RayBooster : An efficient, robust, generic and easy to use basic component for high performance radiative simulations

Jean-Patrick Roccia[1] and Vincent Forest[1]
[1]Research engineer at HPC-SA, Toulouse, France

Abstract

Latest environment concerns and bioclimatic issues are essential in modern building performance simulations. Among those concerns, most of simulations require radiative propagation computation to estimate solar incomes, daylighting, solar panel efficiency, etc. Ray Tracing based methods are a robust approach to radiative simulation. However, a common wisdom is to associate them to slow performances or high demanding computation power. Based on the acquired expertise of HPC-SA in terms of research, industrialization and integration of Ray Tracing technologies over the last decade, we introduce a ray tracer named RayBooster. It helps to reduce significantly the per ray cost of simulations. In fact, this innovative basic component allows both to increase the quality of results for the same amount of time, and to shift off-line simulations, which need a lot of time computations, toward a real-time world. In this paper, we briefly describe the RayBooster ray tracer and its advantages in terms of genericity and performances. We then measure its efficiency on practical benchmarks before illustrating its ease of use by providing a RayBooster 4 powered version of the widely used light simulation toolchain named Radiance (Ward 1994). We show that this integration is not only straightforward but also drastically improves both the robustness and the efficiency of this well-tried codebase.

1 Introduction

Building performance simulation is becoming an essential step in both building conception and building renovation phases. Building performance simulation addresses multiple crucial and challenging concerns. Among these, bioclimatic, environmental and financial considerations predominate. Most of building energy simulations, like solar incomes computation, daylighting, solar panel efficiency, *etc.*, require radiative phenomenon estimations. Now, radiative phenomenon are quite challenging to simulate efficiently and precisely, and can easily become a headache to design and optimize in software. Our researches take place in this context. In order to alleviate radiative phenomenon simulation challenges, we introduce RayBooster 4, the latest evolution of the RayBooster product (HPC-SA 2004). In the first part (section 2), Ray Tracing method is briefly described before introducing RayBooster's purposes and features (section 3). After this functionalities overview, an evaluation methodology is proposed (section 4) and different kind of results are presented and discussed (section 5), specially the RayBooster 4 powered Radiance integration and results. Finally, the conclusion (section 6) revealed the guideline of our future work.

2 Ray Tracing

In this section we provide an overview of the Ray Tracing method, followed by a sensitization to its performance and optimization issues.

Method overview

As pointed out by Macey in (Macey 2008), Ray Tracing foundations have been enunciated by René Descartes in 1637. He uses light rays in order to explain the formation of rainbows. With the appearance of computer assisted simulations, Ray Tracing became quickly an important research field in various domains like optical (Appel 1968), thermal (Eckert 1969), acoustic (Allen & Berkley 1979), *etc.*, that allows high accuracy simulations of radiative phenomenon. Concretely, the Ray Tracing can be split in two distinct parts: the generic intersection request and the domain dependent physics.

Intersection The intersection request is purely geometric. It is a central and common part of all Ray Tracing applications. The physical part of the Ray Tracing uses this request. It allows to compute visibility relationship between objects in a given space. This part typically prevails in terms of execution time. So, it is critical, and requires special attention in order to obtain good performances. The cost of this request often leads to concessions on the physical side, in order to limit the number of intersection to compute.

Physics The physical part of the Ray Tracing is based on a model of rays propagation associated to the simulation domain. Therefore, this part is closely linked to simulated phenomenon, and can be totally independent of the inter-visibility computation request. Concretely, the physics part allows to quantify the impact of radiative emission in a scene, and to propagate this radiative emission by using the intersection request in order to establish visibility relationship.

