Computer Science
University of Manchester

Accelerated Ray Tracing on the KSR1 Virtual Shared-Memory Parallel Computer

M.J. Keates & R.J. Hubbold

Technical Report UMCS-94-2-2

Accelerated Ray Tracing on the KSR1 Virtual Shared-Memory Parallel Computer*

M.J. Keates & R.J. Hubbold

Department of Computer Science
University of Manchester
Oxford Road, Manchester, UK.
keatesm@cs.man.ac.uk, roger@cs.man.ac.uk

February 1994

Technical reports issued by the Department of Computer Science, Manchester University, are available by anonymous ftp from `ftp.cs.man.ac.uk` in the directory `pub/TR`. The files are stored as PostScript, in compressed form, with the report number as filename. Alternatively, reports are available by post from The Computer Library, Department of Computer Science, The University, Oxford Road, Manchester M13 9PL, UK.

Abstract

This report describes the results of experiments with accelerated ray tracing on a virtual shared-memory parallel computer. The machine used, the Kendall Square Research KSR1, is described in sufficient detail for its novel features to be appreciated. The report describes a new ray tracer written specifically to capitalise on the KSR1's features. Models used in the experiments include well-known test cases from the Haines database, permitting comparison with previously reported results. The results show a speed-up which is 76.9% of linear scaling on 256 processors. The report analyses the performance of the software and draws conclusions about the suitability of this type of machine for image synthesis.

Keywords: ray tracing, gridcell acceleration, parallel processing, virtual shared memory.

1 Introduction

During the past decade and a half, ray tracing has become a well established method for generating high-quality images. For complex scenes, naive ray tracing is prohibitively expensive, and this has engendered considerable, and continuing, research into techniques for making things go faster. One line of enquiry has been the derivation of techniques for speeding up ray/object intersection tests, since naive ray tracers typically spend most of their time in this activity [Whitted80]. A second theme to emerge is the use of parallel machines, with the particular goal of achieving linear speed-up with increasing numbers of processors. A third possibility is to design special architectures for ray tracing, but this has the disadvantage of being somewhat inflexible when compared to general purpose machines.

Early attempts to combine accelerated ray tracing and parallelism seem to have been thwarted by limitations of the particular parallel machines used. On machines with modest amounts of distributed memory, using variants of message passing for synchronisation and communication, the major difficulty has been to organise a suitable distribution of data for large models. Shared-memory machines have generally avoided this problem, but have not generally been scalable to more than around 30 processors.

This report reports the results of experiments on a new kind of parallel computer which exhibits the characteristics of shared memory but is also scaleable to thousands of processors. The machine has a novel architecture whose shared-memory programming model makes it easy to use. As part of our work, we wanted to test the use of this machine for interactive visualisation. The questions we wished to answer were: how well does it perform in practice; how well does its performance scale with number of processors used; and does this type of machine hold promise for wider applications in scientific visualisation? In order to answer some of these questions we decided to test the machine's performance for high-quality image synthesis using ray tracing. Ray tracing is a useful benchmark for two reasons: enough experiments have been performed elsewhere to provide a basis for comparison; and complex scenes with large numbers of secondary and shadow rays generate unpredictable accesses to model data — normally a worst case scenario for parallel computers.

Although this report deals specifically with ray tracing, the work described is part of a wider programme of research into the use of general purpose parallel machines for interactive visualisation. Our goal is to build a graphics software architecture which combines ray tracing and ray casting (including various direct volume rendering algorithms), and constructive solid geometry (CSG) operations, and to be able to generate pictures fast enough to permit interactive exploration of models. We aim to achieve this by exploiting parallelism and by using intelligent algorithms to make incremental changes to images. The use of spatial enumeration — which is also the key to accelerated ray tracing — is central to this work.

The remainder of the report is organised as follows. The next section gives a review of acceleration techniques and previous work on parallel processing. Then there is an overview of the architecture and unique features of the machine we have used, the Kendall Square Research KSR1, followed by a section about how the ray tracer is implemented and the reasons for our design decisions. We then give the results of the experiments we have performed, and draw conclusions.

2 Previous work

This section outlines the major ideas which have been pursued and which we believe are relevant for a description of our work. A comprehensive list of references on ray tracing, including acceleration methods and parallel processing can be found in [Speer92] and a good overview of ray tracing on parallel computers has been published by Badouel et al [Badouel90].

2.1 Acceleration techniques

The main techniques for accelerating ray tracing capitalise on the property of spatial coherence. Most of them use some kind of spatial subdivision, in which data structures are built to represent regions of the scene. These structures are then populated with data identifying those objects lying in each region of space. The building of these data structures is a pre-processing step prior to the main ray tracing phase. During the actual ray tracing, individual rays are only tested against the subset of objects which lie inside the regions intersected by the ray. Since we are generally interested in the closest intersection to the ray origin, this checking is best done in depth order along the ray. Four main variants exist:

Uniform subdivision (or gridcell) in which the total model space is subdivided into a regular 3D grid of cells [Fujimoto86]. For each cell, a list of objects which intersect the cell is built, and a ray traversal algorithm is employed to work out which cells are intersected by the current ray. This is normally a simple incremental algorithm based on digital differential analyser (DDA) principles, or a variant of Bresenham's line drawing algorithm, and can therefore be made fast [Cleary88]. The major advantage of the whole approach is its simplicity. Its disadvantages are that the gridcell data structure grows rapidly as the cell size reduces, and the data structure is poorly balanced for models which have uneven spatial distribution.

Octtree subdivision attempts to overcome the last problem — namely the potentially large size of the data structures, and poor spatial distribution — by recursively subdividing the model space, usually by binary division in each dimension [Glassner84], [Fujimoto86], [Spackman90]. Starting with the whole model space, subdivision continues until some minimum number of objects remain in each sub-region, or some number of levels of recursion is reached. The data structure thus generated is a tree, with the leaf nodes containing the object data and each intermediate node pointing at one parent and eight children. This approach is good at representing models with large amounts of local detail in some parts and large spaces in others, and it can skip large, empty spaces fast. It requires a more complex traversal algorithm, but efficient solutions to this do exist [Spackman91]. Octtrees are sensitive to pathological cases which generate large data structures for quite simple scenes. For example, a number of large planar objects, such as the walls of a room, passing through a scene close to a subdivision boundary can cause repeated subdivision. An alternative subdivision scheme uses non-equally sized sub-regions, which is slightly more complicated to implement and has largely been superseded by the next technique [Dippé84].

Hierarchical bounding volumes are constructed by creating a bounding volume for each object in a scene, and then clustering these volumes to form a tree data structure [Kay86]. This approach is popular, because it has the hierarchical advantages of the octtree method — it is good at representing models with large variations in local complexity — with the added advantage that the subdivisions in the structure are tailored to the objects in the scene. The bounding boxes are simple to construct using six planes parallel to the coordinate axes. However, these perform badly for long thin diagonal objects, and so are usually supplemented by a second set of six planes equally inclined to the axes. The preprocessing step needed to construct bounding boxes and to cluster them to build the tree can be time consuming and the ray/box intersection tests are more expensive than the ray traversal algorithms of the gridcell and octtree methods [Goldsmith87].

