A Cross-Platform Framework for Interactive Ray Tracing

Markus Geimer Stefan Müller

Institut für Computervisualistik
Universität Koblenz-Landau

Abstract: Recent research has shown that it is possible to use ray tracing to render complex scenes at interactive frame rates by efficiently exploiting the computational resources provided by current CPUs. However, these implementations are usually written with just a single hardware platform in mind.

In order to overcome this situation, we present a framework that allows easy cross-platform programming of the SIMD instruction sets made available by many of today's CPUs. Although this framework is geared towards our needs in ray tracing, it might be used in other contexts as well. Currently it supports SSE, AltiVec and an FPU emulation mode, but can be easily extended to use other SIMD instruction sets. In addition, we demonstrate how this framework can be used to implement an efficient scene traversal scheme based on a bounding volume hierarchy of axis-aligned boxes, which is the core of our ray tracing system.

As a result, we show that our implementation is able to render medium sized scenes at interactive frame rates on various hardware platforms using only a single CPU, even without any dedicated optimizations such as fine-tuning the cache usage. Because our system can be further improved in many different ways, this approach seems to be promising regarding the use of interactive ray tracing in heterogeneous environments.

1 Introduction

Ray tracing has the reputation of being a very time-consuming algorithm. Since its introduction to the field of computer graphics through [Wh80], a lot of research has been done to speed up the ray-scene intersection process by utilizing additional data structures, such as grids, octrees, bounding volume hierarchies, or BSP trees (see [Gl89] for an overview). Nevertheless, for a long time ray tracing has been judged not to be suitable for the use in interactive applications.

Recently, [Pa99] showed that it is possible to achieve interactive frame rates using a full-featured ray tracer on a shared-memory supercomputer. Their implementation is able to handle arbitrary geometry including parametric surfaces and volume objects, but is carefully optimized for cache performance, i.e. taking advantage of data cache coherency and eliminating false sharing. They proved that ray tracing scales well in the number of processors and that even complex scenes of up to 35 million spheres can be rendered at approx. 15 frames per second. However, they had to use very expensive high-end hardware to achieve this goal.

By contrast, [Wa01, WSB01] implemented a highly optimized ray tracer running on a cluster of commodity PCs. Their system extensively uses Intel's Streaming SIMD Extensions (SSE) to trace packets of rays in parallel. By using explicit data management and prefetching instructions, they are able to hide most of the network and memory latencies. Additionally, they paid careful attention to the layout of their data structures to reduce memory bandwidth requirements that turned out to be the main limitation with respect to rendering performance in traditional ray tracing implementations. However, this system is restricted to x86 architectures supporting SSE.

Another interesting approach introduced by [Pu02] is the use of programmable graphics hardware to implement ray tracing. As graphics processors (GPUs) usually provide a high degree of parallelism, they seem to be well suited for inherently parallel algorithms such as ray tracing. Unfortunately, it is somewhat difficult to efficiently implement traditional scene traversal schemes due to the lack of important functionality on current GPUs, such as flow control in fragment programs. This can only be accomplished by using multiple rendering passes, thus reducing performance. Nonetheless, recent research (e.g. [Bo03, KW03, Pu03]) indicates that this technology offers a great potential.

Anyhow, in this paper we focus on a pure software solution. We present a framework that allows easy cross-platform programming of the SIMD instruction sets provided by many of today's CPUs. Although this framework is geared towards our needs in ray tracing, it might be used in other contexts as well. Furthermore, we demonstrate how this framework can be applied to implement an efficient scene traversal scheme based on bounding volume hierarchies using axis-aligned boxes, which is the core of our ray tracing system.

As a result, we show that our implementation is able to render medium sized scenes at interactive frame rates on various hardware platforms using only a single CPU, even without any dedicated optimizations such as fine-tuning the cache usage. In addition, we present several possible improvements to our system, indicating that this approach seems to be promising regarding the use of interactive ray tracing in heterogeneous environments.

2 SIMD Abstraction Framework

Today, many CPUs offer specific instruction sets for SIMD computations, i.e. the identical processing of several data elements in parallel with a single machine instruction. Well known examples are Intel's Streaming SIMD Extensions [In03] and Motorola's AltiVec [Mo99]. Since the basic functionality provided by these instruction sets is almost identical, it seems quite natural to implement a library that abstracts from the minor differences on the machine level and permits easy cross-platform programming.

