Real-time Ray Tracing through the Eyes of a Game Developer

Jacco Bikker*
IGAD / NHTV University of Applied Sciences
Breda, The Netherlands

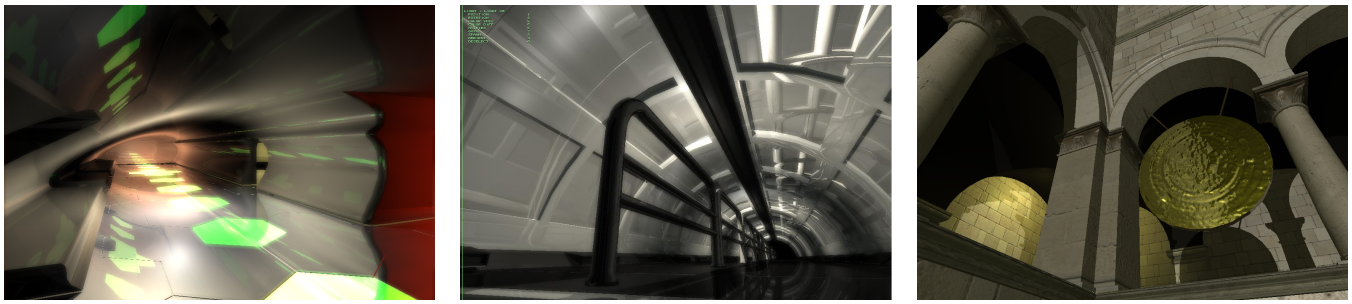


Figure 1: Real-time ray traced images from our experimental ARAUNA engine, used in a student project.

ABSTRACT

There has been and is a tremendous amount of research on the topic of ray tracing, spurred by the relatively recent advent of real-time ray tracing and the inevitable appearance of consumer hardware capable of handling this rendering algorithm. Besides researchers, the prospect of a brave new world attracts hobbyists (such as demo coders) and game developers: Ray tracing promises an elegant and fascinating alternative to z-buffer approaches, as well as more intuitive graphics and games development. This article provides a view from the inside on ray tracing in games and demos, where the emphasis is on performance and short-term practical usability. It covers the way science is applied, the unique contribution of these developers to the field and their link with the research community.

The Arauna ray tracer, developed at the NHTV university of applied science, is used as an example of a ray tracer that has been specifically build with games and performance in mind. Its purpose and architecture, as well as some implementation details are presented.

Index Terms: I.3.7 [Computer Graphics]: Three-dimensional Graphics and Realism—Ray Tracing

1 INTRODUCTION

Ray tracing always has intrigued programmers of all sorts. It is an interesting algorithm: A ray tracer can be implemented in 100 lines of C code [3], yet it produces images that put those produced by rasterizers to shame. The core idea is compact, and not very hard to implement, but a naïve implementation is incredibly slow.

The ray tracing algorithm has attracted a lot of attention: If we measure the complexity of a problem by the number of papers that were written about it, ray tracing certainly is a major challenge. This challenge is not just recognized by researchers: Since the early days of the home computer, demo coders and game developers (with or without an academic background) have built their own tracers. Often with amazing results: Real-time ray tracing in a PC demo was done as early as 1995 [1]. More recently, demos like Heaven7 [5] and the RealStorm benchmark [6] displayed the capabilities of this somewhat invisible demoscene movement [24].

The importance of ray tracing for games has also been recognised by researchers [17]. Ray tracing has been used extensively for offline rendering of game graphics (e.g. Myst and Riven [36]) and for lighting preprocessing as in e.g. Quake 2 [12]. Games are also frequently used as a test-case for real-time ray tracing: At the Breakpoint 2005 demo party, a fully playable real-time ray traced version of Quake 2 was shown by Wächter and Keller [14]. Ray traced versions of animated Quake 3 [19] and Quake 4 [20] walkthroughs where shown running on a PC cluster, and even the PS3 gameconsole is already being used for ray tracing [2].

Today, it has become clear that advancing ray tracing performance is not just a matter of better high-level algorithms: Low level optimization plays a crucial role. It is also clear that ray tracing performance is not dependent of fast ray traversal alone: Shading cost plays a significant role in real-time ray tracing [21, 31].

This article presents a description of the construction of the Arauna ray tracer, which was build as an exercise in applied science and with performance in mind. The Arauna ray tracer project started as an attempt to implement the ideas presented in Wald's PhD thesis on real-time ray tracing [30]. While previous attempts at interactive ray tracing used approximations (e.g., the Render Cache [34], the holodeck ray cache [15]), the OpenRT ray tracer [23] simply sped up the core algorithm to real-time levels on commodity hardware by employing instruction-level and thread-level parallelism, as well as low level optimizations. Since those early days, Arauna evolved into an experimentation platform in order to try out the various ideas presented in recent research papers. Arauna does not claim to be groundbreaking; its aim is to combine and optimize, and, more recently, to offer a platform for students to experiment as an alternative to rasterization.

1.1 Overview

The remainder of this article is organized as follows: Section 2 details the design goals of the ray tracer, which influenced the overall design and determined the focus. Section 3 discusses implementation details of the ray tracer. Attention is payed to the high-level algorithms that were used (Section 3.1) and the shading pipeline (Section 3.2), as well the low level optimizations and their impact on the overall performance (Section 3.3). Some statistics for comparison purposes are presented (Section 4), and areas of ongoing

*e-mail: bikker.j@nhtv.nl

IEEE/EG Symposium on Interactive Ray Tracing 2007
10 - 12 September, Ulm, Germany
978-1-4244-1629-5/07/$25.00 ©2007 IEEE

research are discussed (Section 5). Section 6 finally lists open issues and provides recommendations.

2 Philosophy and Design Decisions

To understand the design decisions taken during the development of Arauna, it is necessary to explain its intended purpose. From the start, Arauna was meant to be a real-time ray tracer; later on this converged to the goal of creating a real-time ray tracing demo, and recently, the aim became to build games using it. A clear goal allows for focused development using limited resources (i.e., time). It also prevents conflicts, when performance needs to be traded for image fidelity or vice versa. The main design decisions are:

Arauna is a real-time ray tracer. It provides a simplified shading model, which is using ambient, diffuse, specular and emissive components, and an approximation to quadratic fall-off for lights, which allows us to discard lights beyond a certain distance (see Section 3.2.1). This enables scenes with many lights, an important feature for game development.

Attractive visual appearance: Ray tracing has the potential to surpass rasterization. Demonstrating and experimenting with this potential is an important goal of this project: The focus is therefore on graphical features that a ray tracer does better (or more elegantly), such as shadows, reflections and refraction, as well as finding elegant solutions for commonly used effects such as fog, glow and particles.

Target performance: Arauna aims to deliver 100M rays/s for typical game environments on a single 1.86Ghz 8-core machine. This kind of hardware is exotic right now, but within a year, this will be high-end consumer hardware. This target is inspired by GPU benchmarks such as 3Dmark06 [7], which often require the very latest 3D hardware to run reasonably well. Note that Arauna does not support network rendering to increase the number of available CPUs. While supporting this makes sense for a more generic ray tracer, it is unlikely that gamers will use such a feature.

