Technical Report TR-2002-01, Computer Graphics Group, Saarland University

A Flexible and Scalable Rendering Engine for Interactive 3D Graphics

Ingo Wald Carsten Benthin Philipp Slusallek

Computer Graphics Group, Saarland University

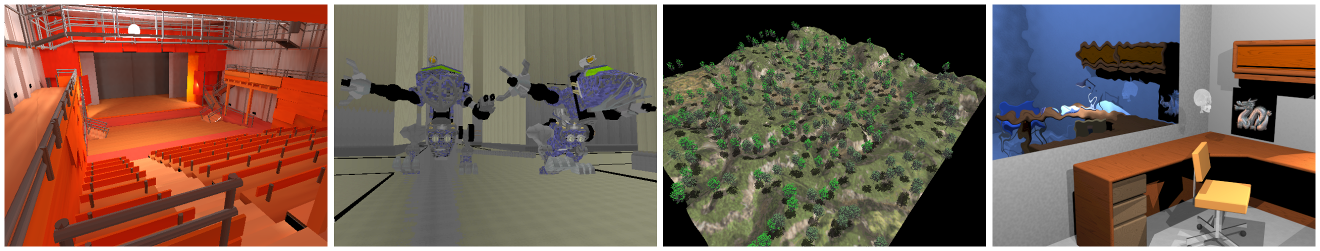

Figure 1: Examples of interactively rendering complex and dynamic scenes with a ray-tracing-based renderer. The scenes show a pre-lighted theatre, robots moving through a city, large numbers of moving trees with sharp shadows, as well as the integration of volumes, lightfields, and procedural shading in an office environment. These examples run interactively at a resolution of 640 × 480 using four to eight dual PCs.

Abstract

Ray-tracing is well-known as a general and flexible rendering algorithm that generates high-quality images. But in the past, ray-tracing implementations were too slow to be used in an interactive context. Recently, the performance of ray-tracing has been increased by over an order of magnitude, making it interesting as an alternative to rasterization-based rendering.

We present a new rendering engine for interactive 3D graphics based on a fast, scalable, and distributed ray-tracer. It offers an extended OpenGL-like API, supports interactive modifications of the scene, handles complex scenes with millions of polygons, and scales efficiently to many client machines. We demonstrate that the new renderer provides more flexibility, more rendering features, and higher performance for complex scenes than current rasterization hardware. Its flexibility enables new types of applications including a system for interactive global illumination.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing, Animation, Color, shading, shadowing, and texture; I.3.2 [Graphics Systems]: Graphics Systems—Distributed/network graphics; I.3.3 [Graphics Systems]: Picture/Image Generation—Display algorithms; I.6.8 [Simulation And Modeling]: Types of Simulation—Animation, Parallel ;

Keywords: Animation, Distributed Graphics, Occlusion Culling, Ray Tracing, Reflectance & Shading Models, Rendering Systems, Volume Rendering

1 Introduction

The ray-tracing algorithm is well-known for its ability to generate high-quality images but it is also famous for its long rendering times. Speeding up ray-tracing so it can be used for interactive applications has been a long standing goal for computer graphics research. Significant efforts have been invested, mainly during the 1980ies and early 90ies, as documented for example in [Glassner 1989; Chalmers and Reinhard 1998].

The interest in fast ray-tracing technology increased again after interactive ray-tracing has been shown on large supercomputers [Muuss 1995; Parker et al. 1999b] as well as on networks of PCs [Wald et al. 2001a; Wald et al. 2001b]. The latter research increased the performance of ray-tracing by more than an order of magnitude on a single processor and scales well to many distributed rendering clients.

These recent developments made it desirable to evaluate ray-tracing as an alternative rendering technique for interactive 3D applications besides the well-established rasterization pipeline. This paper provides several contributions in this direction.

1.1 Contributions

We have built a complete, flexible, fast, and distributed *rendering engine* that offers services on a similar level as OpenGL but provides significantly more flexibility and features. The engine achieves a speed-up by more than an order of magnitude compared to other software ray-tracers and allows efficient distributed ray-tracing without replicating the data to each client. Both contributions rely mainly on effective memory management through caching and prefetching, in combination with algorithmic improvements that allow more efficient execution on today's processors (Section 2).

Next we present a new technique that allows the ray-tracing engine to handle *interactive changes to the scenes*. It is based on the idea to localize changes to the scene so they can be integrated efficiently into the index structures, such as octrees, BSP trees, etc., required for fast ray-tracing (Section 3).

We also describe the new *OpenRT API* we developed to provide a common interface for applications to communicate with a ray-tracing-based renderer. A basic requirement was to keep the API as close as possible to OpenGL in order to simplify porting of existing applications. The approach is demonstrated by an interactive VRML-browser that has been ported from OpenGL to OpenRT. We also show the support for dynamic scenes by using VRML-animations (Section 4).

In the second part of the paper we *focus on the consequences* of using an interactive ray-tracing engine. We discuss and demonstrate a list of *benefits* provided by ray-tracing and compare it with rasterization-based rendering (Section 5).

Next, we present two *new applications* that make effective use of the unique features provided by ray-tracing. The first application demonstrates the simplicity of *integrating different rendering primitives* such as surfaces, volume data sets, and lightfields within a single scene. All the expected effects are automatically and correctly rendered without any special support from the application (Section 6.1.3).

Finally, we present the first *interactive global illumination application*, which uses the rendering engine for a fast and efficient computation of light transport. It uses a new Monte Carlo algorithm that is designed to work well within the constraints imposed by a fast, distributed ray-tracing engine (Section 6.2).

2 A Fast Ray-Tracing Engine

The goal of our research was to design a rendering engine based on ray-tracing for interactive 3D graphics. The design was guided by the following list of requirements.

Interactive Performance: We aimed for a minimum framerate of 5 frames per second (fps) at a resolution of at least 640×480 (video resolution) even for complex scenes while running completely in software.

Plug-in Shaders: Much of the power of ray-tracing comes with the flexibility to easily change the appearance of geometry through the use of programmable shading. The goal was to offer a well-defined interface for shaders that allows them to replace as much functionality in the renderer as possible while keeping interactive performance.

Distributed Computing: Since a single CPU would in general still be too slow, we required transparent support of distributed ray-tracing on a network of workstations with a client-server approach. This poses interesting challenges in the context of dynamic scenes.

Dynamic Scenes: True interactivity requires the ability to perform arbitrary modification to the scene during rendering, which has been a problem with ray-tracing due to the use of pre-computed index structures. The goal was to allow for as general modifications as possible without compromising performance.

API: A new rendering engine must offer a standard API for application developers that is familiar to them, easy to use, and exposes all the interesting features of the ray-tracer. Because the familiar OpenGL API could not be used directly our goal was to stay as close as possible to OpenGL in order to enable easy porting.

Commodity Equipment: We restricted our implementation to commodity computer equipment, such a dual-processor PCs with fast CPUs as well as inexpensive Fast- or Gigabit-Ethernet for the network interconnection. With this setup each client node would cost less than $2500 today with the expected increase in performance over time according to Moore's law.

2.1 Previous Work

Even though ray-tracing has been around for some time [Whitted 1980; Cook et al. 1984], its use for interactive applications is relatively new.

Parker et al. [Parker et al. 1999b; Parker et al. 1999a; Parker et al. 1998] and Muuss [Muuss and Lorenzo 1995; Muuss 1995] demonstrated that interactive rendering can be achieved with a full-featured ray tracer on a large supercomputer. Parker's implementation offers all the usual ray tracing features, including parametric surfaces and volume objects, but is carefully optimized for cache performance and parallel execution in an environment with non-uniform but fast access to shared memory. They have proven that ray tracing scales well in the number of processors, and that even complex scenes of several hundred thousand primitives can be rendered at almost real-time framerates. However, this system required expensive hardware that is not commonly available.

Hardware implementations of ray-tracing are available [Advanced Rendering Technologies 1999], but are currently limited to accelerating off-line-rendering applications, and do not target interactive frame rates.

There has been a tremendous amount of previous work on parallel and distributed ray tracing in general. Detailed surveys can be found in [Chalmers and Reinhard 1998; Reinhard et al. 1998]. We build on this previous work and combine several approaches in order to obtain the required interactive performance.

