# Razor: An Architecture for Dynamic Multiresolution Ray Tracing

PETER DJEU
University of Texas at Austin
WARREN HUNT
Intel Labs
RUI WANG
University of Massachusetts Amherst
IKRIMA ELHASSAN
Microsoft Corporation
GORDON STOLL
Intel Labs
and
WILLIAM R. MARK
Intel Labs

Recent work demonstrates that interactive ray tracing is possible on desktop systems, but there is still much debate as to how to most efficiently support advanced visual effects such as soft shadows, smooth freeform surfaces, complex shading, and animated scenes. With these challenges in mind, we reconsider the options for designing a rendering system and present *Razor*, a new software rendering architecture for distribution ray tracing designed to produce high-quality images with high performance on future single-chip many-core hardware. Razor includes two noteworthy capabilities: a set of techniques for quickly building the kd-tree acceleration structure on demand every frame from a scene graph and a system design that allows for crack-free multiresolution geometry with each ray independently choosing its geometry resolution. Razor's per-frame kd-tree build is designed to robustly handle arbitrarily scene animation, while its per-ray multiresolution geometry provides continuous level of detail driven by ray and path differentials. Razor also decouples shading from visibility computations using a two-phase shading scheme inspired by the REYES system, and caches tessellated representations of freeform surfaces at multiple levels of detail.

We present experimental results gathered from a prototype system implemented on eight CPU cores, and discuss which aspects of the system are most successful and which would benefit from further investigation.

Categories and Subject Descriptors: I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism—*Raytracing*; I.3.3 [**Computer Graphics**]: Picture/Image Generation—*Display algorithms*

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Distribution ray tracing, real-time ray tracing, hierarchical build, lazy build, parallel build, level of detail, multiresolution, subdivision surfaces

**ACM Reference Format:**

Djeu, P., Hunt, W., Wang, R., Elhassan, I., Stoll, G., and Mark, W. R. 2011. Razor: An architecture for dynamic multiresolution ray tracing. ACM Trans. Graph. 30, 5, Article 115 (October 2011), 26 pages.
DOI = 10.1145/2019627.2019634
http://doi.acm.org/10.1145/2019627.2019634



## 1. INTRODUCTION

It has been a longstanding goal in computer graphics to interactively synthesize realistic images, and to provide artists with powerful creative control over image generation. Despite much progress over the past thirty years current interactive graphics systems are still far from these goals.

It is becoming increasingly clear that there are significant limitations with the Z-buffer algorithm used in today's interactive graphics systems. Within the next 5 years, we believe that the Z-buffer algorithm will begin to be augmented or replaced with algorithms such as ray tracing [Whitted 1980] that efficiently support a more general class of visibility queries. This transition to ray tracing is already well under way in offline rendering [Tabellion and Lamorlette 2004; Christensen et al. 2006].

Ray tracing has made significant advances in the interactive domain already. Recently developed interactive ray tracing systems

W. Hunt, I. Elhassan, and W. R. Mark were at the University of Texas at Austin during the period that most of this work was conducted.
This research was funded by Intel Corporation and NSF CAREER award no. 0546236.
Authors' addresses: P. Djeu, Department of Computer Science, University of Texas at Austin, Austin, TX; W. Hunt, Intel Labs; R. Wang, Department of Computer Science, University of Massachusetts Amherst, Amherst, MA; I. Elhassan, Microsoft Corporation; G. Stoll, Intel Labs; W. R. Mark (corresponding author), Intel Labs; email: billmark@billmark.com.

DOI 10.1145/2019627.2019634
http://doi.acm.org/10.1145/2019627.2019634

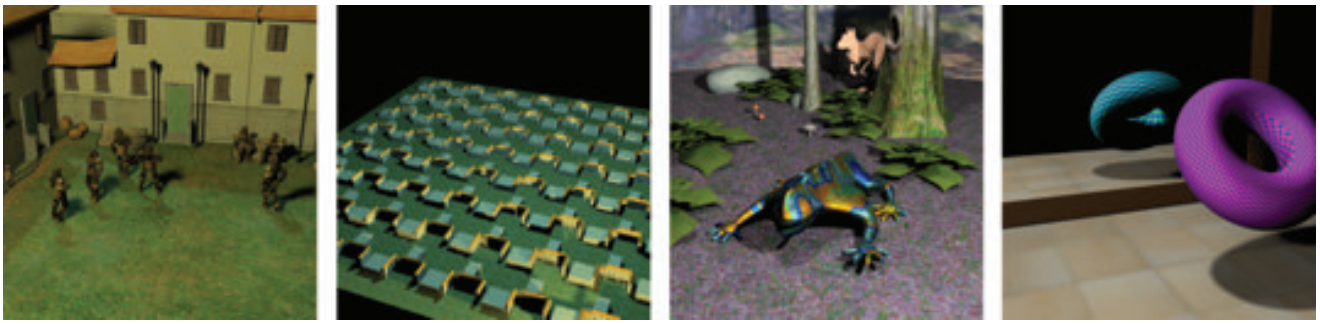

Fig. 1. Images rendered by Razor at near-interactive frame rates. From the left, the first three scenes have millions of visible micropolygons while the last scene is a technical demo. All four scenes are rendered at 1024 × 1024 resolution with a total of 36 rays per pixel (4× supersampling, 2 area lights, and 4 shadows rays/primary ray/light). The two scenes on the left use piecewise bilinear patches while the two scenes on the right consist entirely of subdivision surfaces. The render times are 3.83 sec, 3.77 sec, 6.94 sec, and 3.36 sec using eight Xeon X5355 cores. The models used these images appear courtesy of their respective owners.[1]

[Parker et al. 1999, 2010; Woop et al. 2005, 2006; Reshetov et al. 2005; Wächter and Keller 2006; Lauterbach et al. 2006; Bigler et al. 2006; Wald et al. 2007a; Bikker 2007; Caustic Graphics 2009] demonstrate that interactive ray tracing is starting to become a viable technique. Yet these existing systems typically have serious limitations that make them impractical or insufficient for most mainstream interactive applications. Many of these systems perform poorly for large dynamic scenes, and some of them only implement classical Whitted ray tracing, which for many applications does not provide a compelling improvement in visual quality over state-of-the-art Z-buffer rendering.

The true advantages of ray tracing visibility algorithms only become apparent with the addition of effects that are typically produced using distribution ray tracing [Cook et al. 1984] and/or other advanced techniques. These effects include soft shadows, glossy reflections, diffuse reflections, ambient occlusion, subsurface scattering, final gathering from photon maps, and others. But most current distribution ray tracing systems have several sources of inefficiency. They have high overhead to support dynamic scenes, use excessively detailed geometry for secondary rays, perform redundant computations for shading and secondary rays, and have irregular data access and computation patterns that are a poor match for cost-effective hardware. Successfully addressing these inefficiencies would make ray tracing a more attractive alternative to Z-buffer rendering for interactive applications.

In this article, we discuss some of the inefficiencies in conventional distribution ray tracing systems, and propose a new rendering system architecture that attempts to reduce or eliminate several of these inefficiencies. We focus our innovation on the visibility engine and on the interactions between the visiblity engine and the local shading engine. We do not attempt to innovate in the area of light transport and global illumination algorithms, where we rely on established distribution ray tracing techniques. More specifically, we focus our innovation in three areas: efficient support for arbitrary dynamic motion; efficient support for automatic multiresolution geometry; and an overall system architecture that ties everything together along with existing ideas for decoupling shading computations from visibility hit points. Our approach is forward looking, meaning that Razor is explicitly designed to support a rich set of functionality for future interactive use on future hardware. This strategy contrasts with most other recent work on interactive ray tracing, which takes the opposite approach of restricting functionality (e.g., dynamics) or image quality (e.g., resolution, visual effects, shading) so that the system can run at interactive rates on today's hardware. We present results gathered from an experimental implementation of our design on 2006-era hardware (see Figure 1). As expected, the frame rates achieved on this hardware are not interactive except at very low image quality and resolution settings. We discuss what we have learned about the strengths and weaknesses of various aspects of our design.

It is important to understand that Razor is an experimental system with a complex new design. Not surprisingly, some aspects of the design have been more successful than others, and taken as a whole we consider the system to be only partially successful in meeting its design goals. The findings section near the end of the article attempts to objectively discuss the strengths and weaknesses of the design, with the hope that future efforts can benefit from the lessons that we have learned.

This article is a revised and extended version of two earlier technical reports about Razor [Stoll et al. 2006; Djeu et al. 2007], and is intended to be the archival reference for the Razor system design. The high-level design of Razor was completed in 2005 with some adjustments made in 2006. The vast majority of the implemention and evaluation work was completed by January 2007. We have written three other articles related to Razor. Two articles discuss our techniques for rapidly building a kd-tree acceleration structure [Hunt et al. 2006, 2007], but in the context of a single resolution kd-tree rather than the multiresolution kd-tree described here. A third article, published in a computer architecture conference, analyzes Razor's performance on a simulated single-chip many-core computer architecture [Govindaraju et al. 2008].

## 1.1 Key Techniques

The most important techniques used by Razor are the following.

*Per-frame dynamic and lazy kd-tree build.* Razor incorporates an algorithm for lazily building a multiresolution kd-tree each frame

[1]The character models in Courtyard64 (first scene) are copyright Digital Extremes, all rights reserved. The character models in Courtyard64 use motion capture data from mocap.cs.cmu.edu (NSF EIA-0196217). The dragon model in Forest (third scene) appears courtesy of Jeffery A. Williams and Intel Corporation. The killeroo model in Forest appears courtesy of Phil Dench, Martin Rezard, and headus (metamorphosis) Pty Ltd. The floor in Forest appears courtesy of Jonathan Dale. The background models in Forest appear courtesy of DAZ Productions.

from a tightly integrated hierarchial scene graph representing a dynamic scene. All major system data structures except the original scene graph are rebuilt every frame. The hierarchial scene graph can be thought of as a coarse, poor-quality spatial data structure that is incrementally updated and thus benefits from frame-to-frame coherence. This coarse information is sufficient to allow the high-quality kd-tree acceleration structure to be inexpensively rebuilt on demand each frame. (This article extends the authors' previously published techniques [Hunt et al. 2006; Hunt et al. 2007] to support multiresolution geometry in a micropolygon system.)

*Ray-directed continuous multiresolution geometry.* Razor performs ray tracing on continuous, crack-free level of detail surfaces, while avoiding popping and tunneling artifacts even for camera rays. Each ray independently selects a geometric level of detail, which varies along the ray based on ray and path differentials. We refer to this capability as *ray-directed* level of detail. The continuous levels of detail are synthesized by interpolating on-the-fly between discrete levels of detail stored in the acceleration structure. These discrete levels of detail are generated on demand by adaptive subdivision of input meshes. (These ideas are the primary novel algorithmic contribution of this article.)

*Partially decoupled shading and visibility.* By splitting shading computations into two phases, Razor allows visibility to be computed at a higher spatial frequency than most shading computations. (This approach combines several previous ideas, and in particular can be thought of as an extension of the grid-based shading approach pioneered in the REYES system [Cook et al. 1987] to a 100% ray tracing framework.)

*Partially replicated kd-trees supporting parallel construction.* Razor builds multiple partially complete kd-trees in parallel, by leveraging the laziness of tree construction. This strategy allows acceleration structure build to be parallelized without overhead from communication or mutual exclusion, and allows on-demand construction to be performed at a fine granularity with reasonable efficiency. (This technique is a relatively straightforward extension of lazy build to parallelism).

*A new system architecture.* Razor combines the techniques just mentioned into a prototype system that achieves near-interactive frame rates on dynamic models whose triangle count is similar to that of modern game scenes. (This novel combination of techniques and their evaluation represents the "rendering systems" contribution of this article).

### 1.2 Organization of Article

The remainder of the article is organized as follows: First, we provide an overview of Razor's goals and design at a high level. Second, we describe Razor's design in detail, focusing on the more technically novel aspects of the design. Third, we evaluate various aspects of Razor's design using quantitative results obtained from our prototype implementation. Fourth, we discuss previous work, and how various elements of Razor are related to that work. Finally, we describe what we learned from Razor: which ideas worked well, which didn't, and which would benefit from further investigation. The article is long: It combines a new overall rendering system design with several specific technical innovations needed to make that design viable.

## 2. GOALS

The design of Razor was driven by the following goals.

—*Support distribution ray tracing effects.* These effects include soft shadows, ambient occlusion, and depth of field in addition to traditional Whitted-style effects such as specular reflection.

—*Efficiently support all forms of geometry motion.* Modern Z-buffer rendering systems efficiently support moving, deforming, and changing geometry even in large scenes with high depth complexity; it is important that a ray tracer do the same.

—*Support curved surfaces and detailed geometry.* Most future real-time rendering systems (including Z-buffer ones) are likely to be required to support high-quality geometric detail using mechanisms such as subdivision surfaces and displacement mapping.

—*Efficient real-time performance on future single-chip many-core architectures.* The system should be suitable for use by real-time applications such as games on future single-chip many-core graphics hardware. In particular, we target architectures with a flexible programming model supporting task parallelism across multiple cores and data parallelism with SIMD instructions on each core.

Just as importantly, we defined nongoals for Razor.

—*No backward compatibility.* We reconsider the entire system structure, including the format of the input scenes. We allow Razor to break compatibility with old APIs and to require that scenes be modeled and presented to the system in new ways.

—*No barrier between application and renderer.* Traditional Z-buffer APIs such as OpenGL and DirectX strictly decouple application data structures such as a scene graph from the rendering system. We allow Razor to couple these data structures to each other.

—*No requirement for real-time performance on todays's hardware.* Razor is targeted at *future* hardware with higher performance for ray tracing than today's CPUs or GPUs. In contrast, the designers of most other systems targeted at interactive rendering have imposed an artificial requirement that the systems run at interactive rates on today's hardware, even though it is clear that future hardware with a somewhat different architecture (e.g., that of Seiler et al. [2008]) could provide higher performance than today's CPUs and more flexibility than today's GPUs.

—*Not as flexible as a batch renderer.* Batch rendering systems are expected to be extremely flexible with support for a wide variety of rendering options. In contrast, it is acceptable for systems targeted for interactive use to be less flexible in order to improve performance. This pattern is shown very clearly by today's real-time Z-buffer-based systems.

—*Not a product-quality system.* Razor is an experimental system, and it does not strive for the same level of feature completeness and robustness that would be required for a product-quality system. Razor innovates on a large number of dimensions, and as such we accept that some aspects of its design may be less successful than others.

## 3. THE CHALLENGES

Given the goals just mentioned, several challenges are apparent.

*Overall system performance.* Distribution ray tracing is computationally expensive, so performance must be a key consideration in every design decision.

*Dynamic scenes.* If objects are moving within the scene, it is not possible to treat the construction of a spatial acceleration structure as a "free" preprocessing step; part or all of the work must be performed each frame. Furthermore, if the objects undergo nonrigid motion such as deformation (as is common in skinned characters used in computer games), then it is not possible to use the common optimization of prebuilding acceleration structures for individual objects.

If the scene is complex with many occlusions (such as an entire building with occupants), then building or updating the entire acceleration structure every frame is too costly, even with a fast build technique. This problem is even more acute if we want to represent each object at multiple levels of detail: the time and memory requirments for creating and storing tessellated geometry at the finest level of detail for the entire scene are prohibitive.

*Efficiency of distribution-sampled secondary rays*. Distribution ray tracing systems cast large numbers of secondary rays. For example, many rays are cast to sample area light sources and for ambient occlusion computations. There are many more secondary rays than primary rays, so the cost of tracing the secondary rays and tessellating the geometry they hit dominates the ray tracing time.

Worse yet, the traversal of the acceleration structure by distribution-sampled secondary rays is less coherent than that of primary rays [Christensen et al. 2003; Navrátil and Mark 2006; Mansson et al. 2007], which increases the per-ray cost. In particular, when tracing distribution-sampled secondary rays using a standard acceleration structure, techniques for exploiting SIMD hardware are less effective, and the ray tracer accesses a large memory working set for which caches are less effective than for primary rays.

*Redundant shading computations*. Most ray tracers perform shading computations at each ray hit point. At high screen-space supersampling rates, most of these shading computations are redundant. The situation is even worse for shaders that require arbitrary differential computations, since these shaders must either be run three times at each hit point to compute discrete differentials [Gritz and Hahn 1996], or perform the extra computations needed for automatic differentiation using dual number arithmetic [Piponi 2004; Karrenberg et al. 2010; Gritz et al. 2010]. Redundant shading computations can severely degrade overall system performance, since it is common for a renderer's surface shading costs to be a substantial part of the total rendering cost.

## 4. DESIGN DECISIONS

In this section, we describe the key decisions made in the Razor design. First, we describe several elements of the design that are relatively conventional. Second, we describe design choices that are more unusual and represent the most important differences between Razor and other rendering systems. Third, we describe additional design choices that help to support the novel design choices.

### 4.1 Conventional Decisions

Several of the design decisions in Razor are fairly conventional, but need to be explicitly mentioned to understand the remainder of the system.

The first few of these decisions are so standard that we do not attempt to explcitly justify them; it is perhaps more proper to consider them as additional goals or constraints for the system design. We choose to represent the scene using an explicit boundary representation (rather than, e.g., a volumetric representation); we choose to use discrete rays as the only visibility primitives (rather than, e.g., beams [Heckbert and Hanrahan 1984]); and we choose to use a variant of distribution ray tracing [Cook et al. 1984] to sample various visibility integrals.

In addition, Razor's goal of achieving high efficiency led to several relatively conventional design choices based on experience with earlier high-performance rendering systems. These bottom-up performance-driven choices have the effect of constraining and/or inspiring other aspects of the system design.

—*For actual ray/surface intersection tests, use triangles*. Experience with Z-buffer systems, interactive ray tracing systems, and many offline ray tracing systems (e.g., that of Christensen et al. [2003]) has shown that it is wise to use a single simple geometric primitive (specifically a triangle or quad) for final intersection testing, even if this choice necessitates an earlier tessellation step (as is the case in Razor). Razor uses triangles, and this choice allows the system's inner loops and corresponding data structures to be highly specialized and performance tuned for ray/triangle intersection testing and acceleration structure construction. As we will discuss in Section 5.1.3, these triangles are generated by adaptively tessellating subdivision surfaces.

—*Use a cost-optimized acceleration structure (specifically a kd-tree)*. Previous experience shows that achiving good traversal performance for rays requires the use of a highly efficient acceleration structure. In particular, the partitions in the acceleration structure must be chosen so as to minimize expected traversal cost as predicted by the surface area heuristic [Goldsmith and Salmon 1987]. Razor uses a cost-optimized kd-tree [Havran and Bittner 2002] as its primary acceleration structure, along with modifications described in Section 5.2 to support multiresolution capabilities. For scenes that are animated, this decision raises the question of whether it is possible to efficiently construct or update a cost-optimized kd-tree, and we will address this question in Section 5.2.2. We note that prior to this work, most interactive ray tracing systems supporting arbitrary dynamic scenes avoided the use of cost-optimized acceleration structures, because it was believed that they were too expensive to construct.