Performances

Despite decades of active researches in the Ray Tracing field, obtaining high performances is still a challenging objective. The main bottleneck in term of Ray Tracing intersection request is the ray-primitive intersection step. Indeed, a tri-dimensional scene contains several geometrical primitives, which can be intersected by any rays. Several researches investigate the optimisation of the ray-intersection process for a wide variety of primitives such as triangles (Möller & Trumbore 1997) quadrilaterals (Lagae & Dutré 2005) implicit surfaces (Knoll, Hijazi, Kensler, Schott, Hansen & Hagen 2009), *etc.*

However, optimising the ray-primitive intersection test is not sufficient since a naïve Ray Tracing approach would intersect each ray with all geometrical primitives in the scene. Obviously, this will result in very poor performances, decreasing linearly with the number of primitive in the scene. In order to address this linear complexity, it is necessary to partition the scene to reduce the set of primitives to test for each ray. To do this, several Ray Tracing acceleration structures have been proposed (regular grid (Cleary & Wyvill 1988), *k*d-tree (Bentley 1975), Bounding Volume Hierarchy (BVH) (Rubin & Whitted 1980), octree (Hunter 1978), *etc.*). Each of them have its pros and cons, and the best choice can depend of the final application. We point out that actually this partitioning data structure is the core of any ray-tracer, defining both its properties and its efficiency. Efficiently building such data structure is thus one of the main challenge of a generic ray-tracer as RayBooster 4.

At last, but not least, parallelization is essential in a modern high performance Ray Tracing system. Computing power growth of processing units is expressed by an increasing number

of computing cores on both Central (CPU) and Graphics (GPU) Processing Unit. Ray Tracing can benefit of this parallelism at several stages and with various implementation hardness. Ray Tracing parallelization can be divided into dedicated algorithms and hardware specific implementations. On the algorithmic side, the acceleration structure build and its traversal method must be updated in order to maximize parallelization benefit. The hardware specific side is more tricky, and requires to use various parallelization paradigm like Single Instruction Multiple Data (SIMD) on CPU, with (INTEL 2008) or (ARM 2009), and Single Instruction Multiple Threads (SIMT) on GPU, with (GROUP 2008) or (NVIDIA 2007). This plurality in term of parallelization expression leads to a lot of hardware abstraction difficulties.

3 RayBooster 4

In this section, a description of RayBooster 4 features and benefits is provided. Note that a detailed documentation is available online (HPC-SA 2004).

What is RayBooster?

RayBooster 4 is a Ray Tracing Acceleration Library. It provides software functions and utilities in order to handle geometric and intersection requests in an easy, non-intrusive and efficient way. As shown in the figure 1, RayBooster 4 handles all Ray Tracing specific acceleration data and intersection tests, but the physics used to generate rays and interpret found intersections remains on the user side.

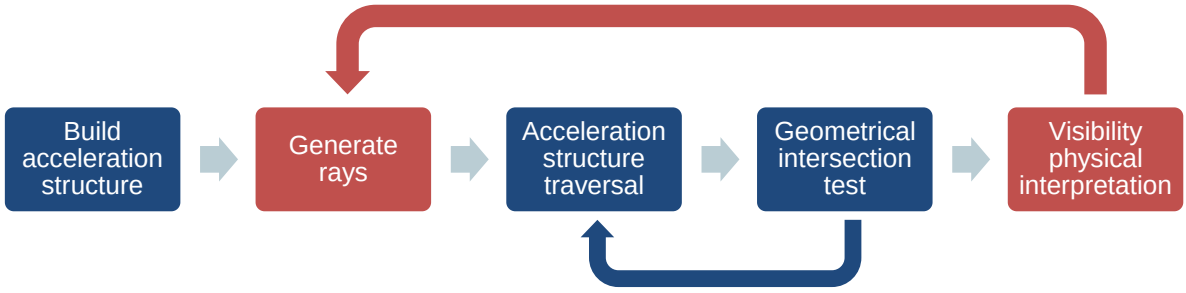


Figure 1: This schema describes a typical and general Ray Tracing process. Blue cells represent steps operated by RayBooster 4, which are independent of the simulation domain. Red cells remain on the user side because of their strong specific simulation domain dependency

Requirements

RayBooster 4 is a C language library. Being essentially a mathematical library, it doesn't need any compatibility requirements. System specific functions are used, particularly for multithreading purpose and advanced low-level optimization. However, these calls are transparent and shall not disrupt the host application.

On the hardware side, the requirements depend of the targeted RayBooster 4 driver. Currently, the only one publicly released driver requires an x86 processor with SSE2 instructions enabled (at least Intel Pentium 4 or AMD Athlon 64). Additionally, RayBooster 4 is available on the following platforms, in both 32 bits and 64 bits: Microsoft Windows, Apple MAC OS, GNU/Linux.

