Benchmarking Ray Tracing for Realistic Light Transport Algorithms

Matthias Raab* Leonhard Grünschloß† Johannes Hanika‡
Manuel Finckh§ Alexander Keller¶

December 11, 2007

1 Introduction

Physically based light transport simulations generally rely on a fast implementation of the ray shooting function $h(x, \omega)$ and a fast visibility test $V(x \leftrightarrow y)$. Those functions directly appear in the transport equations, or reformulations thereof [Vea97], and have to be evaluated very often (they usually make up more than 60% of the total rendering time). Typically the performance of complex surface scattering models is critical as well, their implementations differ widely and cannot easily be compared. In contrast, the functions h and V are universally applicable.

Ray tracing is the natural solution to compute h and V, as those functions actually are what defines ray tracing, and can only be handled insufficiently by other techniques like e.g. rasterization. However, many ray tracing researchers currently focus on rasterization-like techniques, namely accelerating special cases using shafts or ray bundles. While high frame rates for trivially shaded primary rays can be achieved, there is no benefit for a physical simulation that does not fit into the camera-to-scene scenario. Thus, from a physical simulator's view most recent research [WMG+07] in the quite active field of ray tracing can be considered irrelevant for their purpose. That is because the special cases mentioned only account for a negligible fraction of rays usually shot. In addition, the coherence often exploited by these special-case algorithms cannot be provided for physically based algorithms.

We therefore present a framework for benchmarking ray tracing kernels in the context of a realistic light simulation application. By a kernel we mean the functional program unit that provides implementations of h and V, typically using a hierarchical data structure.

*iovis@gmx.de
†leonhard.gruenschloss@googlemail.com
‡hanatos@gmail.com
§manuel@funkyfinckh.de
¶alex@mental.com

2 Related Work

According to our knowledge there exists no previous framework for comparing ray tracing kernels in the context of realistic light transport. However, there has been some related work:

- Smits' global illumination test scenes [SJ00]: A collection of scenes constructed to show some of the difficulties encountered in global illumination simulation.
- Ward's MGF scene repository (`http://radsite.lbl.gov/mgf/`): Scenes with material properties suitable for physically-based rendering.
- RenderPark: a test-bed system for global illumination (`http://www.cs.kuleuven.ac.be/~graphics/RENDERPARK/`): Includes source code and is designed to compare different algorithms for global illumination simulation. However, it does not compare different ray tracing kernels.

In contrast bwfirt has been designed to support multiple ray tracing kernels: you can plug in your own kernel with small effort. Afterwards you can use it to collect some statistics and generate plots for intuitive performance evaluation.

3 Requirements for Realistic Image Synthesis

Realistic light transport algorithms based on (quasi-)Monte Carlo integration, such as Bidirectional Path Tracing or Metropolis Light Transport [Vea97] usually feature a wild mix of incoherent rays to be shot, to both find surface intersection points and to test visibility. Ray tracing kernels that are used in this context should therefore be able to handle arbitrary rays in any order and provide

- a fast implementation of the *ray shooting function* $h(x, \omega)$ for finding the next surface intersection point in the scene surface $\mathcal{S}$ for a ray starting at x in direction ω, i.e.

$$h(x, \omega) := \operatorname*{argmin}_{x' \in \{x + t\omega:\ t \in \mathbb{R}^+\} \cap \mathcal{S}} \|x' - x\|,$$

- and a fast implementation of the visibility test $V(x \leftrightarrow y)$ which is 1 if x and y are mutually visible, i.e.

$$V(x \leftrightarrow y) := \begin{cases} 1, & \text{if } \|h(x, \overrightarrow{xy}) - x\| \geq \|y - x\|, \\ 0, & \text{otherwise.} \end{cases}$$

An important aspect in realistic light transport simulation is the *numerical robustness* of the underlying ray tracing implementation. The geometry is usually provided in single precision floating point representation and due to inaccuracies of many operations in this number system, great care has to be taken to avoid a large error even in the average intensity of a synthesized image. We want to emphasize that numerical precision with regard to ray tracing goes beyond avoiding cracks along triangles.

