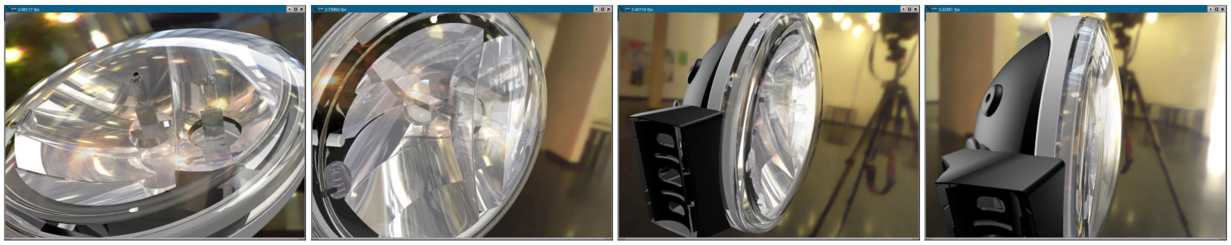
Fourth Eurographics Workshop on Parallel Graphics and Visualization (2002)
D. Bartz, X. Pueyo, E. Reinhard (Editors)

Interactive Headlight Simulation – A Case Study of Interactive Distributed Ray Tracing –

Carsten Benthin, Tim Dahmen, Ingo Wald, Philipp Slusallek

Computer Graphics Group
Saarland University
{benthin,dahmen,wald,slusallek}@graphics.cs.uni-sb.de



Abstract

Todays rasterization graphics hardware provides impressive speed and features making it the standard tool for interactively visualising virtual prototypes early in the industrial design process. However, due to inherent limitations of the rasterization approach many optical effects can only be approximated. For many products, in particular in the car industry, the resulting visual quality and realism is inadequate as the basis for critical design decisions. Thus the original goal of using virtual prototyping — significantly reducing the number of costly physical mockups — often cannot be achieved.
Interactive ray tracing on a small cluster of PCs is emerging as an alternative visualization technique achieving the required accuracy, quality, and realism. In a case study this paper demonstrates the advantages of using interactive ray tracing for a typical design situation in the car industry: visualizing the prototype of headlights. Due to the highly reflective and refractive nature of headlights, proper quality could only be achieved using a fast interactive ray tracing system.

1. Introduction

Interactive visualization of virtual prototypes is becoming an essential tool in the industrial design process. Virtual prototypes are increasingly used as the basis for critical and early design decisions in order to shorten the design process and evaluate more design alternatives. If decisions can be based on accurate data from visualizing the virtual prototypes, costly and time-consuming physical mockups can be avoided.

In this paper we concentrate on the design of car headlights. Headlights are often called the "eyes" of a car and are particularly important for the overall visual impression of the car. Thus, significant efforts are spend on their appropriate appearance. The design is constrained by the required lighting properties, but also by the shape and placement dictated by the design of the car body and other factors. Several design groups need to communicate to reach the final decision and the availability of a realistic virtual prototype is a perfect mean to improve this communication.

Currently, for each new model of a car, several iterations of building and evaluating physical prototypes are performed. Each iteration takes several weeks to months and is fairly expensive due to custom-made tools required to form the front glass and the reflector. Virtual prototyping should be able to reduce the number of physical prototypes, and reduce cost and time in the design process.

However, there are many constraints that must be met in order for a visualization technique to be used in the daily design process.

1.1. Realism

Designers and managers must be able to rely on the data provided by the visualization of a prototype. They cannot accept the loss of realism and accuracy caused by coarse limitations in current graphics hardware. With current graphics technology the accuracy, quality, and realism of virtual prototypes are inadequate as the basis for important design decisions. Hardware heavily relies on approximations for rendering important visual effects. For example, it is impossible to accurately simulate the reflection and refraction of light in curved surfaces with rasterization hardware. Ray tracing, however, is able to simulate these effects but due to its high computional cost, it is still used only as an offline visualization technique.

1.2. Interactivity

Interactivity is another key factor for the effective visualization of prototypes. Certain optical effects may only appear during interaction and movements with the prototype. This is particularly relevant for optical effects, such as highlights and reflection patterns in especially designed reflectors. Such highlights appear only from very specific positions or lighting conditions. Designers must be able to interactively explore such situations in order to find optimal solutions.

