followed by the main branch, which contains the primary image. Each of these subtrees has a similar

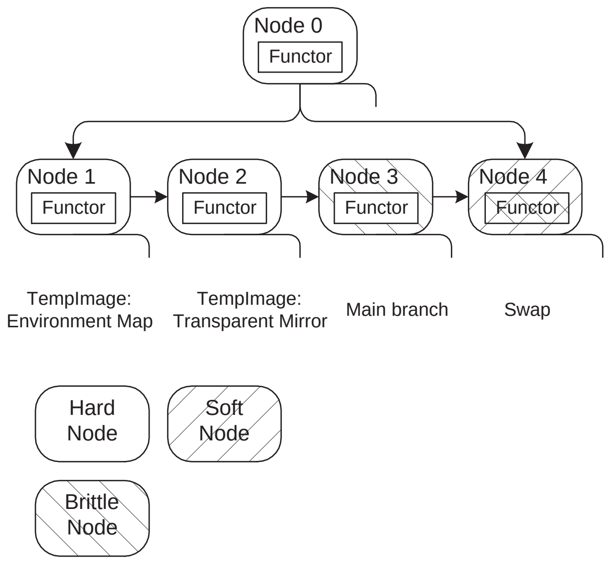


Figure 5.6: Draw Tree example

setup (see fig. 5.7). The first node is used to set up the viewing transformation, followed by the background node. The main part of the tree consists of the visible geometry. In general the geometry will just consist of a soft node with all the visible objects as children. If multi-pass materials are present, they will use a brittle node to wrap their geometry.

For temporary images the last node in the top level is the one that copies the temporary image to its destination, usually a texture. For the main tree the last node sw the back to the front-buffer and thus displays the image.

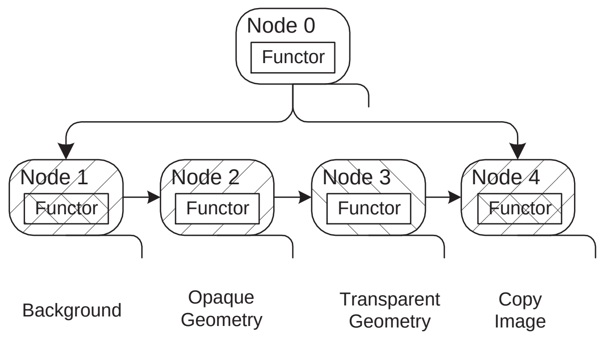


Figure 5.7: Subtree for a single image

5.4.4 Optimization

One purpose of the draw tree is the flexible handling of state change minimizations. As the problem is NP-complete, an optimal solution cannot be found effectively. But the structure of the draw tree allows approximate solutions to be found effectively.

The nodes of the tree can have an associated state, but not necessarily. These state-bearing nodes are the basis for sorting.

The state chunks have an order of effort that is needed to switch between different instances of a chunk. Transformations are relatively inexpensive, texture changes are usually rather expensive. Texture changes show that the order is not a strict order but can heavily depend on context. A texture that is not uploaded into the chip's texture memory will take a significantly longer time to be activated. As this depends heavily on the prior use of this specific texture it skews the sorting order. A local algorithm cannot find the optimal order, but the cost of adding global information can become prohibitively large, alleviating the performance gain of the state sorting. Thus a local sorting based on a hardware-dependent order for the chunks is a good compromise.

The optimizations also depend on the nodes encountered. Hard nodes are fixed, they cannot be changed and thus optimizations have to happen below the level of the hard node.

Soft nodes are the primary base for optimization. All their children can freely be rearranged, thus a simple merge sort on the linked list of children is enough. Soft nodes as children of soft nodes are folded into their parents as long as their children bear their own states, otherwise all the children have to be handled as having the same state anyway. In that case, having them all represented by a single node is more efficient.

Brittle nodes constraint the optimisation operations possible. They can only be shuffled into each other without changing local orders. This is especially effective for several brittle nodes originating from the same material, in the general case the optimizations available are rather limited. Soft nodes can be inserted and sorted into brittle nodes just like into any other nodes.

The described structure allows defining the necessary constraints for multi-pass algorithms with temporary images without unduly restricting the optimisations opportunities necessary to efficiently exploit the available graphics hardware.

5.5 Multi-pipe Handling

There are a number of situations where a single graphics pipeline is not enough. The reasons can be that the resolution that a single pipeline with a single DAC can drive is too small (typically 1920x1200 is the maximum) or that there are multiple projection screens to drive, while nearly every card has only one or two outputs, or multiple pipelines are used for parallelism to increase performance. Multi-pipe approaches are getting increasingly attractive due to the availability of fast cheap graphics hardware in the PC market place. These have to be driven as a cluster (see sec. 4.5), but the general approaches described here still apply.

In situations where there are multiple graphics pipes in a single system they have to be driven by multiple processes to reach maximum performance. Thus the results of sec. 4 are also applied here.

5.5.1 Graphics Library Handling

But the handling of graphics library objects becomes an interesting problem too, in a multi-pipe setup, especially a heterogeneous one. Not all the graphics boards in a system have to have the same type, it can make sense to have a powerful primary display together with a less powerful, cheaper secondary display.

The handling of multiple graphics pipes in one application has its own problems in general. OpenGL is an immediate mode graphics library that uses a currently active state to control the effects of the issued commands. This state is a global state, it is not passed with the function calls. There can only be one active state at any time in a thread. Thus to drive multiple graphics pipes in parallel, every one has to have a separate thread and context.

Object Handling

The separated contexts are a problem, as the context also stores objects that the graphics library manages, like textures and display lists. To simplify applications, these objects should be available to every application on every pipe, regardless of when it was first defined and when it was last changed. At the same time these objects take up precious resources of the graphics pipe, like on-board memory, and as such it doesn't make sense to blindly replicate them across all boards.

The solution lies in a lazy evaluation harness that is supplied by the scenegraph.

Before an object can be used it has to be validated. After that the object is guaranteed to be valid and usable. As this validation has to be done before every use, it has to be as fast as possible. If the object is already valid, a simple table lookup will do nicely. This allows simple and efficient use of OpenGL object without having to care about specifics of pipes and windows.

Invalidating the objects is usually done on the application side, and has to result in updating the objects at the next validation call. To allow flexibility in usage and extension of the object system, actual handling of the creation, updating and deletion of objects is left to the caller. The system is not specialized to handle display lists and texture objects, it is general and open for extensions which are discussed right now, like state objects.

It ke a single integer namespace for OpenGL objects and has a functor associated with every index. It ke track of the current state of the indexed object and calls the functor with parameters that indicate the action that is to be performed on the object, which encompass creation, recreation, updates and deletion.

The presence of multi-threading demands an invalidation/validation handling that is compatible with the the data structures defining in sec. 3.1.1 and the synchronisation approach described in sec. 4.3.3. A time-stamping approach that records the last invalidation and validation time for each object has proven itself to be an efficient solution for this problem.

Extension Handling

OpenGL extensions are an interesting problem in the context of multiple rendering pipes, especially heterogeneous pipes. They are accessed via a function that returns function pointers. These function pointers are specific to the contexts, in which they were acquired. In a heterogeneous environment, they can be different for every context.

Thus it is not possible to just let the user of the extension store and use them, they have to be accessed relative to the currently active pipe. This can be made efficient via a registration mechanism that returns an easy to check index for the registered extension and its associated functions.

Additionally, not all the pipes might support the same extensions, thus it is possible that an extension is not supported not every pipe. A conservative approach should have fall-backs to use in case the extension doesn't exist. This is not always possible or sensible (e.g. for features that would require per-pixel software work). In these cases the feature just has to be ignored and a warning logged.

5.5.2 Cross-screen Consistency

When multiple screens touch each other, as in a Powerwall or Cave setup, care has to be taken to make sure that the touching parts of the images are as similar as possible, otherwise the borders between the screens become visible and the illusion of a seamless display is destroyed.