Additionally, for complex simulation problems the *memory footprint* plays an important role, as performance may suffer severely from swapping from memory to the hard disk.

4 bwfirt - A Simple Ray Tracing Benchmark

In order to account for the requirements above, we created a minimal benchmark framework, mainly cut from an existing renderer. To keep things as simple as possible we only implemented path tracing with next event estimation, e.g. explicit direct light estimation. However, even this rather minimal algorithm provides a realistic mixture of ray shooting and visibility tests. Furthermore, this algorithm generates a rather incoherent distribution of rays through all the visible parts of the scene, which is exactly what we desire for our benchmarking purposes.

If bwfirt is compiled with OpenMP support, it uses all available processor cores for rendering per default, but you can still specify a specific number of threads to use via the `--threads` option. While it may seem odd to include multithreading in a benchmark for an easily parallelizable problem like ray tracing we still included it to allow for testing the thread-safety of your code. Additionally, you can improve your results by parallelizing the construction of your acceleration structure, which usually is far from trivial.

The benchmark provides a python script that automatically generates a report containing graphs, images, compiler, and system information for all available kernels in form of the `bench.pdf` document. Furthermore, it produces text files that contain the information for each individual run of bwfirt.

As most scene descriptions are given in triangles and, more important, the major amount of research in the ray tracing community is spent on triangles, our benchmark currently does only support triangular geometry. New scenes can easily be integrated by a very basic scene file description format: Given a file containing the raw triangles (number of triangles $\cdot$ 9 $\cdot$ `float`) some triangles have to be specified as light sources. This is done in a text file, which might look like:

```
triangles conference.ra2
lights 1 929908 929911 100.0
```

This is the file for the conference room from Ward's MGF file repository. There is one light source consisting of triangles 929908 up to 929911, with a radiant exitance set to 100.0. The triangle indices may be picked in interactive mode using any mouse button.

Providing cameras is simple and can be achieved by the means of another text file, for the conference room a suitable pinhole camera is:

```
type pinhole
position 8.16643 -56.2558 11.4971
direction 0.915894 0.400479 -0.0274723
up 0.0102422 0.0451014 0.99893
depth 1.37374
filmSizeY 1
sensorResponse 1
referenceAverageIntensity (0.857282 0.857282 0.857282)
```

When running in interactive mode a default camera is loaded. It can be used for navigating through the scene (similar to ego-shooters with the left mouse button pressed), in batch mode a camera file has to be specified. The camera file can also be dumped during interactive mode by pressing the key '`c`'.

Another thing worth noting is the last line, containing the average intensity of a reference image which has been computed with 100,000 samples per pixel. This number has to be calculated before the benchmark and manually filled in here to obtain correct error estimates for the numerical comparison graph. If the line is not present, an average intensity of $(0, 0, 0)$ is assumed. Consequently, the numerical comparison graph then displays the squared length of the average image intensity vector.

The full `C++` source code, including a set of standard test scenes, can be downloaded from `http://bwfirt.sf.net`. Besides some included utilities such as a BSD licensed `C++` implementation of the Mersenne Twister [MN98] pseudo random generator, the whole benchmark suite is licensed under the GNU General Public License version 3, but feel free to integrate your proprietary kernels as long as you only want to redistribute the benchmark results.

4.1 Implementing the Ray Tracing Interface

Basically, you have to implement only three functions in order to include your own kernel. You can use the templates located under `src/kernel/yourkernel` to provide:

- `RTKernel::kernelInitialize()`

  This is the method called once before rendering. You should use it to initialize your own data and build your acceleration structure(s) (if you do not use an on-demand construction scheme).

- `RTKernel::kernelIntersectGeometry()`

  Here you can implement your ray shooting function h that returns the nearest intersection found.

- `RTKernel::kernelVisibility()`

  You can put an optimized visibility function V here. If you do not override this method, it is just a frontend to your ray shooting function h.

Handling Surface Offsets. You should not use any custom ε to avoid self-intersections in order to keep different ray tracing implementations comparable (especially with respect to average image intensity). The rendering algorithm guarantees that ray starting points are offset by `RTKernel::NORMAL_OFFSET` times the surface normal. Points on the light source to compute visibility are offset by the same constant if the light source has a geometric representation.