Currently, our framework includes support for three different instruction sets: AltiVec, SSE and an emulation mode that uses the FPU. The latter can be used to compile programs on any unsupported hardware platform. Moreover, we apply it to compare the efficiency of the SIMD code to a serialized version, although it should be noted that this gives only a coarse approximation, since the SIMD programming model imposes some constraints on the implementation creating additional overhead.

Our C++ library provides three new data types: `float4`, `int4` and `bool4`, each one representing a vector of four independent values of the corresponding fundamental data type. Additional types such as `short8` or `char16` could be defined in a similar way if required. Currently, the SIMD abstraction library consists of approx. 45 routines providing the usual arithmetic and bitwise logical operators, comparison functions as well as routines for accessing the individual components of a vector type. Additionally, it contains three new boolean operators to test if none, at least one or all components of a `bool4` are `true`.

To avoid any unnecessary function call overhead, all routines are implemented as inline functions. The main advantage of inline functions over preprocessor macros is that they enable the compiler to do some type-checking, which can be helpful during development. Moreover, it allows us to emulate functionality that is not directly available on a specific platform using a more readable syntax. For example, the AltiVec `vec_select` operation can be expressed by a sequence of three SSE commands. Even so, most functions map directly to the corresponding intrinsics, the high-level language interfaces defined by Intel [In03] resp. Motorola [Mo99] to access their SIMD instructions in a C-like fashion.

Using the intrinsics API has several advantages: First of all, it is easier to program using a C-like interface than writing assembler code. Secondly, the intrinsics are supported by a couple of different compilers, whereas their inline assembler interfaces differ severely thus making the code even more portable. Last but not least, intrinsics enable the compiler to further optimize the code, because it completely handles register allocation and instruction scheduling.

Since the code of only a few functions can be shared among the different architectures (AltiVec, SSE, FPU), each platform has its own implementation of most of the library routines. These are selected via conditional compilation. This way it is fairly easy to add support for other SIMD instruction sets, such as AMD's 3DNow! [Ad00] or MIPS-3D [MI00]. It should even be possible to implement a backend for programmable graphics hardware, although this will probably not be very efficient.

In order to be accessed without performance penalty, all of the SIMD data types need to be properly aligned on a 16-byte boundary on the examined architectures. This issue is handled by the compiler in the case of static resp. automatic variables. Additionally, it takes care of the correct padding when using SIMD data types in structures or as class members. Nevertheless, we have to be careful when allocating objects dynamically, as we have to ensure that all objects start at a valid address. Therefore, our library also provides routines for aligned memory allocation.

3 The Ray Tracing Core

In this section we present our cross-platform ray tracing implementation based on the SIMD abstraction framework described above. We start with a brief overview of the overall system architecture. Afterwards we discuss some important issues in more detail: Memory layout of our basic data structures and scene traversal using bounding volume hierarchies.

3.1 System Architecture

Our main goal was to implement a ray tracing system that is able to achieve a reasonable performance on all of the supported platforms. Therefore, we had to do some careful coding to efficiently use the capabilities of the different architectures. For example, on the lowest level we apply the `vec_nmsub` instruction (negative multiply and subtract) of the AltiVec instruction set to implement a SIMD cross product. This saves three instructions compared to a naive implementation using six multiplies and three subtractions, while not having any performance impact on platforms where this instruction is emulated.

Although ray tracing is capable of rendering arbitrary geometry, we restricted ourselves to triangles. This simplifies the design and avoids branching in the inner loop of our ray tracing core. As processor pipelines get longer, this becomes more and more important on modern CPUs. Hardware features, such as branch prediction and instruction reordering, try to avoid pipeline stalls that decrease performance but their success is strongly dependent on the input code. As a general rule, code should be organized to execute in tight loops with few conditionals. Besides, using only triangles seems to be common practice.