In general this is a problem that needs support from the graphics library. As the graphics library has to clip the primitives at the window border, it has to make sure that clipped and unclipped primitives look the same. This is not true in general.

This can have multiple reasons. One is a full evaluation of the lighting model at the clipping border , another is non-perspective corrected color interpolation in the non-clipped case and perspective correction in the clipping code.

But control at this level is out of the hand of a scenegraph library. The problems can be alleviated by using more finely tessellated geometry, which minimizes the length of clipped edges, but they can not be removed.

Another problem area is using non-coplanar projection screens.The OpenGL lighting model has several places where an absolute direction in the local viewing coordinate system is used. These viewing coordinates are usually aligned with the screen. When the screens are not coplanar this will result in seams.

A closer look at the transformation from model to screen coordinates

$$v_{screen} = M * v_{model}$$

reveals that logically it can be split into multiple transformations

$$v_{screen} = M_{Viewport} * M_{Projection} * M_{Viewing} * M_{Model} * v_{model}$$

For Multi-Screen projections it makes sense to insert two more transformations: a canonical viewer coordinate system and a projection screen system

$$v_{screen} = M_{Viewport}*M_{Projection}*M_{PScreen}*M_{Canonical}*M_{Viewer}*M_{Model}*v_{model}$$

As OpenGL only has two matrices in addition to the viewport transformation there are different variants of how to combine all these transformations into two matrices. Two variants are the most sensible: using a different viewing coordinate system for all screen (i.e. set the OpenGL modelview matrix and the viewer coordinate system, in which lighting calculation is done, to be the same for all screens), or using a uniform one. This is done by by either putting both $M_{Canonical}$ and $M_{PScreen}$ into the `GL_MODELVIEW` matrix, keeping only the $M_{Projection}$ in the `GL_PROJECTION` matrix, or by moving $M_{PScreen}$ into the `GL_PROJECTION` matrix.

There are three spots in the specification that use absolute values in viewing coordinates: infinite viewer specular lighting, environment m and fog.

Infinite Viewer Specular Lighting

For efficiency reasons the lighting model can use an infinitely distant viewer in some calculations. This viewer is assumed to be infinitely far away in the positive z direction in viewing space. To use this feature in a Cave, the local viewing coordinate systems of all the screens have to be aligned to each other, otherwise strongly noticeable artifacts result (see fig. 5.8).

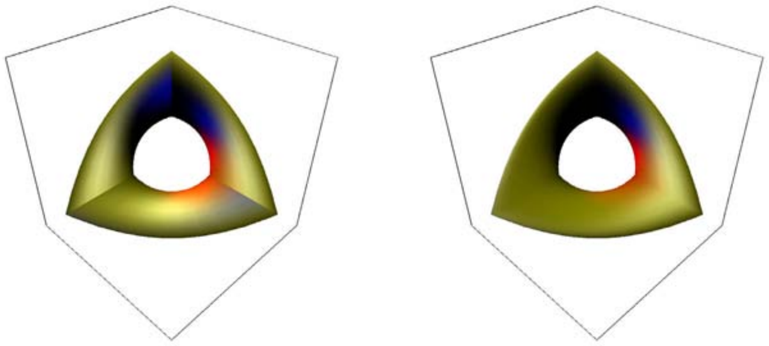


Figure 5.8: Infinite viewer specular lighting.
Left: screen aligned coordinate systems, right: single coordinate system

Environment M

To simulate materials that reflect the environment, spherical environment m are the simplest solution and the only one that is supported by the base OpenGL [14]. The formula used to calculate the texture coordinate assumes that the viewing direction is negative z in the viewing coordinate system. Again, using the standard screen-aligned coordinate system results in severe artifacts. A global coordinate system that is consistent across screens has to be used (see fig. 5.9).

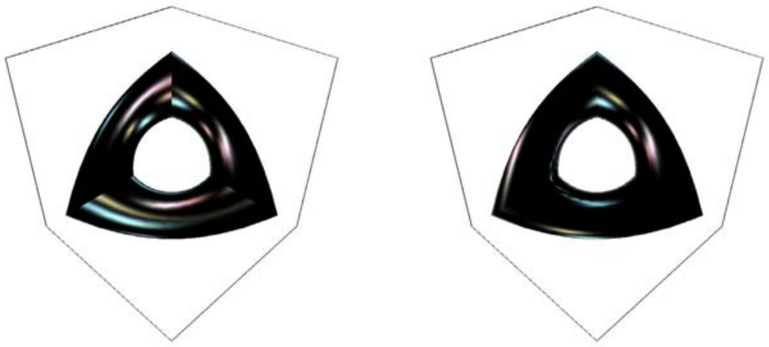


Figure 5.9: Environment map.
Left: screen aligned coordinate systems, right: single coordinate system

Fog

The final area that uses viewing coordinates is fog. The fog formulas use the z coordinate of the viewing coordinate system to calculate the blending factor between the vertex color and the fog color. In this situation using the solution for the other problems, a single viewing coordinate system creates a wall of fog that goes into the direction of the global viewing coordinate system (see fig. 5.10 right). The solution would be to not use the z coordinate but the distance from the viewer for the blending factor calculation. As that would demand changes to OpenGL it can't be done by an application.

Using separate coordinate systems creates separate walls going off in the directions of the screens, which minimizes the artifacts and gives acceptable results (see fig. 5.10 left).

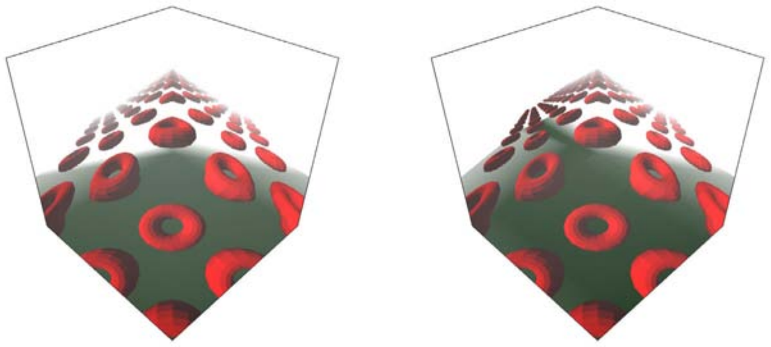


Figure 5.10: Fog.
Left: screen aligned coordinate systems, right: single coordinate system

5.6 Summary

This chapter addresses the efficient and flexible handling of graphics hardware.

The structuring of geometry is important, as it demands flexibility to accommodate a wide range of applications, as well as efficiency, as the geometry represents the largest part of the scenegraph memory-wise, and has to be transfered to the graphics hardware in order to create the image. The GeoProperty concept supports this. To hide the complexity that is associated with flexibility a set of geometry iterators has been introduced that presents a unified and simple image to the user.

The efficient and flexible handling of the graphics library state is an important part of rendering, as this is an area where a lot of development is taking place and where a lot of performance can get lost. By splitting the state into chunks the problem becomes manageable and extensible at the same time, thus providing a stable and efficient basis.

But the level of the state is too low for users, as it is inherently concerned with the limitations of the hardware. A higher, more abstract level is needed that hides the complexity that might be needed to create a desired result behind a simple to use cover. By giving the control of the rendering operation to the material this is possible. It allows transparent use of multi-pass algorithms as well as creating temporary images if needed.

Integrating state handling and material handling is done using the draw tree. It captures the visible objects for the current frame and allows out-of-order insertion of temporary images as well as giving reordering constraints to aid state change minimizations.

A specific aspect that is gaining importance is the consistent handling of multiple non-coplanar screens. By splitting the transformation pipeline into three conceptual parts instead of the two parts defined in OpenGL it is possible to insure cross-screen consistency even in the cases that graphics library does not guarantee and in fact not support.

Chapter 6

Examples

Many of the concepts described in this work have been implemented in the OpenSG scenegraph system.

OpenSG is a high-performance, multi-platform scenegraph for virtual reality applications that is distributed under the LGPL Open Source license and is available from `www.opensg.org`.