2.2 Ray-Tracing Core

The core of our rendering system is based on a re-implementation of the coherent ray-tracing algorithm by Wald et al. [Wald et al. 2001a]. It achieves more than an order of magnitude increase in ray-tracing performance compared to other published ray-tracing results. This is accomplished through reordering of the ray-tracing computations for better coherence and a number of optimization techniques that work in concert to achieve maximum performance.

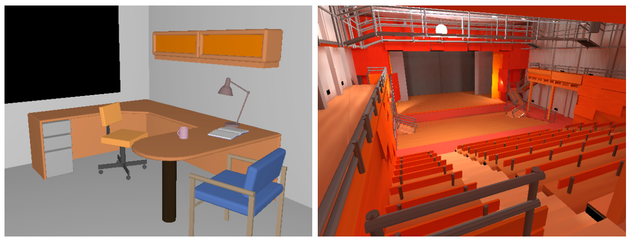
Figure 2: Example scenes rendered with 640×480 pixels on a single CPU: The scene on the left contains 33.600 triangles and renders at 2 frames per second (fps), while the theatre scene with more than half a million pre-lighted triangles still renders at roughly 1 frame per second.

It is important to note that the optimized algorithm does not use any approximations to achieve this speedup. It still performs exactly the same operations on each ray as a basic ray-tracer.

The new core has a more modular structure that allows to replace any code computing the reflection of light at surfaces, sampling of light sources, generating rays from a camera, and processing pixel data. The plug-in code is implemented through shaders that can be dynamically loaded at run-time. Care has been taken that the additional flexibility causes only a small performance decrease in the order of 10%. The additional flexibility of shader plug-ins is instrumental for the results presented in this paper.

2.2.1 Results

Using a single AMD AthlonMP 1800+ CPU, we are able to trace more than half a million rays per second, resulting in a performance of more than 2 frames per second (fps) at video resolution of 640 x 480 pixels for most of our basic test scenes, and still more than 1 fps for highly complex models (see Figure 2). This performance is comparable to the results in [Wald et al. 2001a] given the different hardware.

The performance can simply be doubled by using both processors on a dual workstation, but cannot easily be scaled much beyond that, because the price/performance ratio increases quickly for larger shared memory computers. Instead, we rely on a network of workstations to deliver the necessary performance.

2.3 Interactive Distributed Ray-Tracing

As mentioned above ray-tracing scales nicely with the number of processors if fast access to the scene data is provided, e.g. by using shared memory [Parker et al. 1999b]. The situation is quite different with many clients in a distributed memory environment, which has an impact on the design of the rendering engine and its support for applications.

Most researchers relied on replicating the entire scene database before rendering [Bouville et al. 1985; Potmesil and Hoffert 1989; Singh et al. 1994]. However, this approach has many problems: Replicating the entire geometry, shading, and other data to many clients can take several minutes on standard LANs. Even though most clients often need only a small fraction of this data, it is usually non-trivial for an application to know what data is required by each client.

Two other basic approaches are also available: forwarding rays, e.g. [Cleary et al. 1986; Kobayashi et al. 1987; Reinhard and Jansen 1997], or sending geometry between computers, e.g. [Badouel et al. 1994]. In the first case the entire scene data base is initially distributed across a number of machines usually based on a spatial partitioning scheme. Rays are then forwarded between clients depending on the next spatial partition pierced by the ray. This approach requires too much network bandwidth to be usable in our environment, unless large bi-section bandwidth is available through many fully switched cluster nodes. Furthermore, because rays cannot be reused between frames, no data can be cached at clients.

In the second approach however, scene data is transfered across machines on demand by requests from clients. Because the scene data is read-only and mostly static, caching can be used on the clients. This approach achieves best results for ray tracing algorithms that trace coherent sets of rays and take care to assign similar rays to the same client in each frame, as demonstrated in [Wald et al. 2001b].

We use the same approach in our distributed ray-tracing engine. A master machine is responsible for communicating with the application (see Section 4) and centrally manages the entire scene data — geometry, textures, lights and shaders — without replicating it to all clients. Scene data is managed in small, self-contained parts.

In contrast to [Wald et al. 2001b], our subdivision of the scene data initially also considers information from the application. The application defines individual objects, that may then be further subdivided by the engine if too large.

Load balancing is performed using a demand driven approach, where clients request tasks defined by image tiles from the master machine. In our case load balancing also needs to consider the spatio-temporal coherence between rays in order to make best use of the many client caches. Ideally, each client should get those tiles that contain rays that are similar to rays it has traced in recent frames.

The master machine is also responsible for updating the scene description based on API calls from the application. Currently, changes are broadcasted to all clients including changed shader parameters, textures, geometry, and transformations. While this works reasonably well for relatively small modifications of the environment, it will lead to scalability problems for large changes and many clients.

The distribution process is completely transparent. Applications run only on the master machine and interact with the rendering engine through the API. The distributed computation is also transparent to dynamically loaded shaders except in cases that require communication with other shaders.

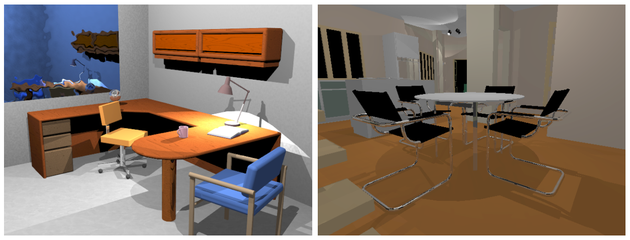

Figure 3: The Office and TownHouse scene used for scalability experiments. In both cases many secondary rays are used for sampling reflections as well as 3 and 25 light sources. With 8 clients these images run at 6 and 2.6 fps, respectively. Distributed ray-tracing allows to scale the frame rate linearly with the number of processors as shown in Table 1. Also note the procedurally bump-mapped mirror in the office scene.

No. of clients	1	2	4	8	16
Office (flat shaded)	3.95	7.85	15.8	26	26
Office (complex shaders)	0.77	1.54	3.08	6.1	10.9
TownHouse	0.3	0.6	1.2	2.4	4.8
Terrain	0.4	0.8	1.6	3.1	6.2

Table 1: Scalability of the distributed ray-tracing engine in frames per second for different numbers of rendering nodes with two processors each. Near-linear scalability is achieved up to a maximum framerate of 26fps, at which the servers network connection gets saturated by the huge amount of pixel data. Note that the scenes require very different numbers of rays for rendering. See Figure 3 for some images.

2.3.1 Results

For the remaining experiments in this paper we use a setup that consists of several dual-processor PCs using AMD AthlonMP 1800+ processors with 512 MB RAM. The nodes are interconnected by a fully switched 100 Mbit-Ethernet using a single Gigabit uplink to the master server to handle the large amounts of pixel data sent for every frame. All experiments are run at video resolution of 640 x 480 pixels, except where noted.

Figure 3 shows images of some of the test scenes used for the scalability experiments as shown in Table 1. We see an almost ideal speedup from 1 to 32 processors, which extends the results previously found in [Wald et al. 2001b] to many more processors. For simple scenes resulting in high frame rates, we are bandwidth limited at the server due to the amount of pixel data. This currently limits our system to roughly 26 frames per second at 640x480, which could be easily alleviated by providing the server with a faster network connection.

3 Dynamic Environments

One of the biggest challenges in interactive ray-tracing is handling dynamic environments. For non-interactive ray-tracing, the time used for building the acceleration structure was insignificant, so that this topic has attracted little research so far [Glassner 1988; Reinhard et al. 2000; Lext and Akenine-Moeller 2001]. This is particularly surprising as any rendering technique that targets sub-linear behavior with respect to scene complexity must maintain such a spatial index.

A simple solution for dynamic environments is to rebuild the whole data structure for every frame. However, this is not feasible for interactive applications. Spatial sorting would again introduce a super-linear complexity of $O(n \log n)$ and would simply be too slow except for the most simple scenes.

A more pragmatic approach is to partition the entire scene into separate *objects*. Each object contains its own index structure that can be changed independently of any other object. These objects are then contained in a single top level scene index structure or can be organized hierarchically. This approach is similar to the one described in [Lext and Akenine-Moeller 2001].

If an entire object's geometry is changed using an affine transformation this can be implemented by inversely transforming rays that intersect the object's bounding box. In this case only the parent object must be notified of the transformation and must update its local index structure. The runtime overhead due to transformations of rays and rebuilding the parent index structures is tolerable and for our implementation using a non-hierarchical scheme is in the order of 10% for the test scenes used in this paper, which contain up to a few thousand objects.