The scenes that will be rendered are mostly static. Games in general use far more static geometry than dynamic geometry, and even though a perfect ray tracer (or a perfect rasterizer, for that matter) handles fully dynamic environments, in practice, this considerably degrades overall performance. Exploiting knowledge about the scene is essential for high performance; a high-performance ray tracer should do this as well.

Arauna is implemented in C++. Low-level optimization is an important part of the project, but this will not include hand-coded assembler. The main reasons for this are: Reduced development time, increased portability and maintainability, and easier experimentation with new ideas.

No GPU ray tracing. Developing a GPU ray tracing path is a significant effort; combining this with rasterization for first hit optimization and perhaps even CPU ray tracing severely complicates development and reduces the flexibility of the experimentation platform.

These design decisions are not presented as "the best choice", but rather as background information for the remainder of this article.

3 Implementation

The Arauna ray tracer is currently being developed and used for a student project (see Figure 1 and 12). The aim of the project is to make a proof-of-concept game based on ray tracing as the prime rendering algorithm, and to expose students to this. The student team consisted of five programmers (one of which focuses on game design) and four visual artists. The artists each have a speciality, ranging from texturing and props modeling to scene modeling. The programmers also specialize: Some focus on game coding, while others like to experiment with the ray tracing core.

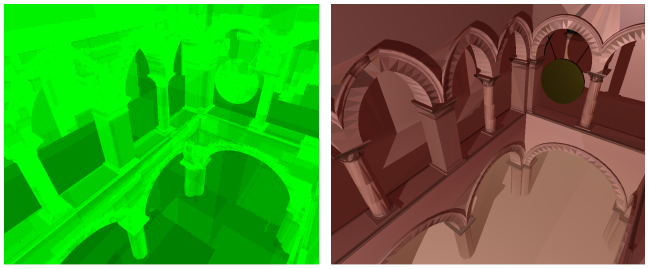


Figure 2: Visualizing kD-tree traversal depth.

Creating art for a ray tracer is different from creating art for a rasterizer. As triangle count is not really an issue, performance tends to depend on other factors instead. Because of this, the artists developed some extremely detailed objects for a game level, which did not hurt performance, but did result in severe aliasing effects caused by the many small triangles. Reducing the triangle count of these objects and using normal mapping instead solved the problem.

3.1 Accelerated Ray Tracing

Finding intersections between rays and primitives in the scene is the core process in a ray tracer. To speed up this process, an acceleration structure is used. In his thesis, Havran recommends the kD-tree for ray tracing static scenes[9]: This structure performs (on average) better than alternatives, such as BVH's, (nested) grids, octrees, and generic BSP trees.

Apart from rendering, ray tracing is also used for the game itself. Rays are used to make grenades bounce, to determine a line of sight, and to detect collisions. Compared to (conservative) polygon-based collision detection, using a ray tracer for this is efficient, even though only approximate, and easy to implement [26].

3.1.1 kD-tree Construction

Arauna uses a kD-tree based on the surface area heuristic [8, 16]. The construction of the tree is a greedy algorithm, based on Wald and Havran's notes [32]. To calculate the best split plane position at each level of the kD-tree, a sorted linked list of primitive boundaries (primitive 'start' and 'end' events, Figure 3b) must be maintained over three axes. To represent this data, a structure that is based on the primitive bounding box is used: the `EventBox` (Figure 3a). An `EventBox` contains two `next` pointers per axis, one for the start event, and another one for the end event.

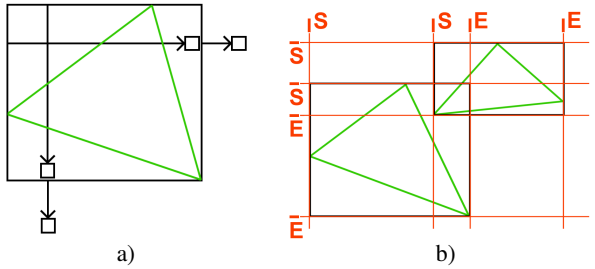


Figure 3: Constructing kD-trees. a) The `EventBox` contains a `next` pointer per axis, per side of the primitive bounding box. b) Sorting events using the `EventBox`: The start event of the first box links to the start event of the second box.

Several researchers suggest that biasing the cost calculated using the SAH improves the quality of the kD-tree (Reshetov et al. [21], Wald et al. [32]). This requires two parameters: The minimum amount of empty space that may be cut off this way, and the 'bonus' multiplier, i.e. the value that the cost is multiplied with to encourage such a split. For the first parameter, Reshetov et al. suggest a value of 10%. For the second parameter, Wald et al. suggest a value of 0.8. Both parameters are apparently determined emperically, and figures about the impact of the bias on the performance of ray queries are not given.

Tests in Arauna indicate that the performance gains of empty space cut-off for ray packet traversal are minimal. Furthermore, optimal values for the percentage of empty space and the bonus multiplier are scene-dependent and even viewpoint-dependent.

The kD-tree builder performs full clipping of primitives during tree construction [11]. This has a considerable effect on tree quality: The fact that a primitive's bounding box intersects a node does not guarantee that the primitive itself intersects the node.

The kD-tree can be visualized in real-time by representing the number of traversal steps by a color. This is shown in Figure 2. This visualization is easy to implement and provides a tool to inspect the results of the kD-tree builder.

A multi-threaded version of the kD-tree builder is also implemented. For this, a fixed number of construction threads is prepared. The threads form a chain: Each thread can activate the next, except for the last thread in the chain. Tree construction is now started by feeding the root node to the first thread. As soon as an optimal split plane is found, the construction thread activates the next one, and assigns the child containing the largest amount of primitives to it. The thread that spawned the next one proceeds with the smaller child. The new thread gets the largest job, because it may in turn spawn another construction thread, while the original one cannot do this. Construction is complete when all threads have signaled the calling thread. This process is illustrated in Figure 4. In practice, this scheme is faster than single-threaded construction, but not much: The first split takes most time, and this work is done by a single thread. After that, it is unlikely that all threads finish at the same time; several will be done while threads further down the chain are still finishing their jobs.

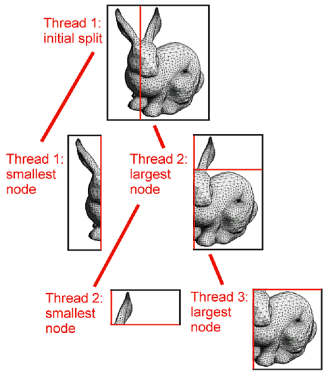


Figure 4: Scheduling threads for parallel kD-tree construction.

Using the multi-threaded version, real-time construction (a full rebuild per frame) is possible for a small amount of triangles. Other approaches exist that can rebuild larger scenes by approximating the surface area heuristic [28]. These approaches typically lead to lower quality kD-trees: E.g., Shevtsov et al. report a 30% performance degradation compared to an offline constructed high-quality kD-tree [27]. The main problem, however, is that construction time still depends on the overall scene size, instead of the number of dynamic triangles: Rebuilding the entire scene because of a simple moving actor seems like a waste. Processing static and dynamic geometry separately removes this dependency.