It runs on a wide variety of platforms, starting from lapt running Windows 98 across a wide range of PC-based systems running Windows and Linux to workstations from all large manufacturers like HP, SUN and IBM up to million dollar or more multi-pipe systems from sgi.

It has been extensively used for different projects:

- Kelvin
  The next-generation VR system developed by IGD and commercialized by vr-com. A number of different projects have been realized using this system. Kelvin is a full-fledged VR system that stresses all parts of the scenegraph. It uses multiple independent threads to work in parallel on problems of collision detection, route propagation and of course rendering. Furthermore it supports rendering to multiple independent graphics pipes for Powerwall and CAVE set.

- Arvika
  Arvika is a German state-funded research project to explore Augmented Reality technology for the construction industry. The Arvika system uses a unified

Figure 6.1: OpenSG

core that is used as an AR component inside a web browser like Internet Explorer running on a laptop with a connected USB camera, up to a high-end sgi Onyx system for augmenting the images of crashed cars with their simulated equivalents. Arvika profits from the flexibility and openness of the OpenSG system that allows integration into a plug-in framework for internet browsers.

- Avalon
  The Avalon [105, 51] VRML-based VR system that has been developed by ZGDV over the last couple years has been ported over to OpenSG from its OpenGL-based low-level structure. In the process Avalon gained the ability to be used in arbitrary projection environments and obtain a significantly higher graphics throughput than before.

- Avatar
  A human motion visualization system to be used to create lifelike avatars for virtual environments has been developed [97]. It uses a skin and bone system using an arbitrary number of bones to create a lifelike human figure in free

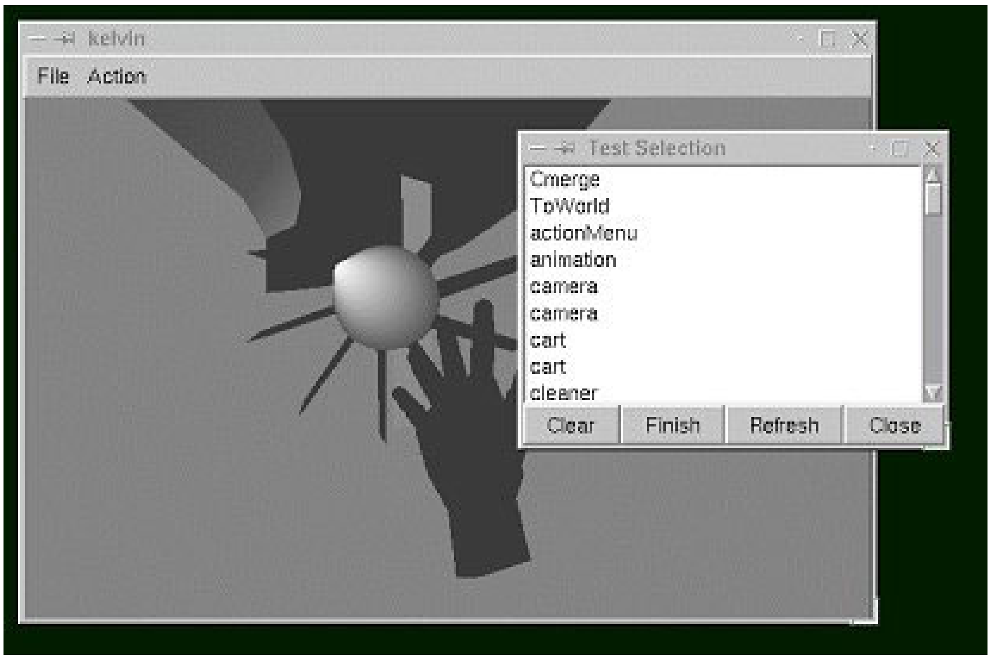


Figure 6.2: Kelvin

motion.

- Dental replacement planner
  A dental replacement planner has been created to visualize measured jaw movements and to help the dentist find problem areas. It uses OpenSGs flexible data management to work in parallel on large datasets of scanned teeth.

- Mass-accident analysis system
  This system is used to visualize the course of mass accidents on german highways [106]. Starting from images of the final setting the whole process of the accident and the involved singular crashes can be set up and visualized, allowing unprecedented insights into the actions in a mass accident.

OpenSG has been downloaded more than 5000 times from the Web and is in active use in a number of research and development institutions worldwide.

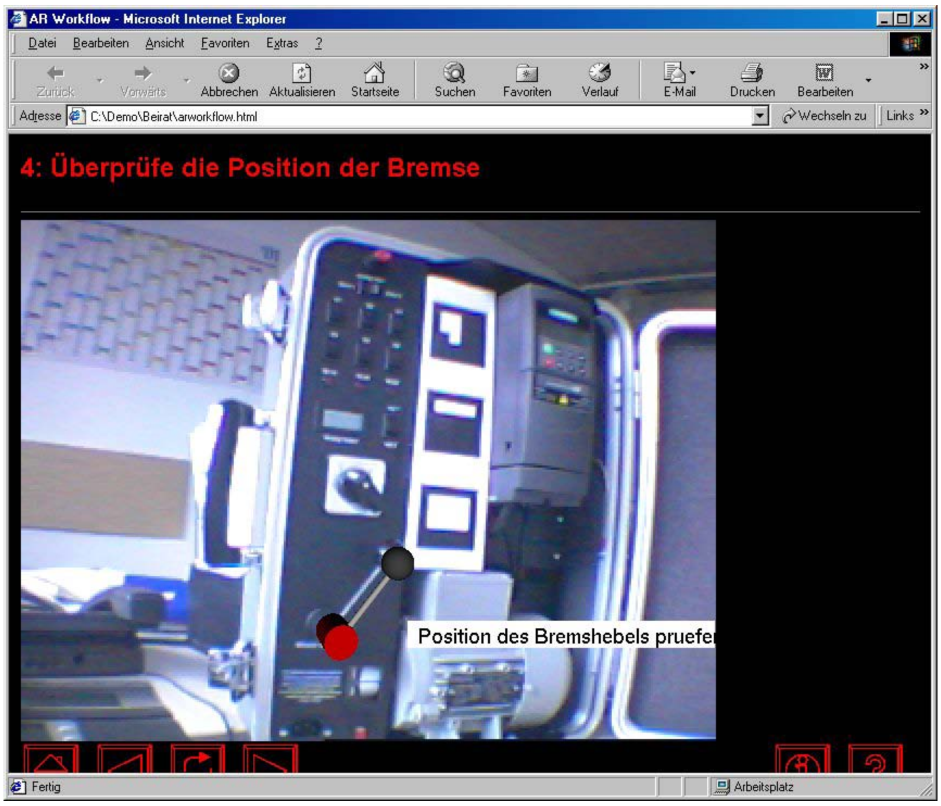

AR Workflow - Microsoft Internet Explorer
Datei Bearbeiten Ansicht Favoriten Extras ?
Zurück Vorwärts Abbrechen Aktualisieren Startseite Suchen Favoriten Verlauf E-Mail Drucken Bearbeiten
Adresse C:\Demo\Beirat\arworkflow.html
Wechseln zu Links
4: Überprüfe die Position der Bremse
Position des Bremshebels pruefe
Fertig
Arbeitsplatz