If the change to an object cannot be described by an affine transformation the local index structure of the object must be rebuilt or updated. In both cases the cost is proportional to the number of the affected objects and possibly their parents if a hierarchical scheme is used.

The resulting speedup depends significantly on how an application splits its scene data into individual objects. In that respect this approach is similar to OpenGL where a different scene structure, e.g. the use of display lists or the grouping of materials, can also result in significant performance differences.

As a side effect the described approach also handles multiple instances of an object. For example, Figure 1 shows a scene that contains hundreds of instances of two tree objects. The resulting roughly 8 million triangles consume little memory and can be rendered with 2–4 fps using four clients.

3.1 Results

Results from testing our system with the BART benchmark scenes [Lext et al. 2000] are shown in Figure 4. The scene contains an entire city modeled with roughly a hundred thousand polygons mostly with textures. Ten robots, each consisting of 16 individually moving parts walk through the city, resulting in 161 independently moving objects. The animation is controlled by an application interpreting the BART scene graph and controlling the renderer through the new API. Using four of our clients, we achieve a smooth frame rate of 7 to 12 frames per second at 640x480 pixels. To our knowledge this is the first time that the BART scene has been rendered at interactive rates.

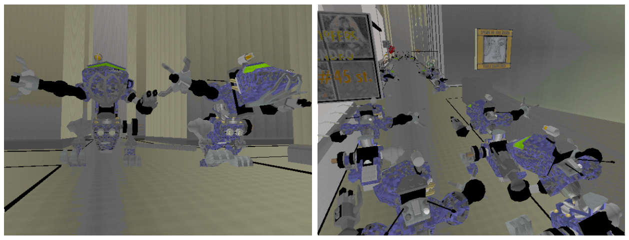

Figure 4: Example images from the BART benchmark. We maintain a framerate of 7–12 fps on four clients.

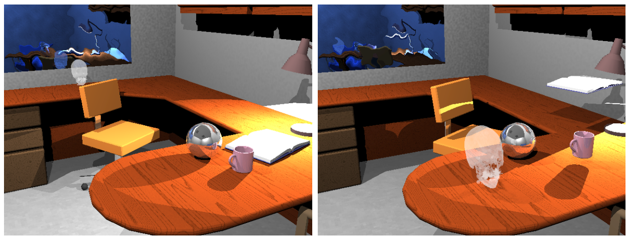

Figure 5: Two example frames from interactively moving the volume head, and other objects. The scene is rendered with shadows from three light sources, reflections, and procedural bump-map and wood shaders. Even during interactions we maintain 5–6 fps on eight clients.

Example images of an interactive session are shown in Figure 5. The user interactively moves the objects in the scene at 5–6 fps using eight clients.

Our current implementation uses a simple flat organization of objects. Thus, it must rebuild the entire top level index structure for every frame even if only a small number of objects have actually been changed. However, the cost for rebuilding the top-level index structure is relative small taking less than 10 milliseconds even for many hundreds of objects as in Figure 6.

The method can reduce the efficiency of occlusion culling as soon as objects overlap in space because two overlapping objects must be traversed sequentially. As a result a ray will continue traversing the first object even though it would have been stopped by geometry contained in the other object. Even though it is easy to construct problematic cases, in practice we see little effect of a few percent on the overall performance.

4 The OpenRT API

In order to make the rendering engine described above available to applications, it must offer a standard interface for applications. Ideally, we could reuse existing APIs for interactive 3D graphics, such as OpenGL [Neider et al. 1993]. OpenGL is well-known to developers and is used by many applications. All OpenGL applications could then directly use the new rendering engine, too.

However, OpenGL and similar APIs are too closely related to the rasterization pipeline to be used directly for our purposes. Their level of abstraction is too low and closely reflect the stream of graphics commands that is fed to the rasterization pipeline.

Other APIs, such as RenderMan, offer better support for ray-tracing, but are not suitable for interactive applications. APIs specifically designed for interactive ray-tracing have simply not been developed yet.

Another option would be to build the ray-tracing API on top of a high-level scene graph API such as Performer, OpenInventor, OpenSG, or others [Rohlf and Helman 1994; Wernecke 1994; OpenSG-Forum 2001]. However, this level is too high for a generic ray-tracing API, as each of the different existing scene graph APIs addresses a specific application. Instead of being limited to a single scene graph API, a low-level API, similar to OpenGL, would be more appropriate. With such support, any scene graph API could later be layered on top of it.

For these reasons, a new API specifically supporting interactive ray-tracing was unavoidable. The design of our new *OpenRT API* is similar to OpenGL but uses frame or retained-mode semantics instead of the immediate-mode semantics of OpenGL. It supports much of the core OpenGL commands and extends them where necessary to better support a ray-tracing engine.

Below, we outline the basic ideas and problems in designing OpenRT. A more detailed specification of the API will be made available elsewhere.

Where meaningful, OpenRT uses similar function names as OpenGL, exchanging the *gl* prefix for *rt*. This is the case for calls specifying *geometry, transformations, and textures*, which have identical syntax and semantics as OpenGL. This simplifies porting applications where large parts of the OpenGL code can be reused directly. Otherwise, OpenRT differs only in three main areas: support for retained objects, programmable shading, and frame semantics.

Instead of *display lists*, which are essentially recorded OpenGL command streams, OpenRT offers *objects*. Objects encapsulate geometry together with references to shaders and their attributes. In contrast to OpenGL display lists, objects do not have any side effects. They are specified using *rtBeginObject(id)/rtEndObject()* pairs. Each object is assigned a unique id that is used to instantiate it later by a call to *rtInstantiateObject(id)*.

In order not to be limited to the fixed reflectance model of OpenGL, OpenRT specifies appearance using *programmable shaders* similar to RenderMan [Pixar 1989]. We currently support shaders that correspond to RenderMan's surface, light, camera, and pixel shaders. In terms of the API, shaders are named objects that receive parameters and are attached to geometry. For this purpose, we have adopted the shader API as suggested by the Stanford Real-time Programmable Shading group [Proudfoot et al. 2001; Mark 2001].

As in the Stanford Shader API, shaders are loaded and instantiated by calls to *rtShaderFile()* and *rtCompileShader()* and are bound to geometry via *rtBindShader()*. Arbitrary shader parameters can be specified by a generic *rtParameter()* call. Depending on the scope for which a parameter has been declared, it is attached to an object, a primitive, or to individual vertices. These different ways to specify parameters allow for optimizing shaders and minimize storage requirements for parameters that change less frequently than at every vertex.

The main architectural difference between OpenRT and OpenGL is the semantics of references. OpenGL stores parameters on its state stack and binds references immediately when geometry is specified. OpenRT, instead, stores state objects, such as textures, shaders, etc., globally and binds them during rendering at the end of each frame. This significantly simplifies the reuse of objects also across frames but means that any changes to such objects will have global effects. These semantic differences require most attention during porting of OpenGL applications.

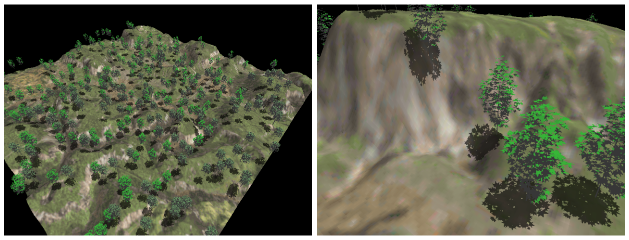

Figure 6: Instantiation: The ground terrain consists of half a million textured triangles and contains several hundred instantiated, highly detailed trees. The total scene consists of roughly 7-10 million triangles, and is rendered together with shadows from a point light source. The right image shows a closeup of the highly detailed shadows cast by the leaves.

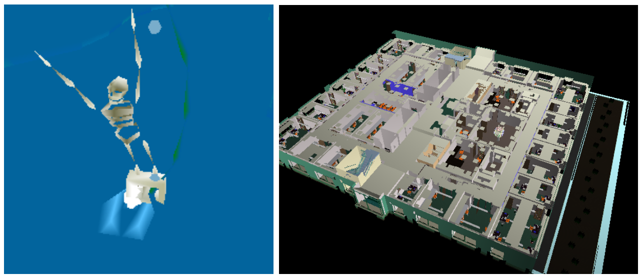

Figure 7: The left image show a frame from an interactive VRML-97 animation displayed with the ported VRML-browser rendering through the OpenRT API with the distributed ray-tracing engine. The image on the right shows one floor of the Soda Hall VRML model containing half a million triangles. Using only 2 clients it renders at roughly 10 fps.