3.1.2 Bounding Interval Hierarchy

In 2006, an alternative acceleration structure for ray tracing dynamic scenes was presented by Wächter and Keller: The Bounding Interval Hierarchy (BIH, [29]). Around the same time, a similar structure was proposed by Woop et al. (the B-KD tree, [35]). The main benefit of these algorithms is the fast build time: Where a kD-tree for $\sim$200k triangles typically takes seconds, a BIH for the same scene is built in a fraction of a second. B-KD trees are not rebuild, but updated; this is a rapid process as well. The BIH construction can be parallelized in the same manner as the kD-tree (with the same restrictions); that way, even for moderately large scenes the structure can be rebuild each frame. The main drawback of the BIH and B-KD tree is that ray traversal is not as fast as using the kD-tree; on average the difference is 20%, but for architectural scenes, the difference is larger (up to 50%). Since most games use large amounts of static architectural geometry, using a BIH or a B-KD tree as the core acceleration structure does not seem optimal.

For this reason, Arauna is using two acceleration structures for tracing rays: A static kD-tree is used for static geometry, while the BIH is used for dynamic geometry. This idea was introduced by Parker et al., who keep dynamic primitives outside the acceleration structure, and intersect these separately [18]. Since dynamic geometry in a game typically consists of player and enemy characters, the BIH is a perfect match. The consequence of having two acceleration structures is however that each ray now has to traverse both. This may seem inefficient at first: E.g., for a scene consisting of 2000 triangles, the average number of traversal steps is 11. Where half of the triangles stored in a second tree, the number of traversal steps needed to traverse both steps is 20. In practice however, most rays will miss the primitives in the BIH, since dynamic geometry typically covers only a small portion of the screen. These rays will traverse a few empty nodes and then terminate; the time this takes proves to be negligible. More importantly: The time it takes to handle changes in the geometry now depends only on the extend of these changes, rather than the overall complexity of the scene.

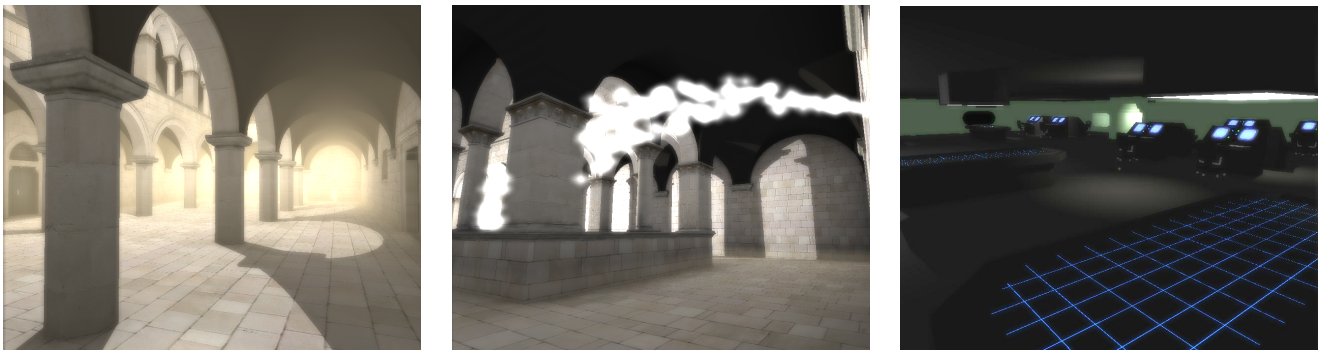
3.1.3 Tree Traversal

Once a high-quality acceleration structure is constructed, the performance of a real-time ray tracer mainly depends on efficient traversal of this structure. As shown by Wald, this process can be greatly sped up by traversing several rays at once [30]. This was later improved by using larger ray packets, or frustums (MLRTA, [21]). The effectiveness of packet traversal depends greatly on the coherence of the rays.

Arauna implements both standard packet traversal and frustum traversal. Frustum traversal requires a common ray origin, and is therefore suitable for primary rays and shadow rays (when traced from a point light source). For other secondary rays, a common origin does not exist, and so regular packet tracing is used instead.

3.2 Realtime Shading

In a real-time ray tracer, shading can easily become the performance bottleneck (Reshetov et al. [21, p.1], Wald et al. [31, p.28]), especially if more than one light source is present in the scene. Using approximation and low-level optimization, the impact of shading can be reduced. This allows the visual artists to use a complex (although fixed) shading model, that includes texture filtering and normal mapping, without sacrificing performance: On average, the shading model uses less time than ray traversal, even when many light sources are used.



a) b) c)

Figure 5: Augmenting realtime ray tracing by a) volumetric fog and b) light particles, whose visibility is determined by ray queries and c) neon glow effect using HDR bloom.

3.2.1 Approximating Quadratic Fall-Off

A game typically requires many lights to define the atmosphere in a level. To reduce the impact of a large number of light sources on the frame rate, Arauna uses a simplified lighting model, inspired by Schlick's approximation of the Phong lighting model [25]. Figure 6 shows the graphs for quadratic fall-off (solid green) and various approximations. Note how quadratic fall-off has an infinite sphere of influence, whereas the three approximations have a finite range. Figure 7 shows the shape of the light spots.

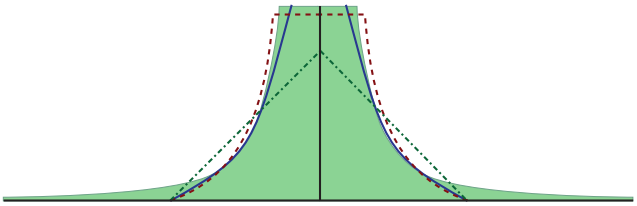
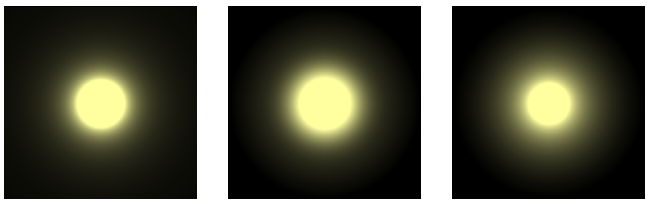


Figure 6: Various lighting models. Dotted green: linear fall-off; blue: cosine + linear fall-off; dotted red: Schlick + linear fall-off; solid green: Quadratic fall-off (for reference).



a) b) c)

Figure 7: Light spot shapes: a) Quadratic fall-off. b) Cosine + linear fall-off. c) Schlick + linear fall-off.

Both the cosine and the Schlick model approximate the original shape well enough, and allow us to quickly discard lights that are too far away. The Schlick model is used in the ray tracer, since it is computationally far less expensive than the cosine model.

