Ray Tracing for the Movie ‘Cars’

Per H. Christensen* Julian Fong David M. Laur Dana Batali

Pixar Animation Studios

Abstract

This paper describes how we extended Pixar’s RenderMan renderer with ray tracing abilities. In order to ray trace highly complex scenes we use multiresolution geometry and texture caches, and use ray differentials to determine the appropriate resolution. With this method we are able to efficiently ray trace scenes with much more geometry and texture data than there is main memory. Movie-quality rendering of scenes of such complexity had only previously been possible with pure scanline rendering algorithms. Adding ray tracing to the renderer enables many additional effects such as accurate reflections, detailed shadows, and ambient occlusion.

The ray tracing functionality has been used in many recent movies, including Pixar’s latest movie ‘Cars’. This paper also describes some of the practical ray tracing issues from the production of ‘Cars’.

1 Introduction

Pixar’s RenderMan renderer (PRMan) is a robust production renderer that is used for many CG movies and special effects [1]. PRMan uses the REYES scanline rendering algorithm [4]. About five years ago, at the request of our external customers, we started a project to add on-demand ray tracing to PRMan.

At roughly the same time, John Lasseter and his team started working on ‘Cars’, a movie that would turn out to be an ideal testing ground and showcase for the ray tracing functionality. There were two main rendering challenges in making the movie. First, ‘Cars’ has scenes that are much more complex than past Pixar movies; for example, wide desert landscapes with many sagebrush and thorn-covered cacti, and a racing oval with 75,000 cars as spectators. Second, ray tracing effects such as correct reflections, shadows, and ambient occlusion were needed to get the desired artistic look. Ray tracing these very complex scenes in manageable time was quite a challenge.

The REYES algorithm is very efficient at handling complex scenes. For ray tracing, we use ray differentials [6, 14] to select the optimal tessellation level of surfaces and the proper MIP map level for textures. A multiresolution geometry cache keeps recently tessellated geometry ready for fast access. Similarly, a multiresolution texture cache keeps recently accessed texture tiles ready for fast access. This combination of ray differentials and caching makes ray tracing of very complex scenes feasible.

This paper first gives a more detailed motivation for the use of ray tracing in ‘Cars’, and lists the harsh rendering requirements in the movie industry. It then gives an overview of how the REYES algorithm deals with complex scenes and goes on to explain our work on efficient ray tracing of equally complex scenes. An explanation of our hybrid rendering approach, combining REYES with ray tracing, follows. Finally we measure the efficiency of our method on a test scene, and present a few production details from the use of ray tracing for ‘Cars’.

Please refer to Christensen et al. [3] for an overview of previous work such as the Toro [10] and Kilauea [7] renderers. Recent related work includes an interactive out-of-core renderer by Wald et al. [15], the Razor project by Stoll et al. [13], and level-of-detail representations for efficient ray tracing of simplified geometry [2, 18]. The focus of those projects is more on interactive or real-time rendering than on movie-quality images.

2 Motivation: why ray tracing?

There are several reasons why the directors chose to use ray tracing for ‘Cars’: realistic reflections, sharp shadows, and ambient occlusion.

Real cars are usually shiny, and the reflections are an important visual cue to the shape and material of the car. With scanline algorithms such as REYES, reflections are usually computed using environment maps. However, this approach breaks down when the reflected points are close to the reflecting points, or if the reflected points are also reflectors. Figure 1 shows two early pre-production test images of Luigi, a yellow Fiat 500 “Topolino”. The images compare environment mapping with ray traced reflections. While the environment map reflection in figure 1(left) shows a good approximation of the distant environment, it does not capture interreflections such as the reflections of the eyes in the hood seen in figure 1(right).