In order to render a single frame, we divide the ray tracing process into three consecutive steps, each done for the entire image:

1. Generate eye rays
2. Find closest intersections (if existent)
3. Shade intersection points resp. store background color

Using this decomposition, we are likely to reduce cache misses due to the smaller working data set of each step, which will result in a performance gain. Although our system traces only primary rays at the moment, it is possible to use this decomposition scheme to handle secondary rays as well. In principle, all three steps can be parallelized using multiple threads, although the time needed for eye ray generation is neglectable compared to the total frame rendering time. However, our implementation currently does not support multithreading.

Similar to [Wa01], we trace packets of four rays in a data parallel way. This applies to the bounding volume hierarchy traversal as well as the ray-triangle intersection testing. To do the latter, a large variety of different algorithms have been proposed. We have chosen the method of [MT97], since it has proven to be very fast and is pretty straightforward to implement using the SIMD framework described in the previous section.

As we use a simple pinhole camera model for eye ray generation, all four rays in such a packet have the same origin. If we connect the intersection points of a ray packet to a single light source, this is also true for shadow rays. This reduces memory requirements for these ray packets and can be exploited in the intersection tests as well (section 3.3).

For shading, we use the well-known Phong reflection model [Ph75]. Since the (at most) four intersection points can have different material properties, data has to be rearranged in order to be processed using the SIMD programming model. This results in some overhead for the setup, but the actual shading computation can then be implemented very efficiently. However, since our shading code is not fully optimized yet, we will not go into more detail in this paper.

3.2 Memory Layout

In order to avoid any unnecessary memory access penalties, we have to take some care of the layout of our basic data structures. As already stated before, all data structures containing SIMD data types have to be properly aligned on a 16-byte boundary. Otherwise, accessing the data will be very slow. Therefore, we pay careful attention on padding our data structures to be a multiple of 16 bytes in size and arrange them in arrays to reduce the overhead due to the alignment.

For instance, the information of a single triangle can be expressed by 9 floats (36 bytes). We pad this data to a total of 48 bytes, storing one vertex and the two edge vectors in separate 16-byte blocks. Beyond that, the memory used for padding keeps additional information that will be useful, such as the triangle id.

As can be seen from the example above, we store data together only if it is used together, i.e. the vertex normals and the material properties used for shading are not part of the triangle data but are kept separately. So we avoid loading data that will not be used, since data transfer between main memory and the caches is always performed in entire cache lines of at least 32 bytes.

3.3 Hierarchy Traversal

As already stated before, we employ a hierarchy of axis-aligned bounding boxes to speed up the ray-scene intersection process. We have chosen the bounding volume hierarchy (BVH) mainly because the traversal can be expressed by a simple and compact iterative algorithm, which is essential for an effective SIMD implementation. This is in contrast to other widely used acceleration structures such as grids and octrees, where each ray in a packet may traverse a different voxel in the next iteration, thus requiring a non-trivial state management. Only BSP trees have a traversal algorithm that has shown to be suitable for an efficient SIMD implementation. However, there are a lot of existing applications using BVHs that might directly benefit from our approach.

To build up the hierarchy, we use a slightly modified version of the construction algorithm proposed by [GS87]. After inserting all triangles into the BVH using the cost-function based heuristic, we additionally sort the children of each node by their surface area in decreasing order. This increases the probability of finding a ray-object intersection early in the traversal process, which gives us an upper bound on the ray length. In our experiments, this reduced the time needed for ray casting by up to 10 percent, depending on the model.

The traversal of our hierarchy is done in depth-first order. If any ray of a packet hits a node, all four rays will continue the traversal of its children. We also tried the traversal scheme proposed by [KK86], but this approach nearly doubled the time needed for ray casting. This is due to the necessary sorting when inserting a bounding box into the heap, which is non-trivial for a packet of up to four ray-box intersection points. Beyond that, the heap management requires additional memory bandwidth.