Figure 6.3: Arvika

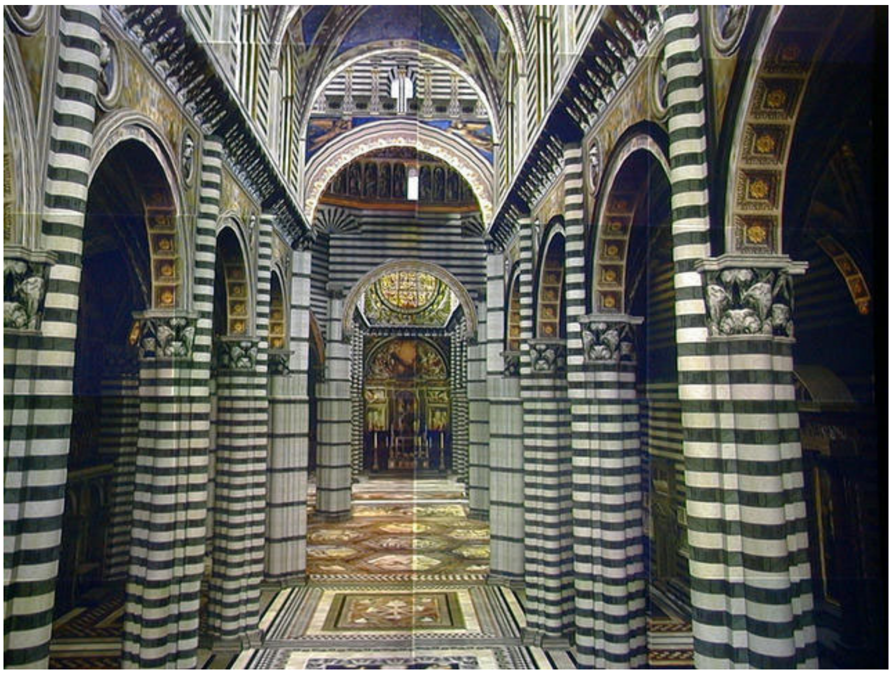


Figure 6.4: The Siena Cathedral, rendered using Avalon

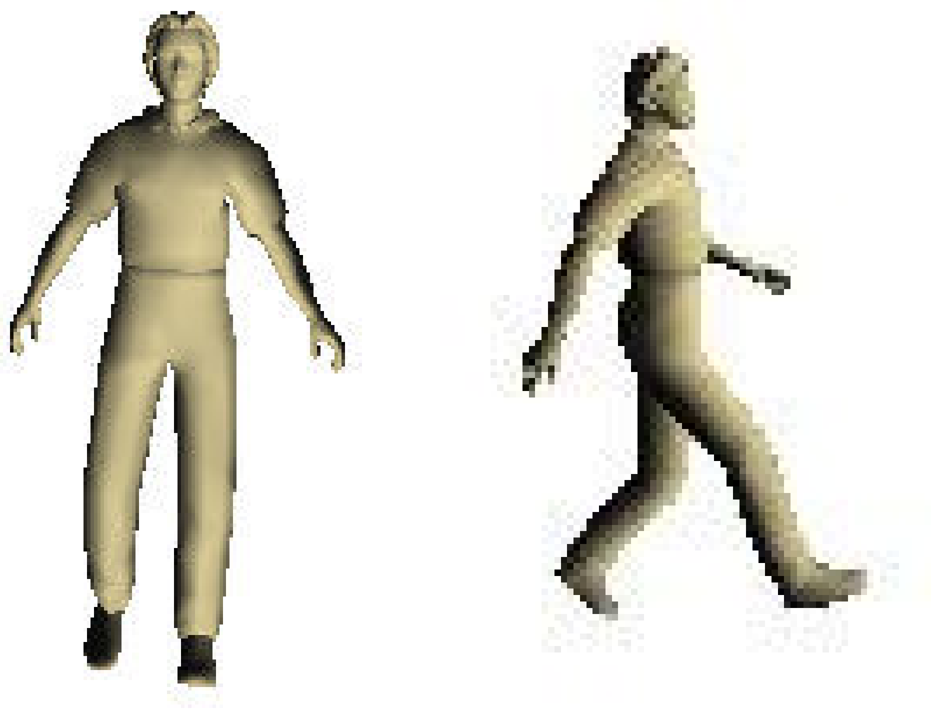


Figure 6.5: Avatar visualization system

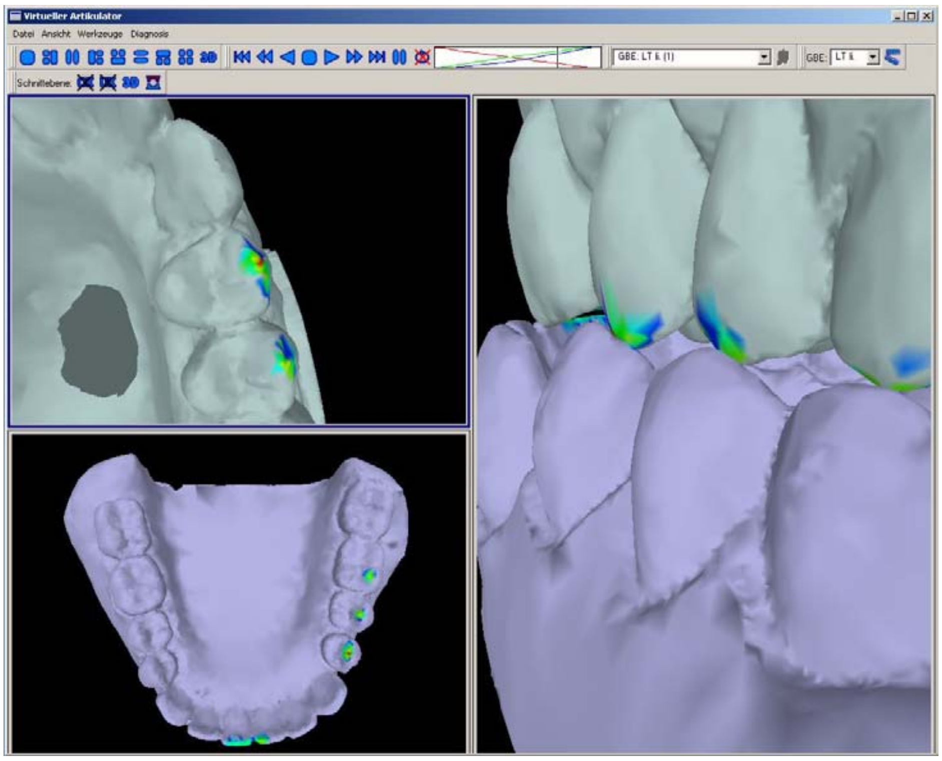
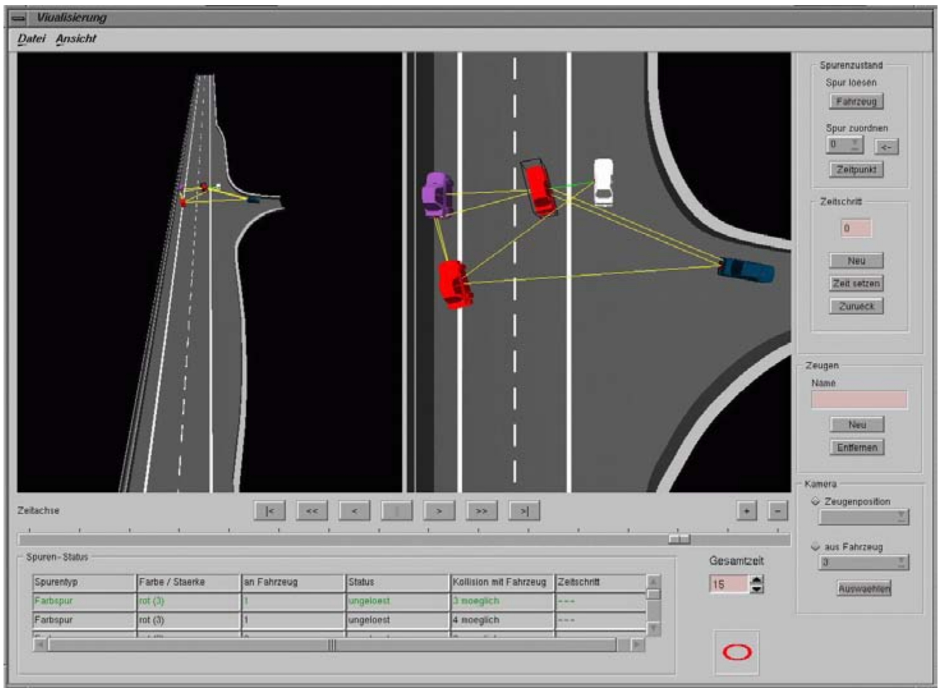


Figure 6.6: Dental replacement planner



Zeitschritt: 0

Gesamtzeit: 15

Spurentyp	Farbe / Staerke	an Fahrzeug	Status	Kollision mit Fahrzeug	Zeitschritt
Farbspur	rot (3)	1	ungeloest	3 moeglich	---
Farbspur	rot (3)	1	ungeloest	4 moeglich	---

Figure 6.7: Mass accident analysis system

Chapter 7

Summary and Future Work

7.1 Summary

This work opens a path to keep scenegraphs a viable paradigm for real-time rendering systems for the foreseeable future, which in the computer and computer graphics area does not exceed three to five years.