Shadows give strong cues about the lighting and about the placement of objects relative to each other. Scanline algorithms traditionally compute shadows using shadow maps [11], and there are still many cases where a shadow map is the most efficient way of generating high-quality shadows. However, some of the scenes in ‘Cars’ use expansive sets but also have a lot of tiny, detailed geometry. This can lead to resolution problems in shadow maps. Furthermore, many scenes contain thousands of light sources, so keeping

*e-mail: {per,jfong,dml,dana}@pixar.com

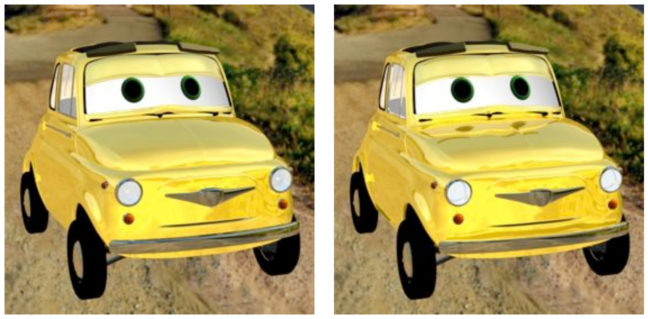

Figure 1: Reflection test: (left) with environment map. (right) with environment map and ray-traced interreflections.

track of the shadow map files can become an asset management problem. Figure 2 shows a frame from the final movie with Lightning McQueen leading a race. This is an example of a large scene requiring very fine shadow detail. The scene contains nearly 1000 light sources. Using ray traced shadows eliminates the resolution and asset management problems.

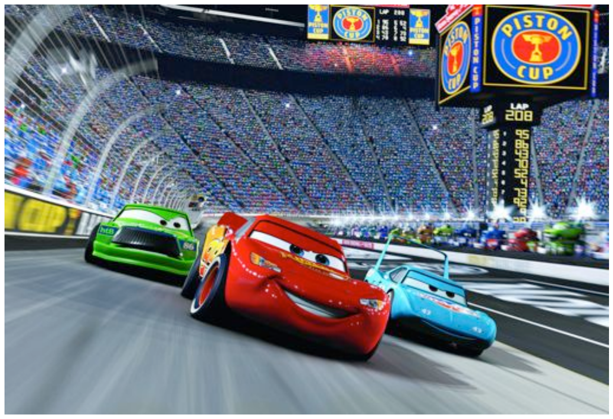

Figure 2: Ray-traced sharp shadows.

Another use of ray tracing is for ambient occlusion. Ambient occlusion [9, 19] is a measure of how much light reaches a point from a uniformly lit hemisphere. Ambient occlusion is widely used in movie production since it gives a good indication of creases on surfaces and spatial proximity of objects, and is a cheap (but crude) approximation of global illumination. Figure 3 shows ambient occlusion on three cars in Radiator Springs. Ambient occlusion is usually computed by shooting many rays to sample the coverage of the hemisphere above each point.

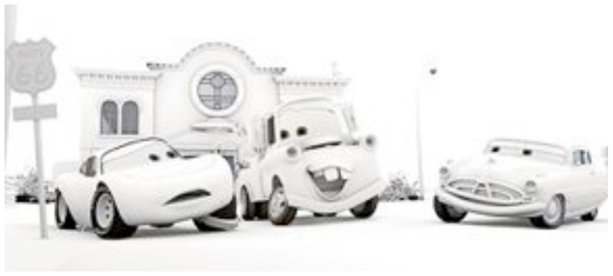

Figure 3: Ambient occlusion.

3 Movie rendering requirements

The rendering requirements in the movie industry are extremely harsh:

- Scene geometry is far too large to fit in memory in tessellated form.
- Many surfaces are displacement-mapped.
- There may be thousands of textures (too many to fit all in memory at full resolution) to control reflection parameters and displacements.
- There can be thousands of light sources.
- All illumination and surface reflection characteristics are controlled by fully programmable, complex shaders.

In addition, images are typically rendered at high resolution with motion blur and depth of field. Furthermore, no spatial or temporal aliasing is acceptable: no staircase effects, “crawlies”, popping, etc.

4 REYES rendering of complex scenes

The REYES algorithm has many desirable properties such as coherent shader execution, coherent access to geometry and texture data, simple differential calculations, efficient displacement, fast motion blur and depth-of-field, and the ability to render very complex scenes.

The REYES algorithm divides each surface into smaller patches, and each patch is tessellated into a regular grid of tiny quadrilaterals (aka. micropolygons). The small patches are easy to place into one (or a few) image tiles. A patch can be thrown away if it is entirely behind other opaque patches.

Shading is done at the vertices of the grid. Shading an entire grid at a time is advantageous for data coherency and differential calculations as needed for e.g. texture filter sizes. The shading rate is decoupled from visibility calculations: there is typically only one shading point (grid vertex) per pixel on average, while the pixel sampling rate typically is 4×4 for static images and even higher for images with motion blur.