4.1 Results

In order to demonstrate the usability of our API, we have not only used it to write all the applications from this paper, but also by porting an existing VRML-97 browser [Bekaert 2001] to the new API.

The browser passes the VRML shading parameters to the renderer via the new shader API. These parameters are then picked up by a special shader that implements the VRML shading model, including texturing and mip-mapping (see Figure 7).

This special VRML shader can also be replaced by other shaders. They have access to the same VRML shading parameters and can, for instance, automatically compute shadows and reflections, or perform global illumination computations.

5 Benefits of Ray-Tracing

In the previous sections we have demonstrated that ray-tracing can be used to build an interactive rendering engine that can be used as an alternative to rasterization-based systems. As shown with the ported VRML browser and the other examples, the new rendering engine offers at least features comparable to rasterization systems and performance for many interactive 3D graphics applications.

For the remainder of this paper we focus in more detail on the additional benefits gained by using a ray-tracing-based system and present unique new applications that become possible with such a system.

5.1 Flexibility

In contrast to rasterization techniques that are limited to computing regular sets of samples, ray-tracing can efficiently handle samples from individual rays. As a result, ray-tracing can be used effectively for applications that require irregular or even adaptive sets of samples.

Example applications include adaptive sampling of the image plane [Akimoto et al. 1989], frameless rendering [Bishop et al. 1994], adaptive or perceptually-driven rendering [Myszkowski et al. 2001], filling holes in image-based rendering [Walter et al. 1999], and many others. We discuss some of the examples below.

5.1.1 Frameless Rendering

Frameless rendering [Bishop et al. 1994] is an approach to adaptively trade image quality for interactivity. Instead of updating all pixels in an image simultaneously, it incrementally updates individual pixels distributed across the image. The update sequence ensures that every pixel is eventually recomputed.

With this technique the image can be displayed at any time and the system can maintain interactive response times even if the performance is insufficient for complete image updates. However, blur-like effects appear for changing image content if the image update frequency is much lower than the display frequency (see Figure 8).

The algorithm has been published severals years ago but could never be implemented efficiently on rasterization technology. With ray-tracing its implementation is trivial. We have used a Quasi Monte-Carlo (QMC) *interleaved sampling pattern* [Keller and Heidrich 2001] to determine the update sequence. It achieves better image quality than using a randomly perturbed pixel update sequence.

5.1.2 Anti-Aliasing

In order to improve image quality we have implemented several anti-aliasing techniques, including supersampling with a fixed sampling pattern per pixel, supersampling using QMC interleaved sampling, and adaptive supersampling. The first two methods decrease performance by a fixed factor because a constant number of rays are computed per pixel. Interleaved sampling, however, achieves better anti-aliasing using less samples [Keller and Heidrich 2001].

Adaptive oversampling, in contrast, starts with a single sample per pixel and sends additional rays in areas where high color contrast is detected between adjacent pixels (see Figure 9). If additional information, like object- or shader-ids, are requested from the renderer this information can also be considered by the detection procedure.

While this method usually requires less rays to be sent, it introduces additional latency because additional samples can only be requested once previous results are returned by the renderer. For the example in Figure 9 we achieve a factor of 2-4 speedup compared to fixed supersampling depending on the maximum number of samples per pixel. With the flexibility of ray-tracing the application can choose which method is best suited for its needs.

5.1.3 Adaptive Quality

Ray-tracing allows to adapt the rendering quality based on relevance information derived from different sources, such as the image being computed, the user, and the environment. Relevance can be based on information about human perception in general, on measured quantities, such as adaptation levels or the user's gaze, and on image features. The ray-tracing engine can use this information for concentrating computations to the more "relevant" parts of the image [Ferwerda et al. 1997; Myszkowski et al. 2001].

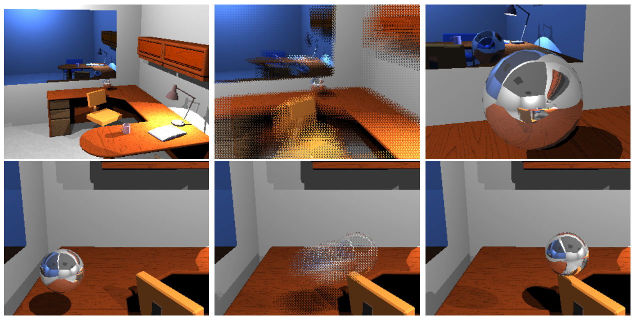
Figure 8: Example images show the effect of frameless rendering during fast camera or object movement. Top: zooming in on the metal ball. Bottom: moving the ball.

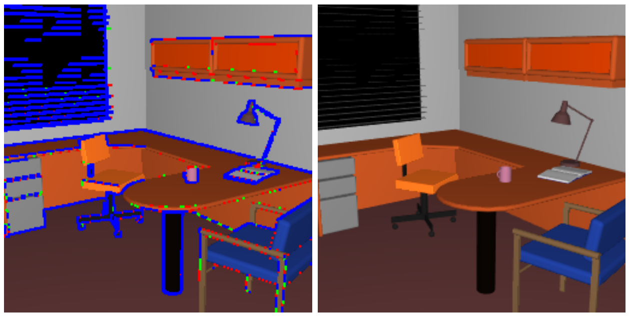
Figure 9: Adaptive oversampling limits anti-aliasing to pixels with strong contrast, marked with color codes.

Once this information is available the application has a number of options to adjust the rendering quality: adjusting sampling density, adjusting shading parameters such as shadow quality, reflection depth, etc. For this purpose the OpenRT API allows to attach additional parameters to each ray that can be accessed from the different shaders. This level of flexibility is not available with rasterization based approaches.

5.2 Logarithmic Complexity

Due to hierarchical index structures (e.g. octrees, BSP-trees, kd-trees etc. [Glassner 1989]) tracing a single ray requires logarithmic time on average in the number of scene primitives to traverse the index structure[1] and results in a roughly constant number of intersection computations before the final intersection is found [Havran 2001].

The plain rasterization algorithm shows linear behavior with respect to scene complexity. Some extensions such as the hierarchical Z-buffer [Greene et al. 1993] or occlusion testing in hardware could help reduce this complexity but are not yet commonly available and also require an interactive update of the index structure.

The logarithmic complexity of ray-tracing with respect to the scene size allows for handling huge models that were previously impossible to render interactively. It also means that the rendering times vary only slowly for models beyond the size of a few hundred thousand triangles, see Figure 2.

The left image in Figure 10 shows the UNC power-plant model [Aliaga et al. 1999] being rendered at 10 fps on four clients (see also [Wald et al. 2001b]). Note that the full detail of the model is being rendered without any simplifications or approximations necessary for hardware rendering [Aliaga et al. 1999]. Instead of complex and time-consuming preprocessing algorithms, ray-tracing requires only simple and fully automatic spatial sorting during creation of the index structure.

The image on the right in Figure 10 shows a terrain data set with several hundreds of trees using instances of two different tree models. The scene contains a total of about 8 million triangles after instantiation. In this application, the user can interactively "slide" the trees along the terrain. We also render this scene with highly detailed shadows caused by the leaves of the trees (see Figure 6). This interactive application maintains a framerate of 2-5 fps on four clients.

[1] In the worst case ray-tracing is linear in the number of primitive but this is highly unlikely to happen in real scenes.

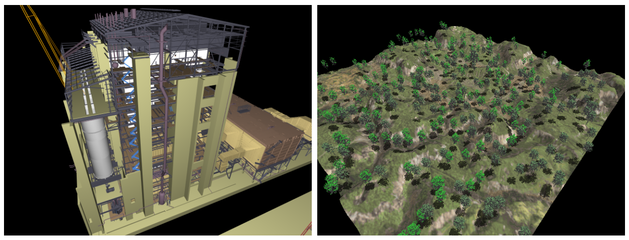

Figure 10: The image on the left shows the UNC power-plant model consisting of 12.5 million individual triangles rendered at roughly 10 fps on 4 clients. The image on the right shows a terrain scene containing more than 8 million triangles. This model is rendered with detailed shadows at 2–5 fps on 4 clients.