Binary space partitioning (BSP) is another hierarchical subdivison method in which the space occupied by the model is divided by a plane. Objects are then tagged according to which side of the plane they lie on. At each level in the hierarchy a new plane is introduced, yielding a new set of objects on either side of the plane. Objects which are cut by a plane must be subdivided into parts lying on each side of the plane. This method is not widely used for ray tracing, but has proved popular for hidden surface removal and geometric modelling [Fuchs80].

The application of acceleration techniques leads to a dramatic improvement in performance. For example, for scenes we have tested, rendering times for a 512 by 512 resolution image can be reduced using the gridcell approach from over 12.5 hours to 107 seconds, on an HP 730 — an improvement factor greater than 420.

2.2 Parallel ray tracing

Ray tracing has become a classic demonstration of parallel processing. It is usually classified as an *embarrassingly parallel* problem, and is easily parallelised by processing each primary ray independently. For small models good speed-up can usually be obtained. However, for realistically large models, such as those in complex computer animations and in scientific visualisation, the model sizes can cause problems. Few sites have sufficiently large-scale parallel machines to examine the effects of running large problems on large numbers of processors, and so reported results have tended to be for smaller systems.

Several different approaches have been tried, dividing principally into ray-parallel and object-parallel methods. With ray-parallel approaches, different rays are processed by different processors which each check for intersections with the scene model. With object-parallel methods, each ray is sent to all processors which then check it against different parts of the model in parallel. When combined with different parallel architectures and the different acceleration techniques described in the previous section, these approaches yield a wide range of alternative strategies. Some of the main ideas which have been explored are summarised below.

On **single instruction/multiple data** (SIMD) and vector processing machines, all processing elements (PEs) operate in lock-step, executing identical instructions on different data.

For ray tracing on such machines a ray-parallel approach is usually adopted — multiple rays are intersected in parallel with each object in the scene. In this case the processing time grows at least linearly with the object count, so that for very complex scenes, quite lengthy processing times can result. Unfortunately, because the same instructions must be executed by every processor, it is very difficult to use acceleration methods, making this type of architecture unattractive [Plunkett85].

On **multiple instruction/multiple data** (MIMD) architectures, there is considerably greater flexibility, as all processors can execute independently. With *static geometric parallelism* the problem is partitioned between processors before rendering begins. Using a gridcell, regions can be subdivided and each processor assigned a subregion. Regions can be cuboid volumes or sets of voxels, and can be allocated to processors using a tiling method. This approach was basically developed for distributed-memory architectures with communication via links between neighbouring processors, and is ray-parallel in nature. It suffers from large numbers of interprocessor messages when tracing secondary rays. One might expect this intuitively for scenes with large numbers of secondary rays, as it is not at all unusual to trace several million rays for a single 512 by 512 image. For high quality images, this number rises very dramatically. For example, for anti-aliased images output to film it is common to trace many millions of primary rays, leading to tens of millions of rays in total. Static partitioning schemes also frequently exhibit poor load balancing due to the different amounts of computation performed by each processor [Cleary86], [Kobayashi88].

An alternative to static scheduling is *dynamic parallelism*, where the work of the processors is adjusted dynamically to improve load balance. The commonest approach to dynamic load balancing is to use a *processor farm*. Here, the parallel processing is managed by a master processor, which "farms out" packets of work to a number of worker processors. The packets must be large enough to minimise communication and synchronisation overheads, but small enough to enable good balancing to be achieved by ensuring minimal idle time at the end of a computation. This is exploited in ray tracing by dividing image space into small regions, with each region treated as a work packet, using a region-parallel approach. On distributed memory machines this approach works well, provided that the whole model can be loaded into the local memory of every processor. If this is not possible then communication overheads increase dramatically and performance can become very poor. Indeed, with poor model distribution it is possible for programs to run more slowly as additional processors are added. On shared memory machines, process farming can also be used, and the incorporation of the various acceleration techniques is also straightforward.

A large amount of research on ray tracing has been conducted using distributed memory, message passing architectures. In large measure, this has been due to the availability, at reasonable cost, of such machines — transputer-based products, for example. Considerable ingenuity has been required to find ways to shoe-horn large models into the limited memory of the local processors, but the results seem mostly to have been disappointing in terms of scalability — see the papers cited below for figures.

One approach to data distribution is to use the hierarchical bounding volume technique to subdivide the model space, and then to distribute the resulting tree data structure across the available processors [Scherson88], [Müller91]. The tree is divided into upper and lower levels, with the upper-level nodes distributed to all processors and sub-branches of the lower levels distributed uniformly across the processors. Clearly there is a relationship here between the topology of the tree and the number of processors available. Rays can now be traced in parallel, with each ray sent to all processors in order to check the upper part of the tree. Eventually, the lower part of the tree is processed only by that processor which contains locally a copy of the objects which filter through the upper levels of the hierarchy. Invoking the processing of the lower part of the tree involves passing a message to the appropriate processor. Load balancing is attempted by sending the initial rays to those processors with the least work to do. In practice, this method leads to quite a large amount of message passing, which eventually limits its performance.

A more general approach was devised by Green, based on a caching scheme for model data [Green91]. Here, parts of the model are passed between processors upon demand. The method capitalises on the coherency of neighbouring rays, which frequently intersect the same objects. After an initial data transfer has taken place, subsequent neighbouring rays, processed on the same processor, can access the same data without any further transfers. A generalisation of this approach was used by Badouel et al [Badouel92], who implemented a virtual shared-memory management system, in software, for a distributed-memory, 64-processor Intel iPSC2. Their results are the best we have seen reported for such machines. For ray tracing, they report scalability better than 78% on 64 processors, although their compute times are significantly greater than those we report later for the same images. Their work suggests that the use of virtual shared-memory is the best approach to this problem on MIMD computers.

In assessing these techniques for parallelising ray tracing with large models, we have found it difficult to avoid the conclusion that most of them were on the one hand motivated, and on the other hand thwarted, by the limitations of the machines being used. Over many years we have been frustrated by the limitations of different hardware systems and have wanted to use a general purpose machine which did not require us to wrestle with its own limitations. In the next section we describe the machine we have used in an attempt to avoid these problems.

3 The Kendall Square Research KSR1

3.1 KSR1 architecture

The KSR1 is a distributed-memory parallel computer whose hardware supports a shared-memory programming model [Frank93]. This shared view of memory is implemented via a paged, virtual-memory addressing scheme. Because the novel memory management lies at the heart of the architecture, the processing elements (PEs) of the KSR1 are custom-designed. Each processor has 32 Mbytes of memory, which functions as a local cache, and a peak computation rate of 20 MIPS and 40 Mflops[1]. Processors are connected via a two-level hierarchy of rings. Level zero rings each connect up to 32 processors and have a communication speed of 1 Gbyte per second, while multiple rings are connected by a level one ring, rated at 4 Gbytes per second. The Manchester machine has 32 processors on a single ring zero, making it a '3G

[1]These figures refer to the KSR1. The recently announced KSR2 has double this performance.

machine' — 1.2 Gflops, 1 GB memory and 1 GB/sec inter-processor communication. Internally, each PE has a sub-cache of 0.5 Mbyte, divided equally between instructions and data. This global view of the architecture is shown in Figure 1. A full-sized KSR1 would have 1040 processors.