The REYES algorithm is very good at handling complex scenes. First, it can completely ignore all objects outside the viewing frustum. Second, it renders only one small image tile (typically 16×16 or 32×32 pixels) at a time. This means that the computation only needs a small fraction of the scene geometry and textures at any given time. This maximizes geometry coherency and minimizes the number of tessellated surfaces that need to be kept in memory at the same time. Furthermore, as soon as a surface has been rendered, its data can be removed from memory since they will no longer be needed. Surfaces are divided and tessellated according to their size on screen, so large distant surfaces automatically get a coarse representation. Likewise, distant objects only need coarse textures, so only coarse levels in the texture MIP maps will be accessed for those objects. For all these reasons, REYES deals gracefully with very complex geometry and huge amounts of texture.

5 Ray tracing of complex scenes

Ray tracing also has several wonderful properties as a rendering algorithm: it is conceptually simple, its run-time only grows logarithmically with scene complexity, and it can easily be parallelized. But ray tracing has an important limitation: it is *only efficient if the scene fits in memory*. If the scene does not fit in memory, virtual memory thrashing slows the rendering down by orders of magnitude.

Ray tracing of complex scenes is inherently harder than REYES rendering of similar scenes. First, objects can't be rejected just because they are outside the viewing frustum: they may cast shadows on visible objects or be reflected by them. Second, even if the image is rendered one tile at a time and the directly visible geometry is ray traced very coherently, the reflection and shadow rays will access other geometry (and textures) in a less coherent manner. Even worse, the rays traced for diffuse interreflections and ambient occlusion are completely incoherent and may access any part of the scene at any time. Hence, we can't delete an object even when the image tile it is directly visible in has been rendered — a ray from some other part of the scene may hit that object at any time during rendering.

For all the reasons listed above, ray tracing of very complex scenes may seem like a daunting task. However, the use of ray differentials and multiresolution geometry and texture caches makes it tractable.

5.1 Ray differentials

A ray differential describes the differences between a ray and its — real or imaginary — "neighbor" rays. Igehy's ray differential method [6] traces single rays, but keeps track of the differentials as the rays are propagated and reflected. The differentials give an indication of the beam size that each ray represents, as illustrated in figure 4. The curvature at surface intersection points determines how the ray differentials and their associated beams change after specular reflection and refraction. For example, if a ray hits a highly curved, convex surface, the specularly reflected ray will have a large differential (representing highly diverging neighbor rays).

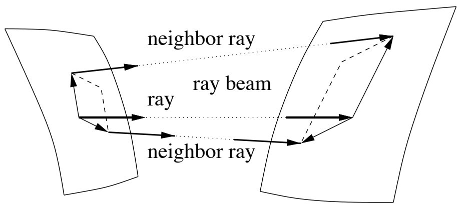

Figure 4: Rays and ray beam.

Suykens and Willems [14] generalized ray differentials to glossy and diffuse reflections. For distribution ray tracing of diffuse reflection or ambient occlusion, the ray differential corresponds to a fraction of the hemisphere. The more rays are traced from the same point, the smaller the subtended hemisphere fraction becomes. If the hemisphere fraction is very small, a curvature-dependent differential (as for specular reflection) becomes dominant.

In Christensen et al. [3] we provided a comprehensive analysis of ray differentials vs. ray coherency. We observed that in all practical cases, *coherent rays have narrow beams and incoherent rays have wide beams*. This is an important and very fortunate relationship that enables ray tracing of very complex scenes. We exploit that relationship in the following sections by designing caches that utilize it.

5.2 Multiresolution tessellation

REYES chooses tessellation rates for a surface patch depending on viewing distance, surface curvature, and optionally also view angle. In our implementation, the highest tessellation rate used for ray tracing of a patch is the same as the REYES tessellation rate for that patch. Subsets of the vertices are used for coarser tessellations, which ensures that the bounding boxes are consistent: a (tight) bounding box of the finest tessellation is also a bounding box for the coarser tessellations. The coarsest tessellation is simply the four corners of the patch. One can think of the various levels of tessellation as a MIP map of tessellated geometry [17]. Figure 5 shows an example of five tessellations of a surface patch; here, the finest tessellation rate is 14×11.

