Razor: An Architecture for Dynamic Multiresolution Ray Tracing

Peter Djeu*, Warren Hunt*, Rui Wang**,
Ikrima Elhassan*, Gordon Stoll***, William R. Mark*

University of Texas at Austin Department of Computer Sciences
Technical Report #07-52
Timestamp: January 24, 2007[†]

* University of Texas at Austin, ** University of Massachusetts at Amherst,
*** Intel Corporation

Abstract

Rendering systems organized around the ray tracing visibility algorithm provide a powerful and general tool for generating realistic images. These systems are being rapidly adopted for offline rendering tasks, and there is increasing interest in utilizing ray tracing for interactive rendering as well. Unfortunately, standard ray tracing systems suffer from several fundamental problems that limit their flexibility and performance, and until these issues are addressed ray tracing will have no hope of replacing Z-buffer systems for most interactive graphics applications.

To realize the full potential of ray tracing, it is necessary to use variants such as distribution ray tracing and path tracing that can compute compelling visual effects: soft shadows, glossy reflections, ambient occlusion, and many others. Unfortunately, current distribution ray tracing systems are fundamentally inefficient. They have high overhead for rendering large 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.

We describe *Razor*, a new software architecture for a distribution ray tracer that addresses these issues. Razor supports watertight multiresolution geometry using a novel interpolation technique and a multiresolution kD-tree acceleration structure built on-demand each frame from a tightly integrated application scene graph. This dramatically reduces the cost of supporting dynamic scenes and improves data access and computation patterns for secondary rays. The architecture also decouples shading computations from visibility computations using a two-phase shading scheme. It uses existing best-practice techniques including bundling rays into SIMD packets for efficient computation and memory access. We present an experimental system that implements these techniques at near-interactive frame rates. We present results from this system demonstrating the effectiveness of its software architecture and algorithms.

Outline of this document

Pages 3-15 of this document constitute the paper submitted to the SIGGRAPH 2007 conference on January 24, 2007. We have not made any changes to the document since that date. The paper was not accepted to SIGGRAPH but has been conditionally accepted (pending major revisions) to ACM Transactions on Graphics. This technical report may be cited as a draft of the ACM TOG

[†] The most significant part of this document, the main report (pp. 3-15), was written on January 24, 2007. This introduction (pp. 1-2) was written on September 27, 2007.

paper until such time as the revisions to the TOG paper have been completed. This technical report supersedes the content of UTCS TR-06-21.

Pages 1-2 of this document provide some updated information that did not appear in the original document, including some missing references to previous work and acknowledgements.

Additional previous work

BENTHIN, C., WALD, I., AND SLUSALLEK, P. Interactive ray tracing of free-form surfaces, 2004, Proceedings of Afrigraph 2004.

This paper describes a system for interactive ray tracing of cubic Bezier patches and Loop subdivision surfaces. It uses a fixed subdivision depth in contrast to Razor which subdivides adaptively. By using a fixed subdivision depth, Benthin et al.'s system avoids the need to address many of the issues with surface cracking and tunneling that Razor must address.

Acknowledgements

Don Fussell and Okan Arikan contributed numerous useful thoughts and suggestions that influenced this research. This work was supported by Intel Corporation and by NSF CAREER award #0546236. The authors would like to thank Bob Liang and Jim Hurley at Intel Corporation for their support. Part of this work occurred while the first three authors and the last author were working at Intel. The character models in the Courtyard are copyright 2003-2006 Digital Extremes, all rights reserved. The characters were animated using motion capture data from mocap.cs.cmu.edu. This database was created with funding from NSF EIA-0196217. The dragon model in the Forest was created by Jeffery A.Williams from Intel Corporation. The killeroo model in the Forest is courtesy of headus (metamorphosis), Phil Dench, and Martin Rezard. The textured Forest floor was made by Jonathan Dale. We would also like to thank DAZ Studio for permission to use the remaining background geometry and textures in the Forest.