Concepts

RayBooster 4 relies on a set of concepts leading to a natural work-flow. Each concept and its associated work-flow is described below.

Driver

A driver is a specific implementation of the RayBooster 4 Application Programming Interface (API). Actually the driver is loaded dynamically according to detected hardware and its capabilities.

Scene

A scene is a set of instantiated geometries. Each Ray Tracing request is applied to a scene, so the scene represent a set of geometries that can be intersected.

Geometry

A geometry, in the meaning of RayBooster 4, is an access to the client triangulated geometry structure. A geometry can be registered in scenes, in order to be instantiated in them.

Instances

An instance is the association of a geometry and a transformation matrix. A geometry can be instantiated several times in a scene, with different matrix. Instantiating similar geometries allows to reduce memory consumption, by not duplicating geometrical data.

Layers

Layers offer the possibility to partition geometric instances inside a scene. An instance can be declared active for only a sub-set of layers. RayBooster 4 allows up to thirty two distinct layers, and each call to the Ray Tracing can be performed on a sub-set of those layers.

Ray Tracing

This part is the core of RayBooster 4. Users provide rays that must be intersected with a given scene, and gets back intersections with geometries. For each set of rays to intersect, users can specify a set of active layers.

Work-flow

The RayBooster 4 work-flow can be divided into five specific stages, as shown in the figure 2. Each stage will be described more precisely in this section.

Idle stage This stage is the initial one for users. The only one feature provided at this step is RayBooster 4 driver loading. A driver is required for any interactions with RayBooster 4. Loading a driver allows to switch to the driver stage.

Figure 2: This schema describes the different RayBooster 4 work flow stages. Each stage provides its own features which are described below.

Driver stage At this stage, the driver offers the possibility to create a RayBooster 4 scene. In order to switch back to the previous stage, it is possible to unload the driver. Creating a scene allows to switch to the scene stage.

Scene stage The scene stage offers features in order to describe the client geometry, in term of RayBooster 4 geometries, instances and layers. Locking the scene switches to the Ray Tracing stage while scene destruction switches back to the driver stage.

Ray Tracing stage At this stage, RayBooster 4 rejects all geometric modifications on the locked scene. The only one step required to start a Ray Tracing session is the RayBooster 4 internal scene data building. After completing this construction step, a Ray Tracing session is enabled, while unlocking the scene switches back to the scene stage.

Ray Tracing session At this point RayBooster 4 offers the possibilities to ray trace an isolated ray, or a set of rays. Unlocking the scene switches back to the scene stage.

RayBooster 4 properties

The RayBooster 4 Ray Tracing solution benefits of more than a decade of active researches and industrialization. This maturity leads to two majors assets : user hardware abstraction and non-intrusive integration.

Despite the fact that RayBooster 4 can use different driver implementations according to runtime resources, user is not impacted by the selected driver. Each driver defines its own type of acceleration structure, intersection test and parallelism strategy. High performances are ensured by state of the art algorithms and hardware specific optimizations, consistently with driver targeted computing units (Roccia, Paulin & Coustet 2012), (Roccia 2013).

RayBooster 4 is generic and uncorrelated to applicative domain, and does not interfere with the application field. Moreover, it can access host software data, ensuring non-duplication without restrictions on host defined types.

Finally, RayBooster 4 is not only free for non-commercial use (HPC-SA 2004) but its features, reliability and usability are also already experienced by both industrial and research products (ArchiWIZARD (HPC-SA 2009), Artlantis (ABVENT 2005), Solfast 4D (HPC-SA 2012), *etc.*).

4 Methodology

Having described the features and the implementation details of the RayBooster 4 library, its efficiency can now be evaluated in both common benchmarks and practical situations. To achieve this, a set of Benchmarks have been designed and are detailed in this section.

Synthetic Benchmark

The RayBooster 4 synthetic benchmark is designed to put the Ray Tracing solution in a very stressful position. Processing units architectures perform well on ”coherent” Ray Tracing. It means that tracing similar rays sequentially improve performances. As RayBooster 4 is optimized to perform real-case applications and assumes nothing on the traced rays, this benchmark launch a huge amount of randomly generated, and so ”incoherent”, rays in a typical complex indoor scene.