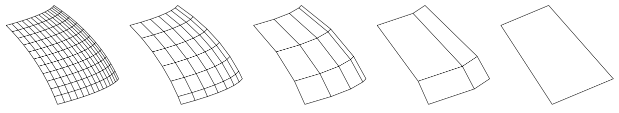

Figure 5: Multiresolution tessellation example for a surface patch: 14×11 quads, 7×6 quads, 4×3 quads, 2×2 quads, and 1 quad.

In our first implementation [3], the tessellation rates used for ray tracing were 16×16, 8×8, …, 1. However, using the REYES tessellation rates and subsets thereof has two advantages: there are fewer quads to test for ray intersection (since we always rounded the REYES tessellations rates up for ray tracing), and there are fewer self-intersection problems if we use a hybrid rendering method (since the vertices of the two representations always coincide when using the current tessellation approach).

The example in figure 5 is a rectangular surface patch. We have a similar multiresolution tessellation method for triangular patches which arise from triangle meshes and Loop subdivision surfaces.

5.3 Multiresolution geometry cache

We tessellate surface patches on demand and cache the tessellations. As shown above, we use up to five different levels of tessellation for each surface patch. However, we have chosen not to have five geometry caches; instead we use three caches and create the two intermediate tessellations by picking (roughly) a quarter of the vertices from the next finer tessellation level.

In our implementation, the coarse cache contains tessellations with 4 vertices (1 quad), the medium cache contains tessellations with at most 25 vertices, and the fine cache contains all larger tessellations (at most 289 vertices). The size of the geometry caches can be specified by the user. By default, the size is 20 MB per thread allocated for each of the three caches. Since the size of the tessellations differ so much, the maximum capacity (number of slots) of the coarse cache is much higher than for the medium cache, and the medium cache has much higher capacity than the fine cache. We use a least-recently-used (LRU) cache replacement scheme.

For ray intersection tests, we choose the tessellation where the quads are approximately the same size as the ray beam cross-section. We have observed that accesses to the fine and medium caches are usually very coherent. The accesses to the coarse cache are rather incoherent, but the capacity of that cache is large and its tessellations are fast to recompute.

5.4 Multiresolution texture cache

Textures are stored on disk as tiled MIP maps with 32×32 pixels in each tile. The size of the texture cache is chosen by the user; the default size is 10 MB per thread.

As with the geometry cache, the ray beam size is used to select the appropriate texture MIP map level for texture lookups. We choose the level where the texture pixels are approximately the same size as the ray beam cross-section. Incoherent texture lookups have wide ray beams, so coarse MIP map levels will be chosen. The finer MIP map levels will only be accessed by rays with narrow ray beams; fortunately those rays are coherent so the resulting texture cache lookups will be coherent as well.

6 Other ray-tracing implementation issues

This section describes other efficiency and accuracy aspects of our implementation of ray tracing in PRMan.

6.1 Spatial acceleration data structure

Good spatial acceleration structures are essential for efficient ray tracing. We use a bounding volume hierarchy, the Kay-Kajiya tree [8]. This data structure is a good compromise between memory overhead, construction speed, and ray traversal speed. The data structure is built dynamically during rendering. While we are quite satisfied with the performance of the Kay-Kajiya tree, it is certainly worth considering other acceleration data structures in the future.

There is one more level of bounding volumes for the finest tessellations: bounding boxes for groups of (up to) 4×4 quads. These bounding boxes are stored in the fine geometry cache along with the tessellated points.

6.2 Displacement-mapped surfaces

Displacement shaders complicate the calculation of ray intersections. If the surface has a displacement shader, the shader is evaluated at the tessellation vertices to get the displaced tessellation. The bounding box of the displaced vertices is computed, and the Kay-Kajiya tree is updated with the new bounding box.

Although there exist techniques for direct ray tracing of displaced surfaces [12], we have found that applying displacement shaders to tessellated grids gives much faster rendering times — at least if the displaced tessellations are cached.

Each object that has displacement must have a pre-specified upper bound on the displacement; such bounds are important for the efficiency of both REYES rendering and ray tracing. Without a priori bounds, any surface might end up anywhere in the scene after displacement, and this makes image tiling or building an acceleration data structure futile.

