Universidade do Minho
Escola de Engenharia

Miguel Dias Coelho da Cruz Nunes

Interactive Ray-Tracing

Outubro de 2009

Universidade do Minho
Escola de Engenharia

Miguel Dias Coelho da Cruz Nunes

Interactive Ray-Tracing

Mestrado em Engenharia Informática

Trabalho efectuado sob a orientação do
Doutor Luís Paulo Santos

Outubro de 2009

Abstract

Ray Tracing is a method capable of producing high quality images by simulating paths of light rays within three dimensional scenes resulting in high quality photo-realism images. Several algorithms have been improved during these past ten years in order to achieve interactive frame rates.

In this thesis we focus on three key issues concerning interactive ray tracing performance: an accelerator data structure to reduce the number of ray-triangle intersections, parallelism within a single processor by making use of Streaming SIMD Instructions and ray coherence and, finally, parallelism at multi-core level with shared memory data.

For the first issue, we developed a single ray shooting method, which tested each ray against all triangles in a scene. We then proceeded to study two different accelerator data structures in order to reduce the number of ray-triangle intersections. The accelerator data structure selected for this thesis was the Bounding Volume Hierarchy method (BVH). The first BVH was implemented by A. Gonçalves[6] and the second one was based on I.Wald's BVH[25] and implemented by us. Tests indicate our BVH has less performance with larger scenes comparing to [6].

Secondly, we developed a vectorial version of the ray tracer, which packs sets of rays into vectors and exploits SIMD instructions. Both scalar and vectorial versions were extensively tested for later comparison. Among several metrics, the time of execution, the frame rate, number of rays and memory accesses were taken into consideration. A speed up from 2.8 to 3.7 was registered, which was less pronounced than the results from previous literature [3].

Finaly, since efficient and reliable sharing of data structures within a shared memory system is becoming a very relevant problem with the advent of many core processors, we decided to develop three multi-threaded solutions: the traditional *lock/unlock* implementation of pthreads, the *local queue* implementation where each thread maintains a local independent working queue, and the *Lock-free* approach which relies, not on locks, but on the Compare-and-Swap atomic synchronisation primitive and on retries. Tests show that the *local* approach outperforms the other two due to very good load balancing conditions, but we were able to demonstrate that the *lock-free* approach outperforms the *lock-based* one for large processor counts.

The general results obtained in this thesis indicate that more work has to be done in order to ensure maximum scalability in general multi-core systems.

Resumo

Ray Tracing é um método eficaz de produzir imagens foto-realísticas de elevada qualidade ao simular os efeitos físicos de raios de luz numa cena tri-dimensional. Durante a última década este método e os respectivos algoritmos têm sido melhorados e optimizados de forma a atingir a interactividade.

Nesta tese focamo-nos em três pontos-chave respeitantes à performance de *Ray Tracing* interactivo: estrutura de aceleração para reduzir o número de intersecções de raios contra triângulos, paralelismo num só processador fazendo uso das instruções SIMD e coerência entre raios, e finalmente, paralelismo entre múltiplos processadores em sistemas de memória partilhada.

Para o primeiro ponto, desenvolvemos um método escalar onde apenas um raio é disparado de cada vez e é testado contra todos os triângulos presentes na cena. De seguida, estudámos duas estruturas de aceleração para reduzirmos o número de intersecções. Foi decidido investigar e implementar o algoritmo de Hierarquia de Volumes (*Bounding Volume Hierarchy*). O primeiro algoritmo que testámos foi o do A. Gonçalves[6] e o segundo foi baseado no trabalho de I.Wald [25] e implementado por nós. Testes indicam que a nossa implementação do trabalho de I.Wald tem menos impacto na performance do *ray tracer* comparativamente a [6].

Em segundo lugar, foi desenvolvido uma versão vectorial, onde raios são colocados em grupos (ou pacotes) de vectores, usando instruções SIMD. Ambas as versões escalar e vectorial foram extensivamente testadas para posterior comparação. Dentro de várias métricas, foram registados o tempo de execução, o número de frames por segundo, o número de raios por segundo e acessos à memória. O ganho obtido nesta vectorização situa-se entre os 2.8 e 3.7 vezes, que foi um valor menor do que o encontrado em trabalhos anteriores[3].

Por último, visto que a eficiente partilha de estruturas de dados num sistema de memória partilhada se tornou num problema relevante com a promessa da presença de um elevado número de processadores por computador, foi decidido desenvolver três soluções *multi-threaded*: a implementação tradicional de *threads POSIX* com *locks/unlocks*, filas de trabalhos locais para cada *thread*, e a versão *lock-free* que, ao invés de usar *locks/unlocks*, faz uso de uma função atómica de sincronização de "comparação e troca". Testes revelam que a segunda versão se comportou melhor que as restantes devido ao bom balanceamento de trabalho, e provamos que a versão *lock-free* é superior à primeira versão para um elevado número de processadores.

Os resultado alcançados nesta tese sugerem que há espaço para melhoramentos e optimizações de modo a garantir a máxima escalabilidade possível para sistemas *multi-core.*

Contents

List of Figures

Listings

List of Tables

Chapter 1

Introduction

Ray tracing is a widely used rendering technique that has the ability to produce high quality images by simulating global illumination and physically based light transport effects[7]. Used originally as an off-line rendering algorithm, the development verified over the last few years, both at the software and hardware levels, has made it possible to achieve interactive ray tracing in accessible hardware[26]. The present condition of CPUs advancements suggests that in the near future these will focus on parallelism and many-core architectures. And since the ray tracing algorithm is embarrassingly parallel, high efficiency is expectable even for high degrees of parallelism. This parallelism is explored along three main techniques: the vectorial properties of modern CPUs, multi-threading in multi-core systems and parallel execution in cluster nodes. This work, among other features, focus on the first two techniques: vectorial calculations and parallelism in different multi-core systems.

Nowadays ray tracing is a powerful tool to deliver fast and physically realistic images. As such, several different approaches have been developed during the last ten years in order to optimize final results. Two of these are about how rays should be shot in order to obtain a frame: the single ray and the packet of coherent rays shooting [23], i.e., scalar versus vectorial approaches. Another approach to achieve interactive frame rates in ray tracing is by exploiting multi-threading on multi-cores by making use of the parallel properties found in the ray tracing algorithm. Another key element of ray tracing is the use of an accelerator data structure (such as kd-trees, grids or bounding volumes hierarchies), which makes possible the improvement of ray-triangle intersections by decrease the number of triangles a ray has to be tested against.

With such hardware and software advancements, it is now possible to achieve interactive frame rates without using supercomputers[28, 1]. Since current personal computers have much more CPU power, interactive ray-tracing has the opportunity to be implemented in all sort of graphics applications. Furthermore, interactive ray-tracing is able to deliver high quality results, solving much of the problems found in rasterization (OpenGL, DirectX), which uses 'fake' effects to generate interactive frame rates. Such methods require high quality fine tuning by artists and programers to camouflage these short-cuts[23].

In this thesis we first focus on the impact a BVH has on ray tracing by studying two different approaches. We then follow to investigate both scalar and vectorial approaches and their impact on the overall performance of ray tracing. Lastly we take a closer look at three methods of work distribution on multi-core systems and how scalable our interactive Ray Tracer (*iRT*) prototype is.

The single scalar algorithm is rather sequential where, after generating a list of rays (a queue), each individual ray is tested against all scene's triangles. If there is a hit, several colour values are calculated, often by shooting secondary rays, and later the pixel is given the final colour. If no hit is detected that ray does not contribute to the final result. After all rays are tested, the final image is displayed and the process is repeated.

With the primary objective to speed up ray tracing without deteriorating image quality, the use of a spatial partitioning algorithm is fundamental to organize triangles present in a given scene. By carefully gathering and storing primitives (triangles and vertices) in a specific way, reducing the number of ray-triangle intersections becomes a simple operation [8]. We have chosen the Bounding Volume Hierarchies (*BVH*) with axis aligned bounding boxes (*AABB*) as our accelerator data structure for the reason that *BVHs* are able to efficiently support dynamic scenes and can be used to efficiently ray trace large static models [25]. Usually, *BVHs* are a specific case of binary trees, where a node gathers information and points to its two children. In the bottom of a *BVH* tree we find leafs that contain the information relative to the actual primitives present in a each voxel.

To build an efficient *BVH* a number of steps must be done. The approach to build the *BVH* is top-down, i.e., for a given initial set of triangles, a root node is created gathering all triangles in a single bounding box. Using some division criteria, two children are created, gathering the first and the second set of triangles. This division goes on until a limit is reached. Then leafs are created containing the bounding box of the set of triangles, and pointers to the respective triangles in the scene.

After the *BVH* is built we can traverse it with rays where each ray will be tested against the root node. In case the ray misses the root's bounding box, the ray will not be tested and will be given a contribution value of 0. Otherwise, the ray is tested against the two children nodes' bounding boxes pointed by the root node. The ray will be tested against nodes recursively until it misses a given bounding box or intersects a leaf's bounding box. After a ray intersects a leaf, it is going to be tested against all triangles present in this leaf. Using this method significantly reduces the number of ray-triangle intersections, which contribute to a major improvement of the overall performance of a ray tracer [8, 25, 6].

In this thesis we compare two different approaches to the BVH algorithm and, accordingly to their behaviour, we chose the best implementation.

Coherence between rays is found by carefully studying each ray's origin and direction. Adjacent primary rays have the tendency to follow very similar paths in space, which, in many cases, hit exactly the same primitive. By making use of this property we gather adjacent primary rays in order to shoot them at the same time and speed up ray-triangle intersections. Likewise, shadow rays and other types of secondary rays tend to share the same coherence. This method allows a better performance in tracing primary rays as well in tracing secondary rays.

While the scalar algorithm is rather easy to understand, the vectorial algorithm uses much more recent technology allowing an internal level of parallelism. As opposed to the single ray shooting, this method allows to test not one, but several rays at once within the same thread. This is possible due to the vectorial SIMD instructions. The main idea is to group rays in packets of four rays and process each packet as a SIMD vector; the size of the packet (four) is determined by the width of the SIMD registers and might change in the future towards 8 and 16 values. Both algorithms follow the same line of logic, but in the packet of rays' method, a list (queue) of packets is generated, instead of the typical list of rays.

The multi-threading/multi-core level of parallelism makes all active processors read and write from the same shared memory address space. The workload distribution structure is stored in this shared memory allowing independent threads to retrieve new tasks [2]. In this context a task is a primary ray/packet that may or may not generate new secondary rays/packets, i.e, new tasks that corresponds to rays being shot at deeper levels of the ray tree. New tasks are stored back in the shared memory. Since all threads read and write on this shared memory, there is a need to control the different simultaneous accesses. For this reason, data access control mechanisms are required to preserve, at any given moment, the consistency of such data. These mechanisms may bring additional costs to a system's performance, which can compromise the interactive levels of the ray tracer. Consequently, it is fundamental to have a well designed and efficient control mechanism to maintain performance.

Besides the vectorial parallel instructions' study and comparison, in this thesis we also discuss and evaluate three different methods to control access to shared memory. The selected data structure is a FIFO-queue structure of sets of packets of rays to be processed by the active threads. Access to shared memory is typically controlled by mutual exclusion mechanism such as locks that guarantees that only one thread accesses a critical section at a time. By carefully re-ordering instructions, it is possible to drastically reduce contention and context switching costs, creating a lock-free environment [9]. We compare this lock-free data control access with the well-known lock-based approach and a conservative local technique where each thread has its own local work queue which prevents any kind of work sharing and avoids access control.

The first part of this work comes as research by comparing both *BVHs* implementations. After choosing the best option we follow to meticulously study both scalar and vectorial algorithms in several aspects concerning hardware, software and performance issues. Among many, this work will help on finding hot spots, i.e., critical places where most time is spent and help optimising both algorithms. The third part of this thesis is focused on multi-threading on multi-core systems and how the data access control mechanisms affect the overall performance of ray tracing.

This thesis is structured in nine different sections. It starts by introducing the context of the subject of study, followed by the state of art, which will point to the most recent researches made in the context of this thesis. We then explain the organisation and pipeline of our interactive ray tracer prototype. Later, the three key points will be addressed separately where a detailed description of the algorithms and results will be explained. This thesis ends with the conclusions of this research, what should the future work be and a section for acknowledgement.

Chapter 2

State of Art and Related Work

Many different interactive ray tracers have been developed throughout the last decade, each one focusing on achieving a specific goal such as accelerating ray-triangle intersections, adapting acceleration data structures to a specific architecture among many other attributes or, on the other hand, integrating several different up-to-date approaches to build a complete ray tracer in order to reach maximum performance. In this section we describe contemporary research and work in ray tracing. Our interactive ray tracer prototype focus on *BVHs* performance, parallelism present within a processor (vectorisation) and between processor cores in a multi-core ambient (multi-threading).