—*Use ray packets for efficiency on SIMD hardware*. One of the best ways for hardware to provide high computational throughput at minimal cost is use register SIMD instructions, such as SSE. Modern GPUs use 16-wide, 32-wide, or 48-wide SIMD hardware, although in some cases the SIMD hardware is hidden behind a nominally scalar ISA [Lindholm et al. 2008]. SIMD hardware amortizes the overheads of instruction fetch, decode, and branching; and improves the coherence of accesses to cache and to DRAM. Previous work has shown that ray tracers can make efficient use of SIMD hardware by aggregating rays into packets [Wald et al. 2001]. Packets are extremely efficient when rays are spatially coherent (e.g., camera rays or hard shadow rays), but become less efficient as rays lose their coherency. One of the major goals of Razor's design was to maximize the efficiency of packet techniques for semicoherent rays such as soft shadow rays. (Concurrent work by others has also shown that that packets are useful for distribution ray tracing [Boulos et al. 2007]). In its default configuration, Razor uses 32-wide packets, implemented as eight 4-wide SSE operations. Even on 4-wide hardware, the 32-wide packets help performance by amortizing branch overhead, but this design will be even more efficient on future, wider SIMD hardware.

### 4.2 Unusual Decisions

There are three key design decisions for Razor that taken together make it substantially different from other ray tracing systems, and in particular make it different from other systems targeted at interactive use. We will describe these decisions, and explain why they were made.

—*Use multiresolution surfaces and a robust multiresolution acceleration structure*. In the past few years, batch rendering systems [Christensen et al. 2003; Tabellion and Lamorlette 2004] have demonstrated that most secondary rays can be traced using coarse

geometric representations of the scene, without harming image quality. Mathematically this situation is explained by the fact that most secondary rays have large ray differentials [Igehy 1999], that is, they diverge strongly from each other as they progress away from their origins. Furthermore, errors in secondary visibility are typically much less perceptually objectionable than errors in primary visibility.

The advantage of using a coarse geometric representation for secondary rays is that it can improve the coherence of secondary ray traversal, which in turn improves overall system performance. In particular, a coarse geometry representation reduces the memory working set required by secondary rays so that cache hit rates are higher, and improves the effectiveness of ray packet techniques needed to achieve high efficiency on SIMD hardware.

For this reason, Razor uses a multiresolution representation of geometry that allows secondary rays to access an appropriate (typically coarse) geometric representation, as determined by ray differentials. The geometry resolution used by secondary rays is not constrained by the resolution used by primary rays as it is in some other systems [Christensen et al. 2006]. Most importantly, there is no longer a single reference point (the camera point) with which to set the resolution of each surface in the scene. Instead, each ray (including secondary rays) may request a geometric resolution that is essentially unrelated to that requested by any other ray. We refer to this capability as *ray-directed* level of detail, in contrast to the more traditional *camera-directed* level of detail. Because of the initial design decision that Razor would use ray tracing for all rays (i.e., primary rays as well as all types of secondary rays), the multiresolution representation and associated intersection testing must be robust enough to avoid objectionable artifacts for primary rays, while still allowing secondary rays to access a different resolution from primary rays without cracking artifacts. This problem had not been solved previously in a multiresolution ray tracer, and led us to develop a novel multiresolution acceleration structure, along with associated traversal and intersection algorithms. This acceleration structure conceptually represents each region of space at all possible resolutions. In the actual algorithm, each region of space is represented at one or more discrete resolutions. The associated traversal and intersection algorithms interpolate on-the-fly between the discrete resolutions, so that rays are intersected with geometry represented at a continuously varying level of detail that is not subject to popping or cracking artifacts.

Details of this design and the considerations that led to it are discussed in Section 5.1.

—*Support dynamic scenes and multiresolution geometry by building the acceleration structure on demand each frame, using the application scene graph as input*. Razor rebuilds the acceleration structure every frame. More precisely, Razor builds only those spatial regions and resolution(s) of the acceleration structure that are needed for the current frame. In contrast, many interactive ray tracing systems choose to preserve the acceleration structure from frame to frame, making updates to node bounds to account for motion.

The decision to rebuild the acceleration structure each frame was made for three reasons: First, this decision supports the goal of allowing arbitrary dynamic motion, including deformation, unstructured motion, and displacement mapping. Second, this decision supports the previous decision of using multiresolution geometry, because the acceleration structure can include exactly those resolution(s) of geometry that are needed for the current frame, while omitting higher resolutions of geometry that would dramatically increase the size of the acceleration structure. Third, this decision supports the goal of efficiency, since acceleration structures that are rebuilt often perform better than those that are incrementally updated (especially if topology cannot be updated). Razor uses as its acceleration structure a kd-tree optimized according to the surface area heuristic. Razor uses several novel techniques that permit rapid construction of this high-quality acceleration structure.

Razor builds its acceleration structure on demand throughout the rendering of the frame, rather than at the start of the frame. This approach avoids wasted work for portions of the scene that are not visible, and ensures that information is available about which resolution(s) of geometry are needed. When a ray enters a previously unbuilt node of the acceleration structure, that node is built at the needed resolution, and traversal is resumed. An application-maintained scene graph provides the coarse scene organization required to identify scene geometry to be inserted into the acceleration structure. This scene graph also provides a coarse spatial sort that reduces the cost of building a high-quality acceleration structure, as compared to approaches that build from a "triangle soup." This design requires a close coupling between the rendering engine and the application scene graph.

The decision to rebuild the acceleration structure on demand every frame from potentially changing geometry places constraints on other aspects of the design. For example, the multiresolution representation cannot require extensive preprocessing of geometry or rely on information about global topology.

—*Decouple shading from visibility to reduce redundant shading computations*. Most ray tracing rendering systems perform shading computations at the ray hit points. However, this approach is inefficient when there is an opportunity to perform shading computations at a lower sampling rate than visibility computations, as is the case when casting many primary rays per pixel. The REYES system [Cook et al. 1987] and the multisampling in modern Z-buffer graphics systems [Akenine-Möller and Haines 2002] avoid this problem by decoupling shading from visibility. However, these previous systems are used only for primary visibility (camera rays), where it can be assumed that all rays originate at a single point (the camera point). This assumption does not hold in a ray tracer.

Razor extends the idea of decoupling shading from visibility to work for all types of rays in a ray tracing system. Razor splits shading into two phases [Pharr and Humpreys 2004], one which is view independent (typically, the computation of the material properties), and another which is view dependent. The view-independent computation is decoupled from visibility by performing it on a regularly spaced grid of points in object space and interpolating the results at visibility hit points. This computation is performed on demand, after a grid has been hit by a ray. More specifically, as inspired by REYES, the system tessellates geometry into micropolygons and performs the view-independent shading computations at the polygon vertices. The view-dependent computations are always performed at hit points, as in a conventional ray tracer. This idea is conceptually simple, but making it work correctly without artifacts in a multiresolution ray tracing system introduces complications, as will be discussed in Section 5.3.

This approach has an additional important advantage that it allows shading computations to be performed in large, efficient batches. In particular, Razor's shading computations use SIMD hardware very efficiently, even if rays are incoherent. This

approach also supports displacement mapping, which can be computed in the first phase of shading.

However, the decision to decouple shading from visibility by evaluating shading in object space interacts in difficult ways with the decision to use multiresolution surfaces. We now have a situation where shading may need to be performed at multiple resolutions for any particular surface. Doing this is straightforward when visibility is coupled to shading, but less so once we decouple them. This challenge is addressed by the detailed system design presented in Section 5.

### 4.3 Additional Key Decisions

There are two other important aspects of Razor's design.

—*Parallelize at a fine granularity and partially replicate acceleration structure*. In an interactive ray tracer, it is not possible to parallelize across frames as it is with batch renderers. Instead Razor parallelizes at a finer granularity, assigning different screen regions to different cores, and using SIMD within each core to parallelize across rays and across shading samples. To eliminate synchronization overhead for building the acceleration structure, each core builds its own instance of the on-demand acceleration structure. We made this choice because it performed better than a shared acceleration structure on the two-socket, 4-core-per-socket platform with large caches that we used to run experiments. In particular, there is surprisingly little replication overhead, due to the fact that each instance of the acceleration structure is built on demand, and the fact that the detailed geometry used by primary rays is mostly not replicated. However, this decision to replicate the acceleration structure is not fundamental to the Razor design; and in follow-on work using a simulated architecture with less per-core cache we have configured Razor to share an acceleration structure across local groups of cores [Govindaraju et al. 2008].

—*Multiresolution geometry is generated by tessellating Catmull-Clark subdivision surfaces on demand*. Razor generates triangle-based geometry at the desired resolution by adaptively tessellating Catmull-Clark subdivision surfaces, then splitting each quad into two triangles. We made this choice based on best-known practices at the time for use of subdivision surfaces. If we were to redesign the system now, we would consider using more recent innovations in subdivision surfaces [Loop and Schaefer 2008]. Since subdivision is performed on demand, it would be possible to confine the triangles generated by tessellation to on-chip storage and use, and Razor was designed to support this technique. The basic idea is to discard triangles when they are evicted from the cache rather than write them to off-chip DRAM. If a triangle turns out to be needed later, it can be regenerated, as inspired by Pharr and Hanrahan [1996]. However, we have not yet implemented the cache management for this technique, and so currently triangles are written to off-chip DRAM when they are evicted from cache, generating spurious off-chip DRAM write traffic.

## 5. SYSTEM ARCHITECTURE

So far, we have discussed the goals of Razor, the high-level design decisions that were made, and the key technical challenges presented by these high-level decisions. It is already clear that the individual design decisions interact with each other in complex ways, as is typically the case for any sophisticated system.

In this section we present the system architecture of Razor, which combines and refines the high-level technical strategies already presented so that they are compatible with each other and form a single integrated system. In particular, the system architecture addresses the key technical challenges presented in the previous section. While some elements of our system adapt well-known approaches, other elements of the system are individually novel and require more detailed explanation. Fortunately, the major components are familiar from any standard ray tracer: the ray/surface intersection technique, the acceleration structure, and the shading system. We will describe these components one at a time.

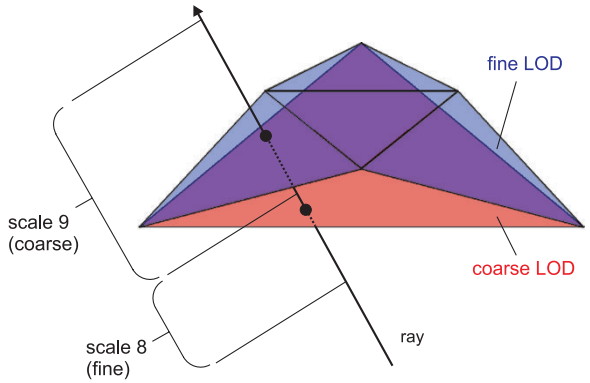


Fig. 2. With discrete LODs, a ray may miss a surface completely if it changes the LOD that it is requesting at a point along the ray that is in between the surfaces produced by two discrete LODs.

### 5.1 Multiresolution Ray/Surface Intersection

As discussed earlier, the ray/surface intersection problem is especially challenging in a system like Razor that uses ray-directed LOD rather than camera-directed LOD, since a single surface region may be represented at multiple resolutions within a single frame.

In order to reuse tessellations and associated vertex data and shading computations within a frame (as is needed when decoupling shading from visibility) these computations must be generated and cached for *discrete* levels of detail. Unfortunately, in a ray tracer, naive discrete LOD approaches suffer from what we call the *tunneling* problem (the name is inspired by the quantum mechanical concept of tunneling). Figure 2 illustrates the simplest version of the problem. In the figure, the level of detail used for intersection between the ray and the surface changes abruptly at a point along the ray. This is a direct result of discretization and of the fact that the required LOD is a function of location along the ray. Unfortunately, the ray switches from wanting the fine version to wanting the coarse version at a point along the ray which is after the intersection with the coarse version but before the intersection with the fine version. This causes the ray to pass through both versions of the surface without any intersection being detected. This problem is closely related to *LOD popping* (the abrupt transition of an object from one LOD to another over the course of an animation), and differs from other typical ray tracing artifacts. Unlike *patch cracking*, where a gap forms between neighboring patches which have been adaptively refined to different LODs along their shared border, holes produced by tunneling can appear even in the middle of individual triangles. Unlike numerical precision problems, tunneling holes can have significant geometric extent. A key challenge in ray tracing multiresolution surfaces is to design a technique that avoids tunneling and patch cracking while satisfying other system constraints.

5.1.1 *Previous Work on Raytracing LOD*. Some production ray tracing systems (e.g., Tabellion and Lamorlette [2004]) rely on manual per-shot tuning of LOD parameters to eliminate possible cracking artifacts; but interactive systems require an automatic technique.

Pixar's production rendering system that hybridizes ray tracing with REYES avoids tunneling and cracking by introducing some additional constraints on tessellation based on the camera point [Christensen et al. 2003, 2006]. Their system avoids tunneling for ray traced secondary rays by choosing a discrete LOD for an entire subpatch when the ray first enters the bounding box of that subpatch. The bounding box is chosen such that all possible LODs for the subpatch are enclosed within it, thereby insuring that tunneling cannot occur. However, for a large base patch such as ground terrain, this technique then raises the question of how to choose the appropriate sized subpatch at which to apply this technique. If the technique is applied to the base patch, then each ray uses a single LOD for the entire base patch, resulting in potentially severe overtessellation and/or undertessellation in some portions of the patch. If the technique is applied to some appropriately sized subpatch, then some mechanism is needed to insure that all portions of a ray choose the same granularity of subpatch; otherwise tunneling could still occur. Also, to make crack avoidance between subpatches tractable, it is useful to insure that *all* rays intersecting a particular region of the patch apply the technique to the same granularity of subpatch. To address these issues, the Pixar system chooses the subpatch based on the LOD used by the primary rays for that patch. More specifically, the subpatch is the final subpatch (a.k.a. grid) produced by the splitting operations of the REYES system. This choice avoids tunneling and also helps to insure consistency between primary "rays" and secondary rays, but has the unfortunate side effect of preventing secondary rays from accessing a LOD coarser than a $2 \times 2$ tessellation of the subpatch, even if the original base patch would have been able to support a coarser LOD for the secondary rays. Thus, the LOD choices of the secondary rays are artifically constrained by the LOD choices of the primary rays even though secondary rays often request LODs which are far coarser than those of primary rays.

With Pixar's approach, crack avoidance between patches becomes complicated. The reason is that two secondary rays intersecting adjacent subpatches (grids) can choose different discrete LODs, leaving the potential for cracks between the subpatches unless extra adjustments are made. Christensen et al. [2006] state that the Pixar system makes these adjustments by moving vertices at subpatch boundaries and/or inserting gap-filling polygons, but provide insufficient detail to understand exactly how the technique works for secondary rays, where there is no canonical LOD for a subpatch as there is for primary rays. Our best guess is that a ray which is close to a boundary between subpatches computes the LOD for both subpatches, and is intersected against gap-filling polygons which are generated for that particular pair of LODs. It is important to realize that these gap-filling polygons can be different for different secondary rays, in constrast to the situation for primary rays, where there is just a single set of gap-filling polygons. This class of techniques for crack avoidance requires that the system be able to find all patches (or subpatches) that neighbor a particular subpatch. This need for neighbor information adds complexity to the system's data structures and makes it more difficult to parallelize computations efficiently at the granularity of subpatches or rays, particularly when tessellation is performed lazily.

Yoon et al. [2006] have developed a multiresolution ray tracing system concurrently with ours. Their system targets massive static models, focusing primarily on memory footprint reduction. It does not make any guarantees about maintaining surface continuity, so cracks/holes are possible.

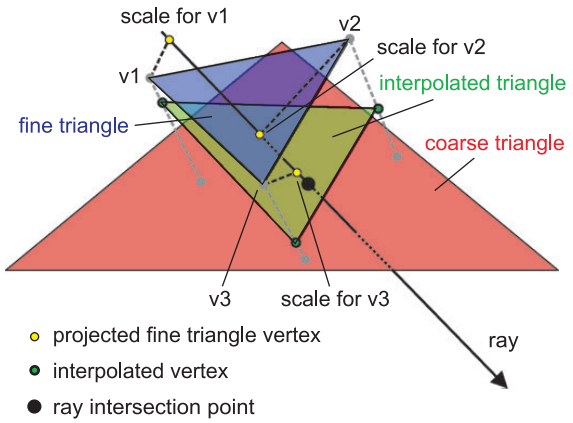


Fig. 3. For each ray/triangle intersection test, the system generates a custom triangle that is specific to that ray. This custom triangle (green) is generated by interpolating between triangles from two discrete levels of detail (blue and red). The interpolation weight for each vertex is determined by projecting the fine-triangle vertex (e.g., v1) onto the ray, computing the scale value at that point on the ray (yellow dot), and determining the fractional distance of this scale value from the two discrete levels.

In work subsequent to ours, Hanika et al. present a mulitiresolution ray tracing system based on patch tessellation where LOD is set on a per-patch basis [Hanika et al. 2010]. Their system avoids cracks by in effect giving the surface a finite thickness at coarser levels of detail. At a patch boundary where a finer level abuts a coarser level, the finer level is guaranteed to abut the thicker edge of the coarser level. More specifically this is done by intersecting rays with small slabs (which are actually the boundaries of a BVH node), whose extent is set such that it is guaranteed to include any finer-level tessellation of the surface, including any possible displacements. This LOD scheme shares the same advantages as our scheme: it avoids cracks while being easy to parallelize.

5.1.2 *Razor's LOD Technique*. For Razor, we developed a different approach to LOD for ray tracing than that used by Christensen et al. that avoids cracking and tunneling, yet does not not restrict the LOD used by secondary rays and facilitates fine-grained parallelization by avoiding the need to exchange information between neighboring patches or subpatches. This approach is used for all rays, including primary rays. The new technique is a hybrid between discrete and continuous approaches to LOD. The system generates discrete levels of detail but continuously interpolates (i.e., geomorphs) between the levels. There are a number of discrete levels of detail (up to fourteen in the prototype), each specified by a world-space edge length threshold. Each level is a distinct version of the *entire scene* that is targeted at the given edge length threshold. The subset of the geometry in each level that is actually needed is built on demand while the remainder is never constructed. The system interpolates between adjacent discrete levels on-the-fly to produce a unique surface for intersection testing against each ray. The continuous interpolation avoids tunneling as well as other LOD popping artifacts. Geomorphing has been used previously with camera-directed LOD to render terrain [Hoppe 1998], but we extend its use to ray-directed LOD.