The value of tight bounding boxes is so high that even if the first access to a displaced surface patch only needs a coarse tessellation, we compute a fine tessellation, run the displacement shader, compute the tight bounding box, update the Kay-Kajiya tree, and throw away those tessellation points that aren't needed. This is a one-time cost that is amortized by the reduction in the number of rays that later have to be intersection-tested against that surface patch. Tight bounding boxes allow us to ray trace displaced surfaces rather efficiently.

6.3 Motion blur

PRMan assumes that all motion is piecewise linear. There is a tessellation for the start of each motion segment plus a tessellation for the end of the last segment. These tessellations are computed on demand and stored in the geometry cache. For ray intersection tests, the tessellated vertex positions are interpolated according to the ray time, creating a grid at the correct time for the ray.

The Kay-Kajiya node for a moving surface patch contains a bounding box for the start of each motion segment plus a bounding box for the end of the last segment. The bounding boxes are linearly interpolated to find the bounding box that corresponds to the ray time.

6.4 SIMD speedups

We use SIMD instructions (SSE and AltiVec) to speed up the computation of ray intersections. When intersection-testing a Kay-Kajiya node bounding box, we test multiple slabs (x, y, and z planes) at once. The bounding boxes for groups of 4×4 quads of finely tessellated patches (stored in the fine geometry cache) are intersection-tested four boxes at a time. And tessellated patches are intersection-tested 4 triangles (2 quads) at a time.

Since each ray is traced independently, our SIMD speedups do not rely on ray coherency. In contrast, Wald et al. [16] used SIMD instructions for tracing four rays at a time. This works fine if the rays are coherent, but fails to deliver a speedup if the rays are incoherent.

6.5 Shading at ray hit points

To compute the shading results at ray hit points, we could shade the vertices of ray tracing tessellations, store the colors in a cache, and interpolate the colors at the hit points. This would be a straighforward generalization of the REYES shading approach. But unfortunately the shading colors are usually view-dependent — highlights move around depending on the viewing direction, for example. The shader may also compute different results depending on the ray level for non-realistic, artistic effects.

Instead we create 3 shading points for each ray hit (similar to Gritz and Hahn [5]). One shading point is the ray hit point, and the other two shading points are created using the ray differentials at the ray hit point. This way, the shader can get meaningful differentials for texture filtering, computation of new ray directions, etc. While most shading functions are executed on all three shading points, some are only executed at the ray hit point — for example, rays are only traced from the ray hit point.

It is worth emphasizing that for production scenes, the dominant cost of ray tracing is typically not the computation of ray intersections, but the evaluation of displacement, surface, and light source shaders at the ray hit points. (This is also why ambient occlusion has gained popularity in movie production so quickly as an alternative to more accurate global illumination solutions: even though it takes a lot of rays to compute ambient occlusion accurately, there are no shader evaluations at the ray hit points.)

6.6 Avoiding cracks

Visible cracks can occur if the two grids sharing an edge have different tessellation rates. See figure 6 for an illustration. This is a potential problem both for REYES rendering and for ray tracing, and has to be dealt with explicitly.

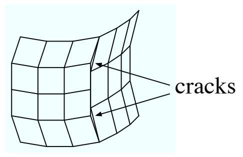

Figure 6: Mismatched tessellation and potential cracks.

The easiest way to fix these cracks requires that all dicing rates are powers of 2. Then every other vertex on the fine tessellation side of the edge can be moved to lie along the straight line between its two neighbor points. This ensures that the vertices on both sides of the edge are consistent so there are no cracks. However, such power-of-two tessellation (aka. "binary dicing") introduces too many shading points compared to more flexible tessellation rates, and it is therefore too expensive in practice when shaders are a bottleneck.

Instead, PRMan uses an alternative algorithm that glues all edges together [1, sec. 6.5.2], thus avoiding cracks. We call this algorithm "stitching". Stitching moves the tessellation points so that the grids overlap, and introduces new quads if needed to fill remaining gaps. For REYES rendering, new quads that are introduced are never shaded, they only copy colors from their nearest neighbor.

We use a similar stitching algorithm for ray tracing. If any new quads are generated, they are stored in the geometry cache.