The simplest way to traverse a hierarchy in depth-first order is to use recursion. However, this appoach suffers from a severe function call overhead. Therefore, we changed the representation of the BVH into an array as described by [Sm98]. Here, all nodes are stored in depth-first order. In addition, each node has an associated skip pointer that points to the next bounding volume that has to be processed if the ray misses the current box and we therefore skip the underlying subtree. This leads to the following compact iterative traversal algorithm:

```
void traverse(Rays& rays, Hits& hits) {
   Box* box = rootnode;

   while (box != stopnode) {
      if (box->intersect(rays)) {
         if (box->hasGeometry())
            box->triangleList()->intersect(rays, hits);

         box++;
      } else {
         box = box->skip();
      }
   }
}
```

Using the array representation has several advantages: First, since we access at least parts of the array in a sequential manner, we may benefit from the hardware prefetchers built into modern CPUs that load possibly needed data into the caches in advance. Secondly, the memory usage of each hierarchy node is fixed and reduced to a minimum, regardless of the number of children. We only have to store the minimum and maximum values along each axis, the skip pointer and a pointer to the geometry. Since a single pointer can be kept in one component of our SIMD vector types, we do not need any padding, thus giving a total of 32 bytes per node.

As a positive side effect, the size of 32 bytes per node is advantageous with respect to cache performance, since this equals exactly the size of a single cache line on many of today's CPUs. As a special exception to the rules presented in section 3.2, we therefore align the hierarchy array on a 32-byte boundary. This prevents the bounding boxes to accidentally straddle a cache line boundary, which would decrease performance.

The ray-box intersection testing is done using the slab algorithm presented by [KK86]. As pointed out by [Sm98], it can be further optimized by taking advantage of the properties of the IEEE floating-point standard. Using the min/max operations provided by the SIMD instruction sets, this test can be done very efficiently without any explicit branching:

```
bool Box::intersect(Rays& rays) {
   float4 origin  = rays.origins(X);
   float4 invDir  = rays.invDirs(X);
   float4 lambda1 = mul(sub(splat(boxMin, X), origin), invDir);
   float4 lambda2 = mul(sub(splat(boxMax, X), origin), invDir);
   float4 lmin    = min(lambda1, lambda2);
   float4 lmax    = max(lambda1, lambda2);
```

```
    for (int axis = Y; axis <= Z; axis++) {
       origin  = rays.origins(axis);
       invDir  = rays.invDirs(axis);
       lambda1 = mul(sub(splat(boxMin, axis), origin), invDir);
       lambda2 = mul(sub(splat(boxMax, axis), origin), invDir);
       lmin    = max(min(lambda1, lambda2), lmin);
       lmax    = min(max(lambda1, lambda2), lmax);
    }

    return any(and(cmplt(lmin, lmax), cmpgt(lmax, 0)));
}
```

However, during our experiments we found out that using a more conservative estimate resulted in better performance. By replacing the `return` statement with

```
return !(all_ge(lmin, lmax) || all_le(lmax, 0));
```

we incorrectly classify some ray packets as hitting the box, thus doing more traversal steps than necessary. Nevertheless, since this is a very rare case when tracing packets of coherent rays, we benefit from the short-cut evaluation of the expression.

Note that the ray-box intersection code can be further improved for primary and shadow rays, because they all have the same origin. If the components of the origin are stored in the same layout as the extents of the bounding box, the subtractions of `boxMin` resp. `boxMax` and the origin can be done at the beginning of the routine, thus saving two data fetches and four subtractions.

4 Results

In this section we present some results measured with our ray tracing implementation that uses the techniques described in this paper. The numbers are based on the ’balls’, ’rings’ and ’tree’ scenes (Figure 1) from the well-known Standard Procedural Databases package proposed by [Ha87]. The screen resolution was fixed to 512×512 pixels for all tests.

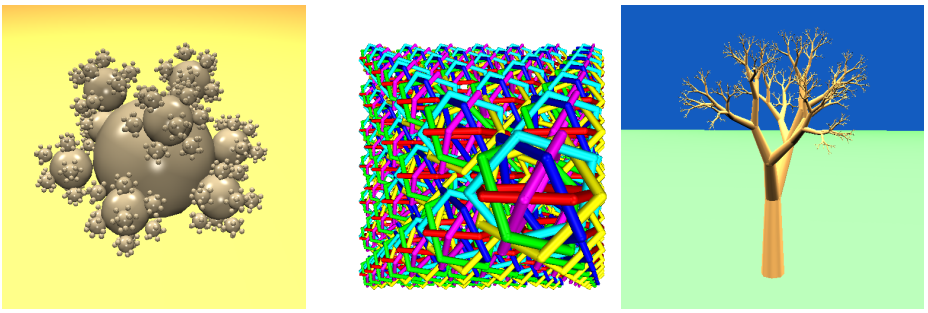

Figure 1: The ’balls’, ’rings’, and ’tree’ models (from left to right) consist of roughly 1.4M, 874k, and 1.7M triangles. The ’tree’ scene is lit by seven, the other two models by three light sources.