Figure 3 illustrates our technique for interpolating between discrete LODs. We refer to the adjacent discrete LODs as the *fine* mesh

and the *coarse* mesh. This pair of adjacent discrete LODs is chosen from the larger set of discrete LODs so as to "bracket" the desired continuous LOD, using a mechanism to be described later in Section 5.2.6. The meshes in our system are generated via subdivision. The first step is to prepare a 1:1 correspondence between triangles in the fine mesh and synthesized triangles lying on the coarse mesh. Each triangle in the fine mesh maps to a portion of a single triangle of the coarse mesh. This property relies on the fact that we use binary subdivision. The system pairs each vertex of the finer triangle with a point on the corresponding triangle in the coarse mesh, thus defining the synthesized triangle lying on the coarse mesh that corresponds to the fine-mesh triangle.

The next step is to compute an interpolated triangle that represents an interpolation between the fine-mesh triangle and the corresponding synthesized triangle lying on the coarse mesh. The system produces this interpolated triangle by interpolating between vertex positions in the fine mesh (three $\mathbf{v}_f$ shown as v1, v2, and v3 in Figure 3), and the corresponding points on the coarse surface (three $\mathbf{v}_c$, which are not necessarily vertices). This interpolation is performed independently for each vertex in the fine mesh, with a separate interpolation weight used for each of the three vertices in a triangle. The interpolation weight for each vertex in the fine mesh is found by projecting the fine-mesh vertex onto the ray, yielding a distance along the ray $t$. With the ray defined by origin $\mathbf{P}$ and direction $\mathbf{D}$:

$$t = (\mathbf{v}_f - \mathbf{P}) \cdot \hat{\mathbf{D}}. \tag{1}$$

Then, the blend weight $b$ between the fine and coarse "vertices" is computed from a continuous scale function defined along the ray

$$t_s = \text{start of this LOD transition, see Section 5.2.6,} \tag{2}$$

$$t_e = \text{end of this LOD transition, see Section 5.2.6,} \tag{3}$$

$$t' = \frac{clamp(t, t_s, t_e) - t_s}{t_e - t_s}, \tag{4}$$

$$b = 3(t')^2 - 2(t')^3, \tag{5}$$

where the last two equations are implementing a "smoothstep" function. The blend weight $b$ is then used to interpolate between the fine and coarse "vertices" to produce a single interpolated vertex $\mathbf{v}$.

$$\mathbf{v} = b\mathbf{v}_f + (1 - b)\mathbf{v}_c \tag{6}$$

Applying this process to all three vertices of the fine triangle produces a single, interpolated triangle that has been custom-built for the incoming ray. Since this triangle is built specifically for this particular ray, it is discarded after the intersection test is complete (see Figure 3).

This projection and interpolation step reduces the overall ray/triangle intersection problem to that of standard (nonmultiresolution) ray/triangle intersection. The overall algorithm is quite efficient. The interpolation weights in this scheme are associated with vertices, not triangles, so if both the fine and the coarse meshes are crack-free, the interpolated mesh is as well. This property relates to a important point. The fact that each discrete level is a consistent view of the entire scene makes it relatively easy to ensure that level's properties (e.g., that it is crack-free). Any such property that is in turn preserved by vertex-by-vertex interpolation is therefore preserved in the interpolated surface that is "seen" by a ray. There is some commonality between this approach and camera-directed LOD techniques for terrain [Luebke et al. 2003; Hoppe 1998].

Note that the crack-free guarantee applies to a single ray, and that we currently make no guarantees about the relation between what geometry will be "seen" by one ray versus another. We also cannot guarantee that a surface will not "misbehave" under interpolation (e.g., folding on itself, etc.), although it will still be crack-free. As mentioned before, an additional limitation of this approach in its current form is that it requires that each discrete LOD be constructed by power-of-two tessellation of the base patch (a.k.a. binary dicing), in order to allow the interpolation between coarse and fine triangles. As Christensen et al. [2006] point out, this requirement can result in tessellations that are finer than strictly needed. For systems that shade at micropolygon vertices as ours does, unnecessarily fine tesselations result in an increase in shading costs as compared to a tessellation at exactly the desired resolution. Table I summarizes the differences between Razor's LOD method and that of Christensen et al. [2006].

Table I. Comparison between LOD Methods Used by Razor and PRMan

| | PRMan | Razor |
|---|---|---|
| LODs available to secondary rays constrained by primary rays | yes | no |
| Information from neighboring patches required to avoid cracks | yes | no |
| Dicing restricted to powers of two | no | yes |
| Requires multi-scale acceleration structure and ray traversal | no | yes |

The technique we have just described allows us to intersect a ray with a blend of geometry from two adjacent discrete levels of detail. The blend weights are computed from a continuous scale function along the ray. This scale function will be described in Section 5.2.6. Note that the system must insure that the appropriate pair of discrete levels of detail is used; this requirement adds complexity to our system and will be discussed along with the scale function.

5.1.3 *Tessellation of Curved Surfaces.* The design of Razor assumes that a surface can be represented at multiple geometric resolutions, with each resolution composed of triangles. These multiple representations of the surface are provided to the multiresolution ray/surface intersector. Razor generates its multiresolution geometry by adaptively tessellating curved surfaces to produce triangles. In its most common configuration, Razor uses Catmull-Clark subdivision patches [Catmull and Clark 1978] as the original representation of the curved surface. Input scenes are provided as geometry meshes. Some of our input scenes were explicitly designed to be rendered as Catmull-Clark patches, while others were originally designed to be rendered as polygonal meshes so that Razor must consider the original mesh vertices to be control points for Catmull-Clark patches.

Our implementation of Catmull-Clark subdivision has support for textures, geometric creases, and texture creases [Halstead et al. 1993; Biermann et al. 2000]. Following the design described by Derose et al. [1998], the Catmull-Clark patches use a generic topology representation (i.e., vertices, edges, faces) in regions of the mesh that contain extraordinary points or crease edges and a uniform bicubic B-spline representation in regions of the mesh that are completely regular [Lane et al. 1980; Peterson 1994]. The latter representation is preferable because it can be split or tessellated anisotropically; that is, it can be split into two subpatches along just one parametric direction, or tessellated at different rates in each of its two parametric directions. Support for anisotropic tesselation is particularly useful since it mitigates the potential for high overtesselation from long, skinny original patches.

In most rendering systems, the purpose of adaptive tessellation is to ensure that curved surfaces appear smooth, especially at silhouette edges. However, in Razor and REYES, where shading occurs

at vertices, adaptive tessellation has the even more important purpose of insuring that the shading rate is sufficient. Unfortunately, Catmull-Clark subdivision is complex and cannot be performed anisotropically near extraordinary points, sometimes leading to significant computational cost. To control for these effects, we implemented a second tessellation subsystem in Razor which has the sole goal of generating vertices at a sufficient density for shading without the complexities of Catmull-Clark subdivision. This second tessellation subsystem treats each mesh element as a bilinear patch (i.e., as close to planar as one can get with four vertices). These bilinear patches can be refined by either splitting them in half along one of their parametric directions or by converting them into a tesselated grid where each parametric direction has its own tesselation rate. The bilinear system removes much of the overhead of Catmull-Clark subdivision: tessellation is inexpensive and fully adaptive in both parametric directions; and it is simple to compute a very tight bounding box for a patch. Evaluating Razor in its bilinear tessellation mode allows us to measure the cost of shading at vertices without the constraints imposed by Catmull-Clark subdivision.

Since both tessellation techniques perform their tesselation adaptively, it is necessary to prevent or fix cracks between adjacent patches. With bilinear tessellation, patch borders are always linear, so no explicit crack fixing is needed even when the tesselation rates of neighboring patches do not match. With Catmull-Clark patches, there is the potential for cracking, so an explicit effort must be made to avoid cracks. Like older versions of PhotoRealistic RenderMan [Apodaca and Gritz 2000], Razor restricts the size of grids in the Catmull-Clark system to powers of two (binary dicing) to simplify the process of finding corresponding vertices along the border. When two patches abut and one tessellates at $2\times$ the rate of the other, cracks are avoided by moving each extra edge vertex to lie on the straight line between its neighbors on the edge [Clark 1979]. It is important to note that this technique results in T-junctions which are generally considered to be a potential source of pinhole artifacts, although we have not observed any such artifacts in our limited tests. The decision process for this technique is performed at each subdivision step, and is entirely local, using only information from the patch under consideration, and specifically from the relevant edge of that patch. No information from the neighboring patch is required, facilitating parallelism and on-demand tessellation. The procedure is similar to that described by Owens et al. [2002]. Moreton discusses the watertight tessellation problem in detail and shows that it is possible to guarantee perfectly watertight tessellation [Moreton 2001].

Different approaches to ray tracing of subdivision surfaces have been used in other systems. In Razor, a patch (or subpatch, if the patch has been split) is completely tessellated and stored once the ray penetrates its bounding volume. The tessellation criteria is adaptive, and cracks are avoided by adjusting the vertex positions of the tessellated polygons. The ShaoLin system [Müller et al. 2003] uses a more complex adaptive scheme, with tighter bounds tests based on projection of the ray onto a test plane. The ShaoLin system relies on additional stitching/gap polygons to prevent cracks, and does not store/cache any results. Benthin et al's interactive system tessellates on-the-fly (no storage/caching) and avoids the complications of adaptive subdivision by always subdividing patches a constant number of times [Benthin et al. 2004].

5.1.4 *Implications of Multiresolution Ray Tracing.* By using a per-ray multiresolution framework, Razor effectively uses multiresolution geometry that (1) transitions continuously between discrete scales and (2) allows rays to independently choose their desired level of detail, without restrictions imposed by the position of the camera. This combination of capabilities is, to our knowledge, not present in any previous ray tracer.

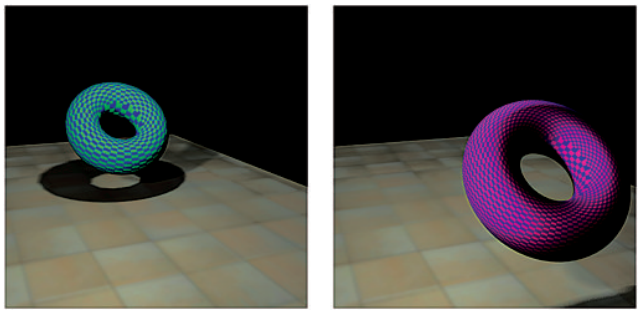

Fig. 4. A torus modeled as a Catmull-Clark subdivision surface. As the viewpoint changes, the number of subdivision refinement iterations increases or decreases, as determined by the ray differentials. (Left) The far viewpoint requires fewer iterations. (Right) The near viewpoint requires more iterations.

For example, Figure 4 shows two views of a torus represented as a Catmull-Clark subdivision surface. As the torus moves towards the camera, it is refined until it is sufficiently well tessellated for the current viewpoint. The number of refinement iterations is governed by the ray differentials of the primary rays. As the torus moves away from the camera, it is refined for fewer iterations. In both cases, subdivision refinements are performed adaptively and the surface remains crack-free, even while each primary ray is processed in isolation, without knowledge of its neighbors.

We encourage the reader to review the last segment of the video that accompanies this article (accessible at the ACM Digital Library) for an example of the multiresolution properties of Razor over the course of an animation. We show that the surface tessellation rate changes continuously from viewpoint to viewpoint, resulting in an artifact-free use of multiresolution. Our video also demonstrates that objects are similarly well-behaved when they are viewed indirectly, for example, via reflection rays. (Please note that some parts of the video are generated in batch rendering mode; see Figure 8 for performance information).

## 5.2 Dynamic Multiresolution Acceleration Structure

As mentioned earlier, Razor builds its acceleration structure on demand, from geometry stored in a scene graph. The scene graph is mostly persistent from frame to frame, while the multiscale kd-tree acceleration structure is rebuilt every frame. We will now discuss the scene graph in more detail. The upper levels of the scene graph are persistent, and are comprised of internal nodes which provide hierarchical structure to the scene graph, and leaf nodes which contain geometry in the form of one surface patch per leaf node. This persistent portion of the scene graph may be updated in response to animation, simulation, or user input just as with any traditional scene graph system. The lower levels of the scene graph are added below the persistent leaf nodes, and contain split and/or tesselated patches that are constructed during the course of rendering a frame. The construction of these transient nodes is performed on demand, and is triggered by rays that require tessellated geometry at a particular level of detail. These lower parts of the scene graph and all acceleration data structures in the system are rebuilt from scratch every frame. Hierarchical bounding volumes are maintained throughout this extended scene graph. The relationship between the scene graph and the multiresolution acceleration structure is shown in Figure 5.

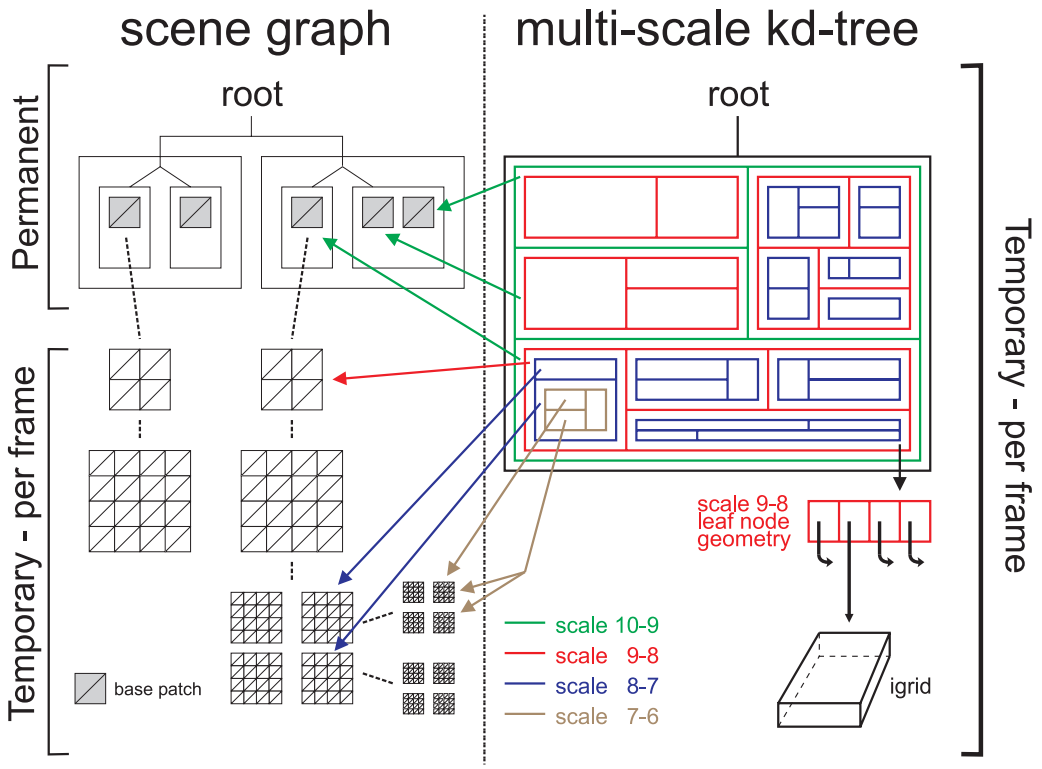


Fig. 5. The scene graph contains a collection of base patches (shaded) which are refined on demand by rays into tessellated patches (bottom left). All components of the scene graph are stored within a bounding hierarchy with permanent upper levels and transient (per-frame) lower levels. Each kd-node represents two neighboring scales and points to items in the scene graph that are appropriate for that range. In general, parent-child nodes in the kd-tree point to nodes that are similarly related in the scene graph (e.g., blue and brown pointers). Base patches and tessellated patches are ultimately converted into an igrid (lower right) when stored in intersectable form in a kd-node.

No restrictions are placed on the quality (in the surface-area-heuristic sense) of the scene-graph hierarchy; it is used only as an accelerator to the kd-tree build process. However, a well-behaved scene graph reduces construction time for the acceleration structure, as discussed in detail in Hunt et al. [2007].

In order to support the interpolating intersection technique described earlier, we would conceptually like to build a separate acceleration structure for every pair of adjacent discrete levels. The geometric primitive at the leaf nodes in each such structure would be a triangle pair consisting of a finer-level triangle paired with the corresponding portion of a coarser-level triangle. We implement this concept with a kd-tree with three key features: (1) the kd-trees for all of the level pairs are merged into a single data structure, (2) this merged data structure is built lazily from the scene graph using fast scanning techniques [Hunt et al. 2006], and (3) the merged data structure stores grids (small regular meshes) of vertices at its leaf nodes rather than storing individual triangle pairs.

5.2.1 *Merged kd-Trees.* Figure 6 illustrates our kd-tree[2]. The multiresolution capability is provided within a single kd-tree by allowing each node to fill a dual role: when traversed at a particular scale the node acts as a leaf node containing geometry at that scale, but when traversed at a finer scale the node acts as an interior node with a split plane and child nodes. This multiscale kd-tree is similar to that described by Wiley et al. [1997] for a multiresolution BSP tree, although our system uses a hierarchical nesting of LODs whereas theirs used n-ary LOD selection nodes. Also, our approach does not restrict the location of cut planes with respect to the geometry as theirs did.

The multiscale kd-tree acceleration structure can be thought of as numerous separate kd-trees, each built for a different discrete scale pair, layered on top of each other. The leaves of a kd-tree built for a single pair become a frontier of internal nodes in the combined tree.

Our kd-tree data structure is specifically designed to utilize known best practices for high-performance kd-tree traversal [Wald et al. 2001; Reshetov et al. 2005], including SIMD packet traversal code and an eight-byte internal node record. During traversal, ray segments descend through the merged tree treating all nodes as internal (split) nodes until they reach either an empty leaf or a node which is a leaf for the segment's scale. Later, we will describe how a ray is split into ray segments so that each ray segment has a single discrete scale. In short, the addition of multiresolution capabilities does not inhibit the use of current best practice approaches for fast traversal with ray packets.

5.2.2 *Fast Construction.* We use several complementary algorithmic approaches to build our kd-tree structures quickly: lazy evaluation of scene graph nodes, use of the scene graph hierarchy for build acceleration [Hunt et al. 2007], and the fast scan

[2]This data structure is perhaps more accurately an axis-aligned BSP tree, but we use the common ray tracing parlance of kd-tree here.