We are perfectly aware that the normal offset by ε "solution" to the self intersection problem is nothing more than a horrible hack and better solutions [Wäc07] do exist. For benchmarking purposes, however, our first goal is to keep different kernels comparable. To achieve this, a constant epsilon is the easiest approach to generate the same behavior for all kernels, sacrificing precision.

4.2 Memory Allocation

In addition to speed, memory requirements are critical for light transport simulations, since even common scenes can consist of hundreds of megabytes of triangle data and the corresponding acceleration structures of ray tracing algorithms can easily exceed the memory available on your system. In order to account for the memory allocation we provided custom allocators in `src/kernel/RTKernel.h` that keep count of memory usage. Just use them instead of `malloc()`, `realloc()`, and `free()`. Note, however, that they contain critical sections for OpenMP parallelization.

- `RTKernel::kernel_malloc()`

  Use this function to allocate all of your memory, an optional alignment may be given.

- `RTKernel::kernel_realloc()`

  Use this function to release parts of the previously allocated memory or if more memory is required.

- `RTKernel::kernel_free()`

  This function releases memory allocated with `kernel_malloc()` and decreases the memory counter.

4.3 Compiling and Running bwfirt

First, please make sure that your system has the following utilities and libraries installed:

- python-matplotlib
- scons
- epstopdf
- optionally SDL, if you want to build the interactive frontend

The benchmark uses *scons* as a build system, a Python-based tool corresponding to autotools and make. Some very basic tests for systems and compilers are included but you can, of course, edit the file `SConstruct` according to your platform. For a standard build, just issue `scons` in the source directory. There are very few build options:

debug Set to 1 for a debug build.

cxx Specify the compiler to use.

dvorak Use *aoe* instead of *asd* for movement in interactive mode.

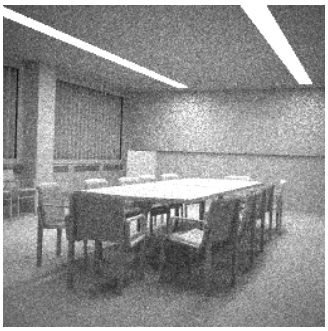
frontend Set to 0 for command line only renderer.

Example: `scons cxx=g++-4.3 dvorak=1`

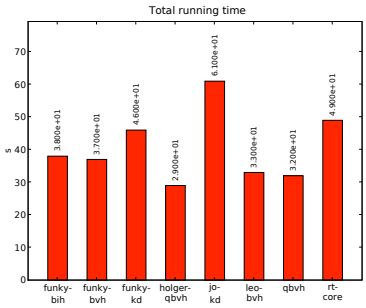
To run the benchmark, simply change to the `bench` directory and issue `python ./bench.py`. You may want to customize the scenes and kernels used. This can be done by editing the lists found at the beginning of the file `bench.py`.

4.4 Some Results and Observations

We ran some tests using a broad range of ray tracing kernels written at Ulm University, featuring kd-trees, bounding volume, and bounding interval hierarchies.



(a) Rendering by bwfirt at 100 passes



(b) Total runnning time

Figure 1: bwfirt's output for the conference room and running time for several kernels on an Intel Core 2 Duo.