3.2.2 Light Tree

Once the sphere of influence for light sources is limited to a specific radius, the number of lights that affect a point in the scene is limited (Figure 8). Determining the set of lights that affects a point can still become a bottleneck, however, since this requires calculating the distance to each light. For hundreds of lights, this is impractical. For this reason, a bounding volume hierarchy (BVH, [22]) is used for the lights in the scene. The BVH is constructed once per frame, so that the positions of all lights can be updated each frame. The BVH is constructed in a bottom-up fashion: Pairs of lights are grouped in an enclosing sphere, until the top level of the BVH is reached, which is a single sphere, containing all the light sources in the scene. The light sources that affect an intersection point can now be quickly determined by traversing the BVH. Using a BVH instead of a kD-tree (as done in [26]) solved problems with the subdivision heuristic: Sometimes, lights overlap significantly, in which case it is hard to find a good split plane position. Also, the box shape of kD-tree nodes does not match the 'shape' of a light well, which leads to considerable overhead for points that will not be lit by a light, yet are in a leaf node containing that light. BVH nodes by nature enclose the light volumes perfectly.

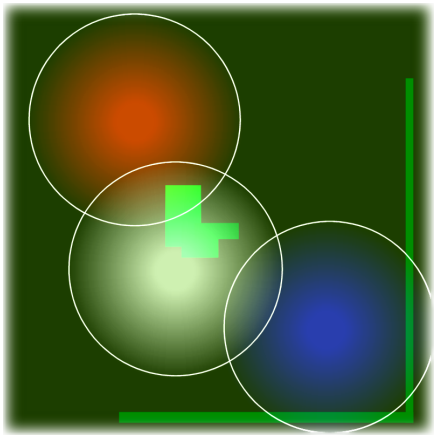


Figure 8: Overlapping lights. The bottom-right walls are lit by the blue light. The structure in the middle is lit by the white light and the red light.

3.2.3 Fog

Several other modifications have been made to the ray tracer to make it more suitable for games. The students added a layer of volumetric fog (Figure 5a): At the end of the shading stage, rays are intersected with a single horizontal plane (the surface of the fog volume). Based on the position of the intersection point along the ray and the position of the ray origin (above or beneath the fog sur-

face), the length of the ray segment that travels the fog volume is calculated. Finally, Beer's law is applied, taking into account fog color and density.

```
dist1 = MAX( 0, foglevel - rayorigin.y );
dist2 = MAX( 0, foglevel - intersection.y );
length = dist1 / raydir.y - dist2 / raydir.y;
float fog = MIN( 1, length * fogdensity );
raycolor = fog * fogcolor + (1-fog) * raycolor;
```

The fog layer is a reasonable approximation, adds a lot of atmosphere to a game scene, and in addition reduces aliasing.

3.2.4 Texture Filtering and Normal Mapping

Texture maps are sampled using a bilinear filter, which is a vast improvement over point sampling. Modern GPUs use far more advanced filters, but these put a heavy strain on the available bandwidth. Besides a texture, materials can also use a (tangent space) normal map (see Section 3.3.2).

3.2.5 Light-Particle Effects

Particles are rendered in the game by treating them as points. For these points, visibility is determined using a single ray between the camera and the particle position. Visible particles are then drawn using a 2D sprite. This only works for points, but in our game, it is quite effective. Particles are used in the game for particle effects such as fountains and fire (see for example Figure 5b), and to visualize the position of light sources.

3.2.6 HDR Effects and Neon Glow

The ray tracer operates on floating point color, and converts this to integer ARGB (32-bit) as a final conversion. On systems that support 128-bit render targets, this conversion can be omitted: The final conversion is then performed by the GPU (which is faster), and postprocessing pixelshaders can operate on the original unclamped color values. In the student project, this is used for HDR glow to emphasize bright areas. This is in turn used to render neon glow: By painting areas of textures using overbright colors (color components exceeding 1), these areas will emit an artificial glow.

3.3 Going for Performance

In this project, a relatively large amount of time was spent on low-level optimizations. Concentrating on performance is quite common for (parts of) game software, but in this case, it is especially relevant: If the goal is to match GPU performance and image fidelity, the ray tracing algorithm is still too demanding for modern high end consumer hardware. Hurley [10] estimated that 450 'ray segments' (i.e., traversal steps; he assumes an average of segments per ray) are needed to make real-time ray tracing 'interesting'. However, to make ray tracing a viable alternative to rasterization, something closer to 300M rays/s and 20 segments per ray is needed to make the ray tracer a viable alternative to rasterization (Figure 9): Gamers are used to (at least) 1280x800 pixels and 30 frames per second; 10 rays per pixel are needed to allow for shadow rays and other secondary rays.

On current high-end hardware, this performance level cannot be achieved. An eight-core machine running at 1.86Ghz delivers real-time performance (30fps @ 800x600) for moderately complex scenes (∼80k triangles), some dynamic triangles (∼5k) and a large amount of lights (see Section 3.2.2). To achieve this, the ray tracer has been designed with performance in mind on a high level. The previous section described the decisions in this regard. Besides that, low-level optimizations have a significant impact.

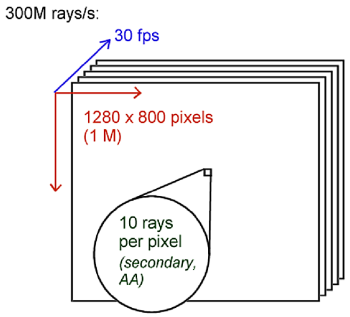


Figure 9: Given a performance of 300M rays/s, 30 frames per second at 1280×800 pixels leave 10 rays per pixel for anti-aliasing and secondary effects.

3.3.1 Parallelism

Modern processor architectures require code that is highly parallel, both on a high level (thread-level parallelism) and on a low level (instruction-level parallelism). Arauna exploits both. On a high level, rendering is evenly distributed over a number of rendering threads. To divide the work evenly, the screen is subdivided in 32x32 pixel tiles: Each tile is a 'task', an atomic workload for a rendering thread. During rendering, the rendering threads take tasks from the task stack and execute them, until no tasks remain.

A few practical notes about this process:

- The number of rendering threads is equal to the number of available cores, and each rendering thread is tied to a single core.
- During rendering, when a thread needs a new task, it checks the stack pointer, retrieves the index of a waiting task using an interlocked decrement of the stack pointer (this is hardware supported), checks if the returned task number is valid, and executes the task. Using this simple scheme, no operating-system-assisted synchronization is needed during actual rendering, which considerably reduces threading overhead.
- When no more tasks are available, the rendering thread signals the main thread, and goes to sleep. The main thread waits for all rendering threads to complete, before it returns control to the host application.

There is one point where this process may not optimally use all available cores: While rendering the last task, one rendering thread will at some point be the only active rendering thread. If the last 32x32 pixel tile happens to be a very expensive tile, a number of cores will be idle for quite some time. In a typical game scene, these cases are rare, since deep recursion is generally prevented by careful object placement. At a 1024x768 screen resolution, it is therefore unlikely that the time to render the most expensive tile will be a significant portion of the overall rendering time.