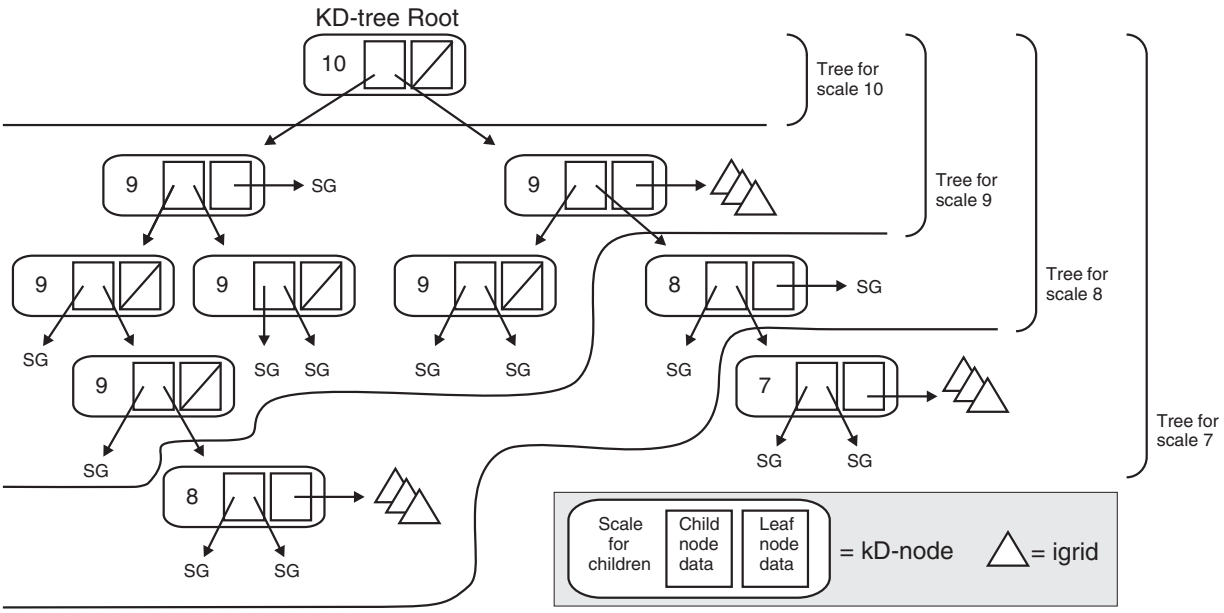


Fig. 6. Multiscale dynamic kd-tree. "SG" designates a pointer into the scene graph. The presence of a pointer to the scene graph rather than a pointer to another kd-tree node indicates that this portion of the kd-tree has not yet been built, but that the information for doing so is available at the designated node in the scene graph.

approximation [Hunt et al. 2006] of the surface area heuristic [Havran and Bittner 2002]. A fourth technique we use is to build down to small regular mesh regions (grids) instead of triangles. These regular mesh regions use their own, more regular, BVH acceleration structure, which has a known topology and can thus be built extremely quickly.

The idea of lazily tessellating and storing geometry has been used for a long time. Arvo and Kirk lazily build a 5D acceleration structure for a ray tracer [Arvo and Kirk 1987]. The RenderMan interface [Pixar 2005] supports a callback to user code for on-demand generation of geometry within a bounding box at the needed resolution, and there are now several ray tracing implementations of the RenderMan interface (e.g., Gritz and Hahn [1996] and Christensen et al. [2006]). Pharr and Hanrahan [1996] build displacement maps on demand in a ray tracer.

In addition to being desirable for efficiency in large or highly occluded scenes, laziness is required in order to support multiresolution geometry. Building out the entire data structure across the entire range of interesting levels of detail would be prohibitively expensive. Thus, our system builds its kd-trees lazily.

A node encountered in our tree during traversal may have been previously marked as "lazy." Such a node has no children or geometry. Instead, it has a pointer to a linked list of as-yet unprocessed nodes in the scene graph. These scene-graph nodes can be any node in the scene graph: an original interior node, an original leaf node (base patch), or a per-frame temporary node consisting of a subpatch produced by earlier subdivision and patch-splitting steps. The information in the lazy kd-node's linked list is sufficient to build the missing portion of the kd-tree if it is needed. This mechanism is similar to the one used by Ar et al. [2002] to build BSP trees for collision detection.

At the beginning of every frame, kd-tree construction is initialized with a single root kd-tree node containing the bounding box of the entire scene and a single pointer to the root of the scene graph. All further kd-tree building is triggered by traversal operations during ray tracing.

The second aspect of our fast tree build is to use the scene graph hierarchy as an acceleration structure for the build process [Hunt et al. 2007]. Since scene graph nodes may be used as proxies for large amounts of geometry, the kd-tree builder can choose to be exposed to a significantly smaller set of proxy candidates when attempting to choose a split plane, dramatically reducing the amount of work required to find a split. This allows for the builder to quickly calculate any split plane (including the top-level kd-tree splits) and eliminates the need to sort all scene geometry before choosing the first split as is required in traditional top-down kd-tree builders. Additionally, the kd-tree builder can actively refine scene graph nodes until it has enough candidates to choose a good split if it initally has too few or poor-quality split candidates.

The use of scene graph nodes as proxies also works to increase the laziness of the system: if a ray does not hit a proxy, that proxy does not need to be refined.

As shown in the results section, although the kd-tree's split planes are chosen from a smaller set of candidates when using hierarchy, the efficiency of the resulting tree is extremely close to the efficiency of a tree built by choosing the planes from all geometry. In summary: the use of hierarchy reduces tree construction time without noticeably impacting tree quality.

We use a scan-based kd-tree build algorithm [Hunt et al. 2006] rather than a sort-based algorithm, because the scan-based algorithm is much faster with almost no degradation in tree quality. The speedup from the scan-based algorithm is particularly important for portions of the scene where the scene graph is flat (has high fanout).

5.2.3 *Low-Level Grid Intersection Structure.* In addition to the fast/lazy kd-tree builder, we organize geometry into grids of small regular meshes rather than individual triangles. We call these grids *igrids* (see Figure 7). The system lazily constructs igrids. A kd-tree

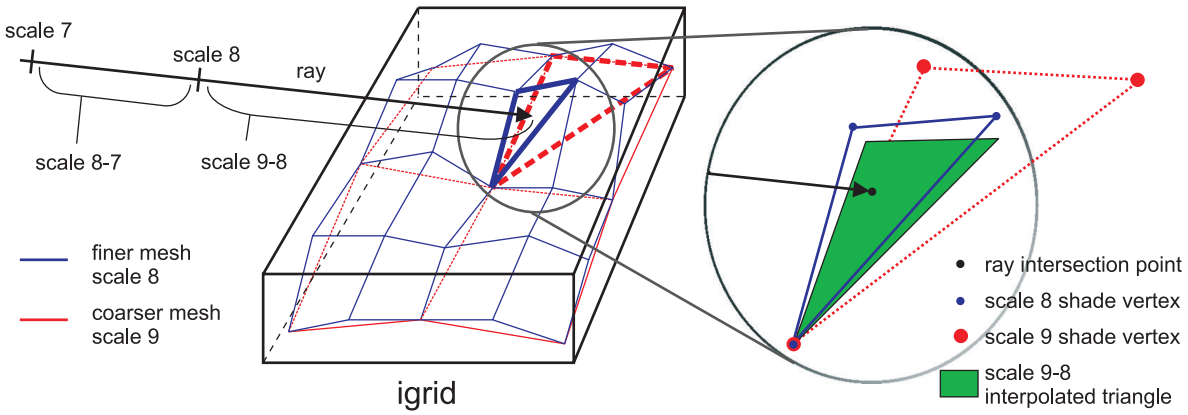


Fig. 7. An igrid holds vertices for a pair of discrete scales. One set of vertices comes from a finer scale of geometry and the other set of vertices comes from a coarser scale of geometry. The grid contains information associating each fine-scale vertex with a point on a coarse-scale triangle. This information is used to generate interpolated triangles (shown in green) for multiresolution intersection (compare with Figure 3). The grid also contains a simple bounding volume acceleration structure (not shown) based on the structure of the tessellation.

leaf node (at a particular scale) may have its associated geometry marked as "lazy." Such a node has a linked list of geometry (patches and subpatches), but the final igrid data structures have not been constructed yet. When the bounding box of the node is first penetrated by a ray, the system computes the final vertex positions for the igrid and builds a simple bounding volume hierarchy based on the internal structure of the tessellation. This acceleration structure for the igrid obviates the need to compute several levels of kd-tree splits at the bottom of the tree, instead taking advantage of the a priori knowledge that triangles within a patch have known connectivity and are usually nearly coplanar. These igrids are also important for our shading model, for enabling features such as displacement mapping, and for improving the performance of the subdivision systems. Other systems have subsequently taken advantage of knowledge about local triangle connectivity to improve the efficiency of the acceleration structure [Lauterbach et al. 2008; Hou et al. 2010], and have shown that using the tessellation topology to build a BVH over a patch provides performance comparable to an SAH-based BVH [Hanika et al. 2010].

5.2.4 *A Note on Efficiency.* The lazy kd-tree-building mechanism is extremely effective. As mentioned before, laziness is required in order to efficiently support multiresolution geometry. What is less obvious is the fact that multiresolution and hierarchical clustering provided by the scene graph make lazy evaluation much more effective.

Standard kd-tree build algorithms build top-down starting from the full geometry description of the scene and the scene's bounding box. Unfortunately this leads to a situation analogous to sifting through individual grains of sand to figure out where to split a beach in half. The time to compute the single split at the root node is linear in the amount of geometry in the scene. This is the case even for an "optimal" O (N log N) build algorithm [Wald and Havran 2006]. The kd-tree is heavily top-loaded in computational cost, greatly impairing the benefits of lazy evaluation (one always touches the root, obviously).

It should be noted that these features are not unique to the kd-tree acceleration structure. A BVH acceleration structure can take advantage of lazy build from heirarchy, fast scan, and the use of grid-specific accelerators, perhaps just as well or even better than a kd-tree. It is not our intention to advocate for use of a kd-tree rather than BVH or vice versa.

5.2.5 *Parallel Construction.* Razor uses a novel method of distributing the kd-tree construction phase across parallel worker threads. Each parallel thread builds its own kd-tree on demand, which has the effect of distributing the kd-tree construction across the threads. Most ray tracers which are interactive on today's hardware use a single acceleration structure which is built by a single thread. More recently, several systems have parallelized the build of a single kd-tree [Shevtsov et al. 2007; Zhou et al. 2008].

Razor's parallel build proceeds as follows: First, primary rays are allocated to the many worker threads. We implement this step using a work queue of screenspace tiles in order to load balance between worker threads and to achieve coherence within a tile. When a worker thread receives a screenspace tile, it begins ray tracing the tile and lazily constructs a kd-tree by adding the scene graph objects that are encountered by the primary and secondary rays for the tile. Ultimately, each worker thread builds its own kd-tree, but by merit of the lazy build only the scene graph objects relevant to the current tile are inserted into the kd-tree. When a worker thread is done with its current tile, it requests a new tile from the work queue and then continues ray tracing and lazy construction from where it left off. Notice that the worker thread inserts only the geometry that is relevant to its current tile, so this approach scales to many threads, although there is a slight caveat in that the top-level splits in the tree will be redundantly computed across the threads. Bottom-level splits, however, will be thread-local. Although secondary rays could in theory force each worker thread to build the entire acceleration structure, in practice this does not happen. There are two reasons for this: First, primary rays require much higher geometric detail than typical secondary rays, so even ill-behaved secondary rays have relatively little impact on tree size. Second, the secondary rays originating from nearby primary rays tend to exhibit reasonable coherence (although this might be less true for some types of rays, such as those used for final gather).

By parallelizing kd-tree construction (which has traditionally been a sequential bottleneck to dynamic ray tracers), we achieve speedup of up to 7.36x with 8 worker threads even though we rebuild the kd-tree at the start of every frame (see Section 6.6).

5.2.6 *Determining the Geometry Scale Needed by a Ray.* As mentioned earlier, each ray in our system has a scale function associated with it that specifies which geometric level of detail should be used for intersection tests with the ray. The value returned by

the scale function varies with the position on the ray. For example, the scale value for a camera ray increases as the distance from the camera gets larger. We will now explain how this scale function is computed.

Our approach builds on the concepts of ray differentials [Igehy 1999] and path differentials [Suykens and Willems 2001], which we briefly summarize here. Ray differentials provide information needed to estimate the origin and direction of the ray's immediate neighbor, or, equivalently, to estimate the geometry of the beam that the ray implicitly represents. At the point where a ray intersects a surface, the cross section of this beam can be used to compute a footprint for texture filtering. Ray differentials are used for this purpose in traditional ray tracing systems. Following Igehy's notation, a ray $\bar{\mathbf{R}} = \langle \mathbf{P}\ \mathbf{D} \rangle$ with origin $\mathbf{P}$ and direction $\mathbf{D}$ has differentials.

$$\frac{\partial \bar{\mathbf{R}}}{\partial x} = \left\langle \frac{\partial \mathbf{P}}{\partial x}\ \frac{\partial \mathbf{D}}{\partial x} \right\rangle \tag{7}$$

$$\frac{\partial \bar{\mathbf{R}}}{\partial y} = \left\langle \frac{\partial \mathbf{P}}{\partial y}\ \frac{\partial \mathbf{D}}{\partial y} \right\rangle \tag{8}$$

Our system uses a ray's differentials to compute the geometric level of detail (i.e., scale) needed by the ray. Inuitively, the idea is to insure that at any particular point on the ray the size of tessellated triangles seen by the ray is proportional to the width of the ray's cross section. In the case of a nonsquare cross section, the minimum of the height and width is used, which guarantees that the system tessellates and shades with geometry that is at least as detailed as needed in each dimension. More formally, the system computes a function that provides a *single, isotropic* world-space scale value at each point on the ray.

Our system currently simplifies the problem of computing footprints from path differentials by retaining just the most important differential pair along with the scale value used at the previous intersection point of the path. Currently, the choice of the most important differential pair is hard coded into our shaders. Area light rays provide an example of how this simplification works: as they first leave the surface, their footprint is a constant determined by the spacing on the surface (which in turn was determined by differentials from the camera ray), but as they move further away from the surface, the area-light differential pair takes over, allowing the footprint to grow rapidly thereafter. For some some effects it would probably be necessary to track more differentials to obtain correct LOD behavior.

In the general case, the footprint $d$ for a single ray differential is a quadratic function of the distance $t$ along the ray, but it can be reasonably approximated or even exactly represented as a linear function in most cases.

$$d = A \pm tB \tag{9}$$

Because we only need to compute an isotropic scale, we can typically reduce a diffential pair to a single linear footprint function. For example, for the diffential pair introduced by shadow rays to an area light, we set $A = 0$ and

$$B = \min\left( \left\| \frac{\partial \mathbf{D}}{\partial x} \right\|, \left\| \frac{\partial \mathbf{D}}{\partial y} \right\| \right). \tag{10}$$

There are some subtle complications that arise when we use the scale function with the multiscale kd-tree. Conceptually, the multiscale kd-tree is equivalent to a collection of several kd-trees, each valid only for a particular range of scales (e.g., one tree is valid for the range of scales between 8.0 and 9.0). Thus, in order to traverse the multiscale kd-tree, the system must determine which segment(s) of the ray fall into each scale range. To make this determination, the system computes the point(s) on the ray at which the scale crosses integer thresholds (8.0, 9.0, etc.). We refer to these points as transition points. This computation requires inverting the scale function. In the general case in which there are many active differentials for a ray, this inversion could be complicated and expensive, but the simplification of maintaining just one linear footprint function plus a starting footprint makes it tractable. To compute the ideal transition point $t_d$ (as a distance along the ray) for a discrete scale whose world-space length is $d$, we compute

$$t_d = \pm \frac{\pm d - A}{B}, \tag{11}$$

where the $\pm$ represents appropriate conditional code, not a dual solution.

In theory, it is possible for a ray to change scale too rapidly, so that tunneling could be caused by switching too rapidly between the discrete scale levels in the multiscale kd-tree. (The tunneling discussion earlier only considered the behavior within a particular scale level such as 8.0–9.0). To insure that such tunneling at discrete scale transitions does not occur, the system enforces a minimum segment length for each scale level in the tree. This length is proportional to $d$, the maximum triangle edge length associated with that scale. We implement this enforcement, along with the choice of the desired initial scale value for a ray, by iteratively assigning the transition points $t_{d,0}$, $t_{d,1}$, etc., and associated scales, beginning at the start of the ray. The assignment of these actual transition points is performed before the start of ray traversal. The system also perturbs the scale function to insure that the scale function is flat at the start and end of the segment associated with a particular discrete scale level, which is also necessary to avoid tunneling. This perturbation is achieved by setting the values $t_s$ and $t_e$, which exist for each scale level, and are used in multiscale triangle intersection, as discussed earlier in Section 5.1.2.

$$t_s = t_{d,n} + \text{constant} * d \tag{12}$$

$$t_e = t_{d,n+1} - \text{constant} * d \tag{13}$$

A side effect of these antitunneling adjustments is that it is possible for very rapidly diverging rays to be intersected with geometry that is more detailed than desired. But tunneling is still avoided; the only penalty is an increase in rendering cost due to the generation of the detailed geometry.

The computation of transition points from ray differentials is complicated and uses messy special-case code that makes it difficult to be certain that the approximations will always be correct. Our advice to future designers of multiscale ray tracing schemes would be to strive to avoid the need to invert the footprint function. This avoidance would probably be greatly facilitated by replacing the kd-tree with a BVH, since the spatial overlap allowed by a BVH would be useful for representing the ambiguity of a surface's location when it is represented at multiple scales.

## 5.3 Split-Phase Shading

The design of our shading system was driven by the desire to decouple shading from visibility. The REYES system [Cook et al. 1987] accomplishes this goal, but in a system that only supports camera rays. Our goal was to extend the REYES approach to a ray tracing framework. Like REYES, our goal is to perform shading computations at the vertices of a finely tessellated polygon mesh and then interpolate to specific hit points, rather than shading at the hit points themselves. The REYES algorithm has amply demonstrated the benefits of this technique: shading calculations can be

performed in highly regular and coherent batches in their natural coordinate space on the surface, and a variety of otherwise tricky operations such as differential calculations and programmable displacement mapping are simplified.

Another critical performance characteristic is that this technique creates a separation between functions which can be band-limited a priori from functions which cannot. In REYES, this means that procedural shaders (expected to band-limit themselves) are separated from visibility calculations. The extremely expensive procedural shading operations can be performed less frequently, at the vertices of the grid, while the less-expensive-to-evaluate but ill-behaved visibility function is supersampled.

Our system uses this concept by leveraging the system's multiresolution representation of geometry. Shading is explicitly factored into two phases. Operations in the first phase are performed at the vertices of grids. The functions calculated in phase one are expected to be band-limited to the frequency of the sampling implied by the tessellation of the grid. Additionally, as the results are cached and reused by the system these values must be independent of viewing direction. The first phase of shading is calculated lazily the first time that a ray strikes the given grid and requires the results.