The analysis of the microprocessor state of the art in chapter 2 and the extrapolations based on it predict that parallel processing of multiple independent threads will be ubiquitous soon, either as separate processors or in a single chip. On the graphics hardware front performance will continue to rise faster than processor performance, but more importantly programmability will spread and the need to differentiate themselves will drive the hardware vendors to keep adding unique features to their systems, demanding high flexibility and extensibility from the scenegraph systems.

The commonly available scenegraph systems Open Inventor, OpenGL Performer, Y and Java3D have been analyzed according to their ability to fulfill these demands. The analysis shows that three areas are not adequately covered:

- extensibility
- handling of parallel tasks
- flexible and efficient handling of graphics hardware.

Thus they have been analyzed and solutions are proposed in this work.

Extensibility Extensibility includes the ability to extend the system in such a way that a new application or system extension can not only be used very easily, but is also able to extend already existing programs to benefit from new developments and extensions without having to be changed. The highest goal is to be able to create systems and tools that can not only use new features as a replacement for the old ones, but also use them and manipulate them natively.

A set of data structures is defined that can give information about themselves, coupled with methods to manipulate that data. Together with a simple to use interface for defining these structures interactively and creating them automatically, building a very generic and efficient system is made possible. The replacement of internal components by versions that are better suited to the task or hardware/software environment, even at runtime, is achieved through the *dynamic combined use of generative patterns*, namely the Factory and Prototype patterns. The flexibility also extends into the specifics of the scenegraph, the nodes and leaves that define the graph, and the methods of traversing this graph. The developed node structure is able to combine the benefits of simple data sharing for efficient replication of scenegraph parts with the usability and consistent node identifiability of single-parent systems by the use of a *node-core split*. The flexibility designed into the graph structure demands equal flexibility in the active parts of the system, the actions that traverse the graph. The design developed in this work is able to efficiently handle the extensibility constraints that the abovementionened structures demand and furthermore supports flexible extension and replacement of node-specific actions itself.

Parallel Processing The different kinds of tasks that can run in a system in parallel and the different demands that they have concerning data independence are described. The big question is if it is necessary to replicate the scenegraph data for parallel tasks to work or not. An analysis shows that some tasks can be usefully handled in parallel without replicating data. This includes scene graph traversals that have a very limited set of cross-node dependencies, primarily parallel traversals that do significant work on a single node.

But in general data replication is needed to allow multiple concurrent tasks to work together without interfering. The continuum between not replicating anything and replicating everything is evaluated showing that there are important tasks that demand a possibly full replication of the scenegraph data. As this is only a possibility, but the typical demands are more limited, a flexible system that allows both the sharing of the bulk of the data, which is the geometry, as well as possibly separating everything is identified as the best solution.

Different approaches to replicate and distribute the data and how to access and synchronize it are compared. Replicating separate fields, replicating containers and replicated the whole tree are the alternative representations, with complete copy, change flags and change lists being the alternatives for synchronization. The results show that the *replicated field container* structuring with *change-list based* synchronization defines the best synthesis of ease of use and caching behavior.

The concepts are based on a shared memory assumption. An important special case for a distributed memory system is a *distributed cluster for multi-screen or large screen rendering*. It is demonstrated how the solution can be extended to cover this case.

Graphics Hardware Handling One important aspect is the handling of the geometric data. It has to be flexible enough to support many different applications and adapt itself to be integrated into other systems, but at the same time it has to exploit the graphics hardware as good as possible. The *GeoProperty* abstraction to define the geometric data developed here is well suited to the OpenGL graphics library that is used to drive the hardware, as well as providing the flexibility to adapt to the different specifics of the different hardware systems and the specifics of the applications using them.

The other main aspect of graphics hardware is the management of the state of the system. Here the conflict is between providing the user with an abstract, efficient interface to define surface properties, while at the same time giving him the flexibility and power to use new features as good as possible, and the need to manage the state in a way that reduces costly state changes, without itself costing too much time. These problems are split into state handling, which is concerned about being able to manage a possibly changing set of graphics state and reduce state changes, and material handling, whose task it is to abstract the internals of the graphics library and provide a useful interface to the rendering properties to the user that abstracts the specifics of the actual hardware and emulates features that are not supported natively as far as possible. An integrated concept solving these problems is developed.

The state handling and state minimization complexity problems are solved by the definition of *state chunks* that cover a subset of the graphics state and allow efficient handling of the whole state. The *material as the rendering controller* concept allows the abstraction of techniques that go beyond the direct capabilities of the graphics library, like multi-pass techniques or techniques involving temporary images. Bringing both of these concepts together in a flexible and efficient manner is done by the *draw tree*. It is a temporary graph that captures the information for the current frame and allows out-of order definition of subtasks as well as supplying the framework for efficient state change minimization.

A problem area that is applied more and more often and is gaining importance is the use of multiple non-coplanar projection screens, e.g. in a CAVE environment. Some assumptions about coordinate systems in the graphics library conflict with the strict demand of cross-screen continuity that is imperative for using these systems. By splitting the usual two-step transformation pipeline of OpenGL into three st and varying the association of the third step it is possible to create *unified* as well as *split viewer coordinate systems*, which allow a finer adaption to the restrictions of the graphics library and solving the continuity problem.

OpenSG The results of this work have been realized in the OpenSG system. OpenSG is a freely available scenegraph that has been used in a number of projects and has proven that the concepts described here are viable and practically useful. These examples cover the range from simple applications that benefit from the simplicity of integrating extensions into the system, through medium-size systems that integrate external components to full-fledged Virtual Reality systems. The daily use of these systems demonstrates the viability of the concepts developed in this work.

7.2 Future Work

Even though a number of solutions to important problem are proposed and described in this work and have been realized in the form of OpenSG, the book on scenegraph systems has by no means been closed. On the contrary, the availability of the kind of system described in this work opens new areas of research.

One large area is the seamless integration of new rendering primitives like NURBS[49, 61], point sets[101, 36, 125, 115] and volumetric data[127, 38, 30, 63]. These have been used successfully in different areas in previous works, but generally only in dedicated specialized systems. Given that each of these has different optimal application areas, an important goal would be a unified system that can handle each of them whenever they are most suited. The interesting problem here is to be able to use all the described generic features on these new primitive types in the same way they are used with standard polygonal surfaces.

Even though computer systems grow in capabilities at an amazing speed, problem sizes grow even faster. An interesting area of reasearch is the extension into the direction of interactively rendering highly complex models, models of a size that cannot be displayed at interactive frame rates even if theoretical hardware performance numbers were achieved and/or that cannot be completely kept in memory at any given time. There are different aspects of the problem, including the ability to automatically create and use a simplified version of the model as well as the ability either page parts of the model or to split the the model over a cluster of machines, which cannot yet be done using the cluster replication model given in section 4.5.

A very general aspect resulting from the availability of a very extensible base system like the one described here is the ability to rapidly prototype new algorithms in a real system that offers a lot of utility functionality. This allows comparisons to existing methods on the one hand as well as, given use at different sites, comparisons to other people's work, even current work, thus stimulating comparative work and relieving many researches from reinventing lots of wheels.