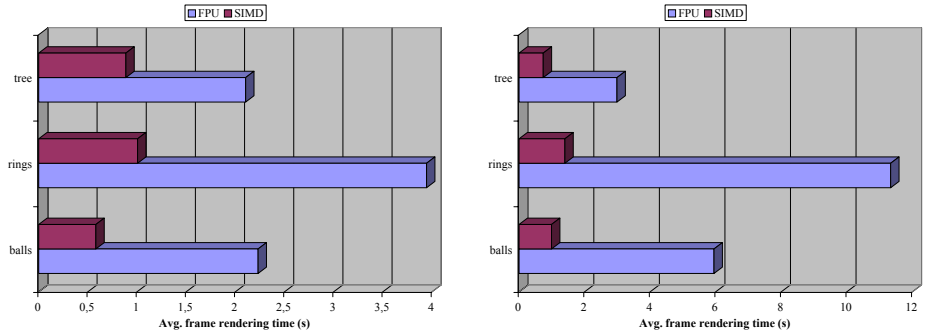

Figure 2: Average frame rendering times for the Xeon machine (left) and the G4 (right), comparing the SIMD implementation with the FPU emulation mode.

Our experiments were performed on the following platforms: The SSE version of our ray tracer was executed under Linux on a dual processor Xeon machine running at 3.06 GHz. This system has 1 GB of memory. The AltiVec tests were measured under Mac OS X on a dual processor PowerMac G4 running at 1 GHz with 1.5 GB of main memory. Since our ray tracing system does not implement multi-threading yet, only one of the processors is used on both platforms. To compile the code, we used the GNU C++ compiler v3.3.

First, we present the results measured on the Xeon machine (Figure 2, left). Using the FPU emulation mode, this system needs average frame rendering times between 2.1 and 3.9 seconds for the three models. When using the SIMD code, these times drop down to 0.58–1 seconds, which gives a total speedup by a factor of 2.4–3.9. Considering only the times for the ray casting step, we achieve a speedup of even 5.2–6.1. This indicates that we efficiently utilize the SSE unit of the Xeon processor.

By contrast, the PowerMac G4 needs approx. 3–11 seconds to render a single frame when using the FPU emulation (Figure 2, right), which is far from interactive. Surprisingly, the G4 performed very well when using the AltiVec enabled version of our code. In this case, we are able to achieve a speedup by a factor of 4–8.1 depending on the model, resulting in average frame rendering times of 0.7–1.4 seconds, which is quite competitive. Again, looking at the ray casting times only, the speedup is in the range of 8–11.

These numbers show that ray tracing is well suited for the SIMD programming model. Keeping in mind that our shading code is not fully optimized yet, these results indicate that our cross-platform approach is rather promising for future work.

5 Future Work

Since our SIMD ray tracing implementation is still at an early stage of development, there is much room left for improvements, both in terms of rendering performance and image quality.

As already stated before, all steps of our frame rendering process could be parallelized by using multiple threads. First experiments indicate that this will nearly double the frame rates on our test platforms. Consequently, the next step would then be to distribute the calculation on a cluster or, since our implementation is already cross-platform, on a network of heterogeneous machines. In addition, we could integrate software prefetching instructions into the code to further improve cache efficiency.

Another interesting approach we will investigate in the future, is the integration of hardware support into our ray tracing implementation by using GPUs to do at least parts of the computation. Similar work has already been done by [CHH02], but we believe that this technology offers even more potential.

In terms of image quality, a ray tracer will not be complete without shadows, reflection and refraction. Additionally, it would be interesting to investigate how global illumination algorithms, such as path tracing or photon mapping, can be adjusted to take advantage of a fast SIMD ray tracing core that relies on packets of coherent rays.