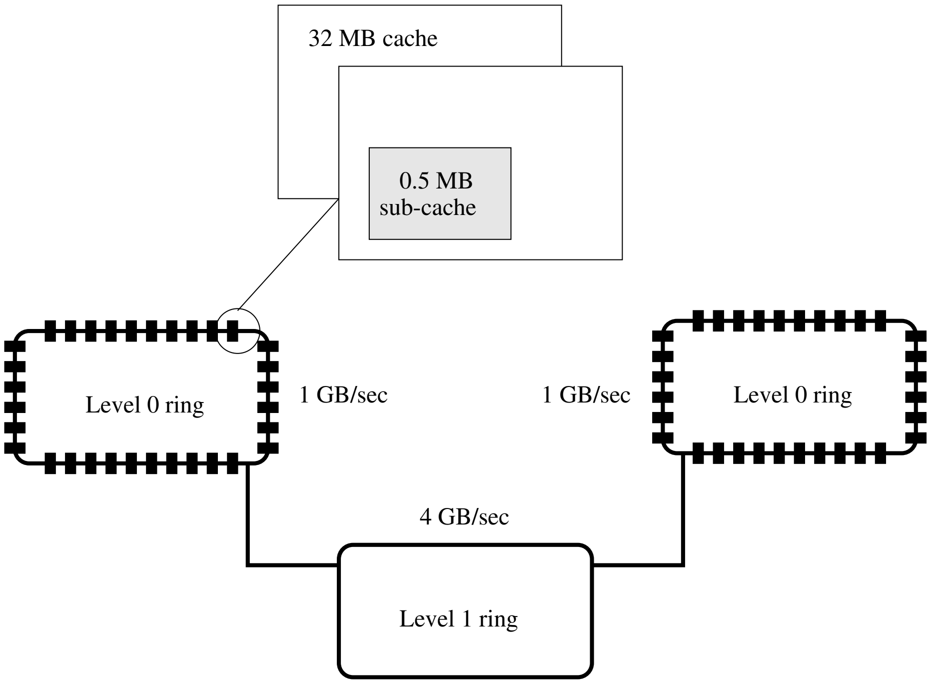

Figure 1: KSR ring architecture

What makes the machine interesting is the way that data is moved around the system, transparently as far as the programmer is concerned. The memory addressing mechanism is organised into 16 Kbyte pages for purposes of memory allocation and paging to/from disc. However, these large pages are divided into 128 byte sub-pages, and it is these small sub-pages which are moved between processors when access to specific memory locations is required. Movement of data is therefore a fairly fine-grained operation.

When a PE requires access to a memory location, a check is first made to see whether that location currently resides in the sub-cache of that PE. If this condition is satisfied then the location is accessed without any inter-processor communication. If the location resides on another PE, then the action taken depends on whether a read or write access is required. In the case of a read operation, a *copy* of the sub-page containing the required virtual address is moved to the local memory of the requesting PE from wherever it currently resides. In the case of a write operation all other copies of the sub-page on other PEs (resulting from previous read operations) must be *invalidated* and then the location can be updated. Subsequent reads by other PEs will propagate new copies of the sub-page containing the updated data. We can see from this that a specific virtual memory location has no single 'home' within the system — addresses migrate between processors according to data access patterns. This means that the whole of memory functions like a large set of caches and gives the architecture its name:

ALLCACHE. Clearly, the problem of maintaining the coherency of the memory during writes is a complex one, and it is the way in which the hardware supports this which is the most novel feature of the system.

An important general tenet of parallel processing — *locality of reference to data* — still holds with this architecture. Accesses to non-local data are not without cost. The latencies associated with accesses at various levels in the hierarchy are summarised in the following table:

Location of data	cycles
Processor register	1
Sub-cache (0.5 MB)	2
Local cache (32 MB)	18
Other PE (ring 0)	175
Other PE (via ring 1)	600
Page fault to disc	20 ms

The feature whereby read accesses automatically yield a *copy* of data is especially powerful for applications where a model is created once and then used many times. This is quite common in visualisation, and especially so in the case of ray tracing. Because data is copied *on demand*, there is no need to write algorithms which decide where to place data *a priori*. Nor is it necessary to explicity move data between PEs because this is performed automatically by the system[2]. Particular care is required when *creating or updating* data to avoid a situation in which locations on the same sub-page are concurrently updated by many different processors. This can create a game of 'ping-pong' in which the master (valid) copy of a sub-page is continually moved between different PEs.

3.2 KSR1 software

The operating system on the KSR1 is a KSR-modified version of OSF/1, which provides a standard UNIX environment. A range of compilers and other widely used products is becoming available on the system. One of these is the X Window System, which our software currently uses for displaying images.

Parallelism on the KSR1 is controlled by using *pthreads*, which are lightweight processes. The KSR1 pthread interface is based on the IEEE POSIX draft standard [POSIX].

A group of pthreads forms a task which shares a single virtual address space, unlike processes which only share data through explicit system calls. Thus pthreads trade data security for low overheads. Although fairly cheap to create, pthreads are generally best used with fairly coarse-grained parallel computations. A number of slave pthreads are created and then subsequently controlled by a master pthread. On the KSR1 it is possible to request that individual pthreads are allocated to specific processors, thereby guaranteeing a one-to-one mapping of threads to PEs. If this is not done then the operating system will place threads on PEs in order to balance the overall load on the machine.

[2] In fact, a copy is obtained from the physically-closest processor which has a valid copy of the sub-page, thereby minimising latencies as far as possible.

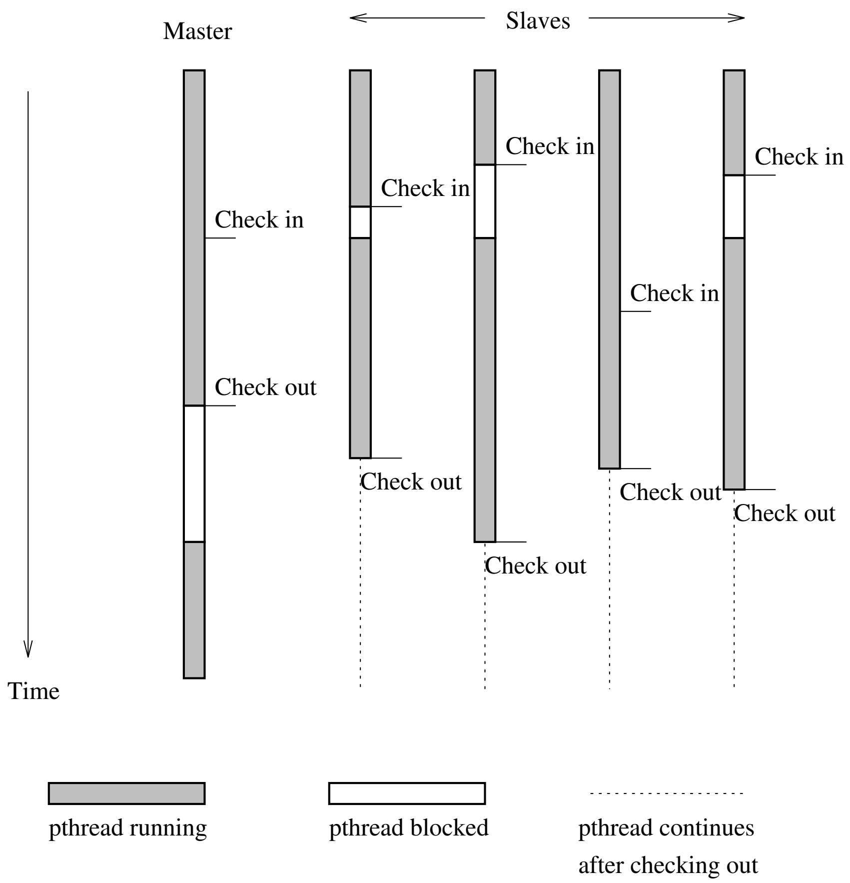

Figure 2: KSR pthread synchronisation

In order to maintain memory consistency the KSR1 provides mutual exclusion and synchronisation constructs. These are the *mutex lock* and the *barrier*.