Practical Benchmark

Monte-Carlo based methods (Metropolis & Ulam 1949), designed to numerically solve integral equations using a high number of independent samples, are widely used to solve complex radiative phenomenon. Those methods are quite challenging to optimize, and to bring to the interactive world. This is why the RayBooster 4 practical benchmark relies on a Monte-Carlo Path-Tracer renderer (Whitted 1980), (Kajiya 1986). This renderer was entirely developed by HPC-SA for Ray Tracing research purposes, it supports transparencies, textures and alpha-textured objects.

Radiance Benchmark

Previous benchmarks measure the RayBooster 4 efficiency in codebases respecting modern efficiency standards like shared-memory parallelism and data oriented programming. Unfortunately, several in-use Ray Tracing solutions were designed with old paradigms that both limits their performances on current hardware and makes the integration of a modern Ray Tracer more challenging.

Radiance is one of such mature codebase. The development of this Ray Tracing solution started in 1984 and it is still today widely used in both lighting and daylighting simulations (Ward 1994). Despite its attractive and well-tried functionalities, its computing time can be quite slow and may vary by several orders of magnitude with respect to the scene layout. By replacing the Radiance ray-tracer by RayBooster 4 one can expect to improve and stabilize its overall performances even though this task may appear quite challenging.

Integration

The first technical issue in the integration of RayBooster 4 in Radiance is that several parts of Radiance was written in K&R C (Kernighan & Ritchie 1978), a pre-standard C language not supported by C++ compilers (ISO 1998). However, thanks to its simple C 90 API (ISO 1990), RayBooster 4 can be used in this pure C codebase without rewriting or hazardous converting it (GNU-C 2006). Actually, a naïve integration of RayBooster 4 in Radiance is quite straightforward.

During the loading of a Radiance “octree” file, the geometric primitives are registered as well as their potential instance data in a temporary data structure. The previous collected informations are then used to create the models and their associated instances in RayBooster 4. Finally

the Radiance “trace” process is replaced by a call to the RayBooster 4 `SceneTraceRay` function and the resulting data is used to fill the output data structures.

We point out that while Radiance supports parametric surfaces as well as any planar polygons, RayBooster 4 only handles triangular faces. Nevertheless, all planar primitives are triangulated on the fly by Radiance when intersection tests occur. So, this drawback is easily alleviated by calling the Radiance triangulation method on all planar polygons as a precompute step, during the load process, instead of calling it during intersection tests. However, it is a deliberated choice to not tessellate parametric surfaces in order to let Radiance intersecting these primitives with an accuracy depending on its own strategy. At least, this approach implies no computing overhead for the RayBooster 4 powered Radiance. Planar surfaces are just triangulated earlier and implicit surfaces are handled the same way as the original Radiance.

5 Results and discussions

This section exhibits the RayBooster 4 robustness and its high performances on a wide variety of scenes, using previously defined benchmarks. We also demonstrate that, thanks to its architecture, a naïve integration of RayBooster 4 in a well-tried and widely used codebase as Radiance (Ward 1994) is not only straightforward but also significantly improves its overall performances.

Synthetic performances

This benchmark results, figure 3, are useful in early software design, in order to determine what can be done in real-time, interactively or off-line with RayBooster 4. For instance, knowing this ”worst-case performances” results, it is reasonable to expect at least 24 Million rays per second for this mainstream computer.