6 Conclusion

For a long time, ray tracing was thought of being too costly to be used in interactive applications. However, recent research has shown that it is possible to achieve interactive frame rates by efficiently using the resources provided by current hardware. Unfortunately, all these implementations are usually written with just a single platform in mind.

The main goal of this paper was to show that it is possible to achieve interactive frame rates using a cross-platform ray tracing implementation. We presented a library that provides an easy way of programming the SIMD instruction sets available on many of today's CPUs, using a higher-level interface. Currently, this library supports two different SIMD architectures and an FPU emulation mode, but can be easily extended to other instruction sets as well.

Furthermore, we have demonstrated that hierarchies of axis-aligned bounding boxes are well suited for a SIMD implementation, tracing packets of four rays in a data parallel way. Storing the nodes in an array in depth-first order, the ray-scene traversal can be expressed by a simple and compact iterative algorithm. In addition, we contributed a SIMD ray-box intersection code without any branching instructions that can be executed very efficiently.

We believe that this cross-platform SIMD approach is promising regarding the application of interactive ray tracing in heterogeneous environments.

Acknowledgements

We would like to thank Marcel Bresink for helping us with the final benchmarks on the PowerMac G4.

References

[Ad00] Advanced Micro Devices, Inc.: *3DNow! Technology Manual*. 2000.

[Bo03] Bolz, J. et al.: Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. *ACM Transactions on Graphics*. 22(3):917–924. 2003. (Proceedings of ACM SIGGRAPH 2003).

[CHH02] Carr, N. A., Hall, J. D., and Hart, J. C.: The ray engine. In: *Graphics Hardware*. pages 37–46. 2002.

[Gl89] Glassner, A. S. (ed.): *An Introduction to Ray Tracing*. Academic Press. London. 1989.

[GS87] Goldsmith, J. and Salmon, J.: Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications*. 7(5):14–20. 1987.

[Ha87] Haines, E.: A proposal for standard graphics environments. *IEEE Computer Graphics and Applications*. 7(11):3–5. 1987.

[In03] Intel Corp.: *IA-32 Intel Architecture Software Developer's Manual, Volume 2: Instruction set reference*. 2003.

[KK86] Kay, T. L. and Kajiya, J. T.: Ray tracing complex scenes. In: *Computer Graphics*. volume 20(4). pages 269–278. 1986. (Proceedings of ACM SIGGRAPH 1986).

[KW03] Krüger, J. and Westermann, R.: Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics*. 22(3):908–916. 2003. (Proceedings of ACM SIGGRAPH 2003).

[MI00] MIPS Technologies, Inc.: *MIPS-3D Graphics Extension*. 2000.

[Mo99] Motorola, Inc.: *AltiVec Technology Programming Interface Manual*. 1999.

[MT97] Möller, T. and Trumbore, B.: Fast, minimum storage ray/triangle intersection. *journal of graphics tools*. 2(1):21–28. 1997.

[Ph75] Phong, B. T.: Illumination for computer generated pictures. *Communications of the ACM*. 18(6):311–317. 1975.

[Pu02] Purcell, T. J. et al.: Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics*. 21(3):703–712. 2002. (Proceedings of ACM SIGGRAPH 2002).

[Pu03] Purcell, T. J. et al.: Photon mapping on programmable graphics hardware. In: *Graphics Hardware*. pages 41–50. 2003.

[Pa99] Parker, S. et al.: Interactive ray tracing. In: *Symposium on Interactive 3D Graphics*. pages 119–126. 1999.

[Sm98] Smits, B.: Efficiency issues for ray tracing. *journal of graphics tools*. 3(2):1–14. 1998.

[Wa01] Wald, I. et al.: Interactive rendering with coherent ray tracing. In: *Computer Graphics Forum*. volume 20(3). pages 153–164. 2001.

[WSB01] Wald, I., Slusallek, P., and Benthin, C.: Interactive distributed ray tracing of highly complex models. In: *Proceedings of the 12th EUROGRAPHICS Workshop on Rendering*. pages 277–288. 2001.

[Wh80] Whitted, T.: An improved illumination model for shaded display. *Communications of the ACM*. 23(6):343–349. 1980.