2.1 Accelerator Data Structures

Space partitioning and ordering is of fundamental importance to guarantee maximum performance by a ray tracer. The choice of one of these accelerator data structures (*ADS*) affects the general traversal performance as well as other algorithms to update or rebuild the *ADS* [28]. Among these, kd-tree, *BVH* and Grids are the most successful *ADS*.

A kd-tree is an axis aligned BSP tree with fixed bounding boxes. This has revealed to be a major problem for dynamic scenes where multiple scene transformations can be found. Kd-trees are known for being the fastest *ADS* for static scenes. The rebuild of this *ADS* is very costly and for that reason a number of algorithms were developed. The surface area heuristics *SAH* algorithm estimates the probability of a ray hitting the bounding box [28, 25]. This method was later adapted to *BVHs*, granting speed ups for this method [25].

Grids, on the other hand, partitions the space in equal and uniform voxels. With a simple method for construction and traverse, it becomes of very low efficiency if the majority of triangles are put together in the same voxel, forcing a ray to be tested against all triangles present in that voxel [27].

As explained in the first section, *BVHs* build voxels accordingly to triangles space position and not accordingly to space itself. This makes possible an efficient way to rebuild *BVHs* [28]. *BVHs* are also successfully used in GPUs by making use of modified versions of the SAH algorithm to achieve maximum efficiency [11].

2.2 Vectorial Ray tracing

Most of the previous work on ray packets showed that using SIMD instructions is of great advantage in terms of cache utilization and memory bandwidth. Such advantage exists for the simple fact that using these vectorial instructions reduces the number of memory accesses. The SIMD instructions explore the coherence between spatially adjacent primary and shadow rays [23, 24]. Taking advantage of ray coherence is fundamental to gather rays with similar properties and reduce the number of ray-triangles intersection tests, i.e., instead of each ray being tested against a triangle at a time, a set of rays is tested at the same time, allowing less memory accesses by requesting data only once per set of rays. First introduced by Ingo Wald, the results achieved by using this method through a kd-tree rapidly propagated the use of ray packetizing in the majority of current ray tracers in various architectures like RTRT, Manta, Razor and POV-ray [28, 1, 22, 17].

Reshetov et al.[20] proposed using larger packets of rays in his Multilevel Ray Traversal Algorithm. In his work, Reshetov excluded objects by applying a combination of frustum culling and interval arithmetic to reduce the number of traversal steps.

Following the above approaches, large ray packet algorithms were fully analysed for scene transversal and frustrum culling [18]. This resulted on a Whitted $16x16$ ray packet tracing system which is robust to degrading coherence; however this ray tracer did not reached real-time results. Also resulting from this study a new partition traversal algorithm was developed.

Peter Shirley took a different approach towards ray-object intersections. He tested the rays against the volumes of the triangles, instead of the typical 2D projection. Shirley optimized this method for single ray shooting, packets of rays with the same origin and general packet of rays. The implementation is also based on SIMD instructions [10].

Boulos et al. [4] decided to implement an alternative approach by grouping rays by its type. Also, by filtering out rays by their importance, more coherence between rays was found. The use of this method and ray tree attenuation granted gains comparing to their single ray implementation.

In [12] SIMD instructions are used to process packets of four rays on a Cell Processor. A problem related to secondary ray coherence is pointed out, thus only using packets for primary rays. It was measured that the majority of rendering time, around 75%, was spent rendering primary rays, while secondary rays rendering took around 25% of the total rendering time, even if the ray tree depth was set to 3. By using ray packets, a speed up of 25% was registered over the use of the single ray shooting method.

James Bigler et al. described Manta interactive ray tracer's architecture. In this article it is claimed that, among several key points, using SIMD instructions and maintaining ray coherence is fundamental for present and future ray tracing systems [1].

Recently, new developments in ray coherence have been made, where, in one case [13], secondary rays are reordered using different methods. The presented methods work the same way as ours: after a packet of primary rays is shot, the new generated secondary rays are stored in a list so new packets of coherent secondary rays can be created. Results in this article showed no improvement over the basic masking techniques of partially fill a new packet with secondary rays. The lack of improvement at the frame per second level is not enough to justify re-writing the ray tracer kernel, however this approach may become more interesting in the future with further research.

2.3 Multi-threading in Ray Tracing

Our interactive ray tracer has several different shared structures. These structures, on run time, are accessed to be read and written upon. The main issue here is the possibility of simultaneous accesses at the same memory address. This is a major problem in multi-threading and should be approached carefully. Considering figures 3.1 and 3.2, it is noted that from one single sequential process of BVH traversing, ray-triangle intersection and shading, our system changes to having several scalar threads of the same type. This parallelisation of rendering pipeline is supposed to bring major speed up performance to the process, where, instead of a single task being run, several tasks can be processed at the same time. But by doing this, a serious problem arises: shared structures may contribute to major errors by, for example, a thread accessing the data structure stored in a memory address that is being used by another thread, thus being in an incoherent state. To resolve this issue we first need to understand which structures are shared and which of them will suffer from simultaneous accesses from different threads.

The shared structures in our interactive ray tracer (iRT) are the scene, the work queue and the frame buffer used to store values of ray shading calculations. From the three structures, the scene is the only one that is not written upon during runtime. Threads only read information from triangles, so having the case of different threads accessing its information is not a concern. As for the frame buffer, this structure is an array, set initially to zero (black colour), where threads update the position of the respective packet of rays. This update is done by simply summing up the present value with the new calculated value by the shader, so there is a need to guarantee that each thread accesses the same buffer position at each time, because each thread has to read the present value, calculate a sum, and finally write the new value in the same buffer position. The working queue issue is also of high importance for it is the most important structure in our system. Access to it has to be robust and must guarantee a correct access by each thread to its correspondent share of work. The work queue is a set of tasks, where each task is a list of set of ray. Each task in the work queue, as stated in the last paragraph, is accessed during run time to add or retrieve tasks. This means the work queue has to be protected from simultaneous reads and writes.

Access control to shared data structures is usually performed by resorting to lock-based (or blocking) mechanisms, which ensure mutual exclusion within critical sections of the code. Access to shared data structures is serialised, resulting in high performance penalties when contention is significant. Since contention increases with the level of concurrency, typically lock-based approaches perform worst as the number of threads increases. Furthermore, locking often requires expensive context switches, which might be intolerable within interactive applications.

Alternatively, one can use lock-free synchronisation methods, which rely on atomic conditional primitives to control access to shared data structures [5, 9] (listing 7.2 presents the functionality of Compare-and-Swap, a well known atomic operation used throughout this work). These algorithms may either be non-blocking or wait free. Non blocking algorithms (lock-free and obstruction free) cannot guarantee termination in finite time because they are based on retries. Wait-free algorithms are guaranteed to finish on a finite number of steps, thus being immune to the priority inversion problem and eliminating deadlock and starvation.

Wait-free and lock-free access control mechanisms are seldom used within the graphics community, in spite of their increasing relevance due to the ever increasing number of cores on modern processors. In [19] the authors present a wait free mechanism to share the irradiance cache among multiple cores. They compare the achieved performance with those achieved with a lock-based approach and a local approach, where threads do not share locally computed irradiance values. This minimized synchronization and work replications overheads that contribute to a weaker performance. A wait-free algorithm has several proprieties that must be respected: all structures are lock free and must guarantee an upper bound of the number of instructions. Wait-free algorithms avoid starvation, deadlock and livelock. This wait-free approach granted a near linear speed up for up to to 24 cores, and authors concluded that the use of lock-based techniques is prohibited on highly concurrent shared memory systems.

Later in this work we will discuss and evaluate a lock-free approach to access a shared queue holding tasks for the rendering threads, and compare its performance with those achieved with a lock-based and local approaches. A wait-free approach was not possible to implement on the account that we cannot guarantee that all insert and remove functions will terminate in a finite amount of time due to the multiple access to data from different threads.

Chapter 3

iRT Description

This chapter is used to explain our interactive Ray Tracer's complete architecture and work pipeline. We first describe our sequential approach (scalar and vectorial versions), explaining the core of the renderer and later the changes that were made in order to allow multi-threading in our software (vectorial version with multiple threads).

3.1 iRT Single Threaded Architecture

iRT's vectorial version works sequentially by, after initiating the application, generating all primary rays, one ray per pixel. As mentioned in the first chapter, rays are packed in sets of four rays, by making use of SIMD instructions. These packets, on their turn, are placed in tasks of 256 elements in the WorkQueue as seen in figure 3.1. Such task size was found after experimenting values ranging from 1 to 1000 elements per task, in which the value of 256 elements per task returned the best results. In listing 3.1, the core of the work queue is shown and in listing 3.2 the methods used to access the work queue can be found. These are the methods which must be protected by making use of access control mechanisms (see section 3.2).

Listing 3.1: iRT - Work queue data structure

```

class PoolNode4 {
 public:
  RayPacket *rps;
  int count;
  PoolNode4 *next;
};
```

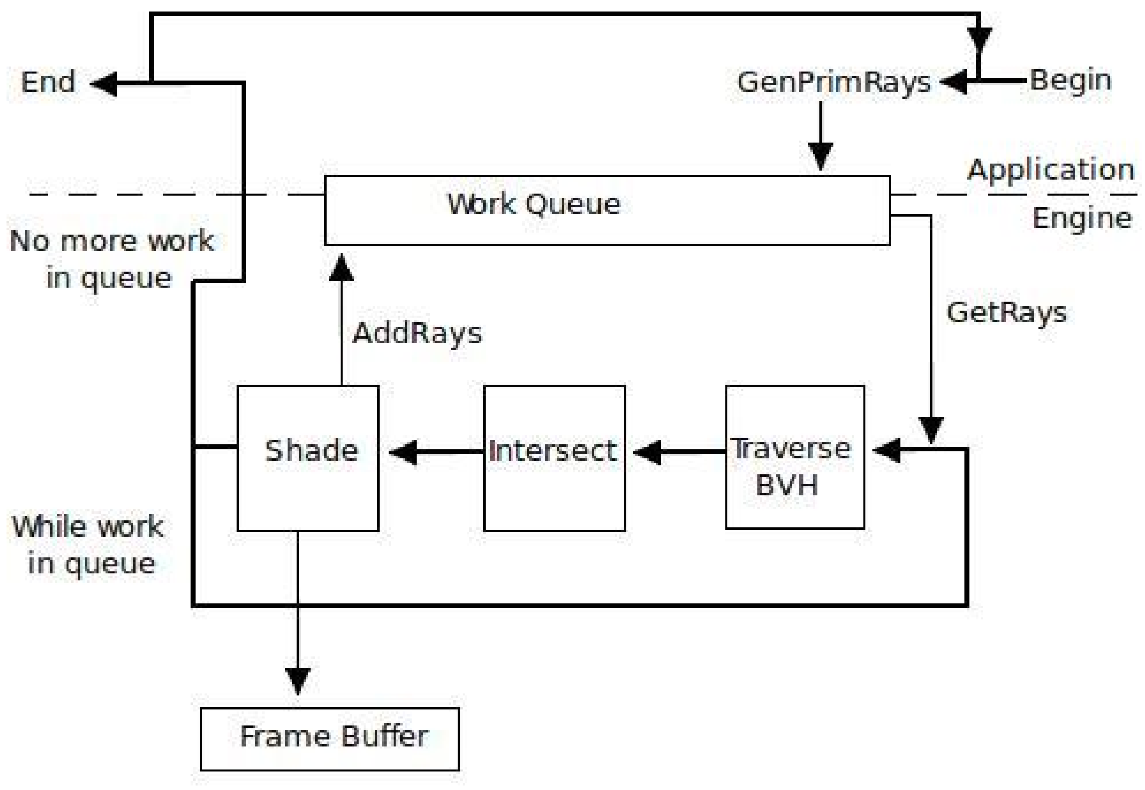


Figure 3.1: iRT sequential architecture

Listing 3.2: iRT - Work queue methods to add and retrieve work

```

// retrives a task from the work queue
// returns false if empty
int RayPacketPool::GetRayPacket (RayPacket **rp) {
  int nbr;
  PoolNode4 *actual;

  if (head->next == NULL) return false;  // pool is empty

  actual = head;
  head = head->next;
  *rp = head->rps;    // Linked list of Ray Packets to be passed
                      // as reference to the rendering structure
  nbr = head->count;
  delete actual;      // Deletes previous task

  return nbr;         // Returns the number of ray packets
                      // present in the linked list
}


// inserts a new task into the pool
// returns false if pool is full
bool RayPacketPool::AddRayPacket (RayPacket *rp, const int count) {

  PoolNode4 * node;
  node = new PoolNode4 (rp, count); // Adds a number of ray
                                    // packets to actual task

  tail->next = node;
  tail = node;

  return true;
}
```

After all tasks are placed in the working queue, the core engine of iRT takes over and the rendering process is started. Here, a task is retrieved from the WorkQueue, and all packets are processed at a time. Each packet runs through the pipeline by traversing the BVH in search of valid bounding box hits. Then, packets are intersected against all triangles returned by traversing the BVH tree. After that, the shader calculates the contribution values for that pixel by accessing the materials stored in the scene. While the shading process is running, new secondary packets of rays may be generated from the intersection points found. For this work, the shader is capable of generating shadow rays and specular rays (see figure 4.1). Transforming the traditional recursive ray tracing algorithm into this iterative one, requires that each ray carries with it information about which pixel it contributes to (this is equivalent to the ID of the parent primary ray) and also a weight factor that is equal to the product of the cosines and BRDFs at all intersection points along the current path. In the end of the shading function, the frame buffer is updated with the new calculated values. When the task is finally done, a new task is retrieved from the working queue, and the process repeats it self until no more tasks are present in the working queue. At this point the core rendering is done, and the application is then responsible to show the resulting frame and decide if a new frame is calculated or if the application is done.

iRT has to be linked with an application program that implements the application logic. Besides supporting all the application functionality, this application program is responsible for initializing iRT, loading the scene and initial tasks (eventually corresponding to primary packets of rays) into the WorkQueue.