1.3. Geometric Complexity and Automatic Processing

The original car models used by design systems are often of high geometric complexity. This results from the fact that these systems usually work with free-form geometry like NURBS, and use tessellation to convert them to triangles, often yielding hundreds of thousands to millions of triangles. Furthermore, modeling tools also use libraries of existing parts, which can have arbitrary geometric complexity themselves, even though they form only a minor part of the whole model.

The geometric complexity of the whole model is already challenging for current graphics hardware, and is commonly reduced using mesh decimation techniques to tractable levels. This process is usually only semi-automatic, and requires significant time and manual effort. It would be highly desirable to have a system that can handle the models with a geometric complexity of several million triangles.

2. Previous Work

There is currently no good solution to visualize car headlights with a high-enough quality to be used in the design process. State-of-the-art is to render an animation using an animation/rendering package based on ray tracing (e.g. Maya/Alias Wavefront[1]). Specifying and rendering the animation alone may take several days and is often performed over the weekend. Even then, the results are not adequate for critical decisions, as they do not allow interactive exploration.

Rasterization based systems do not allow accurate simulation of reflections off of curved surfaces, nor can they handle multiple reflections and self-reflection.

Ray tracing would allow for an accurate simulation of the optical effects in headlights, but has been too slow for interactive use. Recently, systems for interactive ray tracing have been developed: Using a large shared memory supercomputer, Muss et al. [5, 6] have demonstrated that a traditional ray tracer can interactively render several hundred thousands of CSG primitives that would correspond to several million polygons after tesselation.

Similarly, at the University of Utah Parker et al. [8] have built an interactive ray tracing system that is also based on a large shared-memory computer. Beside standard ray tracing their system is being used to visualize volumetric data sets and to render high-quality iso-surfaces[9, 7].

Recently, Wald et al. [16, 14] showed that distributed ray tracing using a cluster of commodity PCs is also able to interactively visualize highly complex models with 12.5 million polygons and more. Wald et al. also demonstrated that their ray tracing engine can be used to build an interactive global illumination system [15]. We build on this distributed ray tracing engine for this case study.

3. Headlight Visualization - A Case Study

In order to provide accurate results for the visualization of car headlights, we use a distributed ray tracing system running on commodity PC hardware. In order to be usable in practice, our visualization system has to meet several imposed concrete requirements imposed by our partners.

First, the system has to meet the quality standard set by the car industry. Even though no actual physical comparisons have been performed yet, the rendering quality achieved by our system match the quality of the current off-line visualization tools. In general, the system has to be 'sufficiently accurate' to help the designers in their design decisions. Therefore, we have been provided with several reference images (see Figure 7) that are used to visually evaluate the quality.

The system should allow the user to move and inspect any desired location with interactive feedback (i.e. a minimum of several frames/sec) at video resolution or higher.

Third, the geometry handling should be fully automatic. In our example, the headlight model consists of roughly 800,000 triangles in total. Modern systems based on triangle rasterization are already able to handle this high number of triangles but due to the logarithmic cost in scene complexity it is a rather moderate complexity for ray tracing. Therefore, we can directly load the original model without the need for geometric simplification and approximations.

Finally, the price of a system has proven to be only a minor issue if a system can solve a problem that would not be solvable at all otherwise. Our system uses a cluster of commodity PCs, which in total costs less than $30.000 for 16 nodes.

4. Implementation

In the following, we will give a brief overview of how our system is implemented and which techniques are used to provide realistic visualization.

4.1. Software Environment

In order to achieve interactive ray tracing performance, we use the OpenRT library [12], which provides a simple interface to a fast distributed ray tracing system, in particular it uses a API very similar to OpenGL.

The OpenRT library supports distributed rendering on a cluster of commodity PCs. It uses a client/server-based architecture where the server hosts the application, and a cluster of client PCs is used for rendering. All OpenRT commands, including scene data is broadcast to all rendering clients. In order to exploit the compute power of all clients the server uses dynamic load balancing [10] where the clients request work from the server on demand. The OpenRT scheduler distributes work based on image tiles. After an image tile has been rendered by a client, the corresponding color information is sent back to the server.