A *mutex lock* is used to protect critical regions of code. When one pthread locks a mutex then any other pthread attempting to lock the same mutex will block until the first pthread unlocks it. Thus, a mutex lock/unlock can be placed around access to shared variables (or code) to guarantee coherency.

A *barrier* is used for synchronising pthreads, and is illustrated in Figure 2. Operation of the barrier is controlled by a *master pthread*, which in general controls a number of other pthreads, called *slaves*. Pthreads check in and out of the barrier. When a slave checks in, its execution is suspended until the master checks in (unless it has already checked in). At this point, the slaves commence executing until they run out of work to do, at which point they check out of the barrier. Having released the slaves, the master can perform useful work itself, after which it checks out of the barrier. The master then becomes suspended until all of the slaves have checked out of the barrier. This provides a means for the master to set up work for the slaves, to release them for processing once the correct data structures have been set up, and then to wait for the work to be completed before moving on to the next phase of the computation.

A feature supported by the compilers is the ability to force sub-page alignment of variables.

This means that variables which need to be accessed independently can be placed on different sub-pages, or alternatively that variables which will be used together can be grouped on the same sub-page(s). Although this is a machine-dependent feature, it is limited to to variable declarations and so is easy to incorporate into programs. It is also possible to pre-fetch data if the patterns of access can be predicted in advance. This mechanism avoids the high latency associated with fetching data from a different PE by transferring it ahead of time, a feature usually called latency hiding. A related post-store mechanism exists to write back copies of an updated sub-page.

The KSR1 provides a performance monitor, called PMON. This gives information on a thread-by-thread basis about execution times and memory hits. The PMON data is obtained from the KSR1 event monitor, hardware registers and the system kernel. There are eighteen PMON counters, which measure statistics such as the number of user clock cycles, the total time that the central execution unit was stalled due to sub-cache misses, and so forth. These can be used to see how long each processor is stalled, and which data fetches (from local cache or from another processor's local cache) affect it most. We have used these facilities to analyse the behaviour of our ray tracer.

4 The KRT ray tracer

The ray tracer is called KRT (KSR Ray Tracer), and is written in C. Rather than use a public domain program such as Rayshade [Kolb91], we decided to use our own code, as we could design and adapt this more readily to suit the KSR1's architecture. The ray tracer supports polygons and quadrics with cutting planes. It has an object-oriented structure which permits new classes of primitives and associated methods to be added easily. Model data is read from standard input, normally in the NFF format [Haines87], although other formats are also supported.

The acceleration technique we chose was the regular gridcell. The main reasons for this choice were:

- It is simple to implement. The cell traversal algorithm is very straightforward and fast, and lends itself to prefetching data, because the list of cells accessed by a ray is easily predicted in advance. In fact, although we planned to use prefetching we have not done so in practice for reasons mentioned later.

- Its regular structure makes it well-suited for parallelising, because cell addresses can be calculated without the need to follow lists of pointers. Complex structures which cause widely-dispersed memory accesses do not perform well with caching schemes. We have evidence from other applications that this is so on the KSR1, but a description of the reasons requires a level of detail which is beyond the scope of this report.

- There is evidence that, in the general case, the gridcell performs at least as well as other methods [Spackman90]. As mentioned previously, it is poor for models which have very uneven spatial distribution. We will return to this point later. A frequent reason not to use gridcells is lack of memory, which is not a problem on the KSR1.

Ray tracing scenes with reflective surfaces and many shadow rays causes unpredictable amounts of computation for each primary ray. For this reason a dynamic load balancing scheme is needed, and we selected the processor farm method. This has the advantage of being simple to implement, with regions of the screen farmed out as work packets. The primary rays cast through pixels then access neighbouring sections of the grid cell data structure, capitalising on the coherency property of neighbouring rays. Although there is nothing original about this approach, this feature is one of the main reasons why we selected it for the KSR1, as it automatically yields locality of reference to the model data.

4.1 The initial implementation

Our first parallel implementation was organised as follows. When the program is first run, it is created as a single master pthread. This master process creates several slave pthreads, and then uses a barrier to synchronise them. Each pthread uses a mutex lock to test and update a shared variable, called `next_packet`, indicating the next area of the image to be computed, as seen in Figure 3. `next_packet` is simply an integer from which the screen region can be calculated by knowing the size of each region. It is initialised as the total number of regions to be processed and is decremented each time a region is processed by a pthread.

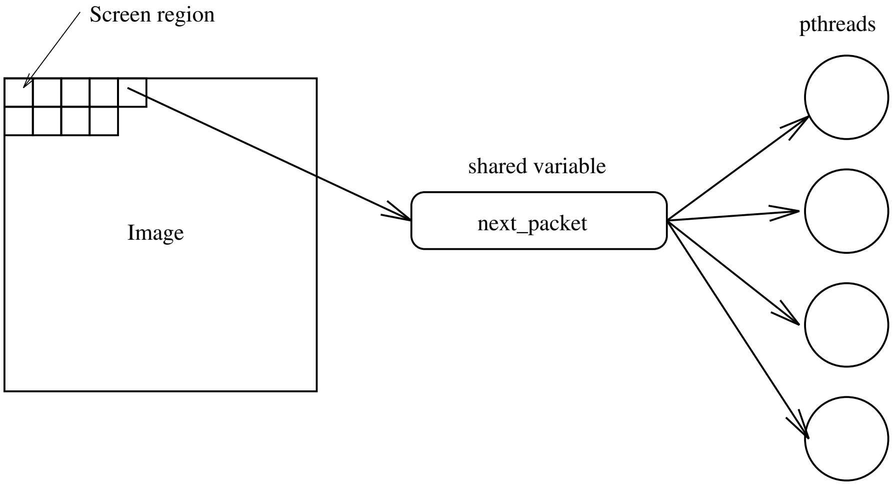

Figure 3: Process farming via a shared variable

When the image has been completely rendered the slave pthreads check out of the barrier, and this allows the master pthread to continue executing. Thus, all pthreads compute continuously as long as there are packets of work for them to undertake. No explicit data transfers take place — parts of the model are paged to individual PEs on demand. This demand paging should suit ray tracing particularly well, as neighbouring rays in a region (work packet) will tend to access common parts of both the gridcell and the geometric model data.

All images are computed in full colour. They can be output to files for subsequent viewing, or displayed directly on a screen using X Windows.

4.2 Initial results (32 processors)

The ray tracer was tested on our 32-processor KSR1 using four different scenes:

1. **A Sphereflake.** A recursive structure of spheres. This is a well-known test model for ray tracing from the Haines database. In our test, we have 7,382 primitives and 3 light sources, shown in Figure 8.