5.3 Output-Sensitive Algorithm

Ray-tracing is an output sensitive algorithm. The performance of ray tracing is roughly linear in the number of rays traced for an image. This number of rays is also closely related to the quality of the resulting image. This includes image resolution, supersampling, reflection and shadow rays, sampling of indirect reflection, and others.

Rendering times obviously scale linearly in the number of initial rays from the camera, with a slight advantage for higher resolutions of the same view (in the order of 5% comparing images of 640x480 with 1280x1024). Higher resolutions lead to more closely spaced rays that show better cache hit rates. Shadow rays show essentially the same behavior if traced together from the end points of a coherent set of rays to each point light source separately [Wald et al. 2001a].

Coherence is lost only for rays reflecting off of geometry with extremely high curvature such as a reflective, strongly bump-mapped surface, or for random sampling of the environment as done in many global illumination algorithms.

This result has some immediate consequences for applications. Lets take a reflective object as an example. For ray-tracing the performance is directly related to the number of pixels covered by the object as well as to the part of the scene being reflected in it.

With a rasterization approach the object requires at least one reflection map. For each map the application must render the *entire* environment, as it is difficult to know what parts will actually be reflected. Thus the cost per map is constant, independent of the amount of reflection visible in the image. Similar arguments hold for other effects.

If multiple rendering effects are combined, such as multiple reflections, the constant cost increases with the number of possible combinations, again independent of the actual visibility in the final image.

5.4 Demand-Driven Operation

Tracing a ray consists of three steps: traversal of the index structure, intersection computations, and shading. Each step is only performed once the previous step determined that the computation is indeed necessary. As a result, the rendering engine can request data on demand avoiding the need to have direct access to the entire scene data at all times.

Together with a scheduling strategy for coherent sets of rays, temporal coherence, and the fact that usually only a small part of the scene is visible in each view, caching and prefetching of geometry data can be made to work well [Wald et al. 2001b]. As occlusion culling is inherently built into ray tracing, occluded parts of the scene are never even requested by the renderer.

The demand-driven mode supported by ray-tracing has already been used effectively for demand-loading of scene data from a master machine. It results in fast start-up times as only the required parts of the scene need to be loaded. It also means that part of the scene that never become visible are never loaded. This is particularly important for highly-occluded scenes, such as the interior of buildings. Clients do not require memory for the entire scene but usually operate efficiently using a cache size of less than 10% of the total model size.

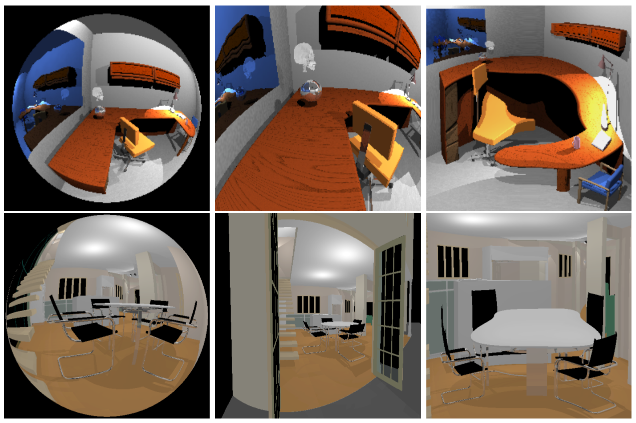

Figure 11: A scene as seen through a fish-eye lens, a cylindrical lens, and some weird, parabolic-like mapping. Arbitrary mappings can be generated by simply replacing the camera that generates primary rays.

One issue with this approach is the additional latency. To some degree it can be handled by sending prefetching rays from anticipated camera positions or using enlarged proxy objects for early notification. However, much more research is required in this area.

This inherently demand driven approach of ray-tracing is in stark contrast to the rasterization-based approach where submission of geometry by the application is decoupled from the Z-buffer that decides what will end up on the screen.

5.5 Programmable Shading

Rasterization-based renderers are limited by the strict pipeline model of shader execution [Mark and Proudfoot 2001]. While it has been shown that arbitrary shader code can be executed in a slightly extended pipeline model, this can become very inefficient [Peercy et al. 2000]. To alleviate this problem more programmable components have recently been added to various parts of the graphics pipeline [NVIDIA 2001; Lindholm and Moreton 2001]. Even though shading language compilers [Mark and Proudfoot 2001] simplify programming, a developer still has to deal with many remaining restrictions imposed by the rasterization pipeline.

Because shading and visibility computations are decoupled, new and advanced shading techniques can easily be added to a ray-tracing implementation. Even complete RenderMan environments with the shading language compiler have been integrated with ray-tracers [Slusallek et al. 1994; Gritz and Hahn 1996]. Ray-tracing has the advantage that only visible samples are ever shaded and that shading can be performed asynchronously with ray-traversal and ray-intersection.

With a ray tracer shaders are no longer restricted to purely local illumination models even for interactive use. Every shader can shoot arbitrary secondary rays as necessary for reflections, refractions, or environment sampling.

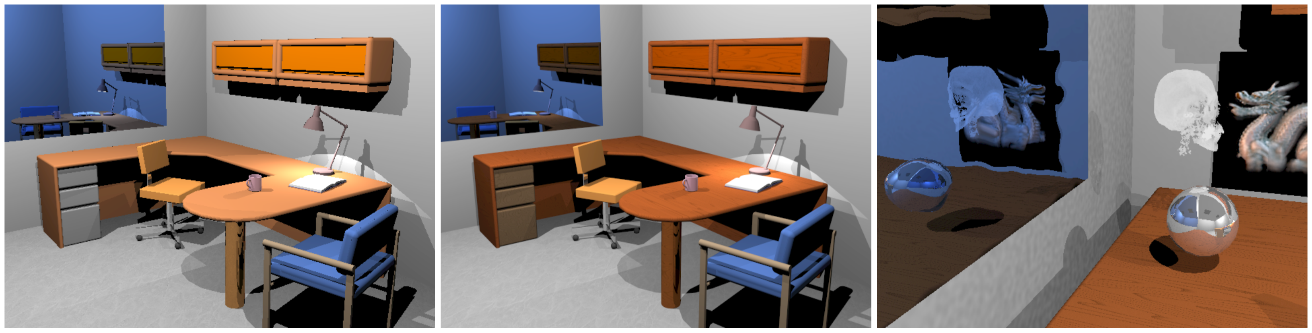

Figure 12: An office environment with different shader configurations: A shader computing reflections and shadows from three lights, additional procedural shaders for wood and marble, and with volume and lightfield objects and procedural bump mapping on the mirror.

5.5.1 Surface and Light Shaders

Figure 12 shows an office environment with several procedural shaders such as *wood* and *marble* applied to some surfaces. Procedurally determining shader parameters allows for more flexibility, arbitrary detail, and better anti-aliasing [Ebert et al. 1998]. Even though both shaders require expensive evaluations of *noise* functions, the overall framerate dropped by about 3%. This indicates that significantly more complex shading code could still be implemented while maintaining interactive performance.

Similar to surface shaders, our system supports arbitrarily programmable light shaders. Our system currently supports the usual light types as supported in OpenGL or VRML, as well as area light shaders for more realistic lighting and global illumination (see below).

5.5.2 Camera Shaders

Rasterization is limited to the simple pinhole camera model. All other models must be approximated by warping and compositing images from different pinhole camera settings, e.g. [Heidrich et al. 1997]. With our ray-tracing based system the user can use *camera shaders* for implementing any desired camera model. The shader simply has to generate the correct primary rays.

This can be used to interactively visualize the scene using simple fish-eye lenses, as shown in Figure 11, or more complex lens systems that simulate real camera lens systems [Kolb et al. 1995; Loos et al. 1998]. The shader is also responsible for computing depth of field and motion blur effects.

5.6 Correctness

Rasterization techniques are limited to approximations for most optical effects because it can only deal efficiently with regular sets of samples derived from a parallel or planar perspective projection.

For example, reflections off of extended objects are approximated by a reflection map that is valid only for a single point. The results are more or less incorrect and cannot contain self-reflection. Similar arguments hold for other effects, such as refraction, shadows, etc. [Diefenbach and Badler 1997; Heidrich and Seidel 1999]. In addition to being approximations, these techniques often require user interaction in determining the best parameters, e.g. for position, resolution, and the contents of reflection maps.