In order to effectively hide communication latency, the application and the OpenRT scheduler are running asynchronously (in seperate threads) on the same host. During job scheduling for the current frame the application(-thread) already sends OpenRT commands for the next frame. These commands are buffered on client side and are only executed after the client has finished all jobs for the current frame. Sending frame data for frame n during rendering of frame n-1 results in one frame latency which is rather negligible. Figure 1 illustrates a simplified version of the communication flow between clients and server.

Due to the ray-tracing-based rendering core a spatial index structure has to be built before rendering on the client side. Transferring all required scene data to the clients and building this data structure takes only a couple of seconds leading to fast startup time.

In order to achieve better image quality without loss in performance OpenRT supports progressive anti-aliasing: if no user interaction occurs, subsequent frames will be accumulated using different random quasi monte carlo [4] pixel sampling patterns per frame.

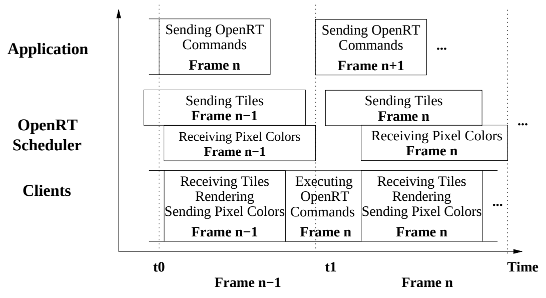

Application
Sending OpenRT Commands Frame n
Sending OpenRT Commands Frame n+1
...
OpenRT Scheduler
Sending Tiles Frame n−1
Sending Tiles Frame n
Receiving Pixel Colors Frame n−1
Receiving Pixel Colors Frame n
Clients
Receiving Tiles Rendering Sending Pixel Colors Frame n−1
Executing OpenRT Commands Frame n
Receiving Tiles Rendering Sending Pixel Colors Frame n
t0
t1
Time
Frame n−1
Frame n

Figure 1: *At time t0 the OpenRT work scheduler starts to distribute work for frame n-1 and the application sends OpenRT commands for frame n to hide communication latency. Between t0 and t1 the scheduler receives color information and distributes work for frame n-1. At time t1 all color information for the current frame have been received and the corresponding frame is displayed. Even if a client has not finished all jobs for the current frame it already receives jobs for the next frame. This avoids client idle times. After time t1 work for frame n is distributed and the application sends OpenRT commands for frame n+1.*

In order to parse all geometry data an OpenGL-based VRML viewer was ported to OpenRT. Due to the very similar syntax porting was done without much effort. Communication between VRML application and library is done exclusively via the OpenRT API. Like RenderMan OpenRT provides the ability to attach arbitrary surface shader to specific geometry written in C/C++. Therefore the simulation code itself is implemented as a special surface shader.

4.2. Glass Simulation

As the reflective and refractive nature of the glass objects forms the core problem of the application, most work has been spent on the glass simulation. However, even with a very fast ray tracing system, some compromises had to be made to achieve interactive performance.

Currently, our system does not handle wavelength-specific effects, and uses only a single index of refraction for all color components. As such, prism-effects will not be simulated. Similarly, we currently ignore polarization effects.

For the actual reflection and refraction calculations, our glass shader is in principle not much different from other glass shaders used in other ray tracers. For each incoming ray, a reflection and a refraction ray are generated, recursively traced through the environment, and weighed with their respective physical contribution based on the respective Fresnel term [3].

However, some important differences do exist: In order to avoid infinite recursion for rays being ’trapped’ in a glass body, most ray tracers usually specify a maximum reflection depth of usually 4 to 8. For a simulation of such a complex glass body as used in our system, this would not be sufficient: A typical ray has to enter and leave the front glass, perhaps pass through two layers of glass surface on the front and two on the back of the light bulb before hitting the reflector, and has to take the same way out, not even considering multiple reflections inside the object. The maximum recursion level used to compute the different pixels can be seen in Figure 2.