Razor: An Architecture for Dynamic Multiresolution Ray Tracing

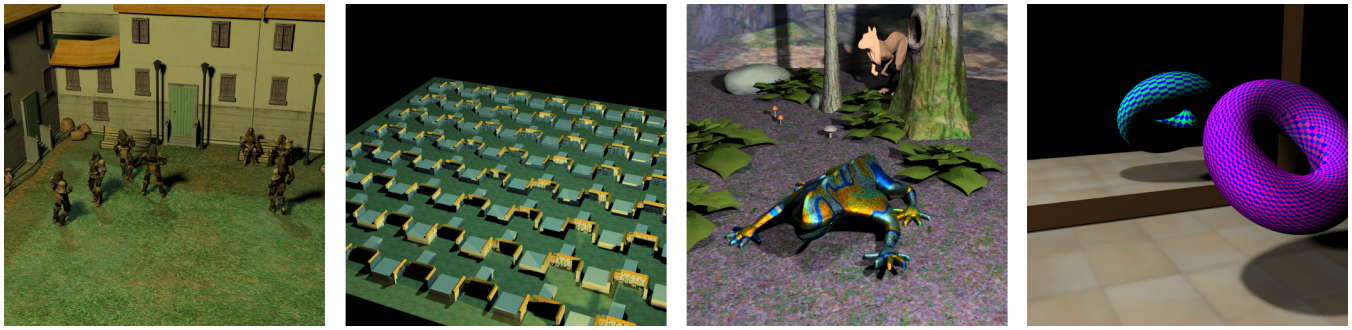

Figure 1: Images rendered by Razor at near-interactive frame rates. The first three scenes on the left have millions of visible micropolygons while the rightmost scene is a technical demo. All four scenes are animated with antialiasing and soft shadows enabled. The two scenes on the right are subdivision surfaces. Starting from the left, the render times are 3.83 sec, 3.77 sec, and 6.94 sec.

Abstract

Rendering systems organized around the ray tracing visibility algorithm provide a powerful and general tool for generating realistic images. These systems are being rapidly adopted for offline rendering tasks, and there is increasing interest in utilizing ray tracing for interactive rendering as well. Unfortunately, standard ray tracing systems suffer from several fundamental problems that limit their flexibility and performance, and until these issues are addressed ray tracing will have no hope of replacing Z-buffer systems for most interactive graphics applications.

To realize the full potential of ray tracing, it is necessary to use variants such as distribution ray tracing and path tracing that can compute compelling visual effects: soft shadows, glossy reflections, ambient occlusion, and many others. Unfortunately, current distribution ray tracing systems are fundamentally inefficient. They have high overhead for rendering large 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.

We describe *Razor*, a new software architecture for a distribution ray tracer that addresses these issues. Razor supports watertight multiresolution geometry using a novel interpolation technique and a multiresolution kD-tree acceleration structure built on-demand each frame from a tightly integrated application scene graph. This dramatically reduces the cost of supporting dynamic scenes and improves data access and computation patterns for secondary rays. The architecture also decouples shading computations from visibility computations using a two-phase shading scheme. It uses existing best-practice techniques including bundling rays into SIMD packets for efficient computation and memory access. We present an experimental system that implements these techniques at near-interactive frame rates. We present results from this system demonstrating the effectiveness of its software architecture and algorithms.

Keywords: ray tracing, rendering, level of detail, hierarchical build, lazy build, multiresolution, subdivision surfaces

1 Introduction

It has been a longstanding goal in computer graphics to synthesize images interactively that are realistic or that achieve a particular artistic look. Despite much progress over the past thirty years, current interactive graphics systems are still far from that goal.

It is becoming increasingly clear that the Z-buffer algorithm used in today's interactive graphics systems is likely to fundamentally limit progress towards photorealism. Within the next 5-10 years, we believe that the Z-buffer algorithm will need 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 [Christensen et al. 2003; Tabellion and Lamorlette 2004].

Recently developed interactive ray tracing systems [Parker et al. 1999; Woop et al. 2005; Reshetov et al. 2005; Wald et al. 2006] compellingly demonstrate that it is no longer possible to dismiss interactive ray tracing as computationally infeasible. Yet these existing systems have serious limitations that make them impractical for most mainstream interactive applications. Many of these systems perform poorly for large dynamic scenes, and most of them implement classical Whitted ray tracing or less, which for most 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 produced using distribution ray tracing [Cook et al. 1984]. These effects include soft shadows, glossy reflections, diffuse reflections, ambient occlusion, subsurface scattering, final gathering from photon maps and others. But current distribution ray tracing systems are fundamentally inefficient, particularly for dynamic scenes. Until these inefficiencies are resolved, ray tracing will not be able to replace Z-buffer rendering for most interactive applications.

In this paper, we explain why current distribution ray tracing systems are inefficient, and propose a new rendering-system architecture that reduces or eliminates the various inefficiencies. Our approach is explicitly designed to be appropriate for future interactive use. We also present an experimental system that implements our approach.

It is important to understand that our motivation for this work is to develop a better understanding of how to build *future* interactive rendering systems that support the *full set of functionality* that one would want in an interactive ray tracing system. This strategy contrasts with most other recent work on interactive ray tracing, which takes the opposite approach of either restricting functionality (e.g. dynamics) or image quality (e.g. resolution, visual effects, shading) so that the system can run at interactive rates today.