Ray-tracing deals with all these effects correctly and automatically. If desired, applications can still use the same approximations but must then deal with the same issues as in OpenGL.

6 Enabling New Applications

The long list of benefits offered by a ray-tracing-based rendering engine enables completely new applications that could not be implemented with rasterization technology. As examples we describe two novel interactive applications that have recently been implemented on top of the OpenRT API.

6.1 Integrating Different Model Types

In addition to the VRML-97 browser mentioned earlier, we also implemented a specialized viewer. It integrates all of the above techniques coherently and also adds new rendering primitives such as volume data sets and light fields. While this functionality is currently in a separate application we plan to integrate it into our VRML browser via custom nodes in a VRML scene graph.

6.1.1 Volume Rendering

We implemented support for volume rendering through a special surface shader attached to a simple box. The shader is called for each ray hitting the box and computes the the emission-absorption model for each voxel along the ray. In case that some transparency remains, an additional ray samples the scene behind the voxel grid. Its intensity is then modulated by the transparency of the volume. The same technique also works for shadow rays where the transparency modulates the shadow intensity.

Since the volume effect has been implemented as a shader object, other applications can simply load and use this shader. No additional support by the application is required.

Figure 13 show an example of a simple volume object rendered this way. We are using a simple implementation of volume rendering that could easily be improved: More advanced and better optimized volume rendering techniques including complex transfer functions, lighting, multiple scattering, etc, could easily be integrated into this system without changing the current ray-tracing core. Also our code does not currently account for other objects penetrating the volume.

6.1.2 Lightfield

Integrating new rendering primitives is extremely simple and fast in a ray-tracing context. In less than half a day we implemented a lightfield renderer, again simply by implementing it as a special surface shader. If applied to a surface it uses the position and direction of the incoming ray for indexing into the lightfield data structure and performing the quadri-linear interpolation. Figure 13 shows an example frame of the Stanford dragon lightfield [Lightpack 2002].

6.1.3 Integration

The ability to easily implement and interactively render new types of objects is already quite useful. In particular, an equivalent implementation using OpenGL would be significantly more complex.

The unique advantage of ray-tracing, however, is that all the techniques described in this paper can easily be combined in a single scene. Figure 14 shows an office environment that contains surface geometry, a volumes data set, and a lightfield object. The surfaces use a mix of simple and procedural shaders, implementing procedurally bump-mapped reflections on the mirror and complex procedural wood.

All the optical effects work as expected: The volume casts transparent shadows onto other objects, the lightfield is correctly visible through the volume as well as through the bump-mapped, wavy mirror on the left. Additionally all the objects can be moved interactively by the user and any parameter can be changed instantly.

From an application point of view, this scene is just as simple as any other scene in this paper. The application simply defines the geometry and assigns the right shaders and parameters, such as the name of the volume data set.

In contrast, implementing such a scene in a rasterization-based renderer, while theoretically possible, would be highly non-trivial, would require numerous rendering passes with the corresponding slowdown, and would still give only approximate results.

The flexibility offered by the ray-tracing algorithm is the key to support all these objects and rendering techniques in a single interactive application. To our knowledge this is the first time such capabilities are available for interactive use.

6.2 Interactive Global Illumination

Global illumination algorithms generate highly realistic images by accounting for the subtle but important effects of indirect illumination in a physically-correct way [Cohen and Wallace 1993; Dutre et al. 2001]. Global illumination simulates the global light transport in an environment between all mutually visible surfaces in the environment. Almost all algorithms use ray-tracing for simulating the transport of light.

Due to the amount and complexity of the computations, rendering with global illumination is usually slow and far from interactive, taking several minutes to hours even for simple diffuse environments. Achieving interactive performance has been a goal for many years, e.g. [Drettakis and Sillion 1997; Bala et al. 1999; Chalmers et al. 2001]. With a fast, flexible, and scalable ray-tracing engine we should now be able to reach this goal.

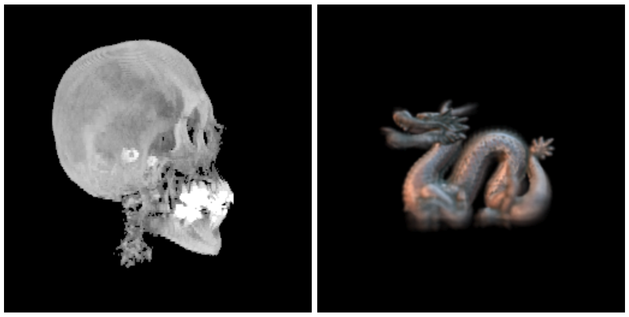

Figure 13: Images of a volume object and a lightfield. Both are simply implemented as surface shaders attached to a box and a rectangle, respectively.

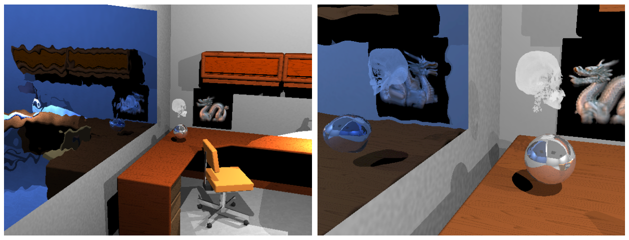

Figure 14: An office scene containing surfaces, volumes, and lightfields models as well as procedural wood, marble, and bump-mapping shaders. Note that all optical effects work as expected: transparent volume shadows, lightfields visible through bump-mapped reflections and through the volume object etc. Of course, the objects can be changed and moved interactively.

6.2.1 Restrictions on Algorithms

However, global illumination algorithms are inherently more complex than classical ray tracing and thus not all algorithms will automatically benefit from a much faster ray-tracing engine. In order to fully exploit the available resources a global illumination algorithm has to meet several constraints:

Parallelism: We were able to achieve good parallelism by computing pixels separately. Many global illumination algorithms, however, require reading or even updating global information, such as the radiosity of a patch [Cohen and Wallace 1993], entries in the photon map [Jensen 2001], or irradiance cache entries [Ward and Heckbert 1992]. This requires costly network communication and synchronization overhead that can easily limit the achievable performance.

Efficiency: In order to be interactive, a suitable algorithm must achieve sufficiently good images with less than about 50 rays per pixel on average given the current performance of our engine and the available machines. Thus we must make the best possible use of the information computed by each ray.

Real-time: Because we aim for interactivity, algorithms can no longer use extensive preprocessing. Preprocessing must be limited to at most a few milliseconds per frame and cannot be amortized or accumulated over more than a few frames as the stalling and increased latency of lighting updates would become noticeable.

Other Costs: The rendering engine can only speed up ray-tracing. So performance must not be limited by other computations, such as nearest-neighbor queries, costly BRDF evaluations, network communication, or even random number generation.

Within the above constraints most of todays global illumination algorithms cannot be implemented interactively based on our engine: All radiosity style algorithms require significant preprocessing of global data structures which seems impossible to implement under these constraints [Cohen and Wallace 1993].

Pure light-tracing or path-tracing [Kajiya 1986] based approaches would require too many rays per pixel for a decent quality at least for non-trivial lighting conditions. The original PhotonMap [Jensen 2001] algorithm requires costly preprocessing for photon shooting and creation of the kd-trees as well as expensive nearest neighbor queries during rendering. The use of irradiance caching for indirect illumination is another problem.

6.2.2 New Algorithm

The above discussion shows that a new algorithm is required to take advantage of a fast ray-tracing engine for interactive global illumination computations: Anonymous [Anonymous 2002] developed a new Monte Carlo based algorithm explicitly designed to run efficiently under the constraints mentioned above. Because the full details of this algorithm are beyond the scope of this paper, we will only give a brief overview: The algorithm is based on the idea of instant radiosity [Keller 1997] for mainly diffuse, direct and indirect illumination, photon maps for caustic effects [Jensen 1997], and interleaved sampling [Keller and Heidrich 2001] for efficient parallelization in a network, and a new filtering technique to avoid disturbing noise artifacts. All techniques recompute the global illumination solution at every frame.

This algorithm has been successfully implemented on top of our rendering system. As it relies mainly on shooting coherent visibility rays, it allows to fully exploit the performance of our ray tracing engine. Minimal preprocessing combined with interleaved sampling across client machines allows for easy parallelization and effective load balancing, resulting in good scalability over more than 32 CPUs.