Using this architecture, rendering is decoupled from the application logic. Although the ray tracing engine was developed essentially for image rendering, it can easily be used for different goals, such as collision detection. It is the application who decides which rays are shot and how are the respective results used. The manner how each ray's contribution is evaluated depends on the particular shader being used (line 8 of listing 3.3); different iRT engines can be built using different shading functions. The shader could actually be a dynamically loadable component - dispensing with building different engines for different shaders - but we decided to keep it statically linked in order not to hurt performance. Also note that the particular implementation of the WorkQueue is hidden behind the respective class interface. In the Multi-threading section, we will be using different work queues without changing the thread RenderLoop code.

3.2 iRT Multi-threaded Architecture

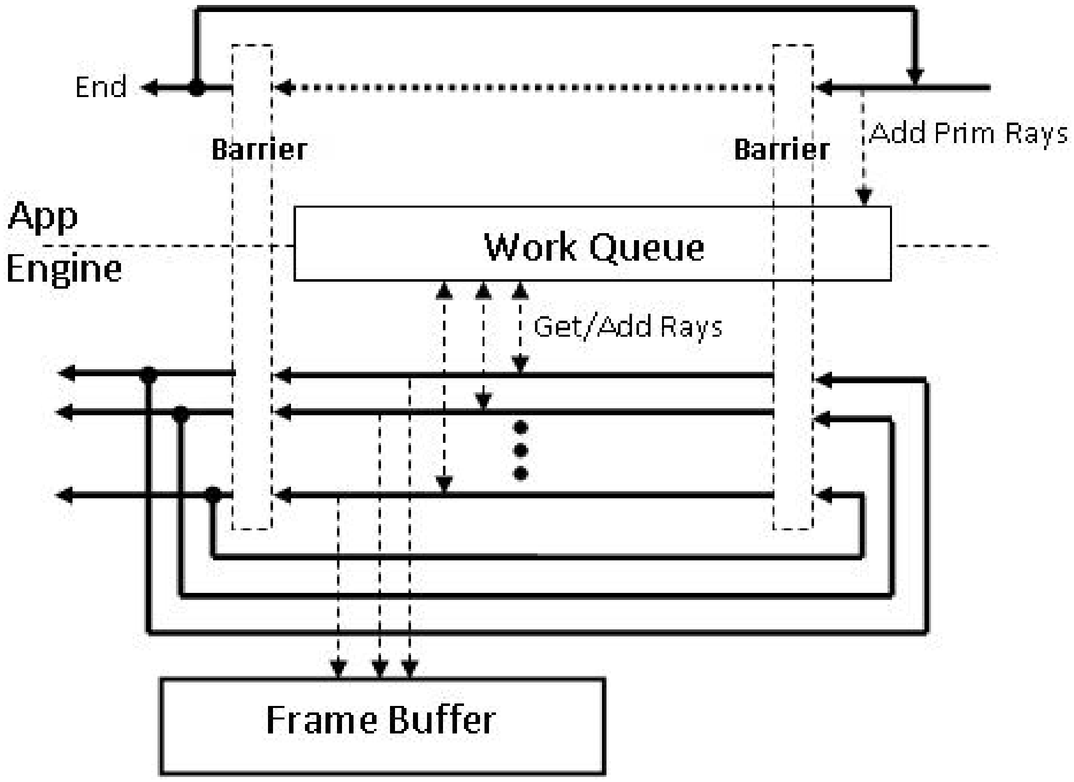


Figure 3.2: iRT multi-threaded architecture

By parallelising our interactive ray tracing engine (iRT) throughout multiple cores, we are able to run a set of symmetric rendering threads, which, as mentioned in the previous section, get their tasks from a global, shared work queue (see figure 3.2). The threads are symmetric in the sense that they all execute the same algorithm, which consists on retrieving tasks from the global queue and, for each packet in the task, traverse the 3D space (using a Bounding Volume Hierarchy), intersect the packet with candidate triangles and then shade the intersection point. Shading may result in shooting additional packets, which is achieved by adding new tasks to the work queue. Shading, specially in the case of shadow rays, may result in contributions to the frame buffer, which are concurrently added by each thread. When the queue is empty this means that the current frame has been rendered; each thread will then join a barrier and wait for further work or a terminate tag that closes the rendering engine (see listing 3.3).

Listing 3.3: iRT - rendering threads main loop

```
1  thread_RenderLoop(WorkQueue wl) {
2      while (!END) {
3          barrier();  // wait for a new frame to start
4          while (wl.getRays (&task) != EMPTY) {
5              for each ray in task {
6                  TraverseBVH ();
7                  Intersect ();
8                  Shade ();  // may add new rays to queue
9                  FrameBuffer.Update ();
10             }
11         }
12         barrier();  // wait for all threads to finish
13     }
14 }
```

Multi-threaded versions, as the sequential one, have to be linked with an application program that implements the application logic. This application program is also responsible for releasing the iRT rendering threads. This step is achieved by joining the barrier where the rendering threads are waiting (line 3 of listing 3.3 and line 7 of listing 3.4). The application program must then wait for the rendering threads to finish, which is achieved by joining the second barrier (line 12 and 9 of listings 3.3 and 3.4, respectively). This main loop is repeated until the application is terminated, in which case the END flag is raised, causing the rendering threads to finish.

Listing 3.4: iRT - application loop

```
1  main() {
2      iRT_init();
3      application_init ();
4      while (!finished) {
5          application_logic1 ();
6          Generate_PrimRays (); //write into the work queue
7          barrier ();  // release render threads
8          application_logic2 ();
9          barrier ();  // wait for frame to finish
10         FrameBuffer.Output ();
11     }
12     set END flag to finish threads
13     barrier();  // release threads and finish
14 }
```

The software architecture, described above and in the past section, implies that two data structures are shared among the application and all rendering threads: the work queue and the frame buffer. Accessing the former is discussed in the multi-threading section. The latter, which is where the rendering results are accumulated by the renderers and read back by the application thread, is protected by a user space spinlock [9], thus reducing context switches among threads. Since the results in the frame buffer are a linear combination of several rays' contributions, access to it could be wait-free among the rendering threads by resorting to hardware-supplied atomic floating-point add instructions (such instruction does not exist in the x86 architecture if one of the operands and the target are in memory).

iRT is a preliminary prototype and most of the development effort focused on the mechanisms used to share tasks among threads, whose results are reported in the Multi-thread section later in this thesis. One very important issue (performance-wise) has been handled in a much straight forward manner, which results in some performance penalties. This issue is the locality of memory accesses [20, 28] which contributes to the final performance of the ray tracer. The worse the locality of memory accesses is, the more is the number of accesses to memory, resulting in a large use of memory bandwidth, which is known to be very slow comparing to direct cache accesses by CPUs.

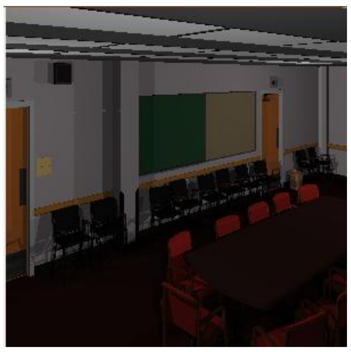
Chapter 4

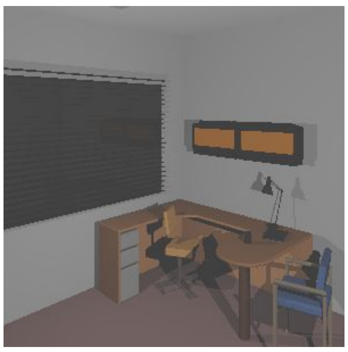
Experimental Methodology and Test Scenes

The equipment used to evaluate our software is a dual quad-core Intel Xeon E5420, 2.5GHz with 8GB of RAM. We used Intel's Vtune 9.1 for extensive profiling and Intel Compilers ICC and ICPC version 11.0.083 installed in a Linux x86 64 bits system. For performance measurements we compiled all of our iRT prototype versions with the option `-O2` for compiler optimisations. Since the core of our software is in the form of inline functions, testing the cache with Vtune does not return values for those functions. For this reason we compiled with `-O0` to test L1 and L2 Data Cache Miss Rates. Accordingly to Vtune's manual, a good L1 Data Cache Miss rate is less than 0.05 and a bad rate is above 0.3. A good value for L2 Data Cache Miss rate is set under 0.01 and a bad value is above 0.1.

The scenes used for testing our software were the conference room (190951 triangles), the office (20769 triangles) and the Stanford bunny with mirror (69463 triangles). Each scene has 4 point light sources and specular materials in order to produce shadow and specular rays. Figure 4.1 (a), (b) and (c) show the respective scenes. All tests are performed using a $300x300$ window. In every version the size of the tasks of rays/packets of rays is set to 256. The size of the window display contributes linearly to the speed up/down of a ray tracer's performance [24]. Besides that, ray tracing is known to be limited by memory bandwidth rather then processing power bandwidth [15]. For this reason it is fundamental to test cache accesses.

In all profiling tests presented, values are an average of 200 frames, while the viewpoint is fixed. We measure the following ray tracing key points:


(a) The conference room


(b) The office scene

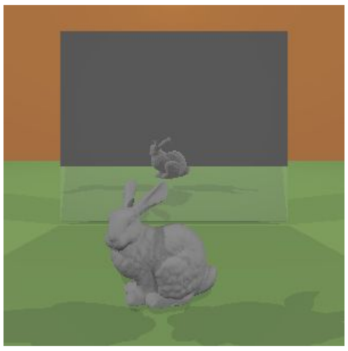

(c) The Stanford bunny

Figure 4.1: The scenes used for the experiments

- Time of Execution (T.Exec) – total time to render a frame;
- Rays per Second (RPS) – indicates how many rays can be traced per second, which is a value that can indicate how close we are from other interactive ray-tracers;
- Frames per second (FPS) – number of rendered frames per second iRT can achieve. The higher this value is, the smoother is the flow of images in the screen;
- Time distribution per function (%) – allow identification of hot spots, i.e., where time is being spent;
- L1 + L2 Data Cache Miss Rate – indicate where our software pays more penalties for accessing the respective Cache.

Profiles of the time distribution per function (%) are used because they allow identification of hot spots in the code, i.e., where time is being spent. For these profiles we decided to separate measured values. First we measure the values of the higher level of rendering:

- `GenPrimRays()` – generates primary rays;
- `RenderLoop()` – initiates the rendering functions;
- `Output()` – function that prints the image into the display.

Then we measure the values of the functions present in the `RenderLoop()` function in order to determine where in the core level are the hotspots present:

- `Traverse()` – function responsible for traversing the BVH;
- `Intersect()` – function that performs all intersections;
- `Shade()` – includes shading and secondary rays generation.

The illumination model used shoots one shadow ray per light source and one specular ray per intersection point, i.e., if the material has a specular reflection coefficient larger than 0. Only one primary ray per pixel is generated.

Chapter 5

Bounding Volumes Hierarchies

In this section we describe both *BVHs'* methods and implementations mentioned in the Introduction section. After explaining each of them, we analyse both construction and traversal performances. Finally we decide which is the best, at the present time, to integrate into our prototype for later optimisations: vectorisation and multi-threading.

5.1 A. Gonçalves's *BVH* Algorithm

Ademar's *BVH* construction is based on the *SAH* binning algorithm. As stated before, the *BVH* is constructed recursively until a leaf is created. The stoppage condition in Ademar's construction is the presence of exactly one triangle per leaf. His *BVH* tree structure is based on pointers (see listing 5.1).