Bibliography

[1] Kurt Akeley. The silicon graphics 4d/240gtx superworkstation. *IEEE Computer Graphics & Applications*, 9(4):71–83, July 1989.

[2] Kurt Akeley. RealityEngine graphics. In James T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 109–116, August 1993.

[3] Kurt Akeley and Tom Jermoluk. High-performance polygon rendering. *Computer Graphics (Proceedings of SIGGRAPH 88)*, 22(4):239–246, August 1988. Held in Atlanta, Georgia.

[4] Gail Alverson, Simon Kahan, Richard Korry, Cathy McCann, and Burton Smith. Scheduling on the tera mta. In *IPPS '95 Workshop on Job Scheduling Strategies for Parallel Processing*. IPPS, April 1995.

[5] Anthony A. Apodaca and M. W. Mantle. Renderman: Pursuing the future of graphics. *IEEE Computer Graphics & Applications*, 10(4):44–49, July 1990.

[6] David Applegate, Robert Bixby, Vasek Chvatal, and William Cook. Traveling salesman problem - home page. http://www.keck.caam.rice.edu/tsp/, 2001.

[7] OpenGL ARB. Opengl architexture review board - arb. http://www.opengl.org/developers/about/arb.html, 2001.

[8] Ulf Assarsson and Tomas Möller. Optimized view frustum culling algorithms. Technical report, University of Technology, Mar 2000.

[9] Peter Astheimer. *Sonifikation numerischer Daten fuer Visualisierung und Virtuelle Realitaet*. PhD thesis, Technische Universität Darmstadt, 1995.

[10] Dirk Bartz, Michael Meißner, and Tobias Hüttner. Extending graphics hardware for occlusion queries in opengl. *1998 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 97–104, August 1998. Held in Lisbon, Portugal.

[11] Johannes Behr, Torsten Fröhlich, Christian Knöpfle, Bernd Lutz, Dirk Reiners, Frank Schöffel, and Wolfram Kresse. The Digital Cathedral of Siena - Innovative Concepts for Interactive and Immersive Presentation of Cultural Heritage Sites. In *ICCHIM 2001 Conference Proceedings*, Milan, Sept 2001.

[12] Yahn Bernier. Half-life and teamfortress networking: Closing the loop on scaleable network gaming backend services. In *Game Developer Conference 2001 Proceedings*, San Jose, 2001.

[13] Allen Bierbaum. Clusterjuggler. In *VR2002 "Open Source VR" Coursenotes*, Orlando, March 2002.

[14] J. F. Blinn and M. E. Newell. Texture and reflection in computer generated images. *Communications of the ACM*, 19:542—546, 1976.

[15] David Blythe. Scene graph rewriting. Personal Communication, 2000.

[16] OpenGL Architecture Review Board, Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. Addison-Wesley, 1999.

[17] J. M. Borkenhagen, R. J. Eickemeyer, R. N. Kalla, and S. R. Kunkel. A multithreaded powerpc processor for commercial servers. *IBM Journal of Research and Development*, 44(6):885ff, Nov 2000.

[18] Ian Buck, Greg Humphreys, and Pat Hanrahan. Tracking graphics state for networked rendering. *2000 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 87–96, August 2000.

[19] Mike M. Chow. Optimized geometry compression for real-time rendering. *IEEE Visualization '97*, pages 346–354, November 1997. ISBN 0-58113-011-2.

[20] J. H. Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547–554, October 1976.

[21] J. D. Cohen, M. C. Lin, D. Manocha, and M. K. Ponamgi. I-collide: An interactive and exact collision detection system for large-scale environments. *1995 Symposium on Interactive 3D Graphics*, pages 189–196, April 1995. ISBN 0-89791-736-7.

[22] Satyan Coorg and Seth Teller. Real-time occlusion culling for models with large occluders. *1997 Symposium on Interactive 3D Graphics*, pages 83–90, April 1997. ISBN 0-89791-884-3.

[23] Carolina Cruz-Neira, Daniel J. Sandin, and Thomas A. DeFanti. Surround-screen projection-based virtual reality: The design and implementation of the cave. *Proceedings of SIGGRAPH 93*, pages 135–142, August 1993. ISBN 0-201-58889-7. Held in Anaheim, California.

[24] Fan Dai, Wolfgang Felger, Thomas Frühauf, Martin Göbel, Dirk Reiners, and Gabriel Zachmann. Virtual Prototyping Examples for Automotive Industries. In *Proc. Virtual Reality World*, Stuttgart, February 1996.

[25] Michael F. Deering. Geometry compression. In Robert Cook, editor, *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 13–20. ACM SIGGRAPH, Addison Wesley, August 1995. held in Los Angeles, California, 06-11 August 1995.

[26] Sim Dietrich. Maximizing texture performance. In *Game Developer Conference 1999 Proceedings*, San Jose, 1999.

[27] Sim Dietrich. Shadow techniques. In *Game Developer Conference 2001 Proceedings*, San Jose, 2001.

[28] Jose Dionisio, Volker Henrich, Udo Jakob, Alexander Rettig, and Rolf Ziegler. The virtual touch: Haptic interfaces in virtual environments. *Computers & Graphics*, 21(4):459–468, July 1997. ISSN 0097-8493.

[29] J. Emer. Simultaneous multithreading: Multiplying alpha performance. In *Microprocessor Forum 1999 Proceedings*, San Jose, Oct 1999.

[30] K. Engel, M. Kraus, and T. Ertl. High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading. In *Eurographics / SIGGRAPH Workshop on Graphics Hardware '01*, Annual Conference Series, pages 9–16. Addison-Wesley Publishing Company, Inc., 2001.

[31] Sidney Fels, Dirk Reiners, and K. Mase. Iamascope: An Interactive Kaleidoscope. In *Visual Proceedings of SIGGRAPH '97: The Electric Garden*, pages 76–77, 1997.

[32] Mark Frohnmayer and Tim Gift. The tribes engine network architecture. In *Game Developer Conference 2001 Proceedings*, San Jose, 2001.

[33] 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.

[34] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[35] Bernhard Geiger. Real-time collision detection and response for complex environments. *Computer Graphics International 2000*, June 2000. Held in Geneva, Switzerland.

[36] J. P. Grossman and William J. Dally. Point sample rendering. In *Proc. 9th Eurographics Workshop on Rendering*, pages 181–192, June 1998.

[37] André P. Guéziec, Frank Bossen, Gabriel Taubin, and Cláudio T. Silva. Efficient compression of non-manifold polygonal meshes. *IEEE Visualization '99*, pages 73–80, October 1999. ISBN 0-7803-5897-X. Held in San Francisco, California.

[38] S. Guthe, S. Roettger, A. Schieber, W. Strasser, and Th. Ertl. High-Quality Unstructured Volume Rendering on the PC Platform. In *Proc. EG/SIGGRAPH Graphics Hardware Workshop '02*, 2002.

[39] Pat Hanrahan. Real time shading languages: 10 years back to the future. *SIGGRAPH/EUROGRAPHICS Graphics Hardware Workshop, 1999*, 1999.

[40] Pat Hanrahan and Jim Lawson. A language for shading and lighting calculations. *Computer Graphics (Proceedings of SIGGRAPH 90)*, 24(4):289–298, August 1990. ISBN 0-201-50933-4. Held in Dallas, Texas.

[41] Volker Haug and Jeanine Indest. Rs/6000 7044 model 270 technical overview and introduction, 2001.

[42] Taosong He. Fast collision detection using quospo trees. *1999 ACM Symposium on Interactive 3D Graphics*, pages 55–62, April 1999. ISBN 1-58113-082-1.

[43] Wolfgang Heidrich, Philipp Slusallik, and Hans-Peter Seidel. Sampling procedural shaders using affine arithmetic. *ACM Transactions on Graphics*, 17(3):158–176, July 1998. ISSN 0730-0301.

[44] Steven Holzner. *Inside XML (Inside)*. New Riders Publishing, 2000.