2. **Reflective spheres.** A simple scene containing 6 reflective spheres, a reflective ground plane and 2 light sources, seen in Figure 9.

3. **The Thinker.** A statue surrounded by large mirrors. It contains 3,036 primitives and 2 light sources, and is shown in Figure 10. This model has a fairly uneven spatial distribution with a large number of polygons around the head. A description of the origin of this model can be found in [Williams93].

4. **An Office.** This scene is the most complex of those tested and contains over 100,000 polygons and 5 light sources. It can be seen in Figure 11. This too has a very uneven spatial distribution.

The same pictures were also computed on a Sun Sparcstation-1 and on a Hewlett-Packard 730 CRX24Z, and the results for these machines are included for comparison. All times are given in seconds. They represent CPU usage, measured with the Unix interval timers, averaged over a number of runs with no other users on the systems. On the KSR1, the PMON tool was also used, and the times quoted are the elapsed times. All of the KSR results reported in this report are for a one-to-one mapping of pthreads to processors. However, the gridcell generation was performed on a single processor, so in measuring speed-up we report the subsequent ray tracing times without the preprocessing step. A parallel grid generation and updating algorithm is currently under development.

4.2.1 Sphereflake

The time for a naive tracer — that is, one without any acceleration — on the HP is 46,000 seconds, or 12hrs:47mins. The times for the HP and Sun versions using the gridcell approach were:

	HP (64^3 gridcell)	Sun (64^3 gridcell)
Build	1.6	5.3
Render	107	369
Total	108.6	374.3

The time taken to build an equivalent gridcell on the KSR1 was 7.0 seconds, while the rendering times for different numbers of pthreads were as follows:

	Number of processors						
	1	2	4	8	16	24	31
Render	414	212	104	53.4	26.8	17.8	14.2
Total	421	219	111	60.4	33.8	24.8	21.2
% of linear (rendering only)	100	97.6	99.6	97.1	96.7	97.1	94.1

4.2.2 Other scenes

Rather than give all of the results here, we present a simple summary of the best case rendering times for each machine, using 31 pthreads in the case of the KSR1. We present the results for the four scenes mentioned previously, which we emphasise are also representative of other scenes we have tried:

	Sun	HP	KSR1
Sphereflake	369	107	14.2
Reflective spheres	136	39.6	4.5
Thinker	183	133	23.7
Office	N/A	686	137

The reflective spheres scene is simple enough to be faster to render *without* using gridcell acceleration, so this is what was done. The office scene was too large to be rendered on our Sun.

4.2.3 Comments on the initial results

The large amount of memory available on the KSR1 meant that the space occupied by the gridcell was not a problem. In practice, we noticed that the gridcell did not work very well with the office data. This model exhibited very uneven spatial distribution. For example, the office contains a number of computer terminals, and the individual keys on these are modelled by polygons. However, we believe that the regular structure of the gridcell has attractions, and we are implementing a two-level recursive version to overcome this problem. It is worth noting that the shared-memory programming paradigm of the KSR makes recursive algorithms very easy to implement; this can be far from the case for message passing machines using large models [Müller91].

The *single processor* time on the KSR1 is between three and four times that of the HP for most scenes. The HP is nominally rated at 76.7 MIPS and 23.7 Mflops, whilst the KSR1 peak figures are 20 and 40 respectively. However, the peak rate of 40 Mflops is only achieved with highly optimised code, and for the general case the maximum theoretical rate is more like 20 Mflops.

The KSR1 monitor, PMON, was used to analyse the performance. Output from the monitor is quite detailed and requires an understanding of the architecture which is beyond the scope of this report, so we will only summarise the results here. The PMON statistics indicated that the processors were only stalled, waiting for data from other PEs, for approximately 1% of the total time. We were surprised by this result. We had expected that the small sub-page size of the KSR would lead to larger overheads, but this shows that in practice the communication ring is not a bottleneck, even when fairly large data structures are demand-paged around the system.

It is worth emphasising that all of the times given in our results include the time to propagate the data between processors — this is frequently *not* included in results reported by other researchers, where model propagation is regarded as a pre-processing step.

The preliminary results show that the performance of the KSR1 continues to scale well, up to 31 processors. For the sphereflake, the speed-up obtained on 31 processors represented 94.1% of the ideal linear value. It is worth noting here that results presented later in this report show an improvement over this, partly as a result of changes to our algorithm, and partly because of improvements to the KSR operating system.

5 Results on 178 processors

The preliminary results reported above were sufficiently encouraging that we wished to discover how well our implementation would scale on a larger machine. Fortunately, we were able to gain access to a machine with 256 processors for a short period, and in this section we report the results of this and of a change in the processor farm algorithm which resulted from it.

Because of limited access to the large machine, we restricted our tests to the sphereflake model. The first experiment was to run the tests with 178 processors, this being the maximum we could access at that time. Figure 4 summarises the results by showing plotting one-over-time, in seconds, for different numbers of pthreads, with one pthread per processor.

It can be seen that above 64 processors, and particularly above 100 processors there is some variability in the performance. With 178 processors the performance has dropped to the equivalent of 73% of the theoretical linear speed-up. Tests, reported later in the report, confirmed that this variability in the results could be eradicated by waiting for a short time between successive test runs. This was caused by background memory garbage collection by the operating system.

A second set of runs was performed with up to 130 processors to investigate the effects of anti-aliasing, by casting nine primary rays per pixel. This results in a higher computation to communication ratio and a correspondingly marked improvement in the scaling. On 130 processors the anti-aliased version yields 95.2% of equivalent linear speed-up.

We discovered by analysing the PMON output that the data propagation delays still only accounted for 1% of the performance loss, so we wanted to explain where the remainder of the time was dissipated. Two possible causes required investigating:

Poor load balancing. Farming out regions of the screen can cause poor load balancing if a large amount of computation is needed for a few regions. Analysis of the PMON statistics showed that reducing the packet size gave a better load balance, but the computation times did not improve. It is not possible with our algorithm to reduce a packet to less than a single primary ray, so if poor load balancing were the only factor then there would

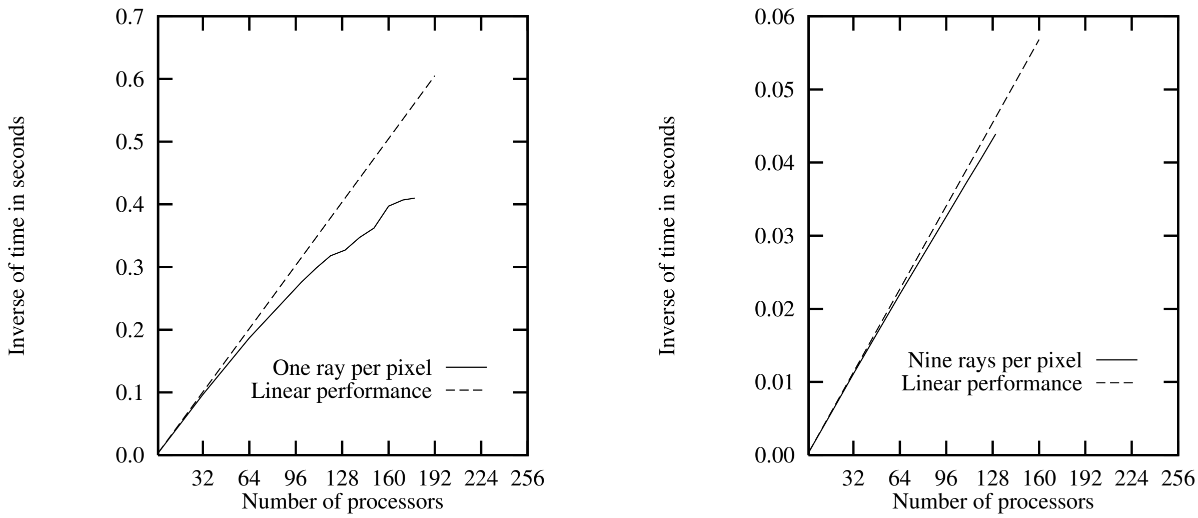

Figure 4: One over time (in seconds) for one ray per pixel, and nine rays per pixel (Sphereflake)