The *SAH* binning algorithm used here is based on [21] where a min-max binning is used. This method allows to keep a record of the exact location of each *AABB*. The algorithm use to build the *BVH* tree consists of two steps: in the first place, the max-min operation is performed over the primitives and, later, the *SAH* are calculated by passing over the bounding boxes. Boxes boundaries are candidates to make plane splits, by making use of the min-max binning algorithm. Then, using the *SAH* calculation, the minimum value is chosen, determining where the *AABB* should be cut; creating, in this way, both child1 and child2 of the BVH_Node structure (refer to 5.1).

During run time, rays have to traverse along the *BVH*: each ray, at a time, is tested against a node. If the node is a leaf, and, at least one of the rays in the ray hits the leaf's bounding box, the triangle present inside the leaf is returned for later intersection. If the ray hits a node and not a leaf, the ray is tested against the node's bounding box. In case there is a hit, the ray is tested against both children. This continues recursively until a leaf is encountered or the ray misses the bounding box, that, in the last case, a empty set of triangles is returned.

Ademar Gonçalves has registered a speed up that varies between 31 and 122 comparing to not using any *ADS* [6] with a previous version of iRT, which did not had tasks. In that version, each ray was obtained directly from the WorkQueue to be processed through the pipeline.

Listing 5.1: iRT - Ademar's *BVH* data structure

```

 bvh_node {

  float bbox[24];                   // bounding box of node
  unsigned int tri_id;              // index of triangle
  unsigned int tri_id_inst;         // index of instance
  struct bvh_node *child1, *child2; // Pointers to children of node

  int axis0, axis1, axis2;          // Axis where cut is found
  int is_leaf;                      // Flag if node is a leaf
}BVH_Node;


// BVH
typedef struct bvh{

  Vector bmin;       // bounding box of the total scene
  Vector bmax;

  BVH_Node *root;    //first element of the BVH
}BVH;
```

5.2 I. Wald's *BVH* Algorithm

We used Ingo Wald's 2007 paper *"Ray Tracing Deformable Scenes Using Dynamic Bounding Volume Hierarchies"* to implement an alternative *BVH*. Wald's results showed that the build time for a scene the size of our conference room was 2.8 seconds and that for the same scene with animation the frame rate was of 1.5.

Both Ademar's and Wald's *BVH* construction is based on a modified *SAH* cost function, for the reason the *SAH* was first derived for kd-trees. In the *BVH* context, building a *BVH* top-down, each recursive construction step consists of partitioning a set of triangles in two. Each of the new subsets is recursively partitioned until a leaf can be created. Wald, in his paper, does not try to find the best *BVH* partitioning, but settles for a "*good*" one. By evaluating this modified version of the *SAH*, the partition is selected with minimal expected cost.

Listing 5.2: iRT - I. Wald's *BVH* data structure

```

struct BVHNode{
  float box_min[3];           // AABB minimum values
  union{
        int firstChildNodeID;  //for inner nodes
        int firstTriangleID;   //for leaf nodes
  };
  float box_max[3];           // AABB maximum values
  short num_triangles;        // flags inner node
  unsigned char axis;         //ordered traversal axis
  unsigned char first;        // first node to be traversal along axis
};
```

By comparing Wald's data structure (see listing 5.2) to Ademar's one, Wald was able to keep a 32 bit aligned structure, which grants a better memory access. Wald uses, instead of pointers, arrays of BVHNodes. Each node has the index to the location in the array of the respective children. To build the *BVH*, Wald uses a function that calculates the area of triangles sets' *AABB*. Then, using the *SAH* algorithm, the best two sets are chosen and these two sets will be the children of the present node. For sorting each set, we have decided to implement the very well known *Quicksort* algorithm.

As for *BVH* traversal, a very similar approach to the previous *BVH* was taken. Wald's *BVH* was focused on optimising the *BVH* traversal for large packets of rays by including Reshtov's frustrum traversal method. In this stage of our work we only developed a scalar version of Wald's *BVH*. Wald's traversal optimisations were not possible to implement in the scalar version, ending up with a very similar traversal function as the Ademar's one.

5.3 Results and Discussion

Three versions of our software were implemented and tested in order to verify which approach was the best. These three versions are the scalar one with no *ADS* (each ray is tested against all triangles), the scalar one with Ademar Gonçalves's *BVH* and, lastly, the scalar one with Ingo Wald's *BVH*.

For the first scalar version only four frames were rendered due to the exceptionally long rendering times. For the other two versions, measurements are based on an average of 200 frames.

In the next sections we will show the results obtained and discuss them inside the context of Interactive Ray Tracing.

5.4 BVH Build Time

Both Ademar's and Wald's *BVHs* did well, but by analysing the values returned by testing both versions, it is shown that Ademar's *BVH* version is slightly faster than Wald's. Furthermore, by looking at Table 5.1 we can see an enormous difference between both *BVHs* build time.

iRT	**Ademar's BVH**	**Wald's BVH**
Conference	1.109	9.28
Bunny	0.397	3.93
Office	0.096	1.01

Table 5.1: Time in seconds to construct each `BVH`.

5.4.1 T.Exec, Frames Per Second, Rays Per Second

iRT	**Scalar**			**Ademar's BVH**			**Wald's BVH**		
	T.Exec	FPS	RPS	T.Exec	FPS	RPS	T.Exec	FPS	RPS
Conf.	7,135	0.00158	687.64	1,068	0.18	130k	1,075	0.18	130k
Bunny	2,538	0.0015	611.13	128	1.56	404.3k	446	0.44	166.1k
Office	831	0.0048	2,107.79	276	0.72	589.2k	335	0.59	331.3k

Table 5.2: Time of Execution (T.Exec) in seconds, Frames per Second (FPS) and Rays per Second (RPS) of *iRT*'s versions: scalar with no *ADS*, scalar with Ademar's *BVH* and scalar with Wald's *BVH*. The three versions are tested with the three scenes (figure 4.1).)

By making use of counters we were able to retrive precise data for the three versions of *iRT*: scalar with no *BVH*, scalar with Ademar's *BVH* and scalar with our implementation of Wald's *BVH*. Table 5.2 shows, for the three versions tested against the three scenes, the total time of execution, frames per second and rays per second. This table clearly shows that the scalar version with no *ADS* has an extremely low rate of frames per second for all scenes. Both versions with *BVHs* completely outrun the first version. Ademar's *BVH*, for instance, renders the conference scene more than 100 times faster than the original one, and the Stanford bunny 1040 times faster. It is obvious that the effect an *ADS* has in rendering is fundamental in interactive ray tracing.

5.4.2 Time distribution per function

Tables 5.3 and 5.4 indicate respectively the percentage of time spent in the application and the percentage of time spent in rendering. The first clearly shows rendering takes more than 99% of the total running time of *iRT* in the three tested versions. In order to obtain performance gains, it is easy to see that optimisations should be done in the rendering core of *iRT*.

iRT	**Scalar**			**Ademar's BVH**			**Wald's BVH**		
	RL	GPR	OU	RL	GPR	OU	RL	GPR	OU
Conf.	99.98%	0.01%	0.01%	99.90%	0.06%	0.04%	99.91%	0.05%	0.04%
Bunny	99.98%	0.01%	0.01%	99.33%	0.38%	0.28%	99.76%	0.14%	0.1%
Office	99.98%	0.01%	0.01%	99.62%	0.21%	0.16%	99.74%	0.15%	0.11%

Table 5.3: Percentage of each application function consuming time for scalar, Ademar's *BVH* and Wald's *BVH iRT*'s versions. GPR - `GenPrimRays`, RL - `RenderLoop` OU - `Output`.

After performance measurements centred in the core of the application, we were able to understand what sections of our algorithm took more time to execute. Table 5.4 shows not only the percentage of time spent in each selected function but also the differences encountered at the core level between using an *ADS* and not using one. Since the scalar version of iRT has no *BVH*, it would not make sense to measure a *BVH* traversal function, for it does not exist in that context. This is why "N/A" is found in the table instead of a percentage value.

The scalar version with no *ADS* shows, as expected, an extremely high consuming time percentage for the intersection function. This is explained by the simple fact that each ray has to be tested against all rays present in the scene. In this case, $90,000$ rays (1 ray per pixel, $300x300$ pixels) had to be tested against scenes with 20k, 69k and 190k triangles. This obligates millions of ray-triangle intersection tests that most probably do not actually return a valid intersection.

By looking at the other two versions, we can clearly see that most work is spent in traversing the *BVH*, while intersecting is reduced to less than 5% in Ademar's *BVH* version and less than 6% in Wald's *BVH* version. In Ademar's version, the percentage of time the intersection function takes is relatively similar among the three scenes, something that does not occur in Wald's version. It is also noticeable

iRT	**Scalar**			**Ademar's BVH**			**Wald's BVH**		
	Int	Tra	S	Int	Tra	S	Int	Tra	S
Conf.	99.97%	N/A	0.01%	4.61%	92.18%	1.48%	5.60%	82.61%	1.54%
Bunny	99.99%	N/A	0.01%	4.17%	85.75%	4.14%	3.20%	64.88%	2.86%
Office	99.94%	N/A	0.03%	4.87%	86.93%	3.34%	4.91%	82.23%	3.34%

Table 5.4: Percentage of each render function consuming time for scalar, Ademar's *BVH* and Wald's *BVH iRT*'s versions, comparing only to `RenderLoop`'s total time. Int - `Intersect`, Tra - `Traverse`, S - `Shade`.

that Wald's traverse time distribution is less than Ademar's. Both facts are justified by memory accessing during run time (see next section), which slows down the rendering, causing, in the end, to Wald's version return lower FPS rate (see Table 5.2). In neither the versions a sum of 100% is achieve, which is justified by smaller work present in the rendering core such as adding new rays to the work queue outside the `shade` function (responsible for adding the majority of secondary rays).

5.4.3 L1 + L2 Data Cache Miss Rate

iRT				
Version	**Function**	**Scenes**		
		Bunny	Office	Conference
Scalar	Gen	0.005	0.005	0.005
	Int	0.005	0.006	0.005
	Tra	N/A	N/A	N/A
	S	0.033	0.023	0.023
Ademar's BVH	Gen	0.002	0.001	0.003
	Int	0.006	0.004	0.007
	Tra	0.009	0.012	0.025
	S	0.001	0.002	0.008
Wald's BVH	Gen	0.004	0.001	0.006
	Int	0.005	0.003	0.008
	Tra	0.055	0.038	0.047
	S	0.004	0.003	0.012

Table 5.5: L1 Data Cache Miss Rate per function for *iRT* scalar with no *BVH*, scalar with Ademar's *BVH* and scalar with Wald's *BVH* versions. Gen - `iRT General ratio`, Int - `Intersect`, Tra - `Traverse`, S - `Shade`

Vtune returned very different values for each version of *iRT* tested in this chapter. Firstly, L1 Data Cache miss rate measurements (table 5.5)in the original scalar version point to low values in the generation of primary rays and in the intersection functions. On the other hand, the shade function is much higher than the other two functions. This happens because of the fact that, for each ray, a number of memory accesses has to be made, in this case, to rays properties and scenes data. By vectorising the algorithm it is expected to see improvements in these values. In Ademar's and Wald's version the Traverse function present a much higher L1 data cache miss rates. Actually, the second presents L1 data cache miss rate values similar or higher than the limit for good rates indicated by Vtune's manual.

L2 data cache miss rate are generally very low and inside Vtune's good limits, except for the shader function in the original scalar version, where rendering the Conference scene, showed results more than 3 times higher than the 0.01 limit table (table 5.6).

High Cache Miss rates is known to contribute to a slow down of performance. With this in mind, and taking in consideration the results from the previous section, we can conclude that an *ADS* is of extreme importance to speed up rendering. Furthermore, since build time is essential in rendering dynamic scenes [28], we choose to use Ademar's *BVH* and advance the development of *iRT* to the next levels: vectorisation and multi-threading.

iRT				
Version	**Function**	**Scenes**		
		Bunny	Office	Conference
Scalar	Gen	0.000	0.000	0.004
	Int	0.000	0.000	0.004
	Tra	N/A	N/A	N/A
	S	0.001	0.000	0.037
Ademar's BVH	Gen	0.000	0.000	0.000
	Int	0.003	0.000	0.000
	Tra	0.003	0.001	0.001
	S	0.001	0.001	0.001
Wald's BVH	Gen	0.000	0.000	0.000
	Int	0.002	0.000	0.001
	Tra	0.001	0.001	0.001
	S	0.001	0.001	0.001

Table 5.6: L2 Data Cache Miss Rate per function for *iRT* scalar with no *BVH*, scalar with Ademar's *BVH* and scalar with Wald's *BVH* versions. Gen - `iRT General ratio`, Int - `Intersect`, Tra - `Traverse`, S - `Shade`

Chapter 6

Scalar and Vectorial Ray Tracing

In scalar ray tracing, after generating all rays and putting them in a work list, a frame is rendered. To render it, the work list is traversed to obtain tasks of rays. At a time, each ray in each task traverses the BVH to generate a small list of triangles. Each ray is tested against the 3D position of all triangles potentially intersected by it. After this, the respective pixel is shaded, and the next ray is tested. While shading, and as stated before, secondary rays may be created and added to the work list. When the work list is empty, the resulting frame buffer is processed later by the application logic. This process may be repeated again to generate the next frame, case the application logic decides that way.