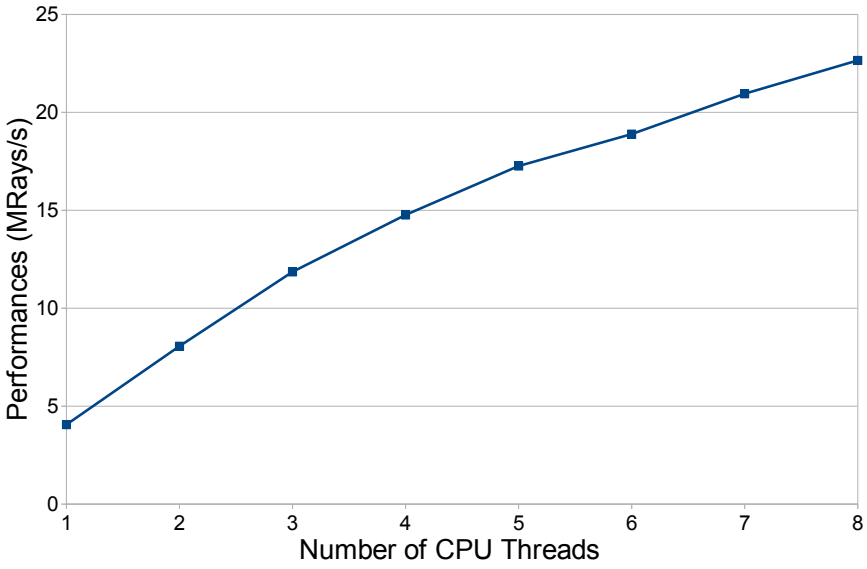
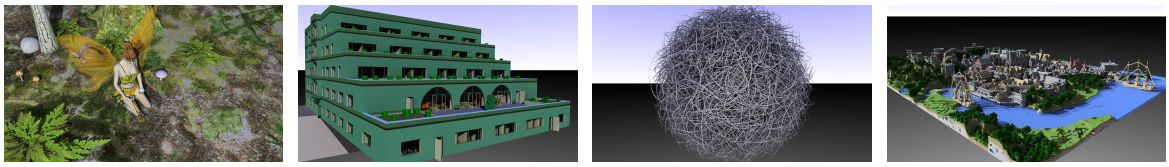


Figure 3: RayBooster 4 synthetic results in a very stressful and unfavorable benchmark. Performances, in Millions of rays traced per seconds, according to the number of active CPU thread. Results obtained on a four core Intel Core-i7 3770 CPU at 3.4GHz

Practical performances

The Path-Tracer performances are measured on set of various scenes (figure 4). Results, available in figure 5, highlight the capacity of RayBooster 4 to perform, in average, nine image realizations per second in full HD resolution (1920x1080 pixels) on a mainstream hardware configuration. This kind of performances allows interactive Monte-Carlo light simulation in Full HD.



(a) Fairy Forest 174,117 triangles | (b) Soda Hall 2,156,683 triangles | (c) Hair Ball 2,850,000 triangles | (d) Rungholt 6,704,264 triangles

Figure 4: Practical benchmark test scenes, rendered with the path-tracer

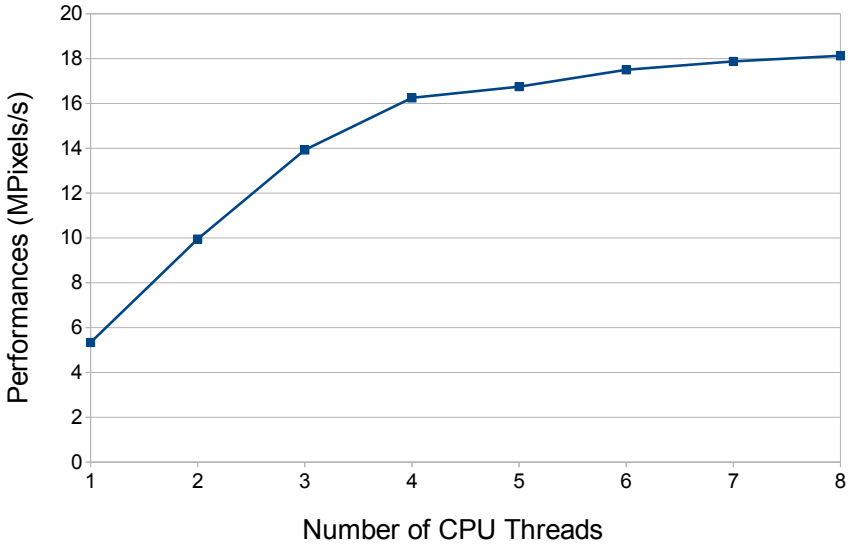
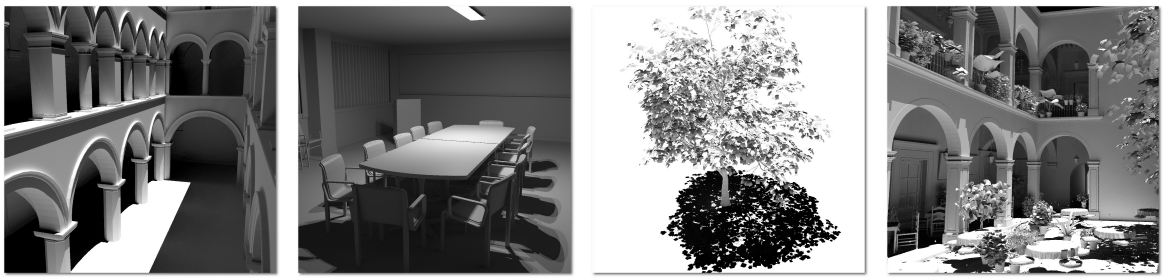