be little we could do about it, other than to design a new strategy. In fact, when a work packet was reduced to a single primary ray the times became worse.

Mutex bottleneck. As the number of processors increases, there is increased likelihood of contention, when accessing the shared variable in order to obtain the next work packet. This could explain why reducing the work packet size, although it produced a better load balance, did not reduce the total times.

6 A modified processor farm strategy

We therefore devised a modified processor farm allocation strategy, based on a two-level hierarchy of regions and packets. With the modified scheme, the screen is statically partitioned into regions, each of which has its own mutex-locked shared variable, depicted in Figure 5. These regions are then statically allocated to pthreads (processors). This minimises the number of pthreads accessing an individual lock.

Within a region, packets of pixels are farmed out as before, until a given screen region has been completed. When this happens, the pthreads associated with that region are re-allocated to another region which still has work to perform. The region is selected which has the minimum number of pthreads competing for its lock, so that once again an attempt is made to minimise lock contention. Clearly, once only a few regions remain, all of the pthreads are competing for access to a reducing number of mutex locks, and eventually, for the last region, for a single lock.

This can be represented as a family of curves, as seen in Figure 6. This shows how the

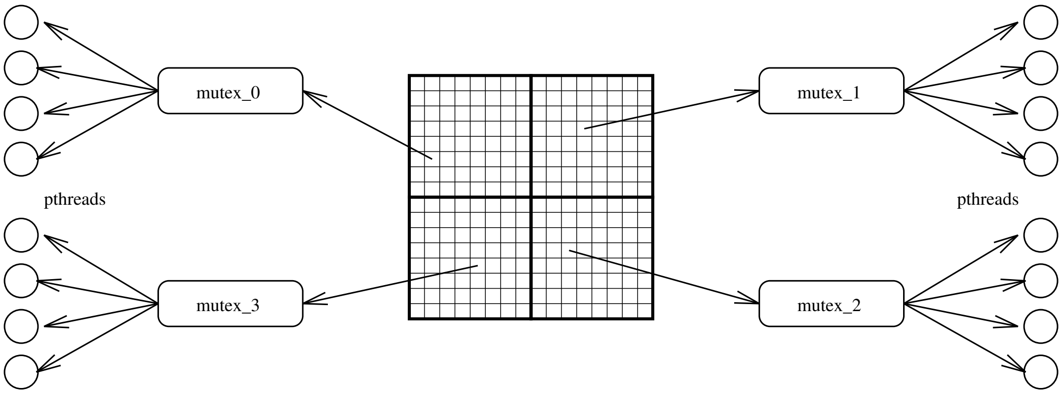

Figure 5: Modified processor farm strategy

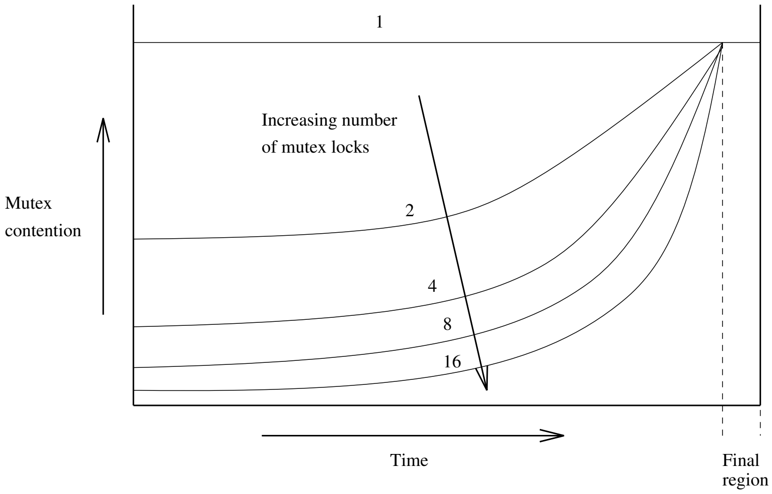

Figure 6: Probability of mutex contention

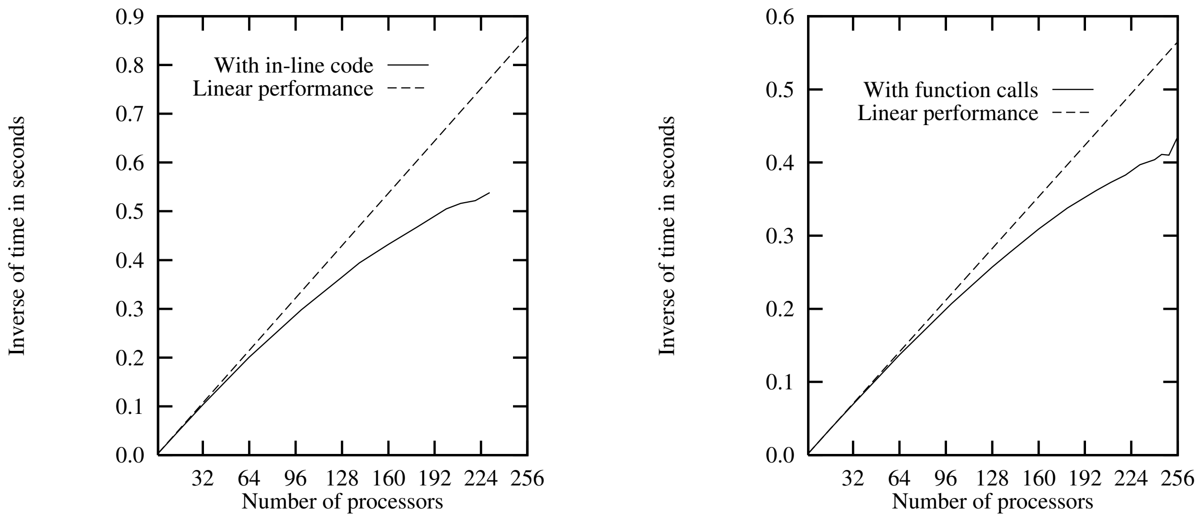

Figure 7: One over time, with in-line code and function calls (Sphereflake)

probability of contention rises over time, as regions are completed. (Note that this figure is a schematic only, not an accurate graph. The probablity of contention will actually rise in a series of steps.) The family of curves is obtained by varying the number of regions, which controls the number of mutex locks. It also illustrates that for the final region, in which all pthreads are accessing a single lock, the contention will rise to the same level as that of the simple scheme tried in the first experiments. By this time, however, considerable savings may already have been made.