3.3.2 SIMD

Besides thread level parallelism, instruction level parallelism is used to increase application performance. In Arauna, SIMD is used for ray setup (especially normalization), ray (packet) traversal, ray (packet) / triangle intersection, shading, and converting the final pixels to 32bit ARGB. Some of these are well documented, especially SIMD packet traversal and triangle intersection. Arauna uses a fixed shading path, and is therefore able to effectively use SIMD code for shading as well.

The shading path consists of the following steps:

1. Texture fetch with bilinear filtering,

2. Normal map lookup with bilinear filtering,
3. Transforming normal from normal map into object space,
4. Lighting model: Ambient, diffuse, specular with linear falloff,
5. A single fog volume, and
6. Conversion to 32bit ARGB,

where the final step is only executed for primary rays in order to display the final result and can be omitted if the render target is 128-bit (32-bit floating point per component).

To reduce the impact of cache misses, materials that have both a texture and a normal map store the image data in an interleaved fashion: Each texel is followed by the corresponding (tangent space) normal. A texel/normal pair is now exactly 32 bytes, and thus fits in a single L1 cache line; fetching the texel ensures that the normal data can be accessed without delay. The result is that normal mapping is fast enough to allow the artists to use it liberally. The drawback of this approach is that the normal map must have the same resolution as the texture, and that materials with the same texture but a different (or no) normal map cannot share texture data.

Texels and normals are fetched for four rays at a time. The bilinear filter requires four texels per ray; a total of 16 texels is thus needed. In case of oversampling or limited undersampling (i.e., a sampling rate >25%), several of these are actually the same texel, further reducing cache misses. Below that point, each ray will sample four points that are not shared with adjacent rays. The fetched texels (in ARGB format, 128-bit) are then converted to RRRR, GGGG, BBBB (alpha is discarded), so that the layout of the 'color component quads' matches other data, which is stored for four rays at a time.

Note that undersampling and oversampling only applies to texture sampling: The ray tracer will always trace exactly one primary ray per screen pixel.

3.3.3 SSE Tricks

When working with four rays at a time, SIMD instructions can be used to speed up several common but expensive operations, such as the normalization of vectors. One of the SSE2 instructions that is provided specifically for this task is the _mm_rsqrt_ps instruction: Using a hardware look-up table, a reciprocal square root is calculated for 4 numbers in a few processor cycles. The drawback of this instruction is its accuracy: The look-up table supports only 22 bits of accuracy. A more accurate result can be obtained using a single Newton-Raphson iteration.

```
__m128 fastrsqrt( const __m128 v )
{
const __m128 nr = _mm_rsqrt_ps( v );
const __m128 muls = _mm_mul_ps( _mm_mul_ps( v, nr ), nr );
return _mm_mul_ps( _mm_mul_ps( _half, nr ),
                        _mm_sub_ps( _three, muls ) );
}
```

The `_three` is the vector $(3,3,3,3)$ and `_half` is the vector $(0.5,0.5,0.5,0.5)$. For calculating the reciprocal of a square root, this function is 8.2 times faster than a regular `1/sqrtf(x)` scalar version.

Similarly, the reciprocal of four values of a vector can be calculated faster:

```
__m128 safercp( const __m128 v )
{
const __m128 nr = _mm_rcp_ps( v );
const __m128 muls = _mm_mul_ps( _mm_mul_ps( nr, nr ), v );
return _mm_sub_ps( _mm_add_ps( nr, nr ),
_mm_andnot_ps( _mm_cmpeq_ps( v, _zero ), muls ) );
}
```

The vector `_zero` is $(0,0,0,0)$ and the masking operation in the last line sets results of a division by zero to 0. For calculating reciprocals, this function is 4.94 times faster than a scalar reciprocal, and has a maximum error of $1.4e10^{-7}$. Just using the `_mm_rcp_ps` intrinsic is even faster, but this has a maximum error of $0.3e10^{-4}$.

3.3.4 Other Low-level Optimizations

Besides parallelism and platform-specific SIMD optimizations, a large number of low-level optimizations influence the quality of the code that is generated from the C++ source files.

Inlining of functions in general improves performance. In many cases, the compiler will already inline suitable functions; in some other cases, the compiler will ignore the `inline` keyword, as it is only a hint. Especially some larger functions that are merely separated from the calling function to increase readability may end up as a function, with the associated function call overhead, even if the user requests such a function to be inlined. Several compilers support the `__forceinline` directive; using this guarantees that the function is inlined as requested.

Expanding variables to vectors: Since many calculations are performed on four rays at a time, many scalar values need to be expanded to a vector to be used in the calculations. By storing data like the position and radius of a light source in the expanded form as well as the original scalar, these calculations are greatly sped up. For the same reason, light and fog color values are stored as RRRR, GGGG, BBBB.

Memory management: Besides aligning objects to cache lines, the memory manager is used to efficiently handle allocation of small objects. These objects are always allocated in groups, so that similar objects are grouped in memory. The memory manager uses simple arrays for objects that will not be recycled (e.g., nodes for the static kD-tree), and linked lists for objects that can be recycled (e.g., nodes for the BIH). Using the linked list, allocation is now simply a matter of getting the first element from a linked list; to recycle an object, it is placed back at the start of the list.

Cache line alignment: All classes and structures are padded with dummy data to make their size a multiple of 32. Allocation of such objects is always done through a custom memory manager, which ensures that the address of each object is a multiple of 32.

Using the `const` keyword is an important hint to the compiler about the intended use of a variable. Declaring all suitable variables and functions 'const' can improve the performance of an already optimized ray tracer by 10

Regarding the `const` keyword: An application that has been modified to maximize the number of constant variables will perform better, even when the `const` keyword is removed afterwards. This is caused by the fact that the modifications that have to be made already make the code 'easier' on the compiler. One example of this:

```
float squared = x * x;
squared += y * y;
float distance = sqrt( squared );
distance = 1.0f / dist;
```

This calculates the reciprocal of the length of vector (x,y). Usually the variable `distance` will not be changed after this, so it can be declared constant. All other variables can also be constant, if some intermediate variables are used:

Table 1: Arauna performance in MRays/s and milliseconds on a dual quad core Xeon system running at 1.86Ghz.

Scene	Triangles	Lights	Shading	MRays/s min.	MRays/s max.	Time max	Time min
Teapots	11040	1	basic	77.3	115.2	26ms	18ms
Teapots	11040	1	full	55.7	96.3	37ms	22ms
Sponza	67324	1	basic	49.7	112.7	42ms	18ms
Sponza	67324	1	full	22.9	87.4	92ms	23ms
KA27	185060	2	basic	33.6	65.9	62ms	31ms
KA27	185060	2	full	31.7	55.2	99ms	57ms
Packard	362900	2	basic	31.8	67.4	68ms	30ms
Packard	362900	2	full	23.8	50.5	84ms	54ms

```
const float sqred_x = x * x;
const float sqred_y = y * y;
const float dist = sqrt( sqred_x + sqred_y );
const float distance = 1.0f / dist;
```

Now the 'constness' of each variable can easily be detected by the compiler, even if we do not explicitly state the `const` keyword.