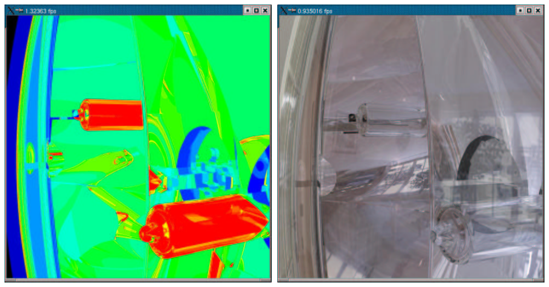


Figure 2: *The left image illustrates the recursion depth required for computing each pixel (accuracy 95%). Each color corresponds to a different depth: black(0), blue(≤5), magenta(≤10), green(≤15), yellow(≤20), red(>25). The right image shows the original image for comparisons (see Figure 8 for a color version).*

On the other hand, tracing rays up to a recursion level of 20 and more creates a huge performance problem: As glass is usually both reflective and refractive, tracing both rays at each recursion level would lead to an exponential explosion of rays traced per pixel (approximately one million for a single pixel with reflection depth 20). Sub-sampling the whole shading tree with Monte Carlo techniques such as Russian Roulette sampling at each recursion level is an option, but would require considerable effort to remove the Monte Carlo noise that would be extremely disturbing in an interactive setting.

To overcome this problem, we currently track the contribution of each path to the actual pixel. Purely local decision – e.g. by comparing each local weight to a minimum threshold- is not sufficient, as the accumulation of weights can lead to a ray becoming unimportant even though every local weight is above the threshold.

By not tracing paths that would have an insignificant pixel contribution, the rays traced per pixel can be reduced to a tolerable level. We define the contribution threshold as 'accuracy'. An accuracy of 99% means that a ray will be terminated if its pixel contribution is below 1%. Tests have shown that using an accuracy of more than 95% produces no visual improvement. Figure 3 presents the number of rays shot per pixel using an accuray of 95%.

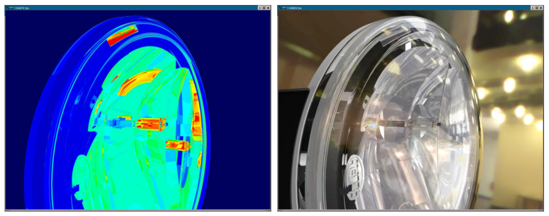


Figure 3: *The left image shows the number of rays used to compute each pixel (accuracy 95%). Each color represents a different number: black(< 1 rays per pixel), blue(≤10), magenta(≤20), green(≤30), yellow(≤40), red(>50). The right image shows the original image for comparisons (see Figure 9 for a color version).*

4.3. High-Dynamic-Range Environment Map

Due to the highly reflective nature of the headlight, almost all illumination comes from the environment. In order to simulate these effects, we took the common approach of using an environment map. Real environments usually cover a high dynamic range of radiance values. As such, using high-dynamic-range environment maps is important, as they increase the contrast and fidelity of the images. As our rendering engine uses full floating point accuracy for all lighting and shading computations, this was easily integrated into our visualization system. See Figure 4 for the impact of high-dynamic range effects. In principle, supporting high-dynamic range environments allows to put the virtual headlight prototype into any kind of measured environment.

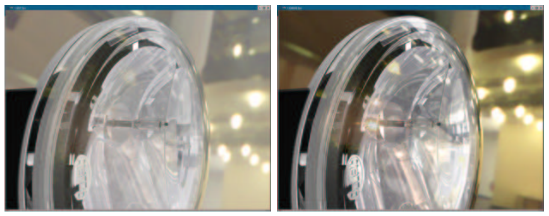


Figure 4: *Enhanced visual quality using a High Dynamic Range Environment Map: Low dynamic range rgb environment map on the left, and a high dynamic range environment map on the right (see Figure 9 for a color version).*

4.4. Tone Mapping

Using high-dynamic-range environment maps also requires tone mapping [17] before displaying the image. We currently use the tone mapping method proposed by Schlick [11]. As our system is purely software-based, implementing tone mapping is straight forward. However, some caution has to be taken due to parallelization: as a client sends only discretized 8-bit RGB values back to the server, tone mapping has to be applied on the clients. This is problematic because the tone mapping algorithms ususally need information about the whole image and a client renders only certain image tiles. We solve this problem by adapting the tone mapping parameters for the next frame based on the results for the current frame. The adapted parameters for the next frame will be transfered from the server to all clients. The improvements due to using high-dynamic-range environment maps and tone mapping can be seen in Figure 4.