[45] Philip M. Hubbard. Approximating polyhedra with spheres for time-critical collision detection. *ACM Transactions on Graphics*, 15(3):179–210, July 1996. ISSN 0730-0301.

[46] Greg Humphreys, Ian Buck, Matthew Eldridge, and Pat Hanrahan. Distributed rendering for scalable displays. In *Supercomputing 2000 Proceedings*, pages 30–30, 2000.

[47] Greg Humphreys and Pat Hanrahan. A distributed graphics system for large tiled displays. *IEEE Visualization '99*, pages 215–224, October 1999. ISBN 0-7803-5897-X. Held in San Francisco, California.

[48] Homan Igehy, Matthew Eldridge, and Kekoa Proudfoot. Prefetching in a texture cache architecture. *1998 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 133–142, August 1998. Held in Lisbon, Portugal.

[49] Ferenc Kahlesz, Akos Balazs, and Reinhard Klein. Nurbs Rendering in OpenSG PLUS. In *OpenSG 2002 Workshop*, January 2002.

[50] Roy Kawalsky. *The Science of Virtual Reality and Virtual Environments*. Addison-Wesley, Wokingham, England, 1993.

[51] Andreas Kühn. Effizientes multimedia-streaming für mixed-reality applikationen. Master's thesis, TU Darmstadt, 2001.

[52] Mark Kilgard. Nv_register_combiners2 opengl extension, 2001.

[53] Mark Kilgard. Nv_texture_shader2 opengl extension, 2001.

[54] Mark Kilgard. Nv_vertex_array_range opengl extension, 2001.

[55] Mark Kilgard. Nv_vertex_program opengl extension, 2001.

[56] Gudrun Klinker, Didier Stricker, and Dirk Reiners. The Use of 3d Model in Augmented Reality. In R. Koch and L. Van Gool, editors, *3D Structure from Multiple Images of Large-Scale Environments, European Workshop at ECCV'98 SMILE 98*. Springer Verlag, 1998.

[57] Gudrun Klinker, Didier Stricker, and Dirk Reiners. Optically Based Direct Manipulation for Augmented Reality. *Computers & Graphics*, 23(6):827–830, December 1999. ISSN 0097-8493.

[58] Gudrun Klinker, Didier Stricker, and Dirk Reiners. An Optically Based Direct Manipulation Interface for Human-Computer Interaction in an Augmented World. In Michael Gervaut, Dieter Schmalstieg, and Axel Hildebrand, editors, *Virtual Environments '99. Proceedings of the Eurographics Workshop in Vienna, Austria*, pages 53–62. Springer-Verlag Wien, 1999.

[59] Gudrun Klinker, Didier Stricker, and Dirk Reiners. Augmented Reality for Exterior Construction Applications. In W. Barfield and T. Caudell, editors, *Augmented Reality and Wearable Computers*. Lawrence Erlbaum Press, 2000.

[60] Wolfram Kresse and Dirk Reiners. Can we trust that image? Photometric Attributes of Current Projection Systems. In *VR2002 Proceedings*, Orlando, March 2002.

[61] Subodh Kumar, Dinesh Manocha, Hansong Zhang, and Kenneth E. Hoff III. Accelerated walkthrough of large spline models. In *Symposium on Interactive 3D Graphics*, pages 91–102, 190, 1997.

[62] Pattie Maes. *Concepts and Experiments in Computational Reflection*. PhD thesis, Vrije Universiteit Brussel, 1987.

[63] M. Magallón, M. Hopf, and T. Ertl. Parallel Volume Rendering using PC Graphics Hardware. In *Pacific Graphics*, 2001.

[64] William Mark, Scott Randolph, Mark Finch, James Van Verth, and Russell M. Taylor II. Adding force feedback to graphics systems: Issues and solutions. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 447–452. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996.

[65] Michael McCool and Wolfgang Heidrich. Texture shaders. *1999 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 117–126, August 1999. Held in Los Angeles, California.

[66] Tom McReynolds. Programming with opengl: Advanced rendering, 1996.

[67] K. Menger. Das botenproblem. In *Ergebnisse eines Mathematischen Kolloquiums*, pages 11–12. Teubner, 1932.

[68] Sun Microsystems. Sun ultra 60 whitepaper, 1999.

[69] Sun Microsystems. Java3d api. http://java.sun.com/products/java-media/3D/, 2001.

[70] Sun Microsystems. Sun enterprise 450 features & benefits, 2001.

[71] Sun Microsystems. Sun ultra 80 datasheet, 2001.

[72] Jack Middleton. Gl_sun_triangle_list opengl extension, 1999.

[73] Jack Middleton. Gl_sun_vertex opengl extension, 1999.

[74] Timothy Miller and Robert C. Zeleznik. The design of 3D haptic widgets. In Stephen N. Spencer, editor, *Proceedings of the Conference on the 1999 Symposium on interactive 3D Graphics*, pages 97–102, New York, April 26–28 1999. ACM Press.

[75] Brian Mirtich. V-clip: Fast and robust polyhedral collision detection. *ACM Transactions on Graphics*, 17(3):177–208, July 1998. ISSN 0730-0301.

[76] John S. Montrym, Daniel R. Baum, David L. Dignam, and Christopher J. Migdal. Infinitereality: A real-time graphics system. *Proceedings of SIGGRAPH 97*, pages 293–302, August 1997. ISBN 0-89791-896-7. Held in Los Angeles, California.

[77] T. Muhammad. Hardware barrier synchronization for a cluster of personal computers. Master's thesis, Purdue University School of Electrical Engineering, Dez 1995.

[78] Stefan Müller and Dirk Reiners. Visual Simulation for Virtual Reality. In *Fiera di Bologna*, 1994.

[79] nVidia. Nvidia opengl specs. http://www.nvidia.com/, 2001.

[80] Marc Olano and Anselmo Lastra. A shading language on graphics hardware: The pixelflow shading system. *Proceedings of SIGGRAPH 98*, pages 159–168, July 1998. ISBN 0-89791-999-8. Held in Orlando, Florida.

[81] Thomas Pabst. High-tech and vertex juggling - nvidia's new geforce 3 gpu, 2001.

[82] Hewlett Packard. Hp 9000-n specifications, 2001.

[83] Hewlett Packard. Visualize center ii overview and features, 2001.

[84] Renato Pajarola and Jarek Rossignac. Compressed progressive meshes. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):79–93, January - March 2000. ISSN 1077-2626.

[85] Mark S. Peercy, Marc Olano, John Airey, and P. Jeffrey Ungar. Interactive multi-pass programmable shading. *Proceedings of SIGGRAPH 2000*, pages 425–432, July 2000. ISBN 1-58113-208-5.

[86] Jovan Popovic, Steven M. Seitz, Michael Erdmann, and Zoran Popovic andrew Witkin. Interactive manipulation of rigid body simulations. *Proceedings of SIGGRAPH 2000*, pages 209–218, July 2000. ISBN 1-58113-208-5.

[87] Kekoa Proudfoot, William R. Mark, Svetoslav Tzvetkov, and Pat Hanrahan. A real-time procedural shading system for programmable graphics hardware. *Proceedings of SIGGRAPH 2001*, August 2001.

[88] Bruno Raffin. Netjuggler. In *VR2002 "Open Source VR" Coursenotes*, Orlando, March 2002.

[89] Dirk Reiners. Hochqualitatives Realtime-Rendering für Virtual Environments. Master's thesis, TU Darmstadt, Dez 1994.

[90] Dirk Reiners. High-Quality High-Performance Rendering for Multi-Screen Projection Systems. In *Proceedings of the 3 rd International Immersive Projection Technology Workshop*, pages 191—200, 1999.

[91] Dirk Reiners. A Flexible Traversal Framework for Scenegraph Systems. In *OpenSG 2002 Workshop*, January 2002.

[92] Dirk Reiners. OpenSG - A Modern Scene Graph for VR Applications. In *Virtual Reality and its Applications in Industry (VRAI) 2002*, April 2002.