The second phase of shading is more typical of a ray tracer. When a ray strikes a grid, the results of the first phase are fetched (following lazy evaluation of the first phase if necessary) and interpolated to the hit point. These values are available as parameters to phase two. Shading in this phase is as flexible as shading in any typical ray tracer. In typical use a BRDF function would be generated from the results available from phase one, and distribution sampling of the BRDF would be performed by casting secondary rays as necessary.

A similar split-phase shading model has been applied previously in physically-based rendering systems [Pharr and Humpreys 2004] in order to enforce properties such as BRDF reciprocity. The separation in our system is more pragmatic and performance-oriented. Shading operations should be factored into phase one as much as possible, with the remainder in phase two, without necessarily considering physical interpretations. Creative abuse of the shading system is certainly an option, such as using various mapping tricks in either of the phases, or casting various physical-or-otherwise secondary rays in phase one. We have already experimented successfully with variants of irradiance caching based on casting rays in phase one (Christensen et al. [2003] do something similar).

Altogether, there are five sources of performance improvement in this shading system. First, redundant shading computations caused by high rates of visibility supersampling are reduced. Second, phase-one is performed on a grid, so that shading "derivative" computations may be computed by discrete differences with neighbors, rather than by potentially more expensive techniques such as executing the shader three times for each hit point [Gritz and Hahn 1996], or automatic differentiation using dual number arithmetic [Piponi 2004; Karrenberg et al. 2010; Gritz et al. 2010]. Third, the grid structure of phase-one shading makes it amenable to acceleration by SIMD mechanisms like $\times$86 SSE. Fourth, grid-based shading improves memory-access locality. Finally, split-phase shading improves the efficiency of SIMD ray packets because there are fewer distinct kinds of phase-two shaders than kinds of combined shaders.

Our experimental system uses simple phase-one shaders that read and filter surface colors from a texture map and compute normal vectors from a bump map. Our phase-two shading currently includes area light source sampling, mirror reflection, hemisphere sampling of ambient occlusion, and simple diffuse and Schlick [Schlick 1994] BRDF evaluation. Shaders are written in C++ within our rendering system; we have not yet defined or implemented a stand-alone shading language for this two-phase shading scheme.

## 6. RESULTS

We have evaluated our prototype system using several scenes, and various rendering configurations which are described in Figure 8.

### 6.1 Overall System Performance

We evaluate the overall performance of our system using a workstation-class desktop PC running Windows XP Pro 64-bit edition. This PC has two 4-core Intel Xeon processors (8 cores total) with a 1333MHz front-side-bus. Most results in this article were gathered using processors clocked at 2.66GHz (Xeon X5355), but due to a machine upgrade during the review cycle, the results in Figure 8 were gathered using processors clocked at 2.83GHz (Xeon E5440). Most of our demos use about 4GB of DRAM.

Figure 8 summarizes performance for our scenes under our three rendering configurations, which are Fast, Balanced, and High Quality. Figure 9 compares the visual quality of the three settings. Our results show that Razor delivers high performance for scenes consisting of millions of visible micropolygons and hundreds of thousands of base patches, even when many secondary rays are cast. Note that performance varies significantly depending on scene characteristics; more complex scenes that have more surfaces touched by secondary rays or that have lots of subpixel geometry (before tessellation) generate more micropolygons and take longer to render.

Most interactive ray tracing systems do not support a combination of features and visual quality (e.g., fully dynamic scenes and soft shadows) comparable to Razor, so is difficult to make unambiguous performance comparisions with other systems. But with this caveat in mind, we attempt some comparisions.

*Comparison to grid acceleration structure*. In terms of support for dynamic scenes, one comparable system is Wald et al.'s grid-based dynamic-scene ray tracer [Wald et al. 2006]. Since Razor is designed to be run with soft shadows enabled while the grid system does not support this feature, we compare the two systems using the metric of rays per second when each system is running in its preferred configuration. For the Courtyard64 scene at 1024 $\times$ 1024 with 605K base patches, 24 million instantiated micropolygons and 72 rays/pixel, Razor traces 1.3 million rays/sec on a single Xeon X5355 core.[3] For other reasonable configurations on large scenes, we have measured rates of up to 2.0 million rays/sec on a single core. The grid system traces about 3.0 millon rays/sec on a 174K triangle model with primary rays and hard shadows, on two 3.2GHz Pentium 4's. For ray tracing code, one Xeon X5355 core is approximately equal to two 3.2GHz Pentium 4 cores. Thus, after adjusting for hardware differences, Razor's performance as measured in ray segments/sec is only slightly less than that of the dynamic-scene grid ray tracer, even though Razor does substantially more useful work. In particular, Razor tesselates to micropolygons, robustly handles large scenes, and traces diverging secondary rays. However, it is important to realize that this comparison may not be completely fair. Razor is likely benefiting from higher ray coherence that occurs when more rays are shot per pixel. In addition, the models are not the same, and one of the article's anonymous reviewers stated that the grid system performs relatively poorly on the model we chose for evaluating it.

*Comparison to BVH acceleration structure*. Two of the highest-performing BVH-based dynamic scene ray tracers are Wald et al. [2007a] and Lauterbach et al. [2006]. Again a direct comparison to either of these systems is difficult to make since neither are distribution ray tracing systems. Wald et al. [2007a] precomputes the

[3]On eight cores, it traces 9.1 million rays/sec.

| Title | Court1 | Court64 | Forest | Court64 | Forest | Court64 | Forest | FairyForst |
|---|---|---|---|---|---|---|---|---|
| Quality | Fast | Fast | Fast | Balanced | Balanced | High | High | Fast |
| Base Patches | 16,690 | 605,308 | 27,341 | 605,308 | 27,341 | 605,308 | 27,341 | 95,916 |
| Tessellation Type | Bilinear | Bilinear | CC | Bilinear | CC | Bilinear | CC | Bilinear |
| Area Lights | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 |
| Image Size | $1024^2$ | $1024^2$ | $1024^2$ | $1024^2$ | $1024^2$ | $1024^2$ | $1024^2$ | $1024^2$ |
| Primary Rays/Pixel | 1 | 1 | 1 | 4 | 4 | 8 | 8 | 1 |
| Shadow Rays/Pixel/Light | 1 | 1 | 1 | 4 | 4 | 4 | 4 | 1 |
| Total Rays/Pixel | 3 | 3 | 3 | 36 | 36 | 72 | 72 | 2 |
| Max Micropolygon Area | 16 $pixels^2$ | 16 $pixels^2$ | 16 $pixels^2$ | 4 $pixels^2$ | 4 $pixels^2$ | 1 $pixel^2$ | 1 $pixel^2$ | 16 $pixels^2$ |
| Micropolygons* | 1,081,594 | 1,250,860 | 1,181,220 | 4,396,510 | 4,017,662 | 15,832,144 | 14,605,880 | 1,185,024 |
| Shaded Microvertices* | 491,370 | 514,794 | 717,566 | 1,729,341 | 2,293,944 | 6,082,358 | 7,850,704 | 617,222 |
| Max Memory Usage | 0.71 GB | 1.3 GB | 1.4 GB | 2.0 GB | 2.2 GB | 3.8 GB | 4.5 GB | 1.5 GB |
| Render Time | 0.42 s | 0.54 s | 1.24 s | 3.10 s | 4.64 s | 6.68 s | 9.72 s | 0.78 s |

| Title | Rabbit | Trio | Rabbit | Trio | Rabbit | Trio |
|---|---|---|---|---|---|---|
| Quality | Fast | Fast | Balanced | Balanced | High | High |
| Base Patches | 383,283 | 469,263 | 383,283 | 469,263 | 383,283 | 469,263 |
| Tessellation Type | Bilinear | Bilinear | Bilinear | Bilinear | Bilinear | Bilinear |
| Area Lights | 2 | 2 | 2 | 2 | 2 | 2 |
| Image Size | $1024^2$ | $1024^2$ | $1024^2$ | $1024^2$ | $1024^2$ | $1024^2$ |
| Primary Rays/Pixel | 1 | 1 | 4 | 4 | 8 | 8 |
| Shadow Rays/Pixel/Light | 1 | 1 | 4 | 4 | 4 | 4 |
| Total Rays/Pixel | 3 | 3 | 36 | 36 | 72 | 72 |
| Max Micropolygon Area | 16 $pixels^2$ | 16 $pixels^2$ | 4 $pixels^2$ | 4 $pixels^2$ | 1 $pixel^2$ | 1 $pixel^2$ |
| Micropolygons* | 3,697,118 | 4,341,230 | 11,803,430 | 13,143,294 | 33,766,042 | 41,306,402 |
| Shaded Microvertices* | 1,312,877 | 1,571,279 | 3,829,106 | 4,660,919 | 11,070,289 | 14,286,980 |
| Max Memory Usage | 2.2 GB | 5.2 GB | 4.6 GB | 6.9 GB | 8.5 GB | 9.9 GB |
| Render Time | 1.81 s | 5.40 s | 9.74 s | 12.5 s | 20.7 s | 25.2 s |

Fig. 8. Performance of Razor for different scenes and quality settings. These results were gathered on a machine with two quad-core Intel Xeon E5440 processors at 2.83GHz (8 cores total). "CC" stands for Catmull-Clark. Unless otherwise specified, the viewpoints shown in the thumbnails are used throughout the Results section. Entries marked with a (*) are measured on a single core to avoid overcount. The Rabbit and Trio scenes are colorless due to an incompatibility in our toolchain with Blender textures.

The models in these images appear courtesy of their respective owners. The character models in Courtyard and Courtyard64 are copyright Digital Extremes, all rights reserved. The character models in Courtyard and Courtyard64 use motion capture data from mocap.cs.cmu.edu (NSF EIA-0196217). The dragon model in Forest appears courtesy of Jeffery A. Williams and Intel Corporation. The killeroo model in Forest appears courtesy of Phil Dench, Martin Rezard, and headus (metamorphosis) Pty Ltd. The floor in Forest appears courtesy of Jonathan Dale. The background models in Forest appear courtesy of DAZ Productions. The FairyForest scene appears courtesy of DAZ Productions. The Rabbit and Trio scenes appear courtesy of the Blender Foundation.

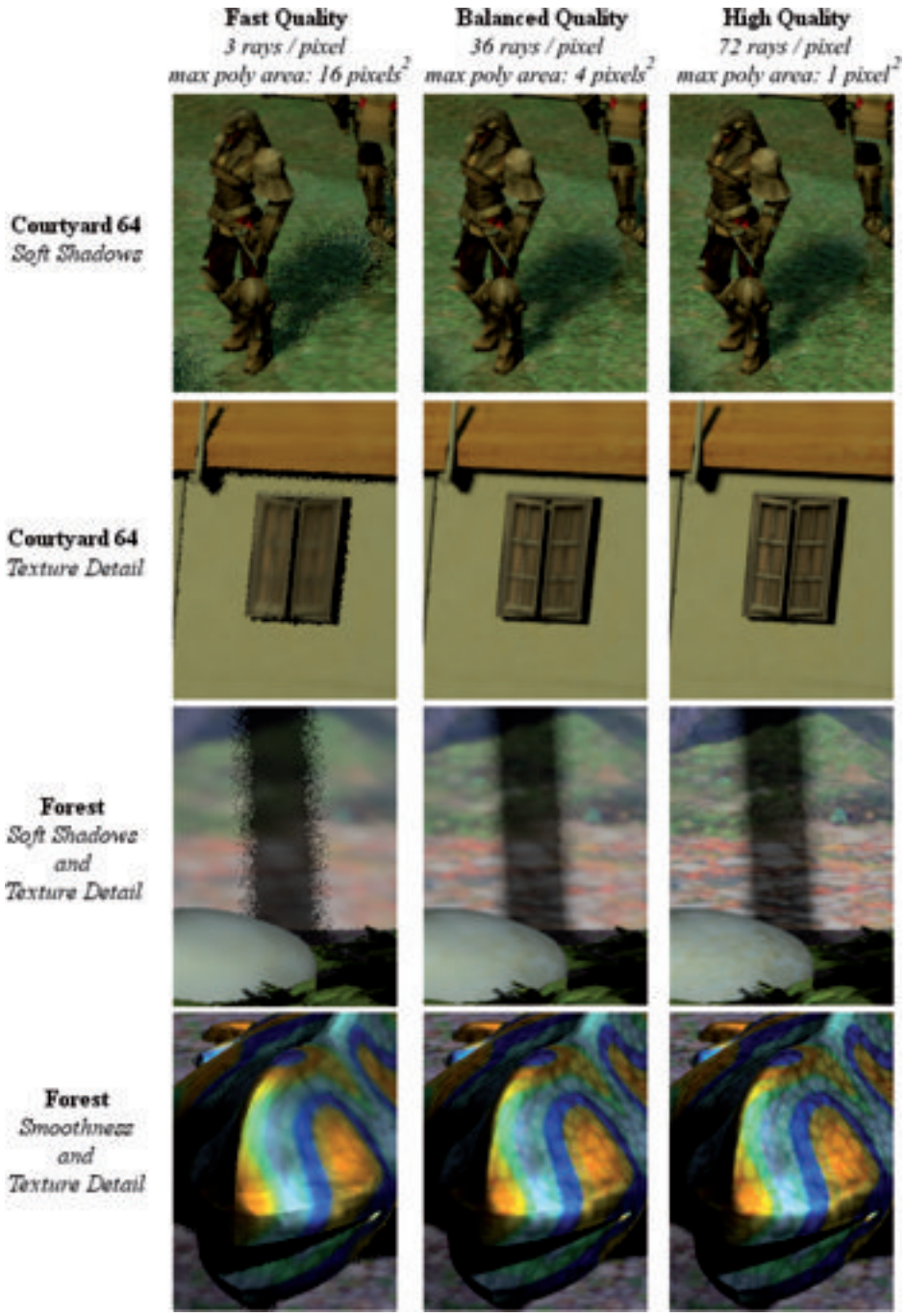


Fig. 9. Comparison of Razor's quality settings. The models in these images appear courtesy of their respective owners. The character models in Courtyard64 are copyright Digital Extremes, all rights reserved. The character models in Courtyard64 use motion capture data from mocap.cs.cmu.edu (NSF EIA-0196217). The dragon model in Forest appears courtesy of Jeffery A. Williams and Intel Corporation. The killeroo model in Forest appears courtesy of Phil Dench, Martin Rezard, and headus (metamorphosis) Pty Ltd. The floor in Forest appears courtesy of Jonathan Dale. The background models in Forest appear courtesy of DAZ Productions.

BVH topology (i.e., which geometry is in which nodes) and only recomputes the bounds each frame. As a result, the performance of the acceleration structure depends on the choice of scene configuration for optimizing the initial BVH topology and on the type of movement. For a scene with arbitrary movement, for example, characters walking around in a game, the initial BVH topology may permit arbitrary degradation of the acceleration structure. Of the scenes analyzed in Wald et al. [2007a], we choose the Fairy Forest

scene for comparison with Razor because this scene is reasonably complex in both geometry and textures. For the Fairy Forest scene, the Wald et al. [2007a] BVH-based ray tracer achieves a frame-rate of 2.3fps on a fully textured 1024 × 1024 image with hard shadows. This corresponds to a rate of 4.6 million rays per second for that scene. This is approximately 3.5× faster than Razor, but on a comparison case (very few secondary rays) that places Razor at a disadvantage. Lauterbach et al. [2006] follows a similiar approach to Wald et al. [2007a], but rebuilds the topology of the acceleration structure when its quality degrades excessively. Lauterbach et al. [2006] do not evaluate their system on any textured scenes, which makes it difficult to make a meaningful quantitative performance comparision with Razor.

More recently, Boulos et al. [2007] have built a BVH-based distribution ray tracing system that uses packets implemented on a machine with 4-wide SSE. On a single 2.4GHz Opteron 880, they achieved performance of 2M – 3M rays/sec for Whitted ray tracing [Whitted 1980] and 0.9M – 1.5M rays/sec for distribution ray tracing [Cook et al. 1984] of various effects. This distribution ray tracing performance level is similar to that achieved by Razor (1.3M rays/sec on 1 core of an X5355 Xeon 2.66MHz), although the comparision is only approximate because of the differences in scenes, rendering recipes, and hardware platforms; and because Razor's performance includes a lazy rebuild of the acceleration structure on every frame. Although Razor is not signifiantly faster than simpler systems, it demonstrates that it is possible to support more features without substantially reducing performance. These features include multiresolution micropolygons and subdivision surfaces.

*Other systems*. The Manta system [Bigler et al. 2006] is targeted at interactive use and can be configured to support distribution ray tracing effects, but its primary focus seems to have been on static scenes; there is not enough published information available about its parallel performance for dynamic scenes to permit an informative performance comparison. Work subsequent to Razor has shown that it is possible to implement a parallel build of a single kd-tree with high performance on CPUs [Shevtsov et al. 2007] and GPUs [Zhou et al. 2008]. Benthin et al. have built a system that directly ray traces Catmull-Clark subdivision surfaces using 64-ray packets without an explicit tessellation step [Benthin et al. 2007]. For that system's strong point of highly coherent rays (camera and shadow rays), they claim a speedup between 2.7× and 16.8× compared to Razor, using similar hardware.

## 6.2 Comparison to Batch Renderers

Razor supports features such as soft shadows and ambient occlusion that have traditionally been associated with batch renderers rather than interactive ray tracers (although as of early 2011 this situation is beginning to change). Figure 10 compares the performance of Razor to that of Mental Ray, a fast batch renderer integrated with the Maya 8.0 modeling package. Both rendering systems are configured with the same image resolution, number of primary rays, and number of secondary rays. Mental Ray is configured to use scan line rendering for primary rays (which is slightly faster than the alternative) and to use ray tracing for all secondary rays. For these comparisons we used a PC with a dual-core Pentium D 3.2GHz processor, with hyperthreading disabled, and 4.0GB of DRAM.