Vectorial ray tracing method is much more complex than the scalar ray tracing, even though it works similarly to the scalar one: a work list is generated, traversed and in the end the frame is displayed, only to start all over again to create the next frame. The main difference resides in the fact that while in the vectorial version a number of rays are tested at a time, in the scalar one each ray is tested individually.

The use of SIMD instructions allows the computation of several values per instruction, instead of the typical single value and therefore we can reduce the rendering time. In our case, we decided to use a set of four rays since the SIMD instructions used apply to a set of four floating points. Regarding primary rays, each of these four rays are positioned at the matrix positions (x, y), $(x + 1, y)$, $(x, y + 1)$ and $(x + 1, y + 1)$, improving the coherence between the four rays. This combination of four primary rays, a packet of rays, is placed in a task, and later the task is placed in the work list. This means that instead of shooting one ray at a time, four rays are shot. This packet tracing method is only efficient when the contained rays are very coherent, i.e., all rays must traverse the same space regions. After one or more of these rays diverge, as it happens when creating packets of secondary rays, some of the rays will become inactive which can lead to a very poor efficiency, down to the scalar version performance.

Taking in consideration N as the SIMD width and n to be the actual useful work with active rays ($n < N$); so when rays are not coherent, $N - n$ remaining slots of the SIMD instruction perform wasted work [26]. For this reason, the ray packet version works very well with primary rays because they tend to be very coherent, while with secondary rays, coherence is of greater complexity.

The vectorisation of the original scalar version is not a direct approach. The majority of functions have to be vectorised (TraverseBVH, Intersect, Shade). All vectorisation implementation requires careful analysis, for example, while traversing the *BVH* with a single ray is of a straight forward implementation, traversing with a packet of rays has to be carefully studied. In scalar traversing, when the ray being tested fails an *AABB* intersection, the function automatically returns a void list of triangles; or if hits a leaf, the triangle contained in the *AABB* is returned. With a vectorised version of traverse, the packet of rays may or may not intersect several different *AABBs*. This makes conditional tests to occur repeatedly, which contributes to a slow down in performance. Another issue is that if all four rays intersect different leafs, the whole packet will be tested against the four triangles returned by the TraversBVH function.

The Intersection function is based on [3], allowing a faster implementation of this function. Only a few adaptations were needed in order to match our scene's data types.

Implementing the Shading function was also carefully done. As with the Traverse function, the shading function vectorisation was far from simple. Actually, there is almost no SIMD instructions present in the shading function to allow access to each ray's data separately. As an example, one ray, in the scalar version, may or may not generate a secondary ray, but in the vectorial version, four rays may or may not generate secondary rays. To resolve this issue, if at least one ray of the packet generated a secondary ray, a new packet had to be created with the new shadow/specular ray, while the other rays of the new packet were set to have a weight of zero, meaning they did not contribute to the final result. As explained before, this kind of packets (with invalid rays or with lack of coherent rays) contributes to a speed down of the ray tracer.

6.1 Results and Discussion

In the next sections we will show and discuss the tests made to the vectorial version obtained by vectorising the scalar version with Ademar's *BVH*. Literature [3] points to a speed up of more than 3 times, while another work points to only a two times speed up increase [24]. In the first section we show the values obtained by testing FPS, T.Exec., and RPS; and how much was gained by vectorising the scalar version. Then we look for hotspots, i.e., where the majority of time is spent and in the last section we compare the number of memory accesses of both versions.

6.1.1 T.Exec, Frames Per Second, Rays Per Second

Vectorisation did not guarantee the desired speed up, which was between 1.2 to 1.8 times faster than the scalar version. Table 6.1 shows that the FPS rate never hits the 2 times speed up factor. In the Conference scene, the vectorial version does not reach the 260 thousand rays per second. In the other two scenes *iRT* goes beyond the 700 thousand rays per second, but are still far from the 1 Million rays per second.

iRT	**Scalar**			**Vectorial**		
	T.Exec	FPS	RPS	T.Exec	FPS	RPS
Conf.	1,068	0.18	130k	571	0.35	259.3k
Bunny	128	1.56	404.3k	106	1.87	731.9k
Office	276	0.72	589.2k	166	1.2	726.7k

Table 6.1: Time of Execution (T.Exec), Frames per Second (FPS) and Rays per Second (RPS) of *iRT*'s versions: scalar with Ademar's *BVH* and vectorial with Ademar's *BVH*. Both versions are tested with the three scenes (figure 4.1).

Initial development results showed that, with simpler scenes, the vectorised version of *iRT* without *BVH* returned speed up values between 3.25 and 4.8 comparing to the scalar version equally without any *BVH* implementation. With this in mind, and looking the returned values by the vectorised version with *BVH*, an issue concerning the traversal of the *BVH* is pointed out. In the next section we further justify why the *BVH* traversal is the hotspot in our software.

6.1.2 Time Distribution per Function

In Tables 6.2, it is noted a small difference between time distribution in the scalar and the vectorial versions. Since the vectorial version produces more frames, it is natural to the `Output` function be called more often, causing its value to increase. The same happens with the `GenPrimRays` function. But the `RenderLoop` function continues to take more than 98% of the total time. The bottleneck can only be present at the core of our software.

iRT	**Scalar**			**Vectorial**		
	RL	GPR	OU	RL	GPR	OU
Conf.	99.90%	0.06%	0.04%	99.80%	0.13%	0.07%
Bunny	99.33%	0.38%	0.28%	98.80%	0.77%	0.43%
Office	99.62%	0.21%	0.16%	99.25%	0.50%	0.25%

Table 6.2: Percentage of each application function consuming time for scalar and vectorial *iRT*'s versions. GPR - `GenPrimRays`, RL - `RenderLoop` OU - `Output`.

iRT	**Scalar**			**Vectorial**		
	Int	Tra	S	Int	Tra	S
Conf.	4.61%	92.18%	1.48%	5.34%	91.04%	2.40%
Bunny	4.17%	85.75%	4.14%	5.36%	83.84%	7.14%
Office	4.87%	86.93%	3.34%	5.96%	83.65%	7.02%

Table 6.3: Percentage of each render function consuming time for scalar and vectorial *iRT*'s versions. Int - `Intersect`, Tra - `Traverse`, S - `Shade`.

By measuring the `RenderLoop` functions we were able to identify where most of the time is spent in rendering. Table 6.3 shows that the traversal of the BVH is still the most time consuming function, taking more than 83% of time while rendering the Stanford bunny and the office scenes, while in the conference scene takes more than 91% of the time. We can clearly point out that optimising the traverse function, a decent gain can be acquired. A graphical interpretation of the vectorial FPS values can be found at image 7.1.

6.1.3 L1 + L2 Data Cache Miss Rate

iRT's vectorial version L1 Data Cache Miss rate values are presented in table 6.4. By refering to Vtune's limit values, no function passes such limit. The overall value of L1 Data Cache Miss rate for each tested scene is low; the highest value measured belongs to the Conference scene test where a 0.004 rate is found. This value is far lower than the 0.05 limit present in Vtune's manual. As for the traversal function, it behaves the same way presented in the previous chapter. This function presents the highest ratio concerning L1 Data Cache misses.

iRT	**Vectorial**			
	Gen	Int	Tra	S
Conference	0.004	0.004	0.020	0.003
Bunny	0.003	0.005	0.014	0.001
Office	0.002	0.003	0.009	0.001

Table 6.4: iRT's vectorial version - L1 Data Cache Miss Rate per function. Gen - `iRT's ratio`, Int - `Intersect`, Tra - `Traverse`, S - `Shade`

iRT	**Vectorial**			
	Gen	Int	Tra	S
Conference	0.000	0.000	0.001	0.001
Bunny	0.001	0.003	0.005	0.001
Office	0.000	0.000	0.001	0.000

Table 6.5: iRT's vectorial version - L2 Data Cache Miss Rate per function. Gen - `iRT's general ratio`, Int - `Intersect`, Tra - `Traverse`, S - `Shade`

L2 Data Cache miss rates shown in table 6.5 indicate that a very low miss ratio is present in any of the test made. Again, the highest value for L2 Cache miss rate is found on the *BVH* traverse function.

L1 and L2 Data Cache miss rates present in tables 6.4 and 6.5 show similar ratios found in the scalar version with Ademar's *BVH* (tables 5.5 and 5.6). This clearly points the *BVH* traverse function as the biggest issue in *iRT*.

Chapter 7

Multi-threading Algorithms

The ray tracing algorithm has been proved to be highly parallelisable. By making use of, not one, but several cores, it is possible to speed up the rendering process almost by the factor of number of cores [23]. By parallelising the vectorial algorithm, the final frame rate value should multiply by that factor, meaning the flow of frames rendered is higher and the final image result is smoother. With this in mind, we use the mentioned three parallel versions in order to achieve such speed up.

The three different data access control mechanisms are presented in the next sections. The lock-based algorithm is referred to as LOCK, the local one as LOCAL and the lock-free as LFREE. As stated in this thesis, it is of most importance to protect data structures shared by different threads. In our software, there is the need to protect the work queue which contains all tasks of rays and the frame buffer, where the values of ray-triangle intersections are stored. The multi-threaded version of iRT is built using POSIX threads, and each of the three different mentioned versions uses different methods to protect such shared data structures. All three versions use a spinlock to protect the access to the film buffer. To access the work queue each version has a different approach to it. LOCK version makes use of the Pthreads mutexes to lock/unlock the queue. The LFREE version, instead of mutexes, uses a Compare-and-Swap atomic operation that allows the access to the queue without resorting to mutexes. It is expected that the LFREE version returns better results than the LOCK one. Finally, the LOCAL version resorts to, not one work queue, but to private work queues. Each thread is assigned a specific work queue; each thread later deals with the generated secondary packets of rays by its own. This means the possibility of work imbalance to be created is added to the rendering process.

Later in this chapter we show the obtained results, by comparing the LOCK, LFREE and LOCAL version with each others and with the original vectorial version, in order to understand how scalable each solution is and what is the performance of each algorithm.

7.1 Lock-Based Queue

The traditional LOCK implementation uses a single lock to protect all accesses to the shared queue. This results in serialising all calls to `getRays()` and `addRays()`, that is, reads and writes may interfere among them. However, reads and writes operate on different ends of the queue, so they must be able to proceed without interference if the queue is whether full or empty. We used the Unbounded Total queue algorithm described in [9]: the queue is a linked list of tasks and different locks are used for reads and writes, thus reducing contention. A sentinel node, whose next field is NULL, is initially inserted. Readers always check whether this is the head node of the queue; if it is, then the queue is empty (see listing 7.1).

Listing 7.1: Lock-based queue

```
addRays (RayType *ray) {
    addlock.lock();
    QueueNode *node = new QueueNode (ray);
    tail->next = node;
    tail = node;
    addlock.unlock();
}

getRays (rayType **ray) {
    getlock.lock();
    if (head->next==NULL) {
        getlock.unlock();
        return EMPTY;
    }
    QueueNode *actual = head;
    head = head->next;
    *ray = actual->value;
    getlock.unlock();
    delete actual;
    return OK;
}
```

7.2 Local Queue

The reasoning behind the LOCAL approach is that each thread maintains its own work queue. The application program rather than writing all primary packets of rays to a single queue, distributes the tasks among all work queues in a round-robin fashion to ensure better load balancing. Each rendering thread then processes its initially assigned packets and adds secondary packets of rays to its own queue. Apart from the scene data and the frame buffer, this approach has no data sharing, thus it requires no data access control mechanism to access the work queue; the work queues implementation is similar to that shown in listing 7.1, but without the locks. Since there is no contention or serialisation of accesses this approach has the potential to outperform the other two if a balanced load distribution is guaranteed. Note that load distribution is statically done by the application program; the round robin distribution of work and the fine granularity of tasks (i.e., the number of packets associated with each of the queue's node) assure a reasonable load distribution for many images and for shallow ray trees, such as those typically found in interactive ray tracing contexts.

The only issue that can be encountered in this approach is that the shader may cause performance to speed down, due to the possibility of having bad load balance when creating new secondary packets of rays. This weak load balance may cause an overload of work in some local work queues, meaning some threads will be working while others will have to wait until all local queues are done. In case of heavy imbalance, this wait time will bring down the overall performance and cause the renderer to be far from optimal.