6.1 Results of the new algorithm on 256 processors

With the modified algorithm we were able to run further tests on a large machine. The timing results are shown in Figure 7. They show a marked improvement, with a much smoother curve and an equivalent speed-up of 76.9% on 256 processors.

It is interesting to note that the modified processor farm algorithm which we are proposing is very similar to one used by Nieh and Levoy for volume rendering on the DASH machine [Nieh92]. In fact, our approach was devised independently, and for different reasons. Nieh and Levoy used it as a way to minimise dynamic data communication; we are using it to avoid shared variable contention.

In order to set these results in context, we show in Table 1 the actual rendering times in seconds. These results were measured for up to 256 processors. The second column shows the effect of using a compiler optimisation which generates in-line code in place of function calls, while the third column gives the corresponding times with normal function calls. In-line code results in a larger executable file, but a 26% improvement in speed, which we attribute

Processors	In-line	Function call
1	298.14	452.90
16	19.16	28.42
32	9.69	14.39
64	4.98	7.30
100	3.36	4.84
128	2.74	3.90
140	2.54	3.62
160	2.31	3.24
180	2.14	2.96
200	1.98	2.76
210	1.94	2.68
220	1.91	2.61
230	1.86	2.52
240	N/A	2.48
245	N/A	2.44
250	N/A	2.44
256	N/A	2.30

Table 1: Rendering times in seconds, improved processor farm (Sphereflake)

to better cache utilisation. However, under these circumstances the improvement in speed-up is only 70% on 230 processors.

It can be seen that the times are quite small, but do not yet approach the speed needed for real-time rendering. If we disable secondary rays then these times reduce by two-thirds. In the software we are now developing, we capitalise on this by using adaptive rendering — by shooting less rays and interpolating pixel values, and by using intelligent "damage repair" methods for making incremental changes to pictures for scenes which are changing dynamically. The software also continues to fill in detail by generating successively higher-quality, anti-aliased images, as long as the user does not change any of the viewing parameters.

7 Conclusions

Our first conclusion is that we have found the KSR1 to be very easy to program. The fact that it copies data for read operations makes it very attractive for this application and, we believe, for graphics and visualisation applications in general. The demand paging of data means that only those portions of the model data actually needed will propagate to individual processors. The programmer is not freed from thinking about correct structure, in terms of attempting to achieve locality of reference to data, but having produced a well-designed program can leave it to the system to manage data movement at run-time.

Movement of data between processors carries a very small penalty — only 1% of the total time. This result is perhaps surprising, given that data is moved on demand in 128 byte sub-pages, but is very encouraging. In part, this is probably due to the locality of reference properties of ray tracing using regions as work packets, and the regularity of the gridcell structure. The feature of the KSR whereby it fetches copies of sub-pages from the physically-closest processor also capitalises on this regularity in our data structures. We had originally thought of using the KSR1's pre-fetch facility to reduce the delays, but have not bothered because the overheads are so small.

As an aside, it is worth commenting that, in common with other researchers, we are reaching the point where we can compute images faster than we can display them — particularly via X Windows, as this system uses a buffer to serialise output. This buffer becomes a significant bottleneck. We are therefore developing our own image display protocols to avoid this problem.

This application shows excellent speed-up — much better than most of the other results we have seen reported. For applications where high-quality images are needed — such as in computer animation — the speed-up, which is better than 95% on 130 processors is notable. We have not had an opportunity to run anti-aliased tests with the improved processor farm algorithm, but believe this should give even better results.

In our view the results reported here demonstrate that the KSR1 has an architecture which is well suited to image generation, and hence for a range of visualisation tasks. For example, we are also developing a variety of volume renderers on the same machine, and these exhibit similar scalability. In situations where a large-scale parallel machine is required for simulation, it is possible to add visualisation of the results at modest cost using the kind of software we have outlined.

8 Acknowledgments

We would like to acknowledge the help from our CNC colleagues Graham Riley and Mark Bull, in optimising our KSR1 code. We gratefully acknowledge the Fraunhofer-Institut für Graphische Datenverarbeitung in Darmstadt, and Siemens, for supplying us with the office model test data, and our colleagues Gareth Williams, Toby Howard and Alan Murta for the Thinker test model.

We would like to give our special thanks to the United States Army Research Laboratory in Maryland, for permission to use their 256-processor KSR1 for larger-scale tests, and to Marshall Presser, Steve Pass, George Mozdzynski and Tim Allen of Kendall Square Research Corporation for assistance with running the tests on that machine.

The financial support of the Science and Engineering Research Council of Great Britain is gratefully acknowledged, under Novel Architecture Computing: Foundation Support (contracts B/18535 and B/28669).

The CNC is a partner in a Network funded under the European Community's Human Capital and Mobility Programme, *Interactive Image Synthesis and Processing on Innovative Computer Architectures*, grant number ERB–CHRX–CT93–0088. This support is gratefully acknowledged.