These experiments show that Razor is 4.2× to 6.6× faster than Mental Ray for similar batch quality ray tracing settings, even though Razor is an experimental system while Mental Ray is a highly optimized commercial rendering system that presumably incorporates years of performance tuning.

| Scene | Razor frame time | Mental Ray frame time | Razor speedup |
|---|---|---|---|
| Courtyard64 | 22.0 s | 146 s | 6.6x |
| Forest | 25.2 s | 107 s | 4.2x |

Fig. 10. Performance comparison between Razor and a batch renderer (Mental Ray) at high-quality settings on a dual-core Pentium D PC. Both renderers are configured for 1024 × 1024 images, 16× supersampling, 96 shadow rays/pixel/light, two lights, and two processing threads. Razor is configured for a maximum micropolygon area of 2 pixels. The Courtyard64 scene uses bilinear geometry in Razor and no subdivision in Mental Ray while the Forest scene uses Catmull-Clark subdivision in both renderers.

| | | | Courtyard64 – 230 | | Courtyard64 – 460 | |
|---|---|---|---|---|---|---|
| Scan | Hier | Lazy | Build | Trace | Build | Trace |
| × | × | × | 15.59s | 1.49s | 15.59s | 1.48s |
| × | ✓ | × | 9.62s | 1.49s | 9.62s | 1.48s |
| ✓ | × | × | 3.89s | 1.47s | 3.89s | 1.46s |
| × | × | ✓ | 3.60s | 1.48s | 9.36s | 1.46s |
| ✓ | ✓ | × | 3.57s | 1.47s | 3.57s | 1.46s |
| ✓ | × | ✓ | 1.14s | 1.47s | 2.10s | 1.47s |
| × | ✓ | ✓ | 0.387s | 1.48s | 3.54s | 1.46s |
| ✓ | ✓ | ✓ | 0.143s | 1.47s | 1.63s | 1.47s |

Fig. 11. Razor's three techniques for fast acceleration structure construction (lazy build, build from hierarchy, and scan/bin SAH approximation) are complementary. Viewpoint information: Courtyard64 frame 230 is the leftmost image in Figure 1, and Courtyard64 frame 460 is the second image from the left in that figure.

Comparisions between such different systems are fraught with difficulties, but this experiment does indicate that the performance of our experimental system already exceeds the performance of one highly optimized rendering architecture used for batch rendering. It is worth nothing that one reviewer of this article mentioned that Mental Ray uses camera-directed LOD, and suggested that it would have been better to have compared performance to that of full multiresolution systems such as Photorealistic Renderman and/or the system by Hanika et al. [2010].

## 6.3 Fast Build of Acceleration Structure

Razor's rendering architecture combines several techniques to rapidly build a high-quality acceleration structure for large scenes.

To evalute the effectiveness of these techniques, we measure the changes in Razor's performance as the various techniques are enabled and disabled. To keep this discussion as simple as possible these measurements are made on a single core of our Xeon X5355 PC (that is, with Razor's parallelism disabled). We decompose rendering time into two components: build time and tracing time. Normally these two kinds of computation are intermingled due to Razor's on-demand build design, so we artifically separate them by first measuring total rendering time, then measuring trace time by rerendering the same frame without deleting the on-demand data structures. Build time is the difference between these two measurements.

Figure 11 shows the performance of Razor's three build techniques for two viewpoints from the Courtyard64 scene. This table reports single-threaded Xeon X5355 build and trace times. To focus

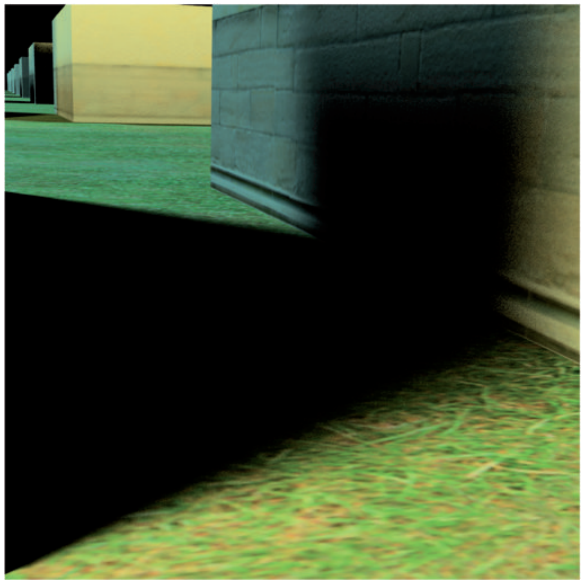

Fig. 12. For this scene and viewpoint, per-ray adaptive LOD drastically improves performance compared to a uniform tessellation of all surfaces hit by rays. Adaptive LOD takes 92 sec, uses 247MB of memory, and has 39,081 VTune L2 cache-miss events. Uniform LOD takes 139 sec, uses 3,694MB of memory, and has 298,377 cache-miss events. The models in this image appear courtesy of their respective owners.[4]

on kd-tree build time, the "fast" rendering settings are used, but with patch tesselation essentially disabled. We consider two cases with different depth complexity. Frame 230 has high depth-complexity and fewer visible polygons while frame 460 has low depth complexity and the polygons in the scene are mostly visible. The three build acceleration techniques in Figure 11 are as follows. Lazy build avoids building portions of the kd-tree which are not touched by any rays. Hierarchical build accelerates kd-tree construction by using scene graph bounding boxes as aggregate stand-ins for the geometry they contain; eventually these stand-ins are unpackaged into their component geometry to insure an adequate number of candidate split planes [Hunt et al. 2007]. Scan build approximates the surface area heuristic [Hunt et al. 2006].

The results indicate that the build acceleration techniques do not degrade acceleration structure performance and work better together than independently. For instance, the trace time remains constant regardless of the build acceleration techniques that are used. Additionally, in frame 230 the three accleration techniques produce more of a speedup when they are all enabled (109×, last row) than the accumulation of the individual speedup when they are enabled one at a time (28.1×, rows 2 through 4). In particular, lazy build and hierarchical build work exceptionally together since the aggregate stand-ins in the scene graph hierarchy allow laziness to more easily identify larger regions of the scene as irrelevant to the current frame. These considerations are discussed in more detail in a companion paper [Hunt et al. 2007].

## 6.4 Evaluation of per-Ray Geometric LOD

In Razor, each ray selects its own LOD for geometry intersections and this LOD varies with the distance along the ray. One of the main advantages of this mechanism is that for a particular surface, shadow rays will often request a LOD that is substantially coarser than that requested by primary rays hitting the same surface. This coarse LOD improves the coherence of shadow rays, which in turn increases packet occupancy and reduces the size of the memory working set.

Figure 12 illustrates a case where adaptive LOD makes a significant difference, especially when one considers the implications for future hardware architectures. We compare the use of a fixed LOD for this entire scene (set such that nearby objects are tesselated correctly) with Razor's standard per-ray adaptive LOD. The nonadaptive LOD takes 50% longer to render, uses 15 times as much memory, and (most importantly) increases the L2 cache miss rate by a factor of 7.6. The primary cause of these differences is the fact that the adaptive LOD algorithm can use a lower tessellation rate for the soft shadow rays hitting the off-screen wall that casts the soft shadow in the forground.

On current hardware architectures, ray tracing is mostly floating-point limited, so the penalty in rendering time for nonadaptive LOD on these architectures is relatively small (50% in this case). However, the best way to improve the price-performance ratio of hardware for a ray tracing workload is to add more cores by reducing the ratio of cache to cores. On such machines, algorithms such as Razor's adaptive LOD that effectively manage the memory hierarchy should have a larger advantage. Christensen et al [2003] make similar arguments in support of their LOD mechanism, although they are concerned primarily with DRAM capacity rather than cache miss rates.

## 6.5 Evaluation of Shading at Micropolygon Vertices

One of the most aggressive design decisions in Razor is to shade at micropolygon vertices rather than at ray hit points. The results presented in Figure 8 can be used to evaluate this approach. We focus on the Forest scene, since most of its geometry was designed for Catmull-Clark subdivision which is needed for the micropolygon approach to work well. At "high"-quality settings with micropolygons targeted to be two pixels or less in area, the system shades 7.9 million microvertices during phase one of shading. Since the supersampling rate for this one-million pixel image is eight samples/pixel, Razor is shading slightly fewer points than a traditional ray tracer would. However, as discussed earlier, this comparison understates the benefits of the micropolygon approach because grid-based shading is more hardware friendly and provides differential information for free. The crossover point between traditional per-ray shading and grid-based shading occurs at a higher supersampling rate than we had originally hoped. The primary culprit is overtessellation, which in turn is primarily caused by two factors: (1) the use of binary subdivision, which is well-known to result in overtessellation, and (2) the need to generate both a fine mesh and coarse mesh, both at discrete scales, with the fine mesh tessellated at a rate higher than is needed for any particular ray. In short, the LOD mechanism causes shading to occur more densely than is optimal. Decoupled shading also requires storage of the results from phase one of shading (12 floats, thus 48 bytes, per vertex in our configuration). Despite these issues, our results do show that that the REYES micropolygon approach can be combined with a ray tracing visibility engine. It is worth noting that Razor could be easily modified to shade at hit points like a conventional ray tracer. In such a mode, the tessellation rate would be reduced to the minimum rate necessary to maintain the appearance of curved surfaces.

[4]The character models in Courtyard64 are copyright Digital Extremes, all rights reserved. The character models in Courtyard64 use motion capture data from mocap.cs.cmu.edu (NSF EIA-0196217).

Courtyard64

| Quality | 1 Core | 8 Cores | Speedup |
|---|---|---|---|
| High | 55.2 s | 7.50 s | 7.36x |
| Balanced | 25.2 s | 3.46 s | 7.28x |
| Fast | 3.23 s | 0.61 s | 5.30x |

Forest

| Quality | 1 Core | 8 Cores | Speedup |
|---|---|---|---|
| High | 73.6 s | 10.8 s | 6.81x |
| Balanced | 33.8 s | 5.17 s | 6.54x |
| Fast | 2.70 s | 0.90 s | 3.00x |

Fig. 13. Razor achieves parallel speedup of up to 7.36× on an 8-core machine, with the best speedups at higher image quality settings.

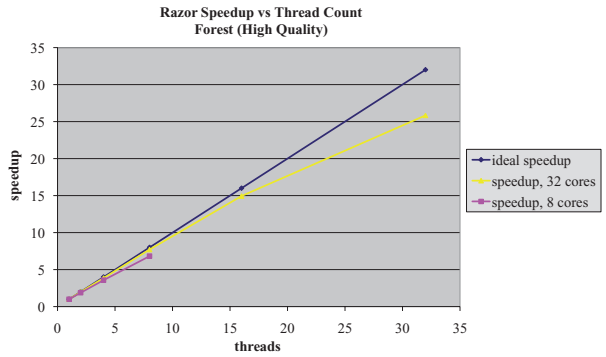


Fig. 14. Razor achieves good parallel speedup using its high-quality settings on an 8-core machine and a newer 32-core machine.

## 6.6 Parallel Speedup

Figure 13 shows that for the balanced and high-quality settings that are specifically targeted by Razor's design, Razor achieves parallel speedup between 6.54 and 7.36 on an eight-core machine. Of particular note is that this speedup includes not only the trace phase but also the kd-tree build phase; each thread produces a newly constructed kd-tree that is suited for its workload on the current frame. These results indicate that Razor's parallel build does not add significant overhead compared to a sequential build. Figure 14 plots speedup for the "high"-quality setting on both the original 8-core machine and a newer 32-core Xeon machine.

At the "fast" quality settings, Razor's speedup is noticeably lower because a larger fraction of the total build time is spent on constructing the upper levels of the kd-tree, which is work that is redundantly performed on each core. We have not gathered results for hundreds of cores at any quality setting, and we acknowledge that the parallelization scheme might need changes in order to scale well to future machines of that size.

## 6.7 Evaluation of Ray Differentials in a Distribution Ray Tracing Setting

In scenes with glossy surfaces, careful management of geometric detail should improve system performance. The reason is that rays reflected from a glossy surface typically have large ray differentials indicating that geometry can be represented coarsely. In this subsection, we evaluate the performance of Razor for distribution ray tracing of glossy surfaces.

The test scene is a version of the Courtyard64 scene that has been modified such that all surfaces in the scene are glossy. We configure Razor to use the High-Quality settings from column 6 in Figure 8. We modify these settings to support glossy reflection using distribution ray tracing: each primary ray spawns four glossy reflection rays, using a 5.0 degree apex angle for the glossy cone. In turn, each glossy reflection ray spawns four shadow rays (per light) when it hits a surface. Thus, Razor will produce one-bounce glossy reflection with shadows in the reflection. Under these settings, Razor casts up to 360 rays per pixel to generate an image with full-scene glossy reflection (Figure 15 (top)).

Since Razor uses ray differentials to control level of detail, we can evaluate the effectiveness of LOD by manipulating the behavior of ray differentials. More specifically, we can disable LOD for glossy rays by constraining the footprint for a glossy ray to remain constant along the entire length of the ray, using the value computed at the

| config | t | mem | micropolys | shaded verts |
|---|---|---|---|---|
| noGlossy | 7.4 s | 3.1 GB | 15,713,038 | 6,063,158 |
| GlossyStd | 52.4 s | 14.2 GB | 28,902,930 | 13,059,549 |
| GlossyNoLOD | 68.3 s | 38.9 GB | 52,540,182 | 25,005,964 |

Fig. 15. LOD provides a performance improvement for glossy rays. (Top) An image of the Courtyard64 scene with 4 glossy rays per primary ray using "High"-quality settings (72 rays/pixel, micropoylgon area $\leq$ 1 pixel$^2$). The scene renders in 52.4 sec on eight cores. (Bottom) "noGlossy" uses no glossy rays, "GlossyStd" uses glossy rays with Razor's standard ray-differential-based LOD, and "GlossyNoLOD" uses glossy rays with ray footprints frozen at each ray's origin (i.e., no LOD). "t" is render time per frame (8 cores), "mem" is memory used (8 cores), "micropolys" is number of micropolygons (1 core), and "shaded verts" is number of shaded microvertices (1 core). The last two measurements use only one core to avoid overcount. The "GlossyNoLOD" trial requires memory flush (Section 6.8). The models in this scene appear courtesy of their respective owners.[5]

[5]The character models in Courtyard64 are copyright Digital Extremes, all rights reserved. The character models in Courtyard64 use motion capture data from mocap.cs.cmu.edu (NSF EIA-0196217).

origin; that is, the ray footprint does not grow with distance along the ray as it normally would.

We compare the performance of Razor under three different configurations: (1) no glossy rays (*noGlossy*); (2) glossy rays traced using Razor's regular LOD driven by ray differentials (*GlossyStd*); and (3) glossy rays traced using a ray footprint frozen at the ray origin (*GlossyNoLOD*). Figure 15 (bottom) presents the results of this comparison. Note that the images produced by GlossyStd and GlossyNoLOD are nearly identical; GlossyStd uses LOD to generate the same quality image at lower cost. We observe that using LOD for the glossy rays provides the expected savings in execution time, memory usage, and micropolygon count, as compared to the configuration with LOD disabled. Specifically, execution time is reduced by 23% and memory usage is reduced by 63%, confirming that Christen et al.'s observations about the value of using LOD [Christensen et al. 2003] hold true for the Razor system for glossy rays.

These experiments also demonstrate that distribution sampling of glossy reflections is costly, even with LOD. With glossy reflection turned off, the system traces 10.2 M rays/sec, but with glossy reflection turned on, performance drops to ∼7.2 M rays/sec. This decrease in performance per ray is due to the increased incoherence of secondary rays with respect to each other and with respect to primary rays. In particular, the shadow rays spawned from glossy rays are less coherent than shadow rays launched from primary hit points. Such performance drops for secondary rays are typical; Razor's per-core glossy ray performance of 0.9M rays/sec/core is similar to the rates of 0.9M – 1.5M rays/sec/core reported in Boulos et al. [2007].

### 6.8 Memory Usage and Memory Flush

For typical runs Razor uses a large amount of memory. We expect this high usage would be significantly reduced by adding a software-managed geometry cache for the REYES-style grids, as originally intended in the design. To estimate the impact on performance of such as cache, we have implemented a crude approximation to it: a system-wide memory flush mechanism, which clears and frees all nonpersistent data in the system once the memory use has exceeded a user-defined threshold. When running with multiple threads, the decision to flush is made independently for each thread, since each thread builds its own per-frame data structures. The non-persistent data that is freed during a flush includes the REYES-style grids, the grid-local BVHs, subdivision intermediaries, and the kd-tree. After a memory flush Razor simply continues the frame from where it left off, lazily regenerating the necessary data structures for the remaining rays. By reducing the memory usage to half of what is reported in Figure 8, we experience only a 5% – 11% reduction in performance for balanced quality runs and under a 3% degradation for high-quality runs. These results indicate that Razor's memory footprint can be significantly reduced without substantially affecting performance. Figure 16 shows how rendering time varies as a function of available memory.

### 6.9 Suitability for Future Hardware

Razor was explicitly designed to perform well on future many-core hardware, although the implementation described in this article is for 2006-era mainstream CPUs. A companion paper [Govindaraju et al. 2008] analyzes Razor's performance on a simulated many-core architecture and shows that its optimizations targeted at making efficient use of the caches on such an architecture are effective.

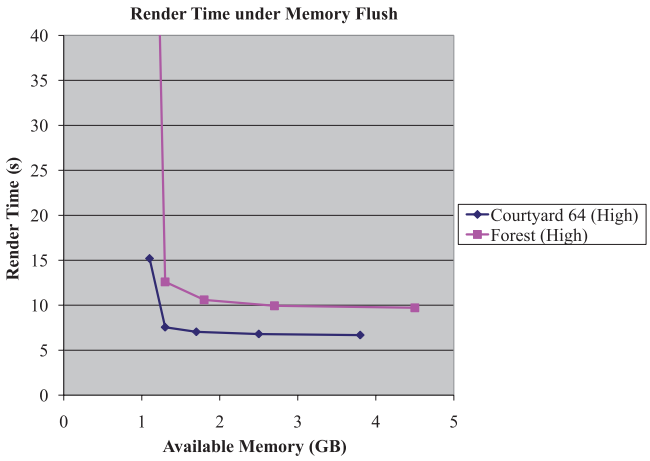


Fig. 16. Razor can limit maximum memory usage, regenerating data structures as needed. Rendering time is not greatly affected so long as the available memory is above a critical minimum threshold (about 1.3 GB for these scenes rendered using a single thread).

## 7. RELATED WORK

Our work builds on five major foundations: (1) the basic principles of ray tracing and distribution ray tracing [Appel 1968; Whitted 1980; Cook et al. 1984; Igehy 1999], summarized nicely in Pharr and Humpreys [2004]; (2) the REYES system for efficient, high-quality rendering using micropolygons [Cook et al. 1987]; (3) work on multiresolution ray tracing [Christensen et al. 2003, 2006] and related data structures [Wiley et al. 1997]; (4) work on efficient ray tracing acceleration structures [Havran and Bittner 2002; Reshetov et al. 2005; Wald et al. 2001]; and (5) work on subdivision surface representations [Catmull and Clark 1978; Halstead et al. 1993; Biermann et al. 2000; DeRose et al. 1998].

In this section we compare some cross-cutting aspects of our system design to previous approaches.

### 7.1 Caching Shading, Irradiance, and Radiance

Razor's mechanism for partially decoupling shading from visibility has two characteristics: First, it interpolates values computed at nearby points on the surface. Second, these values computed at nearby points are computed on demand and reused; that is, they are cached. Razor currently caches and interpolates just material properties (i.e., the BRDF), although the architecture would easily support caching of irradiance [Ward et al. 1988; Ward and Heckbert 1992] or a compact representation of radiance [Arikan et al. 2005].