4 Statistics

The ray tracer was tested for several scenes with varying numbers of triangles and shading complexity. The results are listed in Table 1. About the scenes used: Sponza is a moderately complex architectural model, with complex shading: Almost all materials use a detailed texture and a normal map. The 'teapots' scene is a simple scene, with basic materials. KA27 is a detailed model of a Russian helicopter. Packard, finally, is a highly detailed model of a car, but again with basic materials.

The tests where run on a dual-processor Xeon machine (in total 8 cores), running at 1.86Ghz, at a screen resolution of 1024x1024 pixels. A peak performance of 115 MRays/s is reached for a simple scene, with basic lighting.

As expected, ray tracing performance is rather insensitive to scene complexity. The difference between Sponza and Packard is slightly misleading in this respect: Sponza uses much more time on shading, so the difference in ray traversal time alone is larger. The shading model itself takes between 10% and 30% of the overall render time, depending on material complexity: Texturing is costly. Not mentioned in this table is the overhead of normal mapping: In the Sponza scene test, disabling normal mapping decreases shading time by about 15%.

Adding 2500 dynamic triangles to the Sponza scene increases the frame time by 4ms, where 1 millisecond is used to build the BIH, 2 milliseconds are due to the visible dynamic triangles, and the overhead of checking all rays against the BIH is about 1 millisecond. Increasing the number of dynamic triangles confirms this: For 10k dynamic triangles, BIH construction time is about 4ms, and the overall impact still largely depends on the number of visible dynamic triangles.

Table 2: Multi-threading scalability on a 1.86Ghz system.

Cores	Render time (ms)	MRays/s	x 1 Core
1	359.2	5.85	1.00
2	185.2	11.36	1.94
4	96.4	21.8	3.73
6	68.9	30.7	5.25
8	55.9	37.7	6.44

The scalability of rendering using many cores has been tested on the same system. The results are shown in Table 2: These numbers indicate that beyond eight cores, scalability will become a problem.

5 Future Plans

The Arauna ray tracer now provides sufficient performance for games on an 8-core PC. However, a lot of work remains to be done. The intention is to keep focusing on performance, but besides that, there is now room to increase image fidelity. The most obvious way to do this is by improving the shading model: It is especially attractive to explore the possibilities of global illumination, and approximations such as ambient occlusion [37].

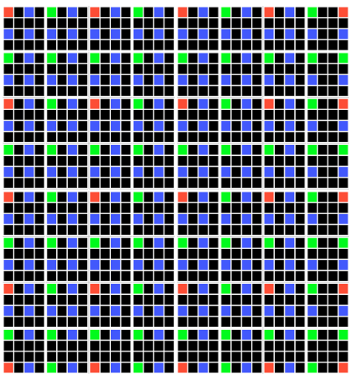


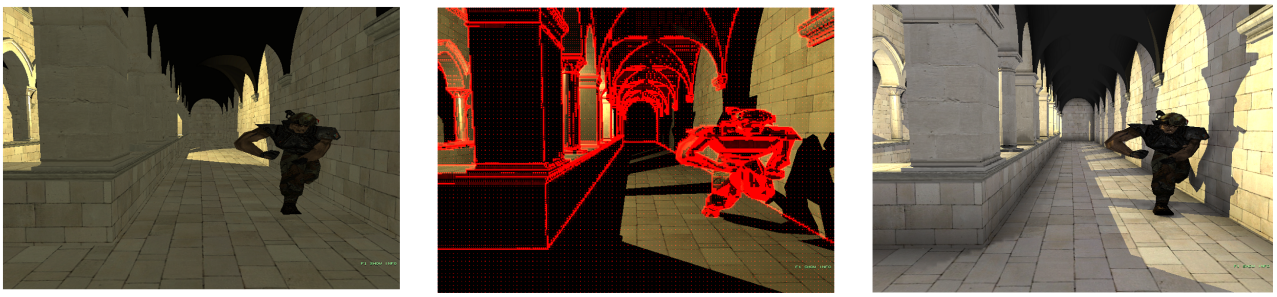
Figure 10: Hierarchical grid used for adaptive ambient occlusion sampling. Red dots form the top-level of the hierarchy; green and blue dots are subsequent subdivisions.

5.1 Ambient Occlusion

A first attempt at an efficient ambient occlusion scheme has been made: For this, a hierarchical version of the discontinuity buffer [13, 33] is implemented. For a 32x32 tile, ambient occlusion samples are first taken for a limited number of points. The grid that is used for this is shown in Figure 10: The red dots form the top-level of the hierarchy (5x5 samples). The pixels between dots identical color are shaded by bilinear interpolation of the ambient occlusion values for the red dots, unless a discontinuity is detected:

- The ambient occlusion values for 4 grid points differ significantly, or
- the intersection points of the primary rays for the 4 grid points are not planar, or
- the normals of the geometry at the 4 grid points differ significantly.

Each of these three conditions can be made more or less strict; this way, a balance between speed and accuracy is chosen. In case of a discontinuity, the top-level grid is subdivided, and the process is repeated for the green dots (9x9 samples), the blue dots (16x16



a) b) c)

Figure 11: Ambient occlusion with a hierarchical discontinuity buffer: a) Basic shading, b) sampling points found, and c) the final result shaded in realtime.

samples) and finally the black dots, in which case interpolation is not used at all. Omitting the last subdivision is acceptable for most applications; this reduces the maximum sampling of the ambient occlusion to 2x2 pixels, with interpolation.

Note that the subdivided tiles are not equally sized. This is a consequence of the otherwise convenient tile size (32x32): Since samples at the edges are needed, 33x33 would have been needed for a regular subdivision. This irregularity does not lead to visible artifacts in the final result; it does however complicate the implementation.

The outlined approach effectively finds discontinuities in the geometry that is detected by the primary rays (Figure 11b), and reduces the number of ambient occlusion samples significantly. In areas where no discontinuities are detected, interpolation results in a good approximation of the low-frequent ambient occlusion (Figure 11c). Further performance improvements are achieved by limiting the maximum range of the ambient occlusion rays. This way, ambient occlusion quality and speed can be balanced.

As mentioned, The discontinuity buffer operates on the 32x32 pixel tiles that each rendering thread renders. Because it is limited to the pixels of a tile, it can be part of the task executed by the rendering threads: Unlike the original discontinuity buffer, it thus runs in parallel if multiple cores are available. The disadvantage is that samples at the edges of the tile are adjacent to samples on neighboring tiles. The samples are thus not evenly distributed over the screen.

The same buffer has also been applied to soft shadows. This does not work well: Area lights cause high frequency changes in illumination near occluders, and the grid will often miss these, leading to objectionable artifacts.

So far, the results of this approach has been somewhat disappointing. This is related to some key issues in real-time ray tracing: The first is the problem of divergent rays. The ambient occlusion rays are extremely divergent by nature, as they are uniformly distributed over a sphere. This makes ray packet traversal impractical, but even single ray traversal suffers: Since each ray traverses vastly different regions of the scene, frequent cache misses reduce its performance. The same problems occur with e.g. reflections from normal mapped or curved surfaces. The other problem is the near-impossibility of combining screen space techniques with recursive ray tracing. While the discontinuity buffer can effectively reduce the number of ambient occlusion samples, it uses the results of primary rays for this; this is not possible when scenery is seen indirectly. The use of so called 'deep buffers' may solve this problem: In this case, a discontinuity buffer per recursion level could be used.