7.3 Lock-Free Queue

The lock-free algorithm [9, 14] does not rely on locks to guarantee mutual exclusion. It relies on the Compare-and-Swap atomic synchronisation primitive described in listing 7.2 and on retries, i.e., it contains a loop (this is the reason why it is not a wait-free method: it is not guaranteed to finish in a finite number of steps). Additionally, the `addRays()` method is lazy, meaning that insertion of new nodes happens in two different steps; in particular, threads may need to help one another in order to advance tail (see listing 7.3).

The `addRays()` method creates a new node (line 2), reads tail and finds the node that appears to be last (lines 4 and 5). It then checks whether that node is still last (line 6) and whether the node has a successor (line 7). If the node does have a successor then it was inserted by other thread; this thread will help the others by trying to advance tail to the next node, but only if tail is still equal to last (line 13) - it will then try again to insert the new node. If, however, the node still does not

Listing 7.2: Compare and swap

```
atomic CAS(addr location, val cmpVal, val newVal)
{
    if (*location == cmpVal)
    {
        *location = newVal;
        return true;
    }
    else return false;
}
```

have a successor (lines 7 – 12), then it performs a trial to append it to the queue (line 8). If it succeeds, it tries to update tail to the new node (line 9); this CAS operation may fail, but this does not represent a problem, since it will only fail if tail has already been advanced by other thread. If, however, appending the node to the queue failed (line 8), then the thread will try again (this CAS operation may fail because some other thread might have appended other node).

The `getRays()` method is very similar to its lock-based homonym. It will check if the queue is empty by verifying if the successor of head is NULL (line 26); if the queue is non empty, then it will try to advance head to its successor and return the previous head node (lines 31 – 35). There is however a subtlety in this lock-free algorithm: before advancing head the algorithm has to make sure that tail is not left referring to the sentinel node that is about to be removed from the queue (this may happen because some thread may have added a node to the queue but was not able to update tail). Thus if head equals tail (line 25) but the head successor is not NULL (line 26), then the queue can not be empty; the thread will try to advance tail to the sentinel's node successor (line 29) and will iterate again.

Listing 7.3: Lock-free queue

```
 1 addRays (RayType *ray) {
 2     QueueNode *node = new QueueNode (ray);
 3     while (true) {
 4         QueueNode *last = tail;
 5         QueueNode *next = last->next;
 6         if (last==tail) {
 7             if (next==NULL) {
 8                 if (CAS(last->next, next, node)) {
 9                     CAS (tail, last, node);
10                     return ;
11                 }
12             } else {
13                 CAS (tail, last, next);
14             }
15         }
16     }
17 }
```

```

getRays (rayType **ray) {
    while (true) {
        QueueNode *first = head;
        QueueNode *last = tail;
        QueueNode *next = first->next;
        if (first==head) {
            if (first==last) {
                if (next==NULL) {
                    return EMPTY;
                }
                CAS (tail, last, next);
            } else {
                *ray = next->value;
                if (CAS (head, first, next)) {
                    delete first;
                    return OK;
                }
            }
        }
    }
}
```

7.4 Results and Discussion

For the multi-threading profiling and performance verification obtained by parallelising our software across several cores, we choose to measure the frame rate (fps), the amount of time it would take to get or add a new task and how long threads have to wait in a barrier for all other threads to finish. Lastly, we compare what is the speed up obtained by comparing the three parallel versions' frames rate with the vectorised single threaded version presented in the previous chapter. These values indicate, respectively, how fast new frames are produced, which contributes to a smoother image refresh rate (as stated before) and to detect a possible bottleneck while retrieving or adding new tasks to the work queue. Measuring the speed up between one parallel version and the vectorial one can indicate how scalable our parallel version. Ideally, the present number of active rendering threads/cores should multiply the speed up by that factor[23]. We also measure the time distribution per function in order to detect which of the functions of *iRT* takes more time, and lastly we test the L1 and L2 Data Cache Miss Rate in order to detect possible bottlenecks associated to bad memory access, which contribute to a reduction of performance.

The above refered tests were made to the three multi-threaded versions: LOCK, LFREE and LOCAL running with 2, 4 and 8 threads.

7.4.1 T.Exec, Frames Per Second, Rays Per Second

In table 7.1, it is shown the time of execution, frame rate and rays per second for the LOCK, LFREE and LOCAL versions for the 2-threaded, 4-threaded and 8-threaded tests respectively. A graphical observation of these values is found in figure 7.1 (a), (b) and (c). Table 7.2 shows the vectorial single threaded version for easier reference and comparison to the multi-threaded versions.

In the 2-threaded rows, we can see the LFREE version is the weakest version, running at lower frames per second than the other two in all scenes. The LOCK and LOCAL versions show very similar results, as the frame rate is very similar across the three scenes. Comparing the best results in this table (LOCK version) to the results obtained in table 6.1, speed ups of 2, 1.7 and 1.975 are found for the Stanford bunny scene, the conference scene and the office scene, respectively. This values points to an almost perfect scalability of our implementation for 2 threads. In the LOCK version we were able to achieve 1.5 million rays per second with the Stanford bunny. The Conference scene has a very low value for RPS. This is due to a not optimised traverse function.

iRT	**T**	**LOCK**			**LFREE**			**LOCAL**		
		T.Exec	FPS	RPS	T.Exec	FPS	RPS	T.Exec	FPS	RPS
C	2	330	0.6	444.3k	396	0.5	370.3k	328	0.6	444.3k
	4	212	0.94	696.2k	197	1.01	747.9k	192	1.04	770.3k
	8	124	1.6	1,184.5k	133	1.49	1.103.4k	108	1.83	1,355k
B	2	51	3.9	1,526.4k	55.7	3.59	1,186.9k	55.9	3.57	1,397.3k
	4	32.58	6.13	2,399.2k	29.27	6.83	2,673.2k	30.5	6.53	2,555.8k
	8	16.85	11.87	4,645.9k	16.62	12.03	4,708.5k	16.16	12.37	4,804.4k
O	2	84	2.37	1,435.2k	97.98	2.04	1,235.5k	85.3	2.34	1,417.2k
	4	52.9	3.78	2,289.2k	48.19	4.15	2,513.2k	47.64	4.19	2,537.6k
	8	26.65	7.5	4,542.2k	26.39	7.58	4,590.5k	23.45	8.53	5,165.6k

Table 7.1: Time of Execution (T.Exec), Frames per Second (FPS) and Rays per Second (RPS) of *iRT*'s versions: LOCK, LFREE and LOCAL for 2, 4 and 8 threads. In the bottom of the table, vectorial original version's values are shown for easier reference. All versions are tested with the three scenes (figure 4.1). C - Conference scene, B - Stanford Bunny scene, O - Office scene, T - number of threads.

Rows for 4 threads, show slightly different results encountered in the last paragraph. Here, the LFREE version returns very similar results to the LOCAL version, while the LOCK version has the worst performance of the three versions. With 4 threads we achieved more than 2.5 Million rays per second with the LFREE and the LOCAL versions with the Office and Stanford bunny scenes. As for the Conference,

iRT	**Vectorial**		
	T.Exec	FPS	RPS
C	571	0.35	259.3k
B	106	1.87	731.9k
O	166	1.2	726.7k

Table 7.2: Time of Execution (T.Exec), Frames per Second (FPS) and Rays per Second (RPS) of *iRT*'s vectorial version. All versions are tested with the three scenes (figure 4.1). C - Conference scene, B - Stanford Bunny scene, O - Office scene.

it is found that RPS is still very low. Comparing to the original vectorised version measurements, we found average speed ups of 3 for the LOCK version, 3.33 for the LFREE version and 3.31 for the LOCAL one.

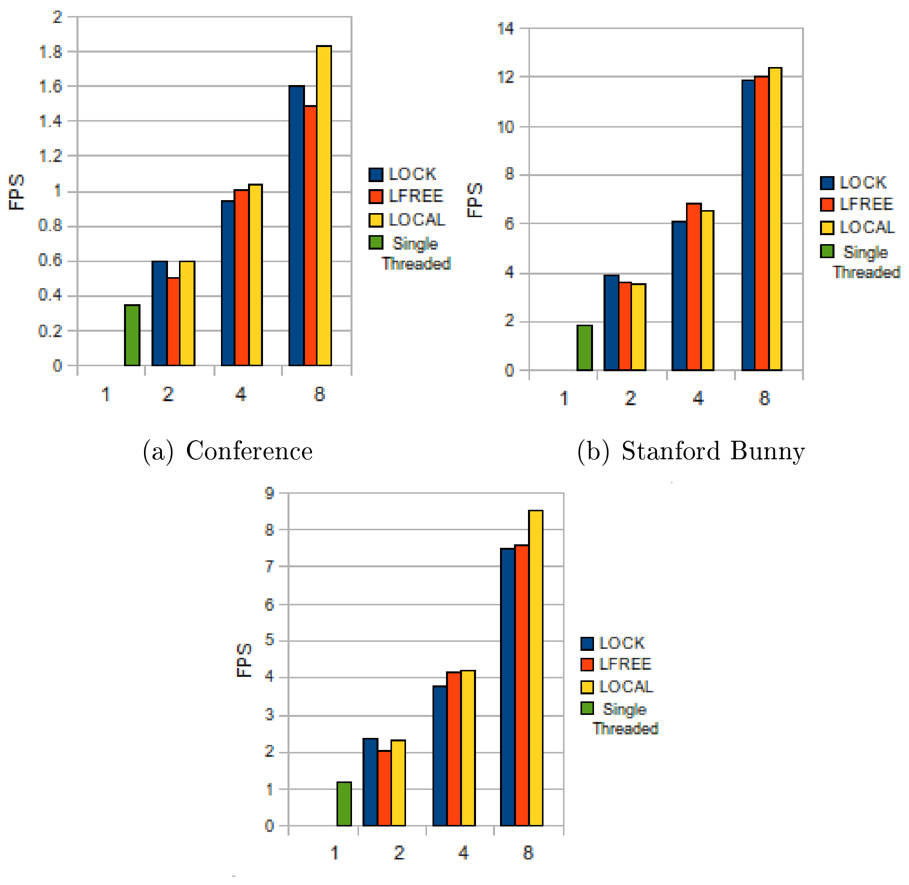
Comparing the T.Exec, FPS and RPS of the LOCK, LFREE and LOCAL versions of *iRT* running with 8 threads, we measured on the LOCAL version and office scene the maximum achieved value concerning RPS with a value of more than 5.165 million rays per second. Also, running with 8 threads we were able to achieve 12.37 frames per second on the LOCAL version with the Stanford Bunny. LOCK and LFREE versions had very similar behaviours while the LOCAL version returned the best results for all scenes. Concerning scalability, comparing to the original vectorial version, the 8-threaded version registered the following speed up values: 5.7 for the LOCK approach, 5.66 for the LFREE approach and 6.3 for the LOCAL approach.

Results from these three tables lead us to conclude there is still margin for optimisations. Ideally, 8-threaded *iRT* should return a speed up of 8, but only an average value of 6.3 was returned.

7.4.2 Time distribution per function

Table 7.3 shows the percentage of time each function takes to run. In each version (LOCK, LFREE, LOCAL) the first and last functions (`GenPrimRays` and `Output`) correspond to the application logic while the `RenderLoop` calls the rendering core; and the three last functions (`Intersect`, `BVHtraversal`, `Shade`) correspond to the core engine and are called by the `RenderLoop` function.

Carefully comparing the values measured, it is noted that percentage values correspond exactly with the behaviour analysed in the previous subsection. For example, the more the `BVHTraversal` functions take time, the less the Frame per Second



(a) Conference

(b) Stanford Bunny

(c) Office

Figure 7.1: Frames per Second for each scene tested with LOCK, LFREE and LOCAL versions, running with 2, 4 and 8 threads, compared with the respective vectorial version. Y-axis - Frames Per second, X-axis - Number of threads.

rate is. The more FPS counted in a test, the higher the values of `GenPrimRays` and `Output` are. This fact has already been described in chapter 6. The `traverse` function is where our bottleneck is present, and thus optimising this function will contribute to better overall results.

In the Conference scene, every single test registered the `RenderLoop` function as always taking more than 99% of the executing time and `BVHtraversal` taking more than 88%. As for the Office and Stanford Bunny, such values are not as high as in the Conference scene, but take as well the majority of rendering time.