The tessellation rate of each surface patch is determined from the size of the patch bounding box relative to the ray beam size. Hence, the tessellation within a patch is kept consistent for each ray, and there are no cracks internally within a patch. (Such cracks are sometimes refered to as "tunnelling" [13].)

7 Hybrid rendering: REYES and rays

PRMan uses a combination of the REYES algorithm and on-demand ray tracing. REYES is used to render objects that are directly visible to the camera. Shading those objects can cause rays to be traced. With this hybrid approach there are no camera rays; all the first-level rays originate from REYES shading points.

With the methods described above, both the REYES and ray tracing algorithms can handle very complex scenes. So why not use ray tracing for primary rendering? It would unify our algorithm, eliminate large parts of the PRMan code base, and make software maintenance easier. However, so far the advantages of coherency, well-defined differentials, graceful handling of displacement mapping, efficient motion blur and depth-of-field, decoupling of shading rate and pixel sample rate, etc. makes the REYES algorithm hard to beat for movie-quality rendering.

8 Test on a complex scene

Figure 7 shows a test example, a scene with 15 cars. The cars are explicitly copied, not instanced. Each car consists of 2155 NURBS patches, many of which have trim curves. The cars have ray-traced reflections (maximum reflection depth 4) and sharp shadows, while the ground is shaded with ray-traced ambient occlusion. This gives a mix of coherent and incoherent rays. The image resolution is 2048×1536 pixels.

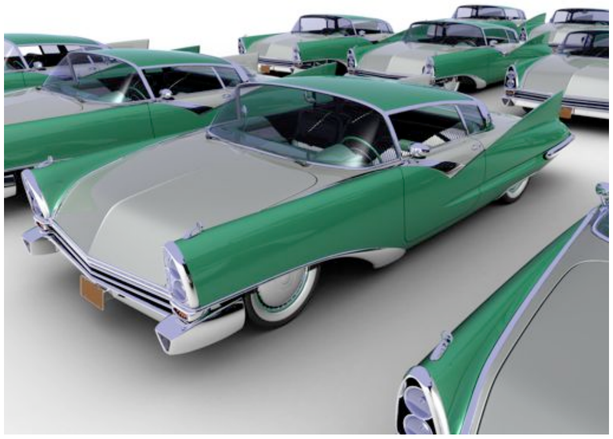

Figure 7: Shiny cars on ambient occlusion ground.

The tests were run on an Apple G5 computer with two 2 GHz PowerPC processors and 2 GB memory.

During rendering the car surfaces are divided into 1.3 million surface patches, corresponding to 383 million vertices and 339 million quads (678 million triangles) at full tessellation. Storing all full tessellations would consume 4.6 GB. Instead, with multiresolution caching, the scene uses a total of 414 MB: Geometry cache sizes are set to their default value (20 MB per cache per thread), a total of 120 MB. The Kay-Kajiya tree uses around 59 MB per thread. The top-level object descriptions use 126 MB plus 50 MB for trim curves.

Rendering this image used 111 million diffuse rays, 37 million specular rays, and 26 million shadow rays. The rays cause 1.2 billion ray-triangle intersection tests. With multiresolution geometry caching, the render time is 106 minutes. The three geometry caches have a total of 675 million lookups and cache hit rates of 91.4%–95.2%. In contrast, if the ray differentials are ignored (the REYES tessellation rates are used for all ray intersection tests) and no caching is done, the render time is 15 hours 45 minutes — almost 9 times slower.

More exhaustive tests and results can be found in Christensen et al. [3]. Although the render times reported there are quite obsolete by now, the time ratios and relative speedups are still representative.

9 Ray tracing for 'Cars'

Figure 8 shows "beauty shots" of two of the characters from 'Cars'. These images demonstrate ray traced reflections, shadows, and ambient occlusion.

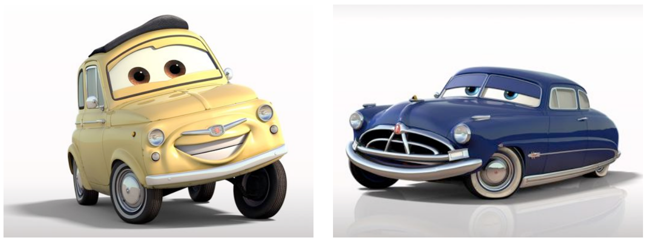

Figure 8: More cars with ray traced effects: Luigi (left) and Doc Hudson (right).