[93] Dirk Reiners. Scenegraph Rendering. In *VR2002 "Open Source VR" Coursenotes*, Orlando, March 2002.

[94] Dirk Reiners and et al. The Elephants and the Ants: Will Large projectors be Replaced By Many Small Ones? In *IPT2002*, Orlando, March 2002.

[95] Dirk Reiners, Didier Stricker, Gudrun Klinker, and Stefan Müller. Augmented Reality for Construction Tasks: Doorlock Assembly. In *First International Workshop on Augmented Reality*. Springer Verlag, 1998.

[96] Dirk Reiners, Gerrit Voss, and Johannes Behr. OpenSG - Basic Concepts. In *OpenSG 2002 Workshop*, January 2002.

[97] Martin Reisinger. Modellierung koordinierter Körperbewegung von Avataren in 3d. Master's thesis, FH Darmstadt, 2001.

[98] John Rohlf and James Helman. IRIS performer: A high performance multiprocessing toolkit for real–Time 3D graphics. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 381–395. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

[99] 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.

[100] Marcus Roth. Integration paralleler rendering-verfahren für lose gekoppelte systeme mit opensg. In *OpenSG 2002 Workshop*, 2002.

[101] Szymon Rusinkiewicz and Marc Levoy. QSplat: A multiresolution point rendering system for large meshes. In Kurt Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, pages 343–352. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[102] Rudrajit Samanta, Thomas Funkhouser, Kai Li, and Jaswinder Pal Singh. Hybrid sort-first and sort-last parallel rendering with a cluster of pcs. *SIGGRAPH/EUROGRAPHICS Graphics Hardware Workshop, 2000*, 2000.

[103] Rudrajit Samanta, Thomas Funkhouser, Kai Li, and Jaswinder Pal Singh. Sort-first parallel rendering with a cluster of pcs. *SIGGRAPH 2000, Sketches*, July 2000.

[104] Rudrajit Samanta, Jiannan Zheng, Thomas Funkhouser, Kai Li, and Jaswinder Pal Singh. Load balancing for multi-projector rendering systems. *SIGGRAPH/EUROGRAPHICS Graphics Hardware Workshop, 1999*, 1999.

[105] Stephan Schneider. Design und implementierung eines event scheduling systems für die interaktions- und animationsverarbeitung in einer vr umgebung. Master's thesis, TH Darmstadt, 1998.

[106] Markus Schütz. Prototypische entwicklung eines visualisierungssystems zur unterstützung der rekonstruktion von massenautounfällen. Master's thesis, FH Darmstadt, 2001.

[107] Mark Segal, Kurt Akeley, Chris Frazier, and Jon Leech. The opengl graphics system: A specification (version 1.2.1). ftp://ftp.sgi.com/opengl/doc/opengl1.2/opengl1.2.1.pdf, 1999.

[108] Mark Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, and Paul E. Haeberli. Fast shadows and lighting effects using texture mapping. *Computer Graphics (Proceedings of SIGGRAPH 92)*, 26(2):249–252, July 1992. ISBN 0-201-51585-7. Held in Chicago, Illinois.

[109] sgi. Unleashing the power of sgi's next generation visualization technology. http://www.sgi.com/software/optimizer/whitepaper.html, 2001.

[110] Michael Sherman. minigl. http://www.dsbox.com/minigl.html, 2001.

[111] silicon graphics. Octane2: Technical info, 2000.

[112] silicon graphics. Onyx3000: Technical info, 2000.

[113] Sandeep Singhal and Michael Zyda. *Networked Virtual Environments: Design and Implementation*. Addison-Wesley, 1999.

[114] John Spitzer. Opengl performance. In *Game Developer Conference 2001 Proceedings*, San Jose, 2001.

[115] Marc Stamminger and George Drettakis. Interactive sampling and rendering for complex and procedural geometry. In K. Myskowski and S. Gortler, editors, *Rendering Techniques 2001 (Proceedings of the Eurographics Workshop on Rendering 01)*, 12th Eurographics workshop on Rendering. Eurographics, Springer Verlag, 2001.

[116] A. A. Stepanov and M. Lee. The Standard Template Library. Technical Report X3J16/94-0095, WG21/N0482, NASI/ISO X3J16/94-0095, WG21/N0482, 1994.

[117] Paul Strauss and Rikk Carey. An object-oriented 3d graphics toolkit. In *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 341–349, Anaheim, California, Aug 1993. ACM SIGGRAPH.

[118] 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.

[119] Didier Stricker, Gudrun Klinker, and Dirk Reiners. A Fast and Robust Line-based Optical Tracker for Augmented Reality Applications. In *First International Workshop on Augmented Reality*. Springer Verlag, 1998.

[120] Seth J. Teller and Carlo H. Séquin. Visibility preprocessing for interactive walkthroughs. In Thomas W. Sederberg, editor, *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 61–69, July 1991.

[121] Matthias Unbescheiden. *Physikalisch basierte Simulation in Virtuellen Umgebungen*. PhD thesis, Technische Universität Darmstadt, 1999.

[122] Alex Vlachos and Jörg Peters. Curved pn triangles. *2001 Symposium on Interactive 3D Graphics*, March 2001.

[123] Alex Vlachos and Jörg Peters. Vertex shading with direct3d and opengl. In *Game Developer Conference 2001 Proceedings*, San Jose, 2001.

[124] Gerrit Voß, Johannes Behr, Dirk Reiners, and Marcus Roth. A Multi-thread Safe Foundation for Scenegraphs and its Extension to Clusters. In *submitted to Fourth Eurographics Workshop on Parallel Graphics and Visualization*. Eurographics, Sept 2002.

[125] M. Wand, M. Fischer, and F. Meyer. Randomized point sampling for output-sensitive rendering of complex dynamic scenes, 2000.

[126] Webopedia. Moore’s law. http://webopedia.internet.com/TERM/M/Moores_Law.html, 2001.

[127] Manfred Weiler and Thomas Ertl. Ein Volume-Rendering-Framework für OpenSG. In *OpenSG 2002 Workshop*, January 2002.

[128] Heather Williamson. *XML: The Complete Reference*. McGraw-Hill Higher Education, 2001.

[129] Paul Woodward. Powerwall. http://www.lcse.umn.edu/research/powerwall/powerwall.html, 2001.

[130] Gabriel Zachmann. *Virtual Reality in Assembly Simulation - Collision Detection, Simulation Algorithms, and Interaction Techniques*. PhD thesis, Technische Universität Darmstadt, 2000.

[131] Hansong Zhang, Dinesh Manocha, Thomas Hudson, and Kenneth E. Hoff III. Visibility culling using hierarchical occlusion maps. *Proceedings of SIGGRAPH 97*, pages 77–88, August 1997. ISBN 0-89791-896-7. Held in Los Angeles, California.

[132] Ziff-Davis. nvidia geforce4 ti 4600, 2002.

Curriculum Vitae

Name:	Dirk Reiners
Geburtsdatum:	17.11.1968
Geburtsort:	Bergisch Gladbach

1975-1979	Grundschule "Am Broich", Bergisch Gladbach
1979-1988	Nicolaus Cusanus Gymnasium, Bergisch Gladbach
1989-1991	Grundstudium Informatik an der Friedrich Alexander Universität, Erlangen
1991-1994	Hauptstudium Informatik an der Technischen Hochschule Darmstadt
1994	Studienaufenthalt am National Center for Supercomputing Applications (NCSA) in Champaign-Urbana, USA
1995-2000	Wissenschaftlicher Mitarbeiter beim Fraunhofer IGD, Darmstadt
1996	Forschungsaufenthalt am Advanced Telecommunications Research Institute International (ATR) in Takanohara, Japan
1997-1998	Wissenschaftlicher Mitarbeiter in der Augmented Reality Gruppe des Fraunhofer IGD, München
2000-2002	Technischer Geschäftsführer des OpenSG Forums