While not all illumination effects are accounted for (e.g. no indirect caustics), all of the most important features are correctly simulated: Direct lighting, soft shadows, indirect illumination, reflections, refractions and caustics. The system allows for arbitrary interactive manipulation of the scene at a framerate of up to 5 fps at 640x480 resolution, with a full update of all effects at every frame. Additionally we can accumulate information in static situations for better image quality.

Some example images from our system can be seen in Figure 15. This is the first time that a system with similar features has been realized at interactive rates.

All global illumination computations are implemented as plug-in shaders applied to the surfaces and to the image on the master machine. This allowed to abstract from the distributed framework and from issues such as dynamic scenes, which are handled transparently by our API. From the application point of view, the global illumination system is a shader like any other, and does not require any special handling.

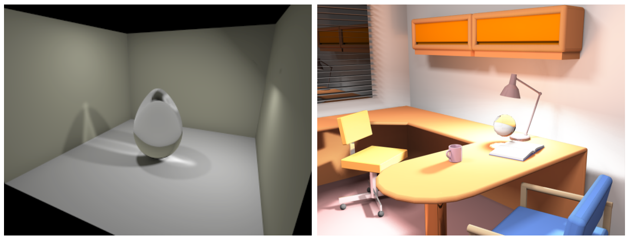
Figure 15: Two frames from an interactive global illumination application: A glass egg with detailed caustics, and an office environment with reflections, caustics, and other lighting and shading effects. These scenes can be changed dynamically and render with up to 5 fps.

7 Conclusions

In this paper, we have presented a new ray-tracing-based rendering engine for interactive 3D graphics. It builds on a fast, distributed implementation of ray-tracing implemented on inexpensive PC hardware hardware and scales to many networked PCs. Other novel contributions include an approach for supporting interactively changing the scene and the design of an OpenGL-like API that facilitates easy porting of existing application to the new rendering system.

This engine supports the full set of rendering features like shadows, reflections, and procedural shading with arbitrary surface, light, and camera shaders. Non-surface models such as volume data sets and lightfields have been integrated seemslessly and efficiently into the system simply through special shaders. All rendering effects, such as transparent shadows of the volume object, automatically work as expected.

The new rendering system offers significant advantages over the current state-of-the-art. The flexibility of ray-tracing allows for the first time to implement many advanced graphics algorithms in an interactive context. This includes frameless rendering, efficient anti-aliasing, adaptive image quality, as well as many special effects. Ray-tracing also delivers “automatically correct” results by default as it is not limited to approximations for basic rendering effects and does not require preprocessing. All computations are done on the fly.

The ray-tracing system scales linearly over a wide range of computational resources and shows logarithmic behavior with respect to scene complexity. Together with its output-sensitivity this allows it to be used for a wide range of applications not limited to computer graphics.

Finally, in order to demonstrate the unique possibilities offered by our new rendering engine, we briefly described a fully interactive global illumination system built on top of the described engine. It uses a new Monte Carlo algorithm that is able to make effective use of the improved ray-tracing performance even in a distributed environment.

We are currently evaluating the practical use of the new system in the car and airplane industry, mainly for highly realistic rendering of complex models including advanced lighting effects.

8 Future Work

While the new rendering system offers high performance, much flexibility, and a wide variety of features there is still room for improvements.

Obviously, higher performance on a single machine is required for challenging existing mainstream solutions for interactive 3D graphics. Boards with multiple standard processors, specialized processors, large scale multiprocessing-on-a-chip, or even special ray-tracing chips are all interesting solutions that need to be carefully evaluated.

In addition we need to investigate better algorithms for reducing the computational load and carefully find options for even more efficient implementations. So far we have used a mostly brute-force, keep-it-simple approach that has often been very successful in graphics. More intelligent approaches would be useful.

The design of the rendering engine could also be improved and extended. Currently all data is managed on the master making it likely to become a bottleneck. More flexible approaches where the scene data is distributed across the clients seems useful, as long as this remains transparent to the application. Other related issues appear with parallel and distributed applications changing the scene simultaneously on multiple master machines.

We need to port and evaluate more applications and locate necessary extensions to the current architecture. This includes work on the OpenRT API. Most interesting candidates are scene-graph libraries where a single port of the library would benefit all applications based on this library. Another interesting challenge would be the design of an interactive RenderMan-compliant renderer.

References

ADVANCED RENDERING TECHNOLOGIES. 1999. The AR250 - a new architecture for ray traced rendering. In *Proceedings of the Eurographics/SIGGRAPH workshop on Graphics hardware - Hot Topics Session*, 39–42.

AKIMOTO, T., MASE, K., HASHIMOTO, A., AND SUENAGA, Y. 1989. Pixel selected ray tracing. In *Eurographics '89*, North-Holland, W. Hansmann, F. R. A. Hopgood, and W. Strasser, Eds., 39–50.

ALIAGA, D., COHEN, J., WILSON, A., BAKER, E., ZHANG, H., ERIKSON, C., HOFF, K., HUDSON, T., STÜRZLINGER, W., BASTOS, R., WHITTON, M., BROOKS, F., AND MANOCHA, D. 1999. MMR: An interactive massive model rendering system using geometric and image-based acceleration. In *1999 ACM Symposium on Interactive 3D Graphics*, 199–206.

ANONYMOUS, A., 2002. Withheld.

BADOUEL, D., BOUATOUCH, K., AND PRIOL, T. 1994. Distributing data and control for ray tracing in parallel. *Computer Graphics and Applications 14*, 4 (July), 69–77.

BALA, K., DORSEY, J., AND TELLER, S. 1999. Radiance interpolants for accelerated bounded-error ray tracing. *ACM Transactions on Graphics 18*, 3 (August), 213–256.

BEKAERT, P., 2001. Extensible scene graph manager. http://www.cs.kuleuven.ac.be/g̃raphics/XRML/, August.

BISHOP, G., FUCHS, H., MCMILLAN, L., AND SCHER ZAGIER, E. J. 1994. Frameless rendering: Double buffering considered harmful. *Computer Graphics 28*, Annual Conference Series, 175–176.

BOUVILLE, C., BRUSQ, R., DUBOIS, J.-L., AND MARCHAL, I. 1985. Generating high quality pictures by ray tracing. *Computer Graphics Forum 4*, 2, 87–99.

CHALMERS, A., AND REINHARD, E. 1998. Parallel and distributed photo-realistic rendering. In *Course notes for SIGGRAPH 98*. ACM SIGGRAPH, Orlando, USA, July, 425–432.

CHALMERS, A., DAVIS, T., KATO, T., AND REINHARD), E. 2001. Practical parallel processing for today's rendering challenges. In *SIGGRAPH 2001 Course Notes, Course 40*. ACM Siggraph.

CLEARY, J. G., WYVILL, B. M., BIRTWISTLE, G. M., AND VATTI, R. 1986. Multiprocessor Ray Tracing. In *Computer Graphics Forum 5*, North-Holland, Amsterdam, 3–12.

COHEN, M. F., AND WALLACE, J. R. 1993. *Radiosity and Realistic Image Synthesis*. Morgan Kaufmann Publishers. ISBN: 0121782700.

COOK, R., PORTER, T., AND CARPENTER, L. 1984. Distributed ray tracing. In *ACM SIGGRAPH Computer Graphics*, vol. 18, 137–144.

DIEFENBACH, P. J., AND BADLER, N. I. 1997. Multi-pass pipeline rendering: Realism for dynamic environments. In *Symposium on Interactive 3D Graphics*, 59–70.

DRETTAKIS, G., AND SILLION, F. 1997. Interactive update of global illumination using a line-space hierarchy. In *ACM SIGGRAPH 97*.

DUTRE, P., BALA, K., AND BEKAERT, P. 2001. Advanced global illumination. In *SIGGRAPH 2001 Course Notes, Course 20*. ACM Siggraph.

EBERT, D., MUSGRAVE, F., PEACHEY, D., PERLIN, K., AND WORLEY, S. 1998. *Texturing and Modeling: A Procedural Approach*. Morgan Kaufmann.

FERWERDA, J. A., PATTANAIK, S. N., SHIRLEY, P., AND GREENBERG, D. P. 1997. A model of visual masking for computer graphics. *Computer Graphics 31*, Annual Conference Series, 143–152.