For convex surfaces like a car body, distant reflections do not need to be very accurate. In many shots, the maximum distance that a ray can hit geometry was set to 12 meters. If the ray didn't hit anything within that distance, it would use a single held environment map instead.

The reflections in the movie were usually limited to a single level of reflection. There were only a few shots with two levels of reflection, they are close-ups of chrome parts that needed to reflect themselves multiple times. Figure 9 shows an example.

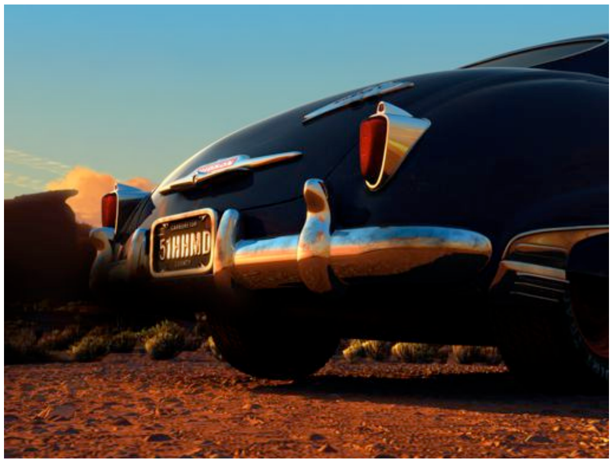

Figure 9: Chrome bumper with two levels of ray-traced reflection.

Figure 10 shows all the main characters in the 'Cars' movie. This is an example of a very complex scene with many shiny cars. The shiny cars reflect other cars, as shown in the three close-ups. The image also shows ray-traced shadows and ambient occlusion.

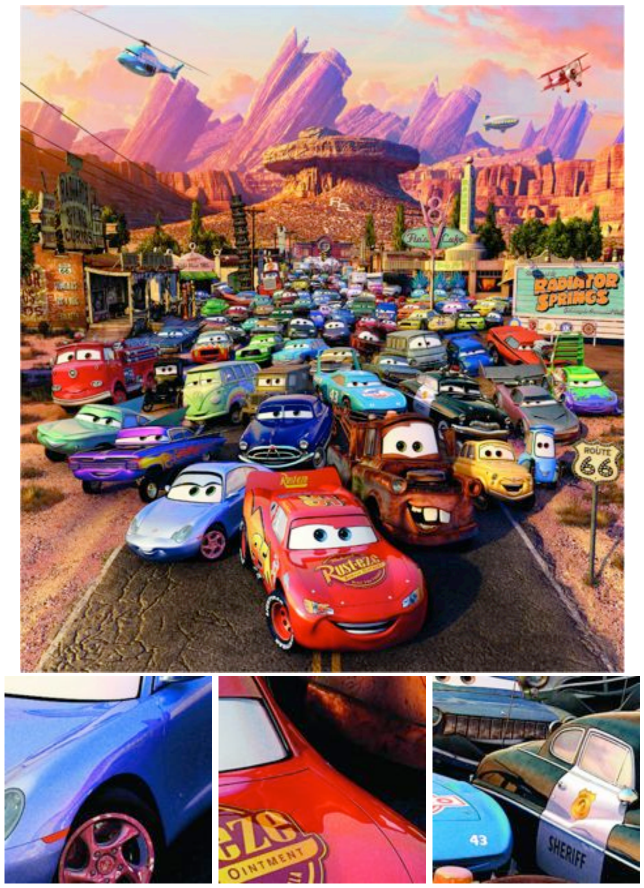

Figure 10: The cast of 'Cars' with three close-ups showing ray-traced reflections.

10 Conclusion

PRMan uses the REYES algorithm for rendering directly visible objects, and offers on-demand ray tracing for reflections, shadows, ambient occlusion, etc. It uses a multiresolution geometry cache and a multiresolution texture cache, and uses ray differentials to select the appropriate resolutions. Due to the observation that coherent rays have narrow beams while incoherent rays have wide beams, the method is efficient for ray tracing of complex scenes. The ray tracing functionality has been used for several movies, including Pixar's 'Cars'.