Figure 5: RayBooster 4 practical results in a real Monte-Carlo based path tracer with three light bounces. Average performances, in Millions of pixels per seconds, according to the number of active CPU thread on test scenes. Results obtained on a four core Intel Core-i7 3770 CPU at 3.4GHz

RayBooster 4 in Radiance

We compare the original version to the RayBooster 4 integration by rendering a wide variety of scenes (figure 6) with the unbiased global illumination algorithm of Radiance. The efficiency of this Path-Tracing-like approach (Kajiya 1986) is particularly dependent of the underlying ray-tracer and is consequently a good candidate to measure its Ray Tracing performances. In addition, all the scenes were designed with one simple diffuse material in order to limit the influence of the material evaluation on the results. We report the performances obtained for

only one rendering definition since we noted that this parameter does not affect the performance ratio between the original and the RayBooster 4 version of Radiance.



(a) Sponza 66,447 triangles | (b) Conference 331,182 triangles | (c) Tree 705,462 triangles | (d) San-Miguel 7,838,612 triangles

Figure 6: Radiance benchmark test scenes

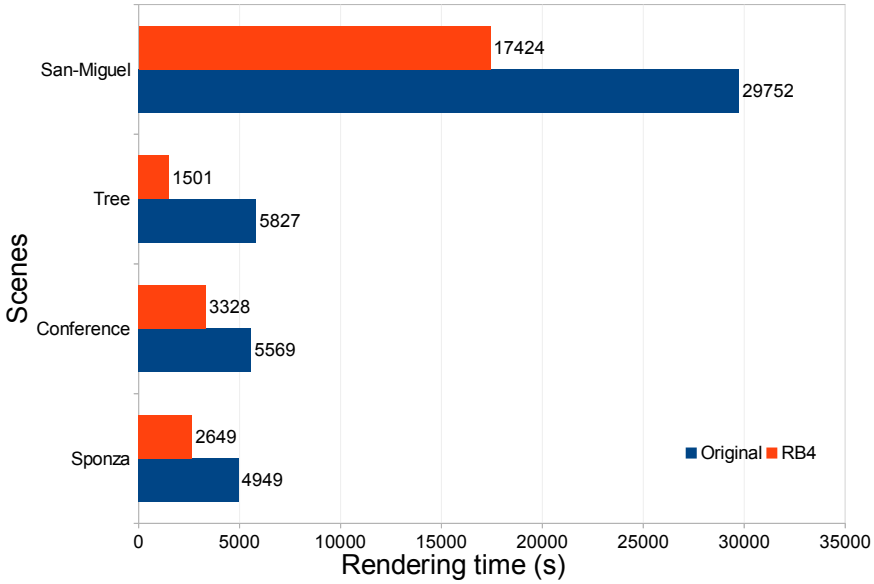


Figure 7: Rendering time in seconds for an image definition of 1826×1080 of the original and RayBooster 4 (RB4) versions of Radiance

The figure 7 presents the rendering time obtained on a Linux workstation composed of an Intel Core i7-860 CPU at 2.8GHz and 8GB of DDR3 at 1333MHz. We observe that the RayBooster 4 version is from 1.7 to 3.9 times more efficient of the original one. As expected the highly detailed and thin geometries of the San-Miguel scene is the most difficult to render. However we point out that while the performance ratio between the original and the RayBooster 4 version of Radiance is roughly the same on Sponza, Conference and San-Miguel (≈ 1.8) it is two times superior for the Tree scene. We supposed that this is due to the spacial complexity of the scene, a single tree lying on a wide ground plane. This so called "teapot in stadium" situation may be particularly difficult to handle by the octree space partitioning data structure used by the original Radiance ray-tracer.