Our caching and interpolation mechanism was inspired by REYES [Cook et al. 1987]. REYES assumes a single viewing-ray direction, and thus can evaluate, reuse, and interpolate the entire shading computation rather than just the BRDF. Both Razor and REYES reuse samples on a grid associated with the surface and use regular data interpolation. This explicit association of samples with a surface neighborhood has the potential to facilitate a large class of interesting shading optimizations. REYES explicitly generates and reuses results for just a single resolution of each surface, whereas Razor can cache results for several different resolutions of a single surface. In both systems, each cached/reused sample is associated with a particular resolution and may thus be prefiltered.

Irradiance caching [Ward et al. 1988; Ward and Heckbert 1992; Tabellion and Lamorlette 2004] and radiance caching [Arikan et al.

2005] just cache irradiance or radiance, rather than caching the results of shading computations. Photon mapping renderers [Wann Jensen 2001] behave similarly. All of these systems typically cache data as individual points in a global 3D data structure such as an octree or kd-tree, and thus do not explicitly associate cached points with a particular 2D surface. This has both the advantage and disadvantage that points from nearby surfaces or from nearby patches on the same surface may be accessed during retrieval, which is not done in our system. These systems also use scattered data interpolation rather than regular interpolation, and treat each sample as a true point rather than as a filtered sample associated with a particular surface resolution as Razor and REYES do.

## 7.2 Ray Tracing Dynamic Scenes

A variety of techniques have been proposed for ray tracing dynamic scenes (see Wald et al. [2007b] for a survey). We discuss the most relevant of these techniques and compare them to our approach.

For the special case of rigid objects, it is possible to prebuild an acceleration structure for each object and transform rays into the object coordinate system during ray tracing [Lext and Akenine-Möller 2001; Wald et al. 2003]. A top-level acceleration structure is still required; some systems use a bounding volume hierarchy, and others rebuild a complete top-level kd-tree every frame [Wald et al. 2003].

It is more difficult to efficiently support unstructured motion (also referred to as nonrigid motion). Several systems rely on building a complete acceleration structure for these objects (e.g., Wald et al. [2003] and Shevtsov et al. [2007]), but this approach performs unnecessary work for occluded objects. It is also possible to directly trace rays through the scene graph since it is a bounding volume hierarchy, which may be used directly as an acceleration structure [Rubin and Whitted 1980]. However, this approach is less efficient than using an explicitly constructed acceleration structure because the topology of the scene graph is not optimized for traversal efficiency.

The trade-offs between updating and rebuilding an acceleration structure are complex, and depend on a variety of factors. For example, as the number of rays traced increases, it becomes worthwhile to invest more time in building a high-quality acceleration structure. Similarly, the choice of acceleration structure (e.g., BVH versus kd-tree) can influence the decision as to whether it is better to update or rebuild; it is generally considered to be easier to update a BVH than a kd-tree. Many systems [Torres 1990; Chrysanthou and Slater 1992; Reinhard et al. 2000; Luque et al. 2005; Wald et al. 2007a; Lauterbach et al. 2006; Woop et al. 2006] dynamically update an acceleration structure rather than lazily rebuilding it each frame as we do. The potential penalty associated with incremental modification of the accleration structure can be reduced if is is known a priori that motion will be restricted to known motions, such as particular animations of deformable characters [Wald et al. 2007a]. Hybrid update/rebuild solutions are also possible; for example Lauterbach et al. [2006] update a BVH acceleration structure each frame but rebuild it when the quality degrades below a certain threshold. However, we reiterate our earlier point that on-demand rebuild is especially valuable when the acceleration structure is multiresolution, since it can be prohibitively costly to update or build an entire multiresolution acceleration structure at all possible levels of detail.

Several systems focus on building a complete acceleration structure extremely rapidly. Many of these systems (e.g., Wächter and Keller [2006]) achieve this speed by avoiding the use of a surface area heuristic. Generally speaking, the resulting acceleration structures are not as effective as those built with a surface area heuristic, resulting in increased trace times for irregular scenes and/or diverging secondary rays.

Along with our work [Hunt et al. 2006], there has been concurrent work for rapidly building a well-optimized kd-tree using approximations that avoid a full sort of geometry [Popov et al. 2006]. We use this class of technique in our system, however, in this article we augment it with several other techniques, including lazy build and build from the scene graph hierarchy, both of which are decribed in more detail in Hunt et al. [2007]. Similar techniques for rapidly building well-optimized acceleration structures have also been developed concurrently for SKD-trees [Havran et al. 2006], which can be thought of as a hybrid between a kd-tree and a conventional bounding volume hierarchy. Concurrently with our work, Wächter and Keller [2006] use a partially lazy build for a bounding interval hierarchy, which is a close relative of the SKD-tree.

Subsequent to our work, Hou et al. have shown that rebuilding a 4D BVH acceleration structure each frame permits efficient support for motion blur within a frame, as well as motion between frames [Hou et al. 2010]. Their accleration structure and build algorithm are specifically designed to provide tight bounds for motion blurred objects.

## 7.3 High-Quality Rendering Systems

Razor is designed to produce high quality images as efficiently as possible, which is a goal shared by many other systems. In addition to REYES [Cook et al. 1987], these other systems include the Maya Renderer [Sung et al. 1998] and Mental Ray [Driemeyer 2000]. Both the Maya Renderer and Mental Ray use a hybrid visibility algorithm: Rasterization is used for primary visibility while ray tracing is used for advanced effects such as reflection and refraction. This design was later adopted by Pixar's RenderMan [Christensen et al. 2003, 2006]. In contrast, Razor uses ray tracing for all visibilty queries. In this regard, Razor follows the philosophy of BMRT [Gritz and Hahn 1996].

There are other similarities among these systems worth noting. The micropolygon approach used by Razor was pioneered by REYES and also used by BMRT. The Maya renderer is able to lazily build its acceleration structure [Sung et al. 1998]. Figure 17 includes additional images produced by Razor along the lines of those produced by other high-quality renderering systems.

In work largely subsequent to Razor, Hanika et al. have built a multiresolution ray tracing system which attacks many of the same technical problems as Razor, but with batch rendering as the target workload [Hanika et al. 2010]. The Arnold batch renderer is 100% ray tracing based (using path tracing) and supports deferred loading of geometry [Fajardo 2010].

GPUs have recently become flexible enough to support production-quality rendering via completely software-defined rendering pipelines. Subsequent to our work, it has been shown that micropolygon rendering can be performed on a 2008 GPU at interactive frame rates [Zhou et al. 2009], and that high-quality micropolygon rendering with motion blur, defocus, and high sampling rates can be performed on a 2009 GPU with frames rates of roughly one minute per frame [Hou et al. 2010].

## 7.4 Interactive Distribution Ray Tracing

Subsequent to completion of our system, there has been an explosion of industry interest in interactive path ray tracing, including hardware built by Caustic Graphics [Caustic Graphics 2009] and NVIDIA's OptiX software for GPUs [Parker et al. 2010]. Since the primary focus of this article is on ray tracing algorithms rather than hardware platform comparisons, we do not attempt to compare Razor to these systems.

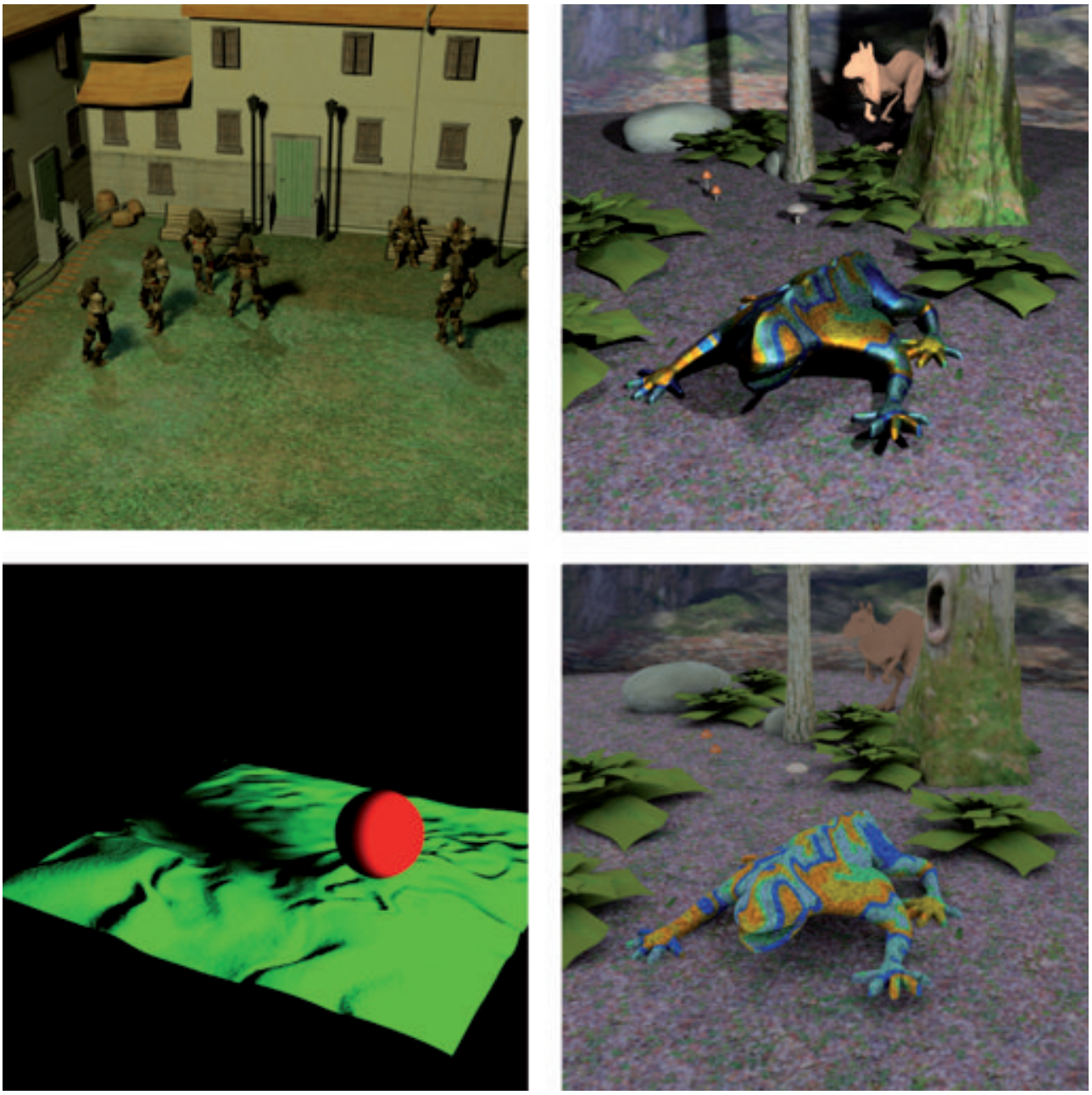

Fig. 17. Four images produced by Razor. Top-left: Courtyard64 balanced quality (36 rays/pixel, micropolygon area $\leq 4$ pixel$^2$). Top-right: Forest balanced quality. Lower-left: Micropolygon grids allow for displacement mapping, which is partially supported in our implementation. Lower-right: Forest ambient occlusion image, balanced quality with 8 occlusions rays instead of 8 shadow rays per primary ray, 15.0 seconds per frame. The models in these scenes appear courtesy of their respective owners. The character models in Courtyard64 are copyright Digital Extremes, all rights reserved. The character models in Courtyard64 use motion capture data from mocap.cs.cmu.edu (NSF EIA-0196217). The dragon model in Forest appears courtesy of Jeffery A. Williams and Intel Corporation. The killeroo model in Forest appears courtesy of Phil Dench, Martin Rezard, and headus (metamorphosis) Pty Ltd. The floor in Forest appears courtesy of Jonathan Dale. The background models in Forest appear courtesy of DAZ Productions.

## 8. LIMITATIONS

While Razor is designed to efficiently produce high-quality images, there are certain situations in which it will not perform well.

For nondistribution ray tracing (i.e., 1 primary ray, 1 shadow ray) of polygonal models, the system incurs high overhead to support multiresolution without receiving the benefits. In these cases it is clearly better to use simpler single resolution designs for fast ray tracing.

Razor's use of a lazy kd-tree build provides maximum benefit for scenes that include occluded or offscreen objects. In a scene where most or all of the geometry is visible, support for lazy kd-tree

construction may add extra overhead compared to eager construction. However, in a multiresolution system, lazy kd-tree construction still provides useful advantages because only the needed resolution(s) of visible geometry are inserted into the acceleration structure.

Razor can produce finer resolution geometry from a subdivision patch, but it cannot combine multiple subdivision patches to produce coarser geometry. Under circumstances where a subdivision patch is not coarse enough for a particular intersection test, Razor uses the geometry anyway, effectively losing the benefits of a multiresolution system. Because of this limitation, the input geometry places significant limits on the theoretical advantages provided by a multiresolution system. More powerful multiresolution geometry representations such as progressive meshes [Hoppe 1996] might help to address this problem, as would input geometry that is specifically designed for a system like this one. We note that most of our test scenes were designed for regular polygon rendering, not subdivision rendering.

Our experimental implementation currently lacks several features that the overall system architecture would easily support. Depth-of-field would be easy to add, just as it is in REYES. For diffuse surfaces, it would be simple to cast hemisphere-sampling secondary rays in phase one of shading, yielding a capability similar to irradiance caching. Displacement mapping is only partially implemented in our system; we have not implemented a solution to avoid cracks between patches when displacement mapping is enabled.

Our experimental system also lacks some useful features that would require more effort to support, including motion blur and more aggressive topology-modifying LOD. We would also like to do a more in-depth study to quantify the benefit of using our multiresolution scheme.

Our system performs shading on an object-space grid generated by tessellating subdivision surfaces. Thus, the density of shaded points is exactly the same as the density of tessellated polygon vertices. In follow-on work (outside of Razor), we have shown that it is possible to decouple shading from tessellation, so that polygon size is increased thereby reducing the cost of acceleration structure build and traversal [Burns et al. 2010]. This decoupling should also provide performance advantages in Razor.

## 9. FINDINGS

Razor is an ambitious design for a rendering system in the sense that it combines a large number of ideas (some new and some old) into a single system that is significantly different from previous rendering systems. Not surprisingly, some aspects of the design have been more successful than others. In this section, we attempt to summarize what we have learned from Razor and to assess which of aspects of the design have been most successful and which are less successful or in need of further investigation.

*Per frame rebuild of high-quality acceleration structures is practical*. Razor demonstrates that it is practical to rapidly rebuild a high-quality (SAH-optimized) acceleration structure each frame, thereby efficiently supporting arbitrary dynamic scenes. Prior to Razor, many researchers believed that this approach was infeasible, with most systems either restricting the types of motion in the scene or using lower-quality (non-SAH optimized) acceleration structures. Razor successfully combines several ideas to rapidly build a high-quality acceleration structure: (1) it builds its acceleration structure from a hierarchical scene graph rather than from a polygon "soup"; (2) it builds its acceleration structure on demand (lazily); (3) it uses a high-quality scan-based approximation to the standard sort-based SAH build; and (4) it uses specialized acceleration structures for meshes at the leaf nodes of the main acceleration structure. For some kinds of motion it is not necessary to rebuild the acceleration structure, so future single resolution systems are likely to hybridize technques similar to those used in Razor with techniques that update or modify an existing acceleration structure. Razor performs its on-demand build at a very fine granularity (a subpatch), but future systems may want to increase the granularity at which the on-demand build occurs to reduce bookeeping overhead and provide additional opportunities for parallelism.

*Multiresolution capabilities can be efficiently supported in an nearly interactive ray tracer, but the benefits are not yet compelling*. Razor demonstrates that it is feasible to fully integrate multiresolution capabilities into a nearly interactive ray tracing system and achieve reasonable performance without cracking artifacts. Previously, it was not clear that this was possible. In particular, Razor shows that it is possible to efficiently represent every region of space at multiple resolutions using a multiresolution kd-tree that is built lazily, while avoiding tunneling and popping artifacts. However, a multiresolution system pays a cost in complexity and computational overhead as compared to a single resolution system, and the benefits of Razor's current multiresolution implementation are less compelling than we had originally hoped. Follow-on work is needed in this area, in particular to explore mechanisms for providing coarser resolutions than those provided by a subdivision patch; to reduce problems associated with overtessellation (see what follows); and to explore the impact of using content designed explicitly for a multiresolution system. These problems are all known to be challenging in their own right, so it is not surprising that additional work remains. We continue to believe that multiresolution capabilities in some form will prove important for future interactive ray tracing systems that use large numbers of secondary rays. The approach recently taken by Hanika et al. seems especially worthy of further investigation [Hanika et al. 2010].

*Binary dicing leads to overtessellation*. When tessellation is performed using subdivision, and the number of quads in each dimension is restricted to be a power of two (as in Razor), the resulting micropolygons can be significantly smaller than required by the minimum shading rate. This problem is known from work on the REYES system (e.g., Apodaca and Gritz [2000], but we note that overtessellation is even harder to control in our system than in a REYES system because of the additional constraints on tessellation imposed by ray-directed LOD. It would be worthwhile to investigate alternative subdivision approaches (e.g., Loop and Schaefer [2008]) and alternative crack prevention approaches (e.g., Fisher et al. [2009]) although it is not immediately obvious if the most promising alternatives could be combined with Razor's interpolation of discrete LODs and shading on grids. One of our findings is that the crack-free surface tessellation problem is very complex in a high-performance system because there are so many interacting constraints. In particular, the need to parallelize at a fine granularity adds additional constraints that are not addressed by batch rendering systems that parallelize primarily by frame.

*Decoupling shading from hit points is achievable and promising but not yet compelling for low to moderate supersampling rates*. Razor shows that decoupling shading computations from ray hit points is feasible in an near-interactive ray tracer. However, the performance benefits were less than originally anticipated due to the previously mentioned overtessellation problem. Whether or not the potential advantages of the decoupled shading approach can be fully realized in practice remains an open question. This question is tied to the larger question of whether future interactive ray tracing systems should be hybridized with REYES or Z-buffer systems, since these systems do succeed in eliminating most redundant shading

computations for primary rays. Decoupling shading from visibility becomes more attractive in a system that supports motion blur and depth of field, since such systems tend to require higher sampling rates.