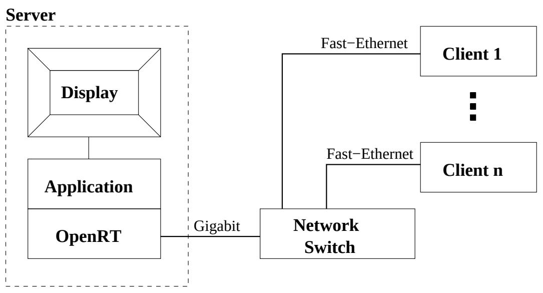


Figure 5: *All rendering clients are connected via a fully switched Fast-Ethernet network to the server which hosts the application.*

5. Results

All tests were performed on a cluster of 16 dual-AthlonMP 1800+ PCs, leading to a maximum of 32 CPUs. The rendering clients were interconnected via a fully switched Fast-Ethernet network. To provide enough bandwidth for transferring pixel colors back to the server, a Gigabit connection from the switch to the server was used (see Figure 5). Using this configuration we are able to achieve frame rates of up to 10 frames per second at video resolution (of 640x480). Figure 6 shows the nearly linear scalability offered by the OpenRT ray tracing core depending on the number of clients.

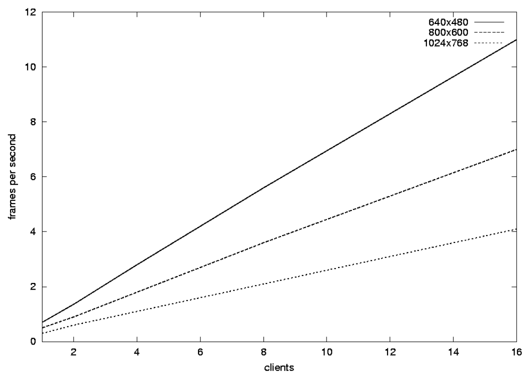


Figure 6: *Frame rate of our system at various image resolutions, and with different numbers of clients: Almost linear scalability is achieved even up to 16 clients (32 CPUs).*

Due to the limitations of the underlying network architecture (bandwidth,latency), our system currently cannot efficiently handle the broadcast of the entire geometry per frame. However, OpenRT offers to ability to interactively change the location of certain (static) parts of the scene [13]. As such, the light bulb can be moved out of the lamp, the glass cover can be taken off, etc. Other missing effects as discussed in Section 4.2 can be added easily.

As described before, we have been provided with reference images created with the previously used off-line visualization tools. Comparing our images to these reference images demonstrate that our system offers at least the same quality – but at interactive speeds. Comparing to actual photographs of a physical model of the headlight show that we are able to achieve a very realistic visualization, as can be seen in Figure 7.

6. Conclusions and Future Work

This paper demonstrates that interactive ray tracing is becoming an important alternative to current interactive visualization techniques. For some applications, like the headlight simulation discussed here, it is the only viable solution. As the trend towards geometrically more complex models and more realism continues, it seems that ray tracing will become increasingly important for future applications.

Of particular importance in this context is the flexibility of extending the basic ray tracing algorithms with custom programmed shaders. With ray tracing, shading is separated from visibility computations and is not constrained by the strict pipeline model of rasterization hardware. Therefore, different shaders are independent of each other and are much simpler to write.

Implementation of the headlight simulation required relatively little effort, as most of the actual work is performed by the underlying rendering engine. Therefore, modifications to the system can also be applied by non-expert users through editing the shaders.

Highly complex lighting simulations, as the one presented here, still require considerable compute power in the form of a small PC cluster. Even though Moore's law will help over time, it seems necessary to add some form of hardware support. Both CPU and GPU designers are currently investigating hardware support for ray tracing.

Our system depends mainly on dynamically loaded shaders, and does not interfere with the underlying interactive distributed ray tracing engine. Therefore, we can exploit the full performance of the OpenRT rendering engine, and can efficiently scale to many CPUs. Using a cluster of 16 Dual-AthlonMP 1800+ PCs, we are able to achieve frame rates up to 10 frames per second for the complete reflection simulation at a resolutions of 640x480 pixels.