7.4.3 L1 + L2 Data Cache Miss Rate

Taking in consideration the L1 Data Cache Miss Rate limits presented in Chapter 4, table 7.4 shows that the minimum limit of 0.05 was never passed, meaning there

iRT	**Function**		**Scenes**		
			Bunny	Office	Conference
LOCK	GenPrimRays		2.89%	2.14%	0.49%
	RenderLoop	Intersect	6.73%	6.80%	6.52%
		Traverse	81.30%	84.08%	90.83%
		Shade	6.44%	5.04%	1.82%
	Output		2.62%	1.94%	0.34%
LFREE	GenPrimRays		2.83%	2.25%	0.39%
	RenderLoop	Intersect	5.95%	6.91%	7.99%
		Traverse	79.60%	83.80%	88.52%
		Shade	7.69%	5.13%	1.91%
	Output		2.62%	1.91%	0.34%
LOCAL	GenPrimRays		3.33%	2.89%	0.41%
	RenderLoop	Intersect	5.61%	5.69%	5.51%
		Traverse	78.65%	78.39%	88.18%
		Shade	8.23%	9.15%	4.19%
	Output		2.73%	2.32%	0.40%

Table 7.3: Percentage of each function consuming time for *iRT* LOCK, LFREE and LOCAL versions running with 8 threads.

is little to no L1 Data Cache Misses. Taking a closer look at each function, some patterns emerge. It is very clear the `traverse` function registers, by far, most of the L1 Data Cache Misses during execution. Also, this can be seen as the complexity of the scene increases, i.e., the more time spent in traversing the *BVH*, the higher the L1 data cache miss value is. Regarding the general L1 data cache miss ratio, it was measured a small value varying between 0.002 and 0.005, which are good values.

As for the L2 Data Cache Miss Rate in *iRT*, the general values quantified were far below the 0.01 value suggesting our renderer has a good L2 memory access. But taking a closer look at each function performance, there can be seen that the maximum limit for a good L2 cache miss rate is passed by the `traverse` function. In table 7.5, the `traverse` function registers a value of 0.011 in the LFREE 2-threaded version while rendering the Stanford Bunny. The L2 data cache miss rate limit is again passed in the 4-threaded version of LOCK and LFREE, for the same scene.

From a careful analysis of the past three section, we choose the LOCAL 8-threaded version as the best version of *iRT* for the reason that it achieved the best frame rates and rays per second in all three scenes. It also presents low values for both L1 and L2 data cache miss rates. The LOCAL version outperformed the LFREE version thanks to a small ray tree depth and good load balancing between queues. A problem arises in the case of a new shader implementation: an increase in the ray tree depth may be introduced, i.e., each ray generates even more secondary rays originating in a very different number of tasks for each local queue. By generating imbalanced work queues, the LOCAL version will most probably loose performance and be outperformed by the LFREE version.

7.5 Scalar Multi-Threaded Experiments

In previous work [16], a similar comparison was done by us in order to evaluate these three algorithms (LOCK, LFREE and LOCAL) in a scalar rendering algorithm. By using Ademar's *BVH* but no vectorisation, obtained results were similar to the ones presented in this chapter. Actually, comparing the results obtained in that work and in this thesis we can see that the vectorial threaded version's FPS rate is only slightly better than the ones presented in [16]. Also, in [16] we tested *iRT* scalar threaded version in two other machines: a 8-core (plus Hyper-Threading) Xeon 5500 server and a 24-core Xeon 7400 server. Maximum scalability was not reached, having its maximum speed up value set at 11, when testing the Office scene with 24 cores. Results are shown throughout figures 7.2, 7.3 and 7.4.

Taking the LOCAL approach in consideration and by comparing the FPS values found in this thesis with the values presented in 7.2, we can see that the vectorial version is only faster by a fraction of a frame (0.1 FPS faster), with 8 threads rendering the Conference scene. In the Stanford Bunny scene, also with 8 threads, the vectorial version achieve 12.37 FPS while the scalar multi-threaded version achieved 11 FPS. Tests of the Office scene returned a 8.53 FPS value in the vectorised version, slightly faster than the 6.4 FPS found in 7.2 (b).

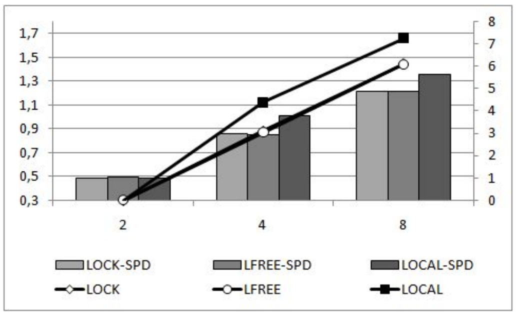
By comparing these results, and only being able to compare performance data from one machine (two quad-core Intel Xeon 5400), we can conclude the vectorised multi-threaded version is not as good as expected has the speed-up found was between this one and the scalar one is less than two. We suppose that, if able to test the vectorised version in the other two machines, similar results were to be found.

iRT					
Version	**Threads**	**Function**	**Scenes**		
			Bunny	Office	Conference
LOCK	2	Gen	0.003	0.005	0.004
		Tra	0.015	0.008	0.020
		Int	0.006	0.003	0.004
		S	0.002	0.002	0.004
	4	Gen	0.003	0.004	0.005
		Tra	0.017	0.017	0.023
		Int	0.005	0.004	0.004
		S	0.002	0.002	0.004
	8	Gen	0.003	0.003	0.005
		Int	0.006	0.002	0.005
		Tra	0.019	0.013	0.024
		S	0.002	0.002	0.004
L-FREE	2	Gen	0.003	0.002	0.004
		Tra	0.002	0.011	0.023
		Int	0.002	0.003	0.005
		S	0.005	0.002	0.007
	4	Gen	0.003	0.003	0.004
		Tra	0.018	0.013	0.023
		Int	0.006	0.003	0.005
		S	0.002	0.002	0.004
	8	Gen	0.004	0.002	0.004
		Tra	0.019	0.013	0.023
		Int	0.005	0.004	0.004
		S	0.002	0.002	0.004
LOCAL	2	Gen	0.003	0.005	0.005
		Tra	0.015	0.008	0.022
		Int	0.005	0.003	0.005
		S	0.003	0.002	0.004
	4	Gen	0.003	0.002	0.004
		Tra	0.016	0.010	0.024
		Int	0.005	0.003	0.005
		S	0.002	0.002	0.004
	8	Gen	0.003	0.002	0.004
		Tra	0.019	0.009	0.025
		Int	0.006	0.003	0.004
		S	0.002	0.001	0.004

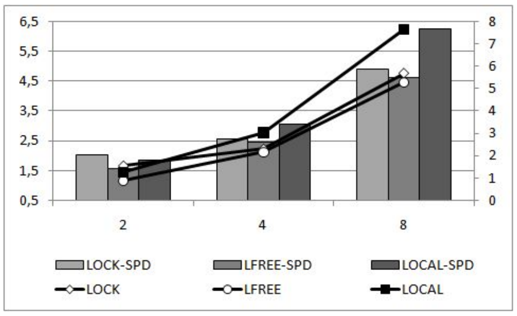
Table 7.4: L1 Data Cache Miss Rate per function for *iRT* LOCK, LFREE and LOCAL versions running with 2, 4 and 8 threads. Gen - `iRT General ratio`, Int - `Intersect`, Tra - `Traverse`, S - `Shade`

iRT					
Version	**Threads**	**Function**	**Scenes**		
			Bunny	Office	Conference
LOCK	2	Gen	0.002	0.004	0.001
		Tra	0.006	0.001	0.001
		Int	0.003	0.000	0.000
		S	0.002	0.002	0.001
	4	Gen	0.001	0.002	0.001
		Tra	0.010	0.010	0.004
		Int	0.004	0.000	0.000
		S	0.002	0.002	0.002
	8	Gen	0.001	0.001	0.000
		Tra	0.009	0.006	0.004
		Int	0.003	0.000	0.000
		S	0.002	0.002	0.002
LFREE	2	Gen	0.001	0.001	0.001
		Tra	0.009	0.005	0.004
		Int	0.003	0.000	0.000
		S	0.002	0.001	0.006
	4	Gen	0.001	0.001	0.000
		Tra	0.010	0.007	0.004
		Int	0.004	0.000	0.001
		S	0.002	0.002	0.001
	8	Gen	0.001	0.001	0.000
		Tra	0.011	0.007	0.005
		Int	0.003	0.000	0.000
		S	0.002	0.002	0.001
LOCAL	2	Gen	0.001	0.004	0.001
		Tra	0.006	0.001	0.001
		Int	0.003	0.000	0.000
		S	0.002	0.001	0.001
	4	Gen	0.001	0.000	0.000
		Tra	0.006	0.001	0.001
		Int	0.004	0.000	0.000
		S	0.001	0.001	0.001
	8	Gen	0.001	0.000	0.000
		Tra	0.009	0.001	0.001
		Int	0.004	0.000	0.000
		S	0.002	0.001	0.000

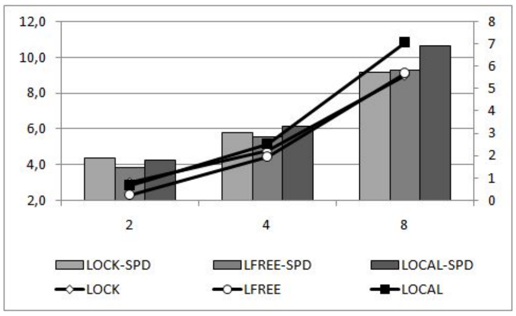
Table 7.5: L2 Data Cache Miss Rate per functionfor *iRT* LOCK, LFREE and LOCAL versions running with 2, 4 and 8 threads. Gen - `iRT General ratio`, Int - `Intersect`, Tra - `Traverse`, S - `Shade`



(a) Conference

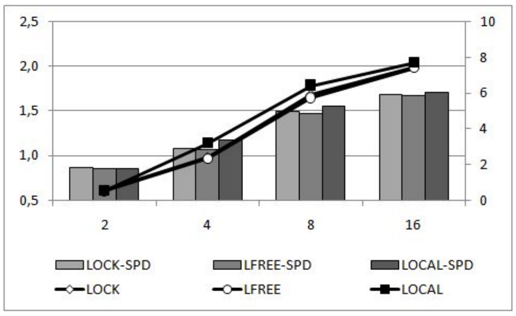


(b) Office

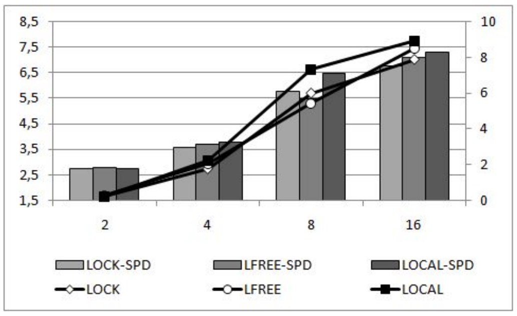


(c) Bunny

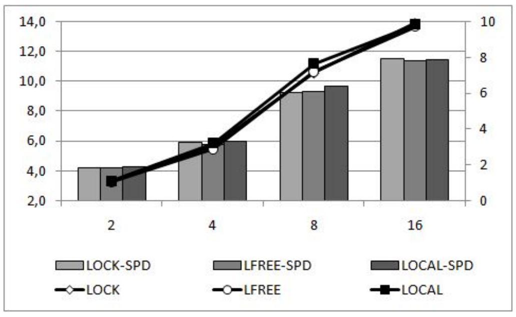
Figure 7.2: Results for the 8-core Xeon 5400 server. Axis: left- fps, right- speed-up, horizontal- number of threads.



(a) Conference

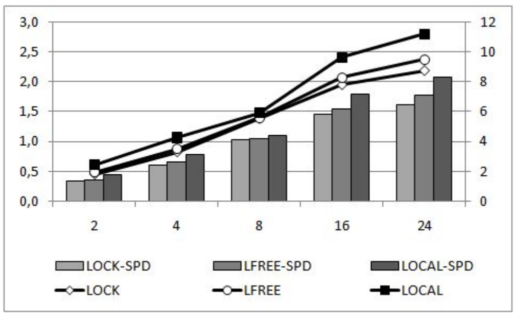


(b) Office

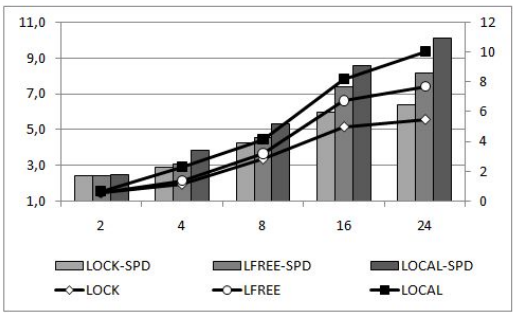


(c) Bunny

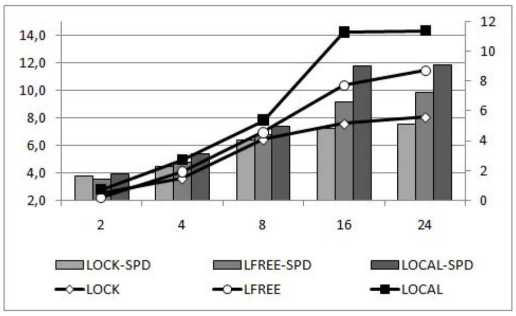
Figure 7.3: Results for the 8-core (plus HT) Xeon 5500 server. Axis: left- fps, right- speed-up, horizontal- number of threads.