*Distribution ray tracing remains costly:* Distribution ray tracing and refinements such as path tracing are widely used in batch rendering systems. Our system and others have shown that with efficient traversal algorithms and data structures the per-ray cost of tracing coherent rays to an area light source can be made fairly low. However, the number of rays is still large, and to our surprise we observed in Razor that the cost of simply *creating* the rays becomes a significant fraction of the total runtime. For mass-market real-time applications such as games, it may be worth making selective trade-offs that sacrifice some of the generality and robustness of pure path tracing techniques in return for improved performance. For example, parts of the global illumination or visibility solution could be precomputed or some dimensions of integration could be ignored for certain effects. There are many examples of special-case approximations in today's rasterization-based games, and it could be fruitful to think about whether some of those approximations could be adapted to a ray tracing context.

## 10. CONCLUSION

We have presented a new software architecture for a dynamic scene distribution ray tracer. The architecture represents surfaces at multiple resolutions with per-ray LOD, integrates scene management with ray tracing, builds most of its per-frame data structures lazily, parallelizes acceleration structure build across worker threads, and partially decouples shading computations from visibility computations.

The system uses many best-known practices (e.g., packet tracing), while also incorporating completely new algorithms. The innovations include a technique for using continuous level-of-detail surfaces without popping or tunneling artifacts; and techniques for rapidly and lazily building a high-quality acceleration structure.

The experimental system that we have built is not production-ready and in its current form leaves some questions unanswered. However, our implementation demonstrates that the key system concepts work together in an nearly interactive rendering system, and allowed us to evaluate their strengths and weaknesses. We hope that these results will be useful to the designers of future interactive rendering systems.

### ACKNOWLEDGMENTS

The authors would like to thank all of our collaborators, especially those who helped with the design, implementation, and funding of this work. Denis Zorin provided useful advice about subdivision surfaces. Andrew Floren enhanced Razor to allow it to share a single acceleration structure across all cores if desired. Don Fussell and Okan Arikan provided numerous useful suggestions. The anonymous reviewers provided many useful suggestions in their detailed reviews. Jim Hurley and Bob Liang provided managenal support for this work.

### REFERENCES

AKENINE-MÖLLER, T. AND HAINES, E. 2002. *Real-Time Rendering* 2nd ed. AK Peters.

APODACA, A. A. AND GRITZ, L. 2000. *Advanced RenderMan: Creating CGI for Motion Pictures*. Morgan Kaufmann.

APPEL, A. 1968. Some techniques for shading machine renderings of solids. In *Proceedings of the AFIPS Spring Joint Computer Conference*. Vol. 32. 37–45.

AR, S., MONTAG, G., AND TAL, A. 2002. Deferred, self-organizing BSP trees. In *Proceedings of the Eurographics Conference*. 269–278.

ARIKAN, O., FORSYTH, D. A., AND O'BRIEN, J. F. 2005. Fast and detailed approximate global illumination by irradiance decomposition. In *Proceedings of the SIGGRAPH Conference*. 1108–1114.

ARVO, J. AND KIRK, D. 1987. Fast raytracing by ray classification. In *Proceedings of the SIGGRAPH Conference*. 55–64.

BENTHIN, C., BOULOS, S., LACEWELL, D., AND WALD, I. 2007. Packet-based ray tracing of catmull-clark subdivision surfaces. Tech. rep. UUSCI-2007-001, University of Utah.

BENTHIN, C., WALD, I., AND SLUSALLEK, P. 2004. Interactive ray tracing of free-form surfaces. In *Proceedings of the AFRIGRAPH Conference*. 99–106.

BIERMANN, H., LEVIN, A., AND ZORIN, D. 2000. Piecewise smooth subdivision surfaces with normal control. In *Proceedings of the SIGGRAPH Conference*. 113–120.

BIGLER, J., STEPHENS, A., AND PARKER, S. G. 2006. Design for parallel interactive ray tracing systems. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing*. 187–196.

BIKKER, J. 2007. Real-time ray tracing through the eyes of a game developer. In *Proceedings of the IEEE/EG Symposium on Interactive Ray Tracing*. Keynote address.

BOULOS, S., EDWARDS, D., LACEWELL, J. D., KNISS, J., KAUTZ, J., SHIRLEY, P., AND WALD, I. 2007. Packet-based Whitted and distribution ray tracing. In *Proceedings of the Graphics Interface Conference*. 177–184.

BURNS, C. A., FATAHALIAN, K., AND MARK, W. R. 2010. A lazy object-space shading architecture with decoupled sampling. In *Proceedings of the High Performance Graphics Conference*. 19–28.

CATMULL, E. AND CLARK, J. 1978. Recursively generated B-spline surfaces on arbitrary topological meshes. *Comput. Aid. Des. 10,* 6, 350–355.

CAUSTIC GRAPHICS. 2009. Introduction to CausticRT. Caustic Graphics website.

CHRISTENSEN, P. H., FONG, J., LAUR, D. M., AND BATALI, D. 2006. Ray tracing for the movie 'Cars'. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing*. 1–6.

CHRISTENSEN, P. H., LAUR, D. M., FONG, J., WOOTEN, W. L., AND BATALI, D. 2003. Ray differentials and multiresolution geometry caching for distribution ray tracing in complex scenes. In *Proceedings of the Eurographics Conference*. 543–552.

CHRYSANTHOU, Y. AND SLATER, M. 1992. Computing dynamic changes to BSP trees. In *Proceedings of the Eurographics Conference*. 321–332.

CLARK, J. H. 1979. A fast scan-line algorithm for rendering parametric surfaces. In *Proceedings of the 6th Annual Conference on Computer Graphics and Interactive Techniques*. ACM, New York, 174–.

COOK, R. L., CARPENTER, L., AND CATMULL, E. 1987. The REYES image rendering architecture. In *Proceedings of the SIGGRAPH Conference*. 95–102.

COOK, R. L., PORTER, T., AND CARPENTER, L. 1984. Distributed ray tracing. In *Proceedings of the SIGGRAPH Conference*. 137–145.

DEROSE, T., KASS, M., AND TRUONG, T. 1998. Subdivision surfaces in character animation. In *Proceedings of the SIGGRAPH Conference*. 85–94.

DJEU, P., HUNT, W., WANG, R., ELHASSAN, I., STOLL, G., AND MARK, W. R. 2007. Razor: An architecture for dynamic multiresolution ray tracing. Tech. rep. TR-07-52, Department of Computer Science, University of Texas at Austin.

DRIEMEYER, T. 2000. *Rendering With Mental Ray*. Springer.

FAJARDO, M. 2010. Ray tracing solution for film production rendering. In *ACM SIGGRAPH Talks: Global Illumination Across Industries*.

FISHER, M., FATAHALIAN, K., BOULOS, S., AKELEY, K., MARK, W. R., AND HANRAHAN, P. 2009. DiagSplit: Parallel, crack-free, adaptive tessellation for micropolygon rendering. In *Proceedings of the SIGGRAPH Asia Conference*. Article 150.

GOLDSMITH, J. AND SALMON, J. 1987. Automatic creation of object hierarchies for ray tracing. *IEEE Comput. Graph. Appl. 7,* 5, 14–20.

GOVINDARAJU, V., DJEU, P., SANKARALINGAM, K., VERNON, M., AND MARK, W. R. 2008. Toward a multicore architecture for real-time ray-tracing. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'08)*. 176–187.

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

GRITZ, L., STEIN, C., KULLA, C., AND CONTY, A. 2010. Open shading language. In *Proceedings of the ACM SIGGRAPH Talks*. ACM, New York, 33:1–33:1.

HALSTEAD, M., KASS, M., AND DEROSE, T. 1993. Efficient, fair interpolation using Catmull-Clark surfaces. In *Proceedings of the SIGGRAPH Conference*. 35–44.

HANIKA, J., KELLER, A., AND LENSCH, H. 2010. Two-level ray tracing with reordering for highly complex scenes. In *Proceedings of the Graphics Interface Conference*. 145–152.

HAVRAN, V. AND BITTNER, J. 2002. On improving kd-trees for ray shooting. In *Proceedings of the 10th International Conference in Central Europe on Computer Graphics, Visualization, and Computer Vision (WSCG02)*. 209–216.

HAVRAN, V., HERZOG, R., AND SEIDEL, H.-P. 2006. On the fast construction of spatial hierarchies for ray tracing. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing*. 71–80.

HECKBERT, P. AND HANRAHAN, P. 1984. Beam tracing polygonal objects. In *Proceedings of the SIGGRAPH'84 Conference*. 119–127.

HOPPE, H. 1996. Progressive meshes. In *Proceedings of the SIGGRAPH Conference*. 99–108.

HOPPE, H. 1998. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *Proceedings of the IEEE Visualization Conference*. 35–42.

HOU, Q., QIN, H., LI, W., GUO, B., AND ZHOU, K. 2010. Micropolygon ray tracing with defocus and motion blur. In *Proceedings of the SIGGRAPH Conference*.

HUNT, W., MARK, W. R., AND FUSSELL, D. S. 2007. Fast and lazy build of acceleration structures from scene hierarchies. In *Proceedings of the IEEE/EG Symposium on Interactive Ray Tracing*. 47–54.

HUNT, W., MARK, W. R., AND STOLL, G. 2006. Fast kd-tree construction with an adaptive error-bounded heuristic. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing*. 81–88.

IGEHY, H. 1999. Tracing ray differentials. In *Proceedings of the SIGGRAPH Conference*. 179–186.

KARRENBERG, R., RUBINSTEIN, D., SLUSALLEK, P., AND HACK, S. 2010. AnySL: efficient and portable shading for ray tracing. In *Proceedings of the Conference on High Performance Graphics*. 97–105.

LANE, J. M., CARPENTER, L. C., WHITTED, T., AND BLINN, J. F. 1980. Scan line methods for displaying parametrically defined surfaces. *Comm. ACM 23,* 1, 23–34.

LAUTERBACH, C., YOON, S.-E., TANG, M., AND MANOCHA, D. 2008. ReduceM: Interactive and memory efficient ray tracing of large models. In *Proceedings of the Eurographics Symposium on Rendering Conference*. 1313–1321.

LAUTERBACH, C., YOON, S.-E., TUFT, D., AND MANOCHA, D. 2006. RT-DEFORM: Interactive ray tracing of dynamic scenes using BVHs. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing*. 39–46.

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

LINDHOLM, E., NICKOLLS, J., OBERMAN, S., AND MONTRYM, J. 2008. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro 28,* 2, 39–55.

LOOP, C. AND SCHAEFER, S. 2008. Approximating Catmull-Clark subdivision surfaces with bicubic patches. *ACM Trans. Graph. 27,* 1, Article 8.

LUEBKE, D., REDDY, M., COHEN, J., VARSHNEY, A., WATSON, B., AND HUEBNER, R. 2003. *Level of Detail for 3D Graphics*. Morgan Kaufmann.

LUQUE, R. G., COMBA, J. L. D., AND FREITAS, C. M. D. S. 2005. Broad-phase collision detection using semi-adjusting BSP-trees. In *Proceedings of the Symposium on Interactive 3D Graphics and Games (I3D'05)*. 179–186.

MANSSON, E., MUNKBERG, J., AND AKENINE-MÖLLER, T. 2007. Deep coherent ray tracing. In *Proceedings of the IEEE/EG Symposium on Interactive Ray Tracing*. 79–85.

MORETON, H. 2001. Watertight tessellation using forward differencing. In *Proceedings of the SIGGRAPH/EUROGRAPHCS Conference on Graphics Hardware*. 25–32.

MÜLLER, K., TECHMANN, T., AND FELLNER, D. 2003. Adaptive ray tracing of subdivision surfaces. In *Proceedings of the Eurographics Conference*. 553–562.

NAVRÁTIL, P. AND MARK, W. R. 2006. An analysis of ray tracing bandwidth consumption. Tech. rep. TR-06-04, Department of Computer Science, University of Texas Austin.

OWENS, J. D., KHAILANY, B., TOWLES, B., AND DALLY, W. J. 2002. Comparing Reyes and OpenGL on a stream architecture. In *Proceedings of the SIGGRAPH/Eurographics Workshop on Graphics Hardware*. 47–56.

PARKER, S., MARTIN, W., SLOAN, P.-P. J., SHIRLEY, P., SMITS, B., AND HANSEN, C. 1999. Interactive ray tracing. In *Proceedings of the Symposium on Interactive 3D Graphics (I3D'99)*. 119–126.

PARKER, S. G., BIGLER, J., DIETRICH, A., FRIEDRICH, H., HOBEROCK, J., LUEBKE, D., MCALLISTER, D., MCGUIRE, M., MORLEY, K., ROBISON, A., AND STICH, M. 2010. Optix: A general purpose ray tracing engine. *ACM Trans. Graph. 29*, 66:1–66:13.

PETERSON, J. W. 1994. Tessellation of NURB surfaces. In *Graphics Gems IV*, P. S. Heckbert, Ed. Academic Press, 286–320.

PHARR, M. AND HANRAHAN, P. 1996. Geometry caching for ray-tracing displacement maps. In *Proceedings of the Eurographics Workshop on Rendering*. 31–40.

PHARR, M. AND HUMPREYS, G. 2004. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann.

PIPONI, D. 2004. Automatic differentiation, C++ templates, and photogrammetry. *J. Graph. GPU, Game Tools 9,* 4, 41–55.

PIXAR. 2005. *The RenderMan Interface ver 3.2.1*. https://renderman.pixar.com/products/rispec/rispec-pdf/RISpec3_2.pdf.

POPOV, S., GÜNTHER, J., SEIDEL, H.-P., AND SLUSALLEK, P. 2006. Experiences with streaming construction of SAH kd-trees. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing*. 89–94.

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

RESHETOV, A., SOUPIKOV, A., AND HURLEY, J. 2005. Multi-level ray tracing algorithm. In *Proceedings of the SIGGRAPH Conference*. 1176–1185.

RUBIN, S. M. AND WHITTED, T. 1980. A 3-dimensional representation for fast rendering of complex scenes. In *Proceedings of the SIGGRAPH Conference*. 110–116.

SCHLICK, C. 1994. An inexpensive BRDF model for physically-based rendering. In *Proceedings of the Eurographics Conference*. 233–246.

SEILER, L., CARMEAN, D., SPRANGLE, E., FORSYTH, T., ABRASH, M., DUBEY, P., JUNKINS, S., LAKE, A., SUGERMAN, J., CAVIN, R., ESPASA, R., GROCHOWSKI, E., JUAN, T., AND HANRAHAN, P. 2008. Larrabee: A many-core x86 architecture for visual computing. In *Proceedings of the SIGGRAPH Conference*. Article 18.

SHEVTSOV, M., SOUPIKOV, A., AND KAPUSTIN, A. 2007. Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes. In *Proceedings of the Eurographics Symposium on Rendering*. 395–404.

STOLL, G., MARK, W. R., DJEU, P., WANG, R., AND ELHASSAN, I. 2006. Razor: An architecture for dynamic multiresolution ray tracing. Tech. rep. TR-06-21, Department of Computer Science, University of Texas at Austin.

SUNG, K., CRAIGHEAD, J., WANG, C., BAKSHI, S., PEARCE, A., AND WOO, A. 1998. Design and implementation of the Maya renderer. In *Proceedings of the Pacific Graphics Conference*. 150–159.

SUYKENS, F. AND WILLEMS, Y. 2001. Path differentials and applications. In *Proceedings of the Eurographics Workshop on Rendering*. 257–268.

TABELLION, E. AND LAMORLETTE, A. 2004. An approximate global illumination system for computer generated films. In *Proceedings of the SIGGRAPH Conference*. 469–476.

TORRES, E. 1990. Optimization of the binary space partition algorithm (BSP) for the visualization of dynamic scenes. In *Proceedings of the Eurographics Conference*. 507–518.

WÄCHTER, C. AND KELLER, A. 2006. Instant ray tracing: The bounding interval hierarchy. In *Proceedings of the Eurographics Symposium on Rendering*. 139–149.

WALD, I., BENTHIN, C., AND SLUSALLEK, P. 2003. Distributed interactive ray tracing of dynamic scenes. In *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics (PVG'03)*. 77–85.

WALD, I., BOULOS, S., AND SHIRLEY, P. 2007a. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph. 26,* 1, Article 6.

WALD, I. AND HAVRAN, V. 2006. On building fast kd-trees for ray tracing, and on doing that in O(N log N). In *Proceedings of the IEEE Symposium on Interactive Ray Tracing*. 61–69.

WALD, I., IZE, T., KENSLER, A., KNOLL, A., AND PARKER, S. G. 2006. Ray tracing animated scenes using coherent grid traversal. In *Proceedings of the SIGGRAPH Conference*. 485–493.

WALD, I., MARK, W. R., GUNTHER, J., BOULOS, S., IZE, T., HUNT, W., PARKER, S. G., AND SHIRLEY, P. 2007b. State of the art in ray tracing animated scenes. In *Proceedings of the Eurographics State of the Art Reports*. 89–116.

WALD, I., SLUSALLEK, P., BENTHIN, C., AND WAGNER, M. 2001. Interactive rendering with coherent ray tracing. In *Proceedings of the Eurographics Conference*. 153–164.

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

WARD, G. J. AND HECKBERT, P. 1992. Irradiance gradients. In *Proceedings of the Eurographics Workshop on Rendering*. 85–98.

WARD, G. J., RUBINSTEIN, F. M., AND CLEAR, R. D. 1988. A ray tracing solution for diffuse interreflection. In *Proceedings of the SIGGRAPH Conference*. 85–92.

WHITTED, T. 1980. An improved illumination model for shaded display. *Comm. ACM 23,* 6, 343–349.

WILEY, C., CAMPBELL, III, A. T., SZYGENDA, S., FUSSELL, D., AND HUDSON, F. 1997. Multiresolution BSP trees applied to terrain, transparency, and general objects. In *Proceedings of the Graphics Interface Conference*. 88–96.

WOOP, S., MARMITT, G., AND SLUSALLEK, P. 2006. B-KD trees for hardware accelerated ray tracing of dynamic scenes. In *Proceedings of the SIGGRAPH/Eurographics Workshop on Graphics Hardware*.

WOOP, S., SCHMITTLER, J., AND SLUSALLEK, P. 2005. RPU: A programmable ray processing engine. In *Proceedings of the SIGGRAPH Conference*. 434–444.

YOON, S.-E., LAUTERBACH, C., AND MANOCHA, D. 2006. R-LODs: Fast LOD-based ray tracing of massive models. *Vis. Comput. 22,* 9, 772–784.

ZHOU, K., HOU, Q., REN, Z., GONG, M., SUN, X., AND GUO, B. 2009. RenderAnts: Interactive Reyes rendering on GPUs. In *Proceedings of the SIGGRAPH ASIA Conference*.

ZHOU, K., HOU, Q., WANG, R., AND GUO, B. 2008. Real-time kd-tree construction on graphics hardware. In *Proceedings of the SIGGRAPH Asia Conference*. Article 126.

Received July 2010; revised March 2011; accepted June 2011