Due to the OpenRT's ability to handle complex scenes, we are able to directly handle the high geometric complexity of 800.000 triangles without the need for geometric simplifications and approximations. As such, our system is fully automatic, and can greatly reduce the turn-around times in the industrial design process.

In future, we are considering to make our system available to the industry by integrating it into commercial applications. Furthermore, our system can easily be adapted to solve similar problems in the industry, e.g. simulating reflections in car windshields, or in instruments of aircraft cockpits. Finally, it would be desirable to integrate the system with a full lighting simulation in order to visualize the lighting effects of the headlight.

Acknowledgements

We would like to thank all the people that have contributed to this paper: Markus Wagner has adapted his VRML-based scene viewer for use with OpenRT. Philippe Bekaert has graciously provided access to his excellent VRML-97 library [2] for building the VRML viewer, and has also provided the high-dynamic-range images of the Max-Planck-Institut foyer used for the environment map.

Finally, we would like to thank the headlight manufacturer Hella for granting permission to use their model for this publication.

References

1. Alias Wavefront. *Maya*. http://www.aliaswavefront.com. 2

2. Philippe Bekaert. Extensible scene graph manager. http://www.cs.kuleuven.ac.be/ graphics/XRML/, August 2001. 6

3. J. Foley, A. van Dam, S. Feiner, and J. Hughes. *Computer Graphics, Principles and Practice, 2nd Edition in C*. Addison-Wesley, 1996. 3

4. A. Keller. *Quasi-Monte Carlo Methods for Realistic Image Synthesis*. PhD thesis, University of Kaiserslautern, 1998. 3

5. Michael J. Muuss. Towards real-time ray-tracing of combinatorial solid geometric models. In *Proceedings of BRL-CAD Symposium '95*, June 1995. 2

6. Michael J. Muuss and Maximo Lorenzo. High-resolution interactive multispectral missile sensor simulation for atr and dis. In *Proceedings of BRL-CAD Symposium '95*, June 1995. 2

7. Steven Parker, Peter Shirley, Yarden Livnat, Charles Hansen, and Peter Pike Sloan. Interactive ray tracing for isosurface rendering. In *IEEE Visualization '98*, pages 233–238, October 1998. 2

8. Steven Parker, Peter Shirley, Yarden Livnat, Charles Hansen, and Peter Pike Sloan. Interactive ray tracing. In *Interactive 3D Graphics (I3D)*, pages 119–126, april 1999. 2

9. Steven Parker, Peter Shirley, and Brian Smits. Single sample soft shadows. Technical Report UUCS-98-019, Computer Science Department, University of Utah, October 1998. http://www.cs.utah.edu/ bes/papers/coneShadow. 2

10. Erik Reinhard. *Scheduling and Data Management for Parallel Ray Tracing*. PhD thesis, University of East Anglia, 1995. 3

11. Christophe Schlick. High dynamic range pixels. *Graphics Gems*, 4:422–429, 1994. 4

12. Ingo Wald, Carsten Benthin, and Philipp Slusallek. OpenRT – A Flexible and Scalable Rendering Engine for Interactive 3D Graphics. submitted for publication, meanwhile available as a Technical Report, TR-2002-01, Saarland University, available at http://www.openrt.de/Publications, 2002. 3

13. Ingo Wald, Carsten Benthin, and Philipp Slusallek. A simple and practical method for interactive ray tracing of dynamic scenes. Technical report, Computer Graphics Group, Saarland University, http://www.openrt.de, 2002. 5

14. Ingo Wald, Carsten Benthin, Markus Wagner, and Philipp Slusallek. Interactive rendering with coherent ray tracing. *Computer Graphics Forum*, 20(3), 2001. 2

15. Ingo Wald, Thomas Kollig, Carsten Benthin, Alexander Keller, and Philipp Slusallek. Interactive global illumination. Technical report, Computer Graphics Group, Saarland University, 2002. to be published at EUROGRAPHICS Workshop on Rendering 2002, available online at http://www.openrt.de/Publications. 2

16. Ingo Wald, Philipp Slusallek, and Carsten Benthin. Interactive distributed ray tracing of highly complex models. In *Proceedings of the 12th EUROGRPAHICS Workshop on Rendering*, June 2001. London. 2