5.2 Incoherent Secondary Rays

As mentioned in Section 2, demonstrating the potential of ray tracing is an important goal. Much of this potential is coupled to secondary effects. Of these, only shadows can be efficiently handled using ray packet tracing. Finding ways to improve the performance of reflection and refraction is therefore important.

One approach that looks interesting is to intersect each ray with four triangles at a time, instead of the commonly used method of intersection four rays with a single triangle. Rays may still traverse in packets for frusta, but single rays will now at least exploit instruction level parallelism for triangle intersection. In their Master's thesis, Bonnedal and Pettersson state an overall speedup of 1.3 to 2 compared to single ray traversal / triangle intersection [4]. This may require significant changes to kD-tree construction: Leaves should contain (a multiple of) four primitives, and the average expected cost of triangle intersection should be reduced, leading to smaller trees and larger leaves.

6 Open Issues

There are several areas that need more research:

1. Divergent rays: The tremendous speed-up that is obtained by bundling rays in packets appears to primarily benefit primary rays and shadow rays. Finding coherence in a 'ray soup' is needed to speed up all rays.

2. Reducing the number of rays: Several features that considerably affect image quality, such as soft shadows, global illumination, and anti-aliasing require many rays. Finding efficient schemes to reduce the number of rays will let ray traced games use more realistic graphics.

3. The transition of rasterization to ray tracing: Finding ways to gradually introduce ray tracing in games is needed to 'educate' both gamers and game developers, without requiring large investments in ray tracing hardware.

4. An efficient programmable shading model: The Arauna shading model is fixed and limited, while the ray tracing algorithm in principle offers an elegant way to support complex effects that involve many secondary rays.

Many of these are typical research tasks; others (such as the programmable shaders) simply require an efficient implementation, making optimal use of the available hardware. Advancing in these areas would greatly benefit from a closer relation between academic researchers and game developers, both in a formal setting and in more informal ways.

ACKNOWLEDGEMENTS

Many of the techniques applied in the Arauna ray tracer have originated from or discussed on the ompf.org forum. Detailed low-level optimization tips where provided by Thierry Berger-Perrin, webmaster of the ompf forum (http://www.ompf.com/forum).

Several game-specific features of the ray tracer are the idea of NHTV/IGAD students: The volumetric fog and the BVH for lights where implemented by Rutger Janssen; texture glow was proposed by Jan Pijpers. The GL1/RTRT project team consists of the following students: Mike van Mourik (lead), Erik Verboom (design), Frans Karel Kasper (coding and design), Rutger Janssen and Wilco Schroo (engine), Jan Pijpers, Ramon Plaisier and Titus Lunter (modelling), Pablo van den Homberg (props) and our special team member: Octopussy, our friendly 8-core.

The author wishes to thank Carsten Wächter, Alexander Keller and Per Christensen for proofreading and useful comments.

REFERENCES

[1] Abaddon. Chrome Real-time Ray Tracing Demo. Presentation at the Assembly 95 demo party, Helsinki, Finland, 1995.

[2] C. Benthin, I. Wald, M. Scherbaum, and H. Friedrich. Ray Tracing on the CELL Processor. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 15–23, 2006.

[3] T. Berger-Perrin. The Sphereflake, in 100 lines of c code. Website: http://ompf.org/ray/sphereflake/, with link to source code, 2005.

[4] R. Bonnedal and M. Pettersson. Master thesis: SIMD Accelerated Ray Tracing, 2002.

[5] Exceed. Heaven seven. Presentation at the Mekka and Symposium demo party, 2000.

[6] Federation Against Nature. RealStorm Benchmark 2004. Website: http://www.realstorm.com.

[7] Futuremark Corporation. 3dmark06. Website: http://www.futuremark.com.

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

[9] V. Havran. *Heuristic Ray Shooting Algorithms*. PhD thesis, Czech Technical University, Praha, Czech Republic, 2001.

[10] J. Hurley. Ray Tracing goes Mainstream. *Intel Technology Journal*, 9(2), 2005.

[11] J. Hurley, R. Kapustin, A. Reshetov, and A. Soupikov. Fast Ray Tracing for Modern General Purpose CPU. In *Proceedings of Graphicon*, pages 255–261, 2002.

[12] ID Software. Quake 2. Website: http://www.idsoftware.com, 1997.

[13] A. Keller. *Quasi-Monte Carlo Methods for Photorealistic Image Synthesis*. Ph.D. thesis, Shaker Verlag Aachen, 1998.

[14] A. Keller and C. Wächter. To Trace or Not To Trace, That is the Question. Presentation for the Breakpoint 2005 demo party seminar, 2005.

[15] G. W. Larson. The Holodeck: A Parallel Ray-caching Rendering System. *Proceedings of the Second Eurographics Workshop on Parallel Graphics and Visualisation*, pages 17–30, Sept. 1998.

[16] J. MacDonald and K. Booth. Heuristics for Ray Tracing using Space Subdivision. *The Visual Computer*, 6(3):153–166, June 1990.

[17] J. A. Oudshoorn. Ray Tracing as the Future of Computer Games, 1999.

[18] S. Parker, W. Martin, P. Sloan, P. Shirley, B. Smits, and C. Hansen. Interactive Ray Tracing. In *Symposium on Interactive 3D Graphics*, pages 119–126. ACM SIGGRAPH, 1999.

[19] D. Pohl. Quake 3 Ray Traced. Website: http://graphics.cs.uni-sb.de/ sidapohl/egoshooter/, 2004.

[20] D. Pohl. Quake 4 Ray Traced. Website: http://www.q4rt.de, 2007.

[21] A. Reshetov, A. Soupikov, and J. Hurley. Multi-Level Ray Tracing Algorithm. *ACM Transactions on Graphics (ACM SIGGRAPH 2005 Conference Proceedings)*, 24(3):1176–1185, 2005.

[22] S. M. Rubin and T. Whitted. A 3-Dimensional Representation for Fast Rendering of Complex Scenes. *Computer Graphics (Proceedings of SIGGRAPH '80)*, 14(3):110–116, 1980.

[23] SaarCOR. The Openrt Real-time Ray Tracing Project. Website: http://www.openrt.de.

[24] V. Scheib, T. Engell-Nielsen, S. Lehtinen, E. Haines, and P. Taylor. The Demo Scene. In *Conference Abstracts and Applications (SIGGRAPH '02)*, pages 96–97, New York, NY, USA, 2002. ACM Press.

[25] C. Schlick. A Customizable Reflectance Model for Everyday Rendering. In *Rendering Techniques '93 (Proceedings of the Fourth Eurographics Workshop on Rendering)*.