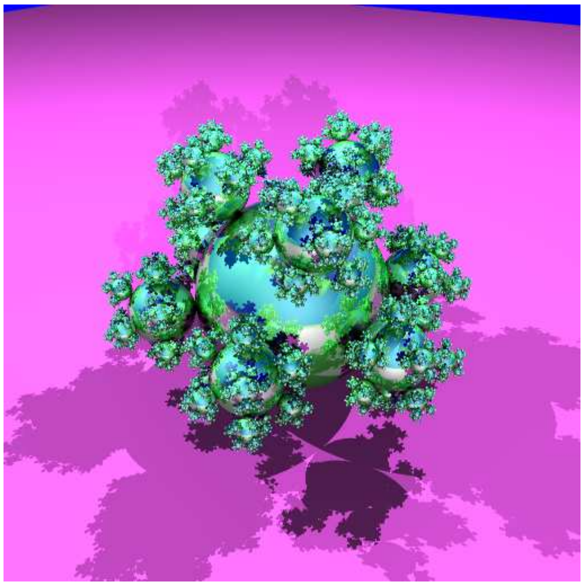

Figure 8: Sphereflake

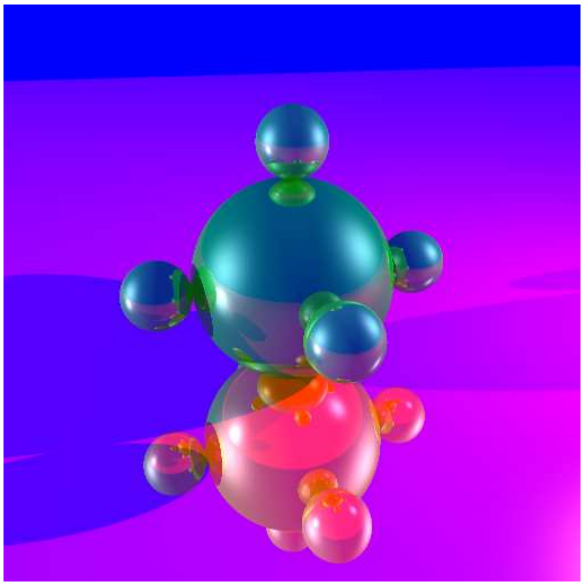

Figure 9: Simple scene

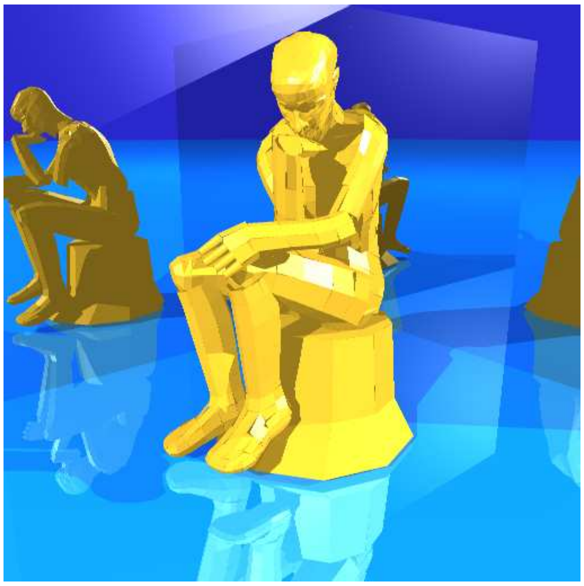

Figure 10: The Thinker

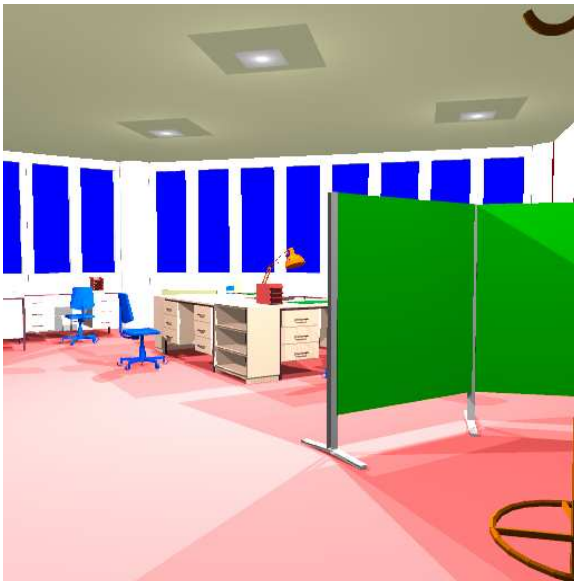

Figure 11: An office

References

[Badouel90] Didier Badouel, Kadi Bouatouch and Thierry Priol, **Ray Tracing on Distributed Memory Parallel Computers: Strategies for Distributing Computations and Data**, Course Notes for Course 28, ACM Siggraph, 1990, pp. 185–198.

[Badouel92] Didier Badouel and Thierry Priol, **An Efficient Parallel Ray Tracing Scheme for Highly Parallel Architectures**, *Advances in Computer Graphics*, R.L. Grimsdale and A. Kaufman (Eds), Springer-Verlag, 1992, pp. 93–106.

[Cleary86] Cleary J. G., Wyvill B. M., Birtwistle G. M. and Vatti R., **Multiprocessor Ray Tracing**, *Computer Graphics Forum, 5 (1), 1986*, pp. 3–12.

[Cleary88] Cleary J. G. and Wyvill G., **Analysis of an Algorithm for Fast Ray Tracing Using Uniform Space Subdivision**, *The Visual Computer, 4 (2), 1988*, pp. 65–83.

[Dippé84] Dippé M. and Swensen J., **An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis**, *ACM Computer Graphics 18 (3), July 1984*, pp. 149–158.

[Fuchs80] Fuchs H., **On Visible Surface Generation by A Priori Tree Structure**, *ACM Computer Graphics 14 (3), July 1980*, pp. 124–133.

[Fujimoto86] Fujimoto A., Tanaka T. and Iwata K., **ARTS : Accelerated Ray Tracing System**, *IEEE Computer Graphics and Applications, 6 (4), 1986*, pp. 16–26.

[Frank93] Frank S., Burkhardt III H. and Rothnie J., **The KSR1: Bridging the gap between shared memory and MPPs**, *Proceedings Compcon '93, IEEE, February 1993*, pp. 285–294.

[Glassner84] Glassner A. S., **Space Subdivision for Fast Ray Tracing**, *IEEE Computer Graphics and Applications, 4 (10), 1984*, pp. 15–22.

[Goldsmith87] Goldsmith J. and Salmon J., **Automatic Creation of Object Hierarchies for Ray Tracing**, *IEEE Computer Graphics and Applications, 7 (5), 1987*, pp. 14–20.

[Green91] Stuart Green, **Parallel Processing for Computer Graphics**, *Pitman, 1991.*

[Haines87] Haines E. A., **A Proposal for Standard Graphics Environments**, *IEEE Computer Graphics and Applications, 7 (11), 1987*, pp. 3–5.

[Kay86] Kay T. L. and Kajiya J. T., **Ray Tracing Complex Scenes**, *Computer Graphics, 13 (2), 1986*, pp. 158–64.

[Kobayashi88] Kobayashi H., Nishimura S., Kubota H., Nakamura T. and Shigei Y., **Load Balancing Strategies for a Parallel Ray Tracing System Based on Constant Subdivision**, *The Visual Computer, 4 (4), 1988*, pp. 197–209.

[Kolb91] Kolb Craig E., **Rayshade User's Guide and Reference Manual**, *Yale University, Department of Mathematics, July 1991.*

[Müller91] Müller S., **A Parallelised Intersection Calculation for Radiosity and Ray Tracing**, *Fraunhofer Institut für Graphische Datenverarbeitung, Darmstadt Technical Report*, 1992.

[Nieh92] Nieh J. and Levoy M., **Volume Rendering on Scalable Shared-Memory MIMD Architectures**, *Proc. ACM Workshop on Volume Visualization, 1992*, pp. 17–24.

[Plunkett85] Plunkett D. J. and Bailey M. J., **The Vectorization of a Ray Tracing Algorithm for Improved Execution Speed**, *IEEE Computer Graphics and Applications, 5 (8), 1985*, pp. 52–60.

[POSIX] IEEE POSIX Draft Standard, **Threads Extension for Portable Operating Systems, (P1003.4a).**

[Scherson88] Scherson I. D. and Caspary E., **Multiprocessing for Ray Tracing : A Hierarchical Self-Balancing Approach**, *The Visual Computer, 4 (4), 1988*, pp. 188–96.

[Spackman90] Spackman J. N., *University of Bath Computer Science Technical Report 90–33, 1990, (also Phd. Thesis).*

[Spackman91] Spackman J. N. and Willis P. J., **The SMART Navigation of a Ray through an Oct-tree**, *Computers and Graphics, 15 (2), 1991*, pp. 185–194.

[Speer92] Speer L. R. **An Updated Cross-Indexed Guide to the Ray Tracing Literature**, *Computer Graphics, 26 (1), 1992*, pp. 41–72.

[Williams93] Williams J.G., Murta A.D. and Howard T.L.J., **Modelling Rodin's Thinker: A case study combining PHIGS and ray tracing**, *Proceedings First Bilkent Computer Graphics Conference, Ankara, Turkey, July 1993*, pp.243–254.

[Whitted80] Whitted T., **An Improved Illumination Model for Shaded Display**, *CACM, 23(6), June 1980*, pp.343–349.