17. G. Ward. A Contrast-Based Scalefactor for Luminance Display. In P. Heckbert, editor, *Graphics Gems IV*, pages 415–421. Academic Press Professional, 1994. 4

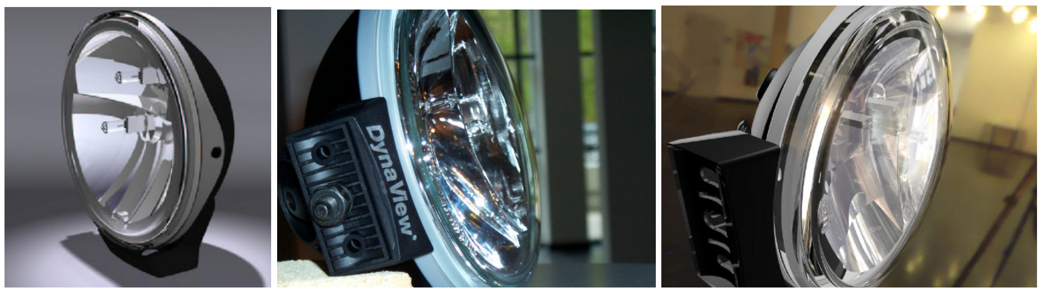


Figure 7: *Rendering quality achieved by our visualization system: The image in the middle illustrates the reference quality produced by an off-line ray tracer, and the right image shows the quality achieved by our system. For comparison, the third image is a photo of the physical model.*

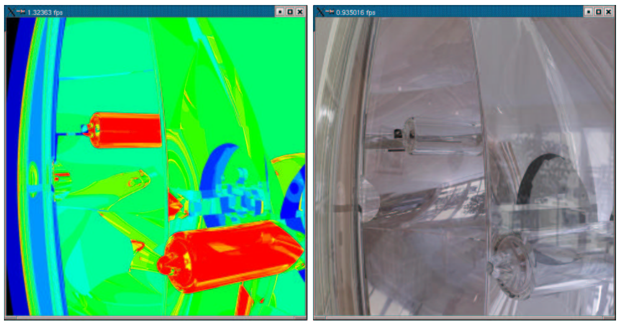


Figure 8: *Recursion depth required for computing each pixel (accuracy 95%). Each color corresponds to a different depth: black(0), blue(≤5), magenta(≤10), green(≤15), yellow(≤20), red(>25). The right image shows the original image for comparisons.*

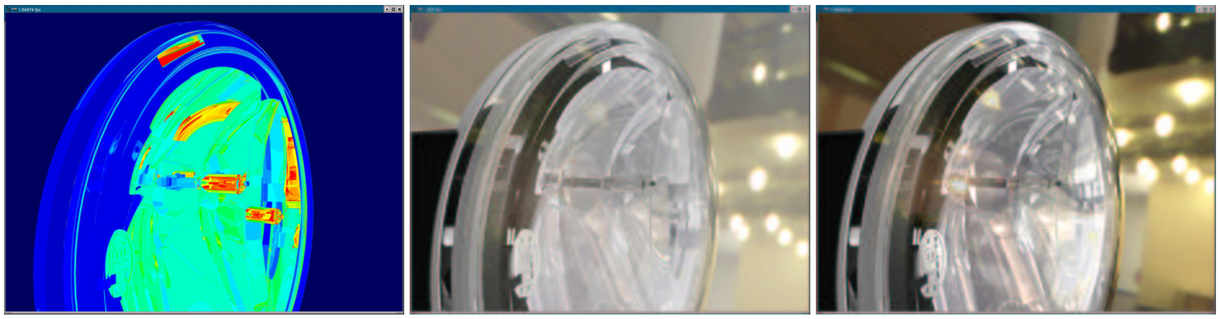


Figure 9: *Number of rays used to compute each pixel (accuracy 95%). Each color represents a different number: black(< 1 rays per pixel), blue(≤10), magenta(≤20), green(≤30), yellow(≤40), red(>50). The second and third images illustrate enhanced visual quality using a High Dynamic Range Environment Map: low dynamic range (second image), high dynamic range (third image).*