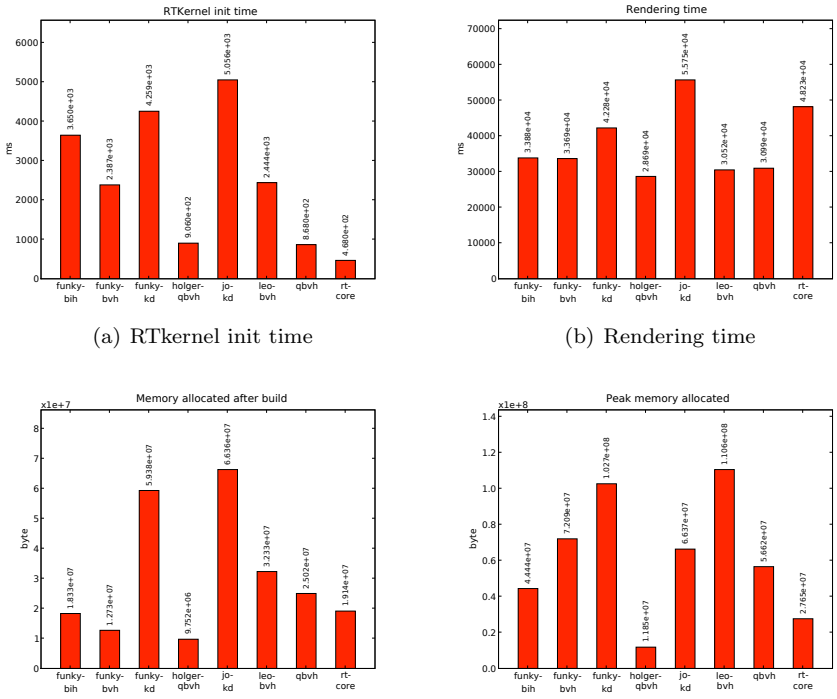
The two kernels *funky-kd* and *simple-bvh* are included in the bwfirt source code package.

- *simple-bvh:* A simple implementation of a very basic bounding volume hierarchy, featuring bounding box and triangle intersection tests in double precision to provide a higher numerical precision for reliable average intensity computations.
- *funky-kd:* A fairly simple kd-tree implementation with decent speed that can be used as reference point for speed comparisons. Note, however, that this kernel is intentionally simple and does not include the surface area heuristic.

Numerical precision. It turned out to be quite hard to produce the same average image intensity using different ray tracing cores. Most differences are due to different triangle intersection codes or custom ε parameters spread through the code (like in the original implementation of the Möller-Trumbore test). Additionally, an only slightly modified ε for the surface offsets has huge impacts on the result, and is able to distort the average image intensity in the first 3 digits.

However, with a fixed triangle test, different kernels using different acceleration structures can achieve very similar results.

Similiar performance. With variations from scene to scene, most of our optimized codes achieve comparable performance. A very notable case is *leo-bvh*, which is a decent implementation of a BVH, with comprehensible code lacking hardcore optimizations such as SSE and still offering competitive performance (at the cost of a high peak memory footprint). Can a major speed gain still



(a) RTkernel init time

(b) Rendering time

(c) Memory allocated after build

(d) Peak memory allocation

Figure 2: Additional statistics for the conference room scene.

be expected or will the only achievable advances be found in the area of memory consumption (see the exceptionally good performance of *holger-qbvh* at low memory footprint)? Please note that the source code for both of these kernels is not included in the bwfirt source code package.

5 Conclusion

bwfirt is an approach to a ray tracing benchmark for path tracing based algorithms. It can be used to compare ray tracing algorithms by different researchers quite easily with regard to single rays and we hope some people actually use it to give comparable numbers.

We want to hear from you! A final conclusion can only be drawn when several ray tracing kernels have been tested using this framework. So if you have a kernel which you believe is doing well, we hope you will try the benchmark on it and let us know the results.

6 Acknowledgments

bwfirt has been funded by the project *information at your fingertips - Interaktive Visualisierung für Gigapixel Displays*, Forschungsverbund im Rahmen des

Förderprogramms Informationstechnik in Baden-Württemberg (BW-FIT).

References

[MN98] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, 1998.

[SJ00] B. Smits and H. Jensen. Global illumination test scenes. Technical Report UUCS-00-013, University of Utah, 2000.

[Vea97] E. Veach. *Robust Monte Carlo Methods for Light Transport Simulation*. PhD thesis, Stanford University, 1997.

[Wäc07] C. Wächter. *Quasi-Monte Carlo Light Transport Simulation by Efficient Ray Tracing*. PhD thesis, Ulm University, 2007.

[WMG+07] I. Wald, W. Mark, J. Günther, S. Boulos, T. Ize, W. Hunt, S. Parker, and P. Shirley. State of the Art in Ray Tracing Animated Scenes. In *Eurographics 2007 State of the Art Reports*, 2007.