(a) Conference



(b) Office



(c) Bunny

Figure 7.4: Results for the 24-core Xeon 7400 server. Axis: left- fps, right- speed-up, horizontal- number of threads.

Chapter 8

Conclusions

Ray tracing has demonstrated to be an accurate and reliable tool to produced high quality and physicality correct images. The interactive ray tracer prototype (iRT) was able to perfectly simulate hard shadows and reflections by making use of only a small ray tree depth (see figure 4.1).

In this thesis we presented and implemented our interactive ray tracer, in order to measure three optimisation techniques which were expressed in results suggesting a margin for future optimisations. Such results were obtained by resorting to Intel's Vtune software and counters in our source code.

To begin with, we implemented a basic version of our renderer, where only one ray per pixel was shot at a time and had to traverse the complete scene (list of triangles) to test if a valid intersection with a triangle was found. In the end the ray was shaded and this value added to the final display buffer. This has proven to be quite inefficient. Subsquently, we included in our software the first optimisation. This optimisation consisted in experimenting two different Bounding Volumes Hierarchies approaches. One of the versions was a direct implementation of Ademar Gonçalves's work [6] and the other was our implementation of Ingo Wald's approach [25]. Although Wald's had previously shown exceptional results we were not able to achieve them. Profiling these three versions showed that with the use of a *BVH* would speed up our ray tracer by more than 100 times. In the Stanford Bunny scene we were able to achieve a 1040 times speed up with Ademar's *BVH.* Our implementation of Wald's *BVH* showed a high *BVH* construction time and a lower frame per second rate. For these reasons we decided to incorporate Ademar's *BVH* into iRT. It was also detected that the traversal function takes the majority of rendering time, presenting much higher values of L1 and L2 Data Cache miss rates, comparing to the other measured functions. An *ADS* with fast building and traversing times is of major importance since dynamic scenes require constant updating on the *BVH*'s shape and fast traversal to avoid hurting the renderer's performance.

Secondly, we decided to vectorise our scalar version of *iRT*. This vectorisation was only possible due to two simple facts: the ray tracing algorithm is highly parallelisable and ray coherence is present among adjacent primary rays. By making use of SIMD instructions we were able to gather four adjacent rays into a single packet. By doing this, it became possible to test four rays at once against one triangle, instead of only one ray at a time, as previously indicated in the scalar version. We then proceeded to test both versions in the matters of memory access performance, frames per second, rays per second and time distribution per function. Tests indicated that the speed up found was below our expectations, staying below 2, when literature points to an almost 4 time fold improvement. By profiling the vectorized version we were able to identify the *BVH* traverse function as the hotspot for rendering, for the reasons that this function takes, in average, more than 80% of rendering time and that initial tests without an *ADS* showed a speed up value near 4. A decrease in L1 and L2 Data Cache Miss rates suggests that less memory accesses are performed and data is more correctly aligned in memory.

Regarding the last optimisation considered in this thesis, we decided to compare three different data access control mechanisms, used to share access to a FIFO-queue holding tasks (sets of rays) for a multi-threaded interactive ray tracer. One approach is based on using locks to provide mutual exclusion to critical regions. This approach, which uses different locks for reading and writing thus reducing contention and serialisation, is referred to as LOCK. The lock-free synchronisation approach (LFREE) avoids all locks by carefully reordering instructions. Finally, within the local approach each thread maintains a local work queue, preventing work sharing but also dispensing with access control mechanisms. Our results have shown that the LOCAL approach outperforms the other two both in raw performance and scalability. This result can be explained by the fact that this approach incurs no data access control overheads. Overheads due to load imbalance do not occur due to both the fine granularity of the tasks and to the homogeneity and shallowness of the per pixel ray trees depths. LOCK and LFREE perform similarly for a moderate number of cores. However, as the core count increases, the time spent waiting to enter critical sections with locks starts to grow exponentially. With the lock free approach this overhead increases sub-linearly, having a significant impact on the achieved frame rates. This result becomes specially relevant due to the ever increasing core count in modern processors. The performance of future shared memory many-core systems will be very likely dependent on the ability to efficiently and robustly share data structures.

iRT versions	**Scenes**	**FPS**	**RPS**
Scalar (Single Threaded))	Conf	0.00158	687.64
	Bunny	0.0015	611.13
	Office	0.0048	2,107.79
Scalar with BVH (Single Threaded)	Conf	0.18	130,000.32
	Bunny	1.56	589,187.04
	Office	0.72	404,381.52
Vectorial with BVH (Single Threaded)	Conf	0.35	259,250.6
	Bunny	1.87	731,910.52
	Office	1.2	726,748.8
LOCAL	Conf	1.83	1,355,078.4
	Bunny	12.37	4,804,458.52
	Office	8.53	5,165,665.64

Table 8.1: Comparison between the progression found in the several presented versions of *iRT*: the basic scalar version, the scalar version with Ademar's *BVH*, the vectorised version, and the best 8-threaded vectorial version. FPS - Frame per seconds, RPS - Rays per second.

These three methods used to optimise our first approach to interactive ray tracing, from the sequential version all the way to a complete multi-threaded and vectorised version (table 8.1 summarises the best values measured throughout this work), proved to bring efficiency and to be of extreme importance to the core of the software here presented. Unfortunately, we were not able to reach interactive frame rates; and based on the present literature on the subject [28, 25, 3, 19], our results in this thesis clearly indicate that there is still margin for further optimisations and, consequently, increase the general performance.

Chapter 9

Future work

Ingo Wald's work on *BVHs* is quite interesting which makes us think this should be targeted as a major objective of further study. An efficient and well implemented *BVH* as Wald's one should improve immensely the ray-triangle intersection function time.

Different alternatives to this ray-triangle intersections method will be studied in order to obtain better results during rendering, being one of them Peter Shirley's Optimising Ray-Triangle Intersection via Automated Search methods[10]. Preliminary results showed a decrease in performance, possibly due to data structure/access management.

The major argument against packetising rays and using explicit SIMD code is that it only increases performance as far as these rays are coherent, i.e., traverse the same regions of space and have high probability of intersecting the same triangle. This is usually true for neighbouring primary and shadow rays, since all do share the same origin and similar directional paths. Coherence drops as we descend in the ray tree, since reflected and transmitted rays usually have very different directions (e.g., for curved surfaces). Studying performance for secondary rays will thus be a major step and the results will help deciding whether these rays should be traced with the sequential or the vector approaches.

The parallel versions of iRT are not close to be linearly scalable in relation to the number of cores/threads. The presented results may be influenced by other parts of the software not directly related with the multi-threading algorithms. As it was proved, the *BVH* traverse function influences the FPS rate and optimisation of this functions should allow better scalability.

Ray-tracing become interactive because there was a need of not only speed up the production of frames, but to allow dynamic scenes to be showed at high frame rates. The next step of our *iRT* will be to incorporate such animations. For this to work there is a need to reconstruct or change the *BVH* structure during execution time. For example, in [25], it was demonstrated that it is possible to achieve such interactive frame rates while including a *BVH* reconstruction between frames rendering.

Finally, in order to generate better quality images, an improvement of the shaders' quality and their diversity is going to be studied in the future, as well as raising the number of rays shot per pixel.

Chapter 10

Acknowledgements

This work was partially supported by research project *IGIDE* – FCT grant PTD-C/EIA/65965/2006.

Bibliography

[1] J. Bigler, A. Stephens, and S.G. Parker. Design for parallel interactive ray tracing systems. *Symposium on Interactive Ray Tracing*, 0:187–196, 2006.

[2] J. Bigler, A. Stephens, and S.G. Parker. Design for parallel interactive ray tracing systems. *Symposium on Interactive Ray Tracing*, 0:187–196, 2006.

[3] Jacco Bikker. Interactive ray tracing, 2008. http://software.intel.com/en-us/articles/interactive-ray-tracing/.

[4] Solomon Boulos, Dave Edwards, J. Dylan Lacewell, Joe Kniss, Jan Kautz, Peter Shirley, and Ingo Wald. Packet-based whitted and distribution ray tracing. In *GI '07: Proceedings of Graphics Interface 2007*, pages 177–184, New York, NY, USA, 2007. ACM.

[5] Kurt Debattista, Kevin Vella, and J. Cordina. Wait-free cache-affinity thread scheduling. *IEE Proceedings - Software*, 150(2):137–146, 2003.

[6] Ademar Gonçalves. Ray tracing interactivo: Estrutura de aceleração. 2009. UCE 15.

[7] Donald P. Greenberg. A framework for realistic image synthesis. *Commun. ACM*, 42(8):44–53, 1999.

[8] Herman Haverkort. Introduction to bounding volume hierarchies. 2004.

[9] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* 2008.

[10] A. Kensler and P. Shirley. Optimizing ray-triangle intersection via automated search. *Symposium on Interactive Ray Tracing*, 0:33–38, 2006.

[11] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast bvh construction on gpus. In *RT '07: Proceedings of the 2009 Eurographics Symposium*, 2009.

[12] Bo Li, Hai Jin, Zhiyuan Shao, Yong Li, and Xin Liu. Optimized implementation of ray tracing on cell broadband engine. In *MUE '08: Proceedings of the 2008 International Conference on Multimedia and Ubiquitous Engineering*, pages 438–443, Washington, DC, USA, 2008. IEEE Computer Society.

[13] Erik Mansson, Jacob Munkberg, and Tomas Akenine-Moller. Deep coherent ray tracing. In *RT '07: Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, pages 79–85, Washington, DC, USA, 2007. IEEE Computer Society.

[14] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. 15th ACM Symp. on Principles of Distributed Computing*, pages 267–275, 1996.

[15] Paul Arthur Navrátil and William R. Mark. An analysis of ray tracing bandwidth consumption. Technical Report TR-06-40, The University of Texas at Austin, November 21 2006.

[16] Miguel Nunes and Luís Paulo Santos. Workload distribution for ray tracing in multi-core systems. In *Encontro Português de Computação Gráfica*, October 2009. accepted for publication.

[17] Persistence of Vision Pty. Ltd. Persistence of vision raytracer (version 3.6.2), 2009. Retrieved from http://www.povray.org/download/.

[18] Ryan Overbeck, Ravi Ramamoorthi, and William R. Mark. Large Ray Packets for Real-time Whitted Ray Tracing. In *IEEE/EG Symposium on Interactive Ray Tracing (IRT)*, pages 41—-48, Aug 2008.

[19] L. Santos P. Dubla, K. Debattista and A. Chalmers. Wait-free shared-memory irradiance cache. In *Eurographics Symposium on Parallel Graphics and Visualization*, pages 57–64. Eurographics, March 2009.

[20] Alexander Reshetov, Alexei Soupikov, and Jim Hurley. Multi-level ray tracing algorithm. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 1176–1185, New York, NY, USA, 2005. ACM.

[21] Maxim Shevtsov, Alexei Soupikov, and Alexander Kapustin. Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes. *Comput. Graph. Forum*, 26(3):395–404, 2007.

[22] Gordon Stoll, William R. Mark, Peter Djeu, Rui Wang, and Ikrima Elhassan. Razor: An architecture for dynamic multiresolution ray tracing. Technical report, University of Texas at Austin, 2006.

[23] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination.* PhD thesis, Computer Graphics Group, Saarland University, 2004.

[24] Ingo Wald, Carsten Benthin, Markus Wagner, and Philipp Slusallek. Interactive rendering with coherent ray tracing. In Alan Chalmers and Theresa-Marie Rhyne, editors, *Computer Graphics Forum Proceedings of EUROGRAPHICS 2001*, volume 20, pages 153–164. Blackwell Publishers, Oxford, 2001. available at http://graphics.cs.uni-sb.de/ wald/Publications.

[25] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics*, 26(1), 2007.

[26] Ingo Wald, Christiaan P Gribble, Solomon Boulos, and Andrew Kensler. SIMD Ray Stream Tracing - SIMD Ray Traversal with Generalized Ray Packets and On-the-fly Re-Ordering. Technical Report UUSCI-2007-012, 2007.

[27] Ingo Wald, Thiago Ize, Andrew Kensler, Aaron Knoll, and Steven G. Parker. Ray tracing animated scenes using coherent grid traversal. *ACM Trans. Graph.*, 25(3):485–493, 2006.

[28] Ingo Wald, William R. Mark, Johannes Günther, Solomon Boulos, Thiago Ize, Warren Hunt, Steven G. Parker, and Peter Shirley. State of the art in ray tracing animated scenes. In Dieter Schmalstieg and Jirí Bittner, editors, *STAR Proceedings of Eurographics 2007*, pages 89–116, Prague, Czech Republic, September 2007. Eurographics Association.