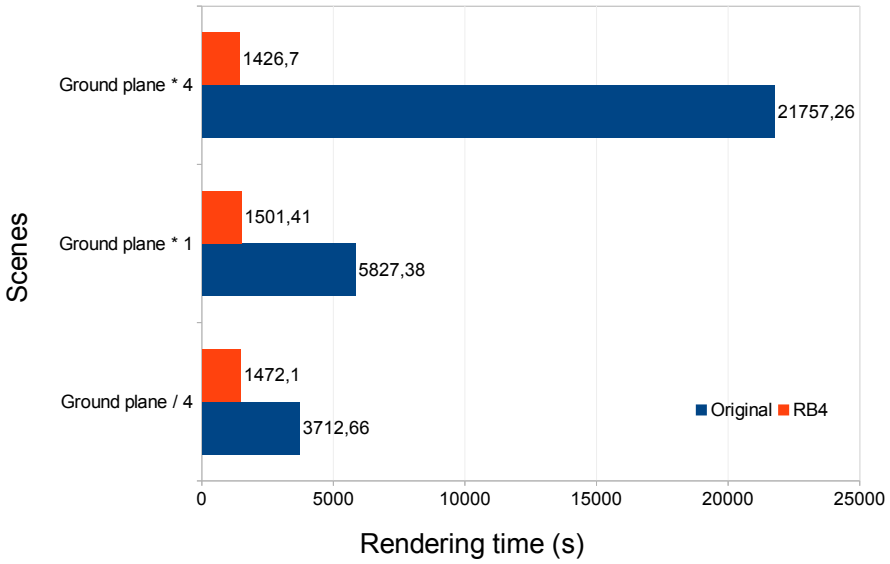


Figure 8: Rendering time in seconds of the Tree scene on the original and RayBooster 4 (RB4) versions of Radiance (image definition of 1826×1080). The reported performances was measured on the initial scene (Ground plane * 1) as well on two variants from which the ground plane size was multiply (Ground plane * 4) or divided (Ground plane / 4) by four.

The figure 8 exhibits this drawback by rendering this Tree scene with a smaller and a greater ground plane. In both situations, the rendered images are the same but while on the RayBooster 4 version the performances remain roughly the same, the efficiency of the original version of radiance drastically changes; dividing or multiplying by 4 the ground plane size decrease by 1.56 or increase by 3.73 its rendering time, respectively. In this simple case, one can alleviate this issue by instancing the tree geometry in a separate octree priorly to its insertion in the final scene. Unfortunately such data reorganization may be particularly difficult and time consuming for more complex scenes. This highlights the RayBooster 4 robustness and adaptability to users data, by maintaining high performances for all spatial configurations.

(a) Original Radiance (b) RayBooster 4 powered Radiance (c) (1 - Difference) ×5

Figure 9: Difference (right) between an original Radiance rendering (left) and a RayBooster 4 powered Radiance rendering (middle).

The figure 9 shows that images computed by both Radiance versions are substantially the same. The per pixel differences are explained by the random nature of the Radiance light paths

sampling algorithm. One cannot strictly reproduced the random pattern due to differences in the numerical schemes of the two geometric intersection implementations. However, this phenomenon is amplified by the iterative, heuristic-based, image refinement process of Radiance. Note that without this Radiance refinement optimization, results will tends to the same image for both versions, with an increased number of light path explored.

6 Conclusion and future work

We have discussed radiative simulations issues, introducing RayBooster 4, an efficient, robust, generic and easy to use basic component for high performance radiative simulations. This software assets in terms of usability, robustness, integration and performances have been discussed and applied to different kind of applications.
Future works naturally emerge from introduced researches. The first one consist of RayBooster 4 improvements, by adding features and releasing new drivers. Another axis is to push further the RayBooster 4 integration in Radiance, in order to provide a robust, fast and complete replacement for the Radiance standard ray-tracer, without modifying Radiance interface and users codes. Finally, we plan to promote RayBooster 4 inside the ArchiWIZARD Daylight simulator, in order to achieve both extreme performances and accuracy.

7 Acknowledgments

The authors would like to thank our partner GEOMAP GIS America (GEOMAP GIS America 2000), the ITEA organisation (ITEA 2012) and all used data sets owners (Ingo Wald, the UC Berkeley Walkthrough Group, Samuli Laine, Tero Karras, Kescha, Guillermo M. Leal Llaguno, Marko Dabrovic and Greg Ward) for sharing them.