[26] J. Schmittler, D. Pohl, T. Dahmen, C. Vogelgsang, and P. Slusallek. Realtime Ray Tracing for Current and Future Games. In P. Dadam and M. Reichert, editors, *34. Jahrestagung der Gesellschaft für Informatik*, volume 50 of *LNI*, pages 149–153. GI, 2004.

[27] M. Shevtsov, A. Soupikov, and A. Kapustin. Highly Parallel Fast KD-tree Construction for Interactive Ray Tracing of Dynamic Scenes. In *Proceedings of Eurographics*, volume 26, page to appear, 2007.

[28] G. S. W. Hunt, W.R. Mark. Fast kd-tree Construction with an Adaptive Error-Bounded Heuristic. *IEEE Symposium on Interactive Ray Tracing*, pages 81–88, 2006.

[29] C. Wächter and A. Keller. Instant Ray Tracing: The Bounding Interval Hierarchy. In T. Akenine-Möller and W. Heidrich, editors, *Rendering Techniques '06 (Proceedings of 17th Eurographics Symposium on Rendering)*, pages 139–149, 2006.

[30] I. Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Saarland University, 2004.

[31] I. Wald, S. Boulos, and P. Shirley. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics (SIGGRAPH 2007 Conference Proceedings)*, 26(1):6, 2007.

[32] I. Wald and V. Havran. On building fast *k*D-trees for Ray Tracing, and on doing that in O(N log N). *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 61–69, 2006.

[33] I. Wald, T. Kollig, C. Benthin, A. Keller, and P. Slusallek. Interactive Global Illumination using Fast Ray Tracing. In P. Debevec and S. Gibson, editors, *Rendering Techniques 2002 (Proceedings of the 13th Eurographics Workshop on Rendering)*, pages 15–24, 2002.

[34] B. Walter, G. Drettakis, and S. Parker. Interactive Rendering using the Render Cache. In D. Lischinski and G. Larson, editors, *Rendering techniques '99 (Proceedings of the 10th Eurographics Workshop on Rendering)*, volume 10, pages 235–246, New York, NY, Jun 1999. Springer-Verlag/Wien.

[35] S. Woop, G. Marmitt, and P. Slusallek. B-KD Trees for Hardware Accelerated Ray Tracing of Dynamic Scenes. In *Proceedings of Graphics Hardware*, pages 67–77, 2006.

[36] C. Worlds. Myst PC Game. Website: http://www.cyan.com, 1993.

[37] S. Zhukov, A. Iones, and G. Kronin. An Ambient Light Illumination Model. In *Rendering Techniques '98 (Proceedings of the Eurographics Workshop on Rendering)*, pages 45–55, 1998.

CURRICULUM VITAE: JACCO BIKKER

Jacco Bikker is a lecturer for the International Architecture and Design course of the University of Applied Sciences, Breda, The Netherlands. Before that, he worked in the Dutch game industry for ten years, for companies such as Lost Boys Interactive, Davilex, Overloaded PocketMedia and Woedend!Games. Besides his job, he wrote articles on topics such as ray tracing, rasterization, visibility determination, artificial intelligence and game development for developer websites such as Flipcode.com and Gamasutra.

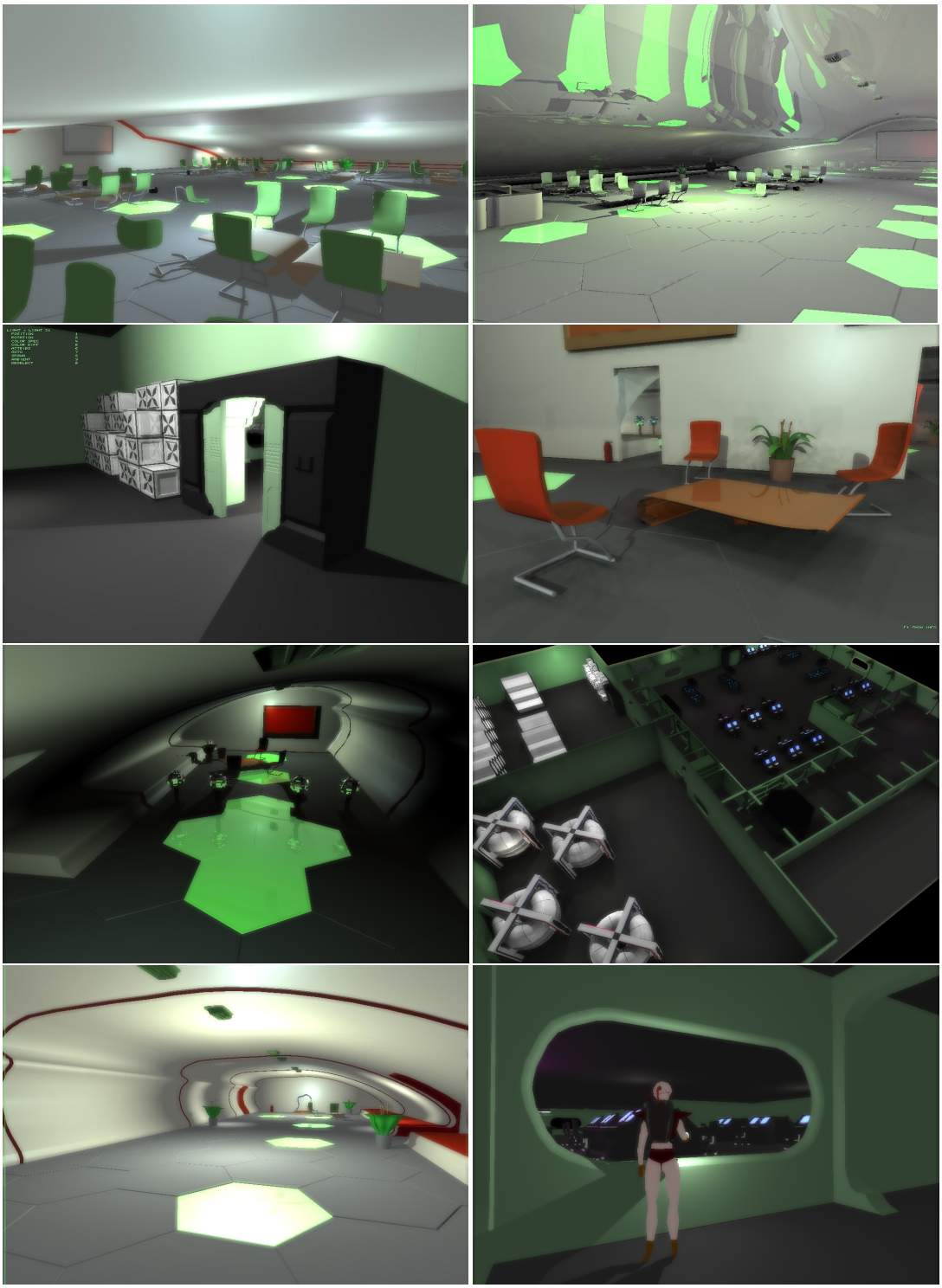


Figure 12: More images taken from a game implemented on the Arauna real-time ray tracing platform.