GLASSNER, A. S. 1988. Spacetime ray tracing for animation. *IEEE Computer Graphics and Applications 8*, 2, 60–70.

GLASSNER, A. 1989. *An Introduction to Raytracing*. Academic Press.

GREENE, N., KASS, M., AND MILLER, G., 1993. Hierarchical Z-buffer visibility.

GRITZ, L., AND HAHN, J. K. 1996. BMRT: A global illumination implementation of the RenderMan standard. *Journal of Graphics Tools: JGT 1*, 3, 29–47.

HAVRAN, V. 2001. *Heuristic Ray Shooting Algorithms*. PhD thesis, Czech Technical University.

HEIDRICH, W., AND SEIDEL, H.-P. 1999. Realistic, hardware-accelerated shading and lighting,. In *Proceedings of SIGGRAPH'99*, 171–178.

HEIDRICH, W., SLUSALEK, P., AND SEIDEL, H. 1997. An image-based model for realistic lens systems in interactive computer graphics. In *Graphics Interface '97*, W. A. Davis, M. Mantei, and R. V. Klassen, Eds., 68–75.

JENSEN, H. W. 1997. Rendering caustics on non-Lambertian surfaces. *Computer Graphics Forum 16*, 1, 57–64.

JENSEN, H. W. 2001. *Realistic Image Synthesis Using Photon Mapping*. AK Peters.

KAJIYA, J. T. 1986. The rendering equation. In *Computer Graphics (SIGGRAPH '86 Proceedings)*, D. C. Evans and R. J. Athay, Eds., vol. 20, 143–150.

KELLER, A., AND HEIDRICH, W. 2001. Interleaved sampling. In *Rendering Techniques 2001*, Springer, London, S. Gortler and K. Myszkowski, Eds., 269–276.

KELLER, A. 1997. Instant radiosity. In *Computer Graphics Proceedings, Annual Conference Series, SIGGRAPH 97*, 49–56.

KOBAYASHI, H., NAKAMURA, T., AND SHIGEI, Y. 1987. Parallel processing of an object space for image synthesis using ray tracing. *The Visual Computer 3*, 1, 13–22.

KOLB, C., MITCHELL, D., AND HANRAHAN, P. 1995. A realistic camera model for computer graphics. *Computer Graphics 29*, Annual Conference Series, 317–324.

LEXT, J., AND AKENINE-MOELLER, T. 2001. Towards rapid reconstruction for animated ray tracing. In *Eurographics 2001 – Short Presentations*, pp. 311–318.

LEXT, J., ASSARSSON, U., AND MOELLER, T. 2000. BART: A benchmark for animated ray tracing. Tech. rep., Department of Computer Engineering, Chalmers University of Technology, Goeteborg, Sweden, May. Available at http://www.ce.chalmers.se/BART/.

LIGHTPACK, 2002. Stanford lightfield archive. http://graphics.stanford.edu/software/lightpack/gamma-corrected/lifs.html.

LINDHOLM, E., AND MORETON, M. K. H. 2001. A user-programmable vertex engine. In *Computer Graphics (Proc. of Siggraph)*, 149–159.

LOOS, J., SLUSALLEK, P., AND SEIDEL, H.-P. 1998. Using wavefront tracing for the visualization and optimization of progressive lenses. In *Computer Graphics Forum (Proc. EUROGRAPHICS '98)*, 255–266.

MARK, W., AND PROUDFOOT, K. 2001. Compiling to a VLIW fragment pipeline. In *Graphics Hardware 2001*, ACM Siggraph, 47–56.

MARK, W. 2001. Shading system immediate-mode API, v2.1. In *SIGGRAPH 2001 Course 24 Notes – Real-Time Shading*, ACM Siggraph.

MUUSS, M. J., AND LORENZO, M. 1995. High-resolution interactive multispectral missile sensor simulation for atr and dis. In *Proceedings of BRL-CAD Symposium '95*.

MUUSS, M. J. 1995. Towards real-time ray-tracing of combinatorial solid geometric models. In *Proceedings of BRL-CAD Symposium '95*.

MYSZKOWSKI, K., TAWARA, T., AKAMINE, H., AND SEIDEL, H.-P. 2001. Perception-guided global illumination solution for animation rendering. In *Computer Graphics (Proc. of Siggraph)*, ACM Siggraph, Los Angeles, 221–230.

NEIDER, J., DAVIS, T., AND WOO, M. 1993. *OpenGL Programming Guide*. Addison-Wesley, Reading MA.

NVIDIA. 2001. *NVIDIA OpenGL Extension Specifications*. March.

OPENSG-FORUM, 2001. http://www.opensg.org.

PARKER, S., SHIRLEY, P., LIVNAT, Y., HANSEN, C., AND SLOAN, P. P. 1998. Interactive ray tracing for isosurface rendering. In *IEEE Visualization '98*, 233–238.

PARKER, S., PARKER, M., LIVNAT, Y., SLOAN, P. P., HANSEN, C., AND SHIRLEY, P. 1999. Interactive ray tracing for volume visualization. *IEEE Transactions on Computer Graphics and Visualization 5*, 3 (July-September), 238–250.

PARKER, S., SHIRLEY, P., LIVNAT, Y., HANSEN, C., AND SLOAN, P. P. 1999. Interactive ray tracing. In *Interactive 3D Graphics (I3D)*, 119–126.

PEERCY, M. S., OLANO, M., AIREY, J., AND UNGAR, P. J. 2000. *Interactive Multi-Pass Programmable Shading*. ACM Siggraph, New Orleans, USA, July.

PIXAR. 1989. *The RenderMan Interface*. San Rafael, September.

POTMESIL, M., AND HOFFERT, E. M. 1989. The Pixel Machine: A parallel image computer. In *Computer Graphics (SIGGRAPH '89 Proceedings)*, J. Lane, Ed., vol. 23, 69–78.

PROUDFOOT, K., MARK, W., TZVETKOV, S., AND HANRAHAN, P. 2001. A real-time procedural shading system for programmable graphics hardware. In *Computer Graphics (Proc. of Siggraph)*, ACM Siggraph, 159–170.

REINHARD, E., AND JANSEN, F. W. 1997. Rendering large scenes using parallel ray tracing. *Parallel Computing 23*, 7 (July), 873–885.

REINHARD, E., CHALMERS, A., AND JANSEN, F. 1998. Overview of parallel photorealistic graphics. In *Eurographics '98, State of the Art Reports*, Eurographics Association.

REINHARD, E., SMITS, B., AND HANSEN, C. 2000. Dynamic acceleration structures for interactive ray tracing. In *Proceedings Eurographics Workshop on Rendering*, 299–306.

ROHLF, J., AND HELMAN, J. 1994. IRIS Performer: A high performance multiprocessing toolkit for real-time 3D graphics. *Computer Graphics 28*, Annual Conference Series, 381–394.

SINGH, J. P., GUPTA, A., AND LEVOY, M. 1994. Parallel visualization algorithms: Performance and architectural implications. *IEEE Computer 27*, 7, 45–55.

SLUSALLEK, P., PFLAUM, T., AND SEIDEL, H.-P. 1994. Implementing renderman–practice, problems and enhancements. In *Computer Graphics Forum, Proceedings of Eurographics '94.*, Blackwell, vol. 13, 443–454.

WALD, I., BENTHIN, C., WAGNER, M., AND SLUSALLEK, P. 2001. Interactive rendering with coherent ray tracing. *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2001 20*, 3. available at http://graphics.cs.uni-sb.de/ wald/Publications.

WALD, I., SLUSALLEK, P., AND BENTHIN, C. 2001. Interactive distributed ray tracing of highly complex models. In *Proceedings of the 12th EUROGRPAHICS Workshop on Rendering*. London.

WALTER, B., DRETTAKIS, G., AND PARKER, S. 1999. Interactive rendering using the render cache. *Eurographics Rendering Workshop 1999*. Granada, Spain.

WARD, G. J., AND HECKBERT, P. 1992. Irradiance Gradients. In *Third Eurographics Workshop on Rendering*, 85–98.

WERNECKE, J. 1994. *The Inventor Mentor*. Addison-Wesley, Reading, Massachusetts, U.S.A.

WHITTED, T. 1980. An improved illumination model for shaded display. *CACM 23*, 6 (June), 343–349.