8 References

ABVENT (2005), Artlantis. http://www.artlantis.com/.

Allen, J. B. & Berkley, D. A. (1979), 'Image method for efficiently simulating small-room acoustics', *The Journal of the Acoustical Society of America* **65**(4), 943–950.

Appel, A. (1968), Some techniques for shading machine renderings of solids, *in* 'Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference', AFIPS '68 (Spring), ACM, New York, NY, USA, pp. 37–45.

ARM (2009), ARM Architecture Technologies. http://www.arm.com/products/processors/technologies/neon.php.

Bentley, J. L. (1975), 'Multidimensional binary search trees used for associative searching', *Commun. ACM* **18**(9), 509–517.

Cleary, J. & Wyvill, G. (1988), 'Analysis of an algorithm for fast ray tracing using uniform space subdivision', *The Visual Computer* **4**, 65–83.

Eckert, E. R. G. (1969), 'Radiative transfer, h. c. hottel and a. f. sarofim, mcgrawhill book company, new york, 1967. 52 pages'.

GEOMAP GIS America (2000). http://www.geomapgis.ca/en.

GNU-C (2006), 'protoize'.

GROUP, K. (2008), Opencl - the open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/.

HPC-SA (2004), RayBooster: raytracing acceleration library. http://www.hpc-sa.com/en/products/raybooster.

HPC-SA (2009), ArchiWIZARD. http://www.archiwizard.fr/.

HPC-SA (2012), Solfast 4D. http://www.hpc-sa.com/en/products/solfast4d.

Hunter, G. M. (1978), Efficient Computation and Data Structures for Graphics., PhD thesis, Princeton, NJ, USA. AAI7823520.

INTEL (2008), INTEL Intrinsic guide. http://software.intel.com/fr-fr/avx.

ISO (1990), Programming languages – C, Technical Report 9899:1990, ISO/IEC.

ISO (1998), Programming languages – C++, Technical Report 14882:1998, ISO/IEC.

ITEA (2012). https://itea3.org/.

Kajiya, J. T. (1986), 'The rendering equation', *SIGGRAPH Comput. Graph.* **20**(4), 143–150.

Kernighan, B. W. & Ritchie, D. M. (1978), *The C Programming Language*, 1st edn.

Knoll, A., Hijazi, Y., Kensler, A. E., Schott, M., Hansen, C. D. & Hagen, H. (2009), 'Fast ray tracing of arbitrary implicit surfaces with interval and affine arithmetic.', *Comput. Graph. Forum* **28**(1), 26–40.

Lagae, A. & Dutré, P. (2005), 'An efficient ray-quadrilateral intersection test', *Journal of Graphics Tools* **10**(4), 23–32.

Macey, J. (2008), Ray-tracing and other rendering approaches. http://nccastaff.bournemouth.ac.uk/jmacey/CGF/slides/RayTracing4up.pdf.

Metropolis, N. & Ulam, S. M. (1949), 'The Monte Carlo Method', *Journal of the American Statistical Association* **44**(247), 335–341.

Möller, T. & Trumbore, B. (1997), 'Fast, minimum storage ray-triangle intersection', *J. Graph. Tools* **2**(1), 21–28.

NVIDIA (2007), CUDA Zone - NVIDIA Developer Zone. http://developer.nvidia.com/category/zone/cuda-zone.

Roccia, J.-P. (2013), Ray tracing acceleration on heterogeneous parallel systems, PhD thesis, University Paul Sabatier, TOULOUSE.

Roccia, J.-P., Paulin, M. & Coustet, C. (2012), Hybrid cpu/gpu kd-tree construction for versatile ray tracing, *in* 'Eurographics (Short Papers)'12', pp. 13–16.

Rubin, S. M. & Whitted, T. (1980), 'A 3-dimensional representation for fast rendering of complex scenes', *SIGGRAPH Comput. Graph.* **14**(3), 110–116.

Ward, G. J. (1994), The RADIANCE lighting simulation and rendering system, *in* 'Proc. SIGGRAPH', pp. 459–472.

Whitted, T. (1980), 'An improved illumination model for shaded display', *Commun. ACM* **23**(6), 343–349.