Acknowledgements

We would like to thank our colleagues in Pixar's RenderMan Products group for providing an inspiring and creative environment and for many helpful discussions. Loren Carpenter implemented most of the SIMD speedups and Brian Smits helped us optimize other aspects of the ray-tracing efficiency. Tony Apodaca headed the Cars "Nitro" speed team. Erik Smitt clarified some of the movie production details.

References

[1] Anthony A. Apodaca and Larry Gritz. *Advanced RenderMan — Creating CGI for Motion Pictures*. Morgan Kaufmann, 2000.

[2] Per H. Christensen. Point clouds and brick maps for movie production. In Markus Gross and Hanspeter Pfister, editors, *Point-Based Graphics*, chapter 8.4. Morgan Kaufmann, 2006. (In press).

[3] Per H. Christensen, David M. Laur, Julian Fong, Wayne L. Wooten, and Dana Batali. Ray differentials and multiresolution geometry caching for distribution ray tracing in complex scenes. *Computer Graphics Forum (Proceedings of Eurographics 2003)*, 22(3):543–552, 2003.

[4] Robert L. Cook, Loren Carpenter, and Edwin Catmull. The Reyes image rendering architecture. *Computer Graphics (Proceedings of SIGGRAPH '87)*, 21(4):95–102, 1987.

[5] Larry Gritz and James K. Hahn. BMRT: A global illumination implementation of the RenderMan standard. *Journal of Graphics Tools*, 1(3):29–47, 1996.

[6] Homan Igehy. Tracing ray differentials. *Computer Graphics (Proceedings of SIGGRAPH '99)*, pages 179–186, 1999.

[7] Toshiaki Kato. The Kilauea massively parallel ray tracer. In Alan Chalmers, Timothy Davis, and Erik Reinhard, editors, *Practical Parallel Rendering*, chapter 8. A K Peters, 2002.

[8] Timothy L. Kay and James Kajiya. Ray tracing complex scenes. *Computer Graphics (Proceedings of SIGGRAPH '86)*, 20(4):269–278, 1986.

[9] Hayden Landis. Production-ready global illumination. In *SIGGRAPH 2002 course note #16*, pages 87–102, 2002.

[10] Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. Rendering complex scenes with memory-coherent ray tracing. *Computer Graphics (Proceedings of SIGGRAPH '97)*, pages 101–108, 1997.

[11] William T. Reeves, David H. Salesin, and Robert L. Cook. Rendering antialiased shadows with depth maps. *Computer Graphics (Proceedings of SIGGRAPH '87)*, 21(4):283–291, 1987.

[12] Brian Smits, Peter Shirley, and Michael M. Stark. Direct ray tracing of displacement mapped triangles. In *Rendering Techniques 2000 (Proceedings of the 11th Eurographics Workshop on Rendering)*, pages 307–318, 2000.

[13] Gordon Stoll, William R. Mark, Peter Djeu, Rui Wang, and Ikrima Elhassan. Razor: an architecture for dynamic multiresolution ray tracing. Technical Report TR-06-21, University of Texas at Austin, 2006.

[14] Frank Suykens and Yves D. Willems. Path differentials and applications. In *Rendering Techniques 2001 (Proceedings of the 12th Eurographics Workshop on Rendering)*, pages 257–268, 2001.

[15] Ingo Wald, Andreas Dietrich, and Philipp Slusallek. An interactive out-of-core rendering framework for visualizing massively complex models. In *Rendering Techniques 2004 (Proceedings of the Eurographics Symposium on Rendering 2004)*, pages 81–92, 2004.

[16] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Michael Wagner. Interactive rendering with coherent raytracing. *Computer Graphics Forum (Proceedings of Eurographics 2001)*, 20(3):153–164, 2001.

[17] Lance Williams. Pyramidal parametrics. *Computer Graphics (Proceedings of SIGGRAPH '83)*, 17(3):1–11, 1983.

[18] Sung-Eui Yoon, Christian Lauterbach, and Dinesh Manocha. R-LODs: Fast LOD-based ray tracing of massive models. In *Proceedings of Pacific Graphics '06*, 2006.

[19] Sergei Zhukov, Andrei Iones, and Gregorij Kronin. An ambient light illumination model. In *Rendering Techniques '98 (Proceedings of the 9th Eurographics Workshop on Rendering)*, pages 45–55, 1998.