The most important new ideas in this paper are:

- The system architecture as a whole.
- A novel algorithm for representing and intersecting continuous level-of-detail surfaces in a ray tracer.
- A practical technique for lazily building a multiresolution kD-tree each frame from a tightly-integrated scene graph holding a dynamic scene. All major system data structures except the original scene graph are rebuilt every frame.
- An approach to surface shading that partially decouples shading computations from visibility computations. This approach extends the grid-based shading approach pioneered in the REYES system [Cook et al. 1987] to a ray tracing framework.

2 The Challenges

There are several challenges to building an efficient distribution ray tracing system:

Overall system performance:

Distribution ray tracing is computationally expensive, so systems must use a variety of best-practice techniques to achieve high performance at reasonable cost. First, geometry must be tessellated into triangles or quads before intersection testing (see e.g. [Christensen et al. 2003]). Second, the system must use an efficient acceleration structure such as a cost-optimized kD-tree [Havran and Bittner 2002] [1]. Third, the system must support aggregation of rays into packets [Wald et al. 2001]. By bundling rays into packets, cache hit rates are improved, branch mis-predict penalties are reduced, and use of register SIMD hardware such as SSE is improved. These practical considerations constrain other aspects of the system design.

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 non-rigid motion such as deformation (as is common in skinned characters used in computer games), then it is not even possible to use the common optimization of pre-building acceleration structures for individual objects.

If the scene is complex with many occlusions (such as an entire building with occupants), then it is unacceptably expensive to build the entire acceleration structure every frame. This problem is even more acute if we want to represent each object at multiple levels of detail; in this case the finer levels of detail will cause the system to run out of memory if we store tessellated geometry in the acceleration structure.

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, to sample incoming BRDF directions, 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.

Redundant shading computations:

Most ray tracers perform shading computations at each ray hit point. At high screen-space super-sampling rates, most of these shading computations are redundant. The situation is even worse for shaders that require arbitrary differential computations, since these shaders must be run three times at each hit point to compute discrete differentials [Gritz and Hahn 1996]. 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.

3 High-level solutions

Once the challenges above are understood, a set of potential solutions emerges. At the conceptual level these solution strategies are simple, but they each uncover more detailed challenges. In this section we explain these solution strategies and corresponding detailed challenges.

Use multiresolution surfaces to reduce the cost of tracing secondary distribution rays:

As [Christensen et al. 2003] and [Tabellion and Lamorlette 2004] have demonstrated, most secondary rays can be traced using a very coarse geometric representation of the scene. Mathematically the reason for this is that most secondary rays have large ray differentials [Igehy 1999] – i.e. they diverge strongly from each other as they progress away from their origins.

Efficient distribution ray tracing for large scenes requires a multiresolution scene representation. Without this capability, secondary rays will make effectively random accesses to the fully detailed scene database. In particular if the scene is dynamic, generating and accessing this data will be prohibitive. In addition to improving memory performance, and reducing the cost of tessellation and shading, these techniques potentially improve SIMD packet tracing efficiency for the same reasons.

Multiresolution and level-of-detail techniques are well understood for Z-buffer systems, but using them in a ray tracing system presents additional challenges. Most importantly, there is no longer a single reference point (the eye point) with which to set the resolution of each surface in the scene. Instead, each ray – including secondary rays – may request an LOD that is essentially unrelated to that requested by any other ray. An important implication of this situation is that any particular surface region may be accessed at multiple levels of detail by different rays. Under these conditions, the problem of guaranteeing that surfaces are watertight is much harder than it is in a Z-buffer system. This guarantee is important to insure that reflections, refractions, and shadows do not have crack artifacts. In future interactive systems these guarantees

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

must operate automatically; it will be unacceptable to rely on manual per-shot tuning of LOD parameters as is done in some offline ray tracing systems [Tabellion and Lamorlette 2004].

Adding multiresolution capability to a ray tracing system makes the design of the acceleration structure more complicated. Standard space-partitioning data structures represent each surface once at a single level of detail. To store each surface at multiple resolutions, the system must use multiple acceleration structures or be able to represent the same surface more than once in a single acceleration structure. Similarly, the ray traversal algorithm must be able to select the appropriate representation of a surface for intersection tests with the ray.

These challenges are more serious in a system that builds its acceleration structure on demand from dynamic geometry. In particular, solutions that require extensive preprocessing of geometry or that require global topological knowledge are unlikely to be acceptable.

Thus the challenges are: 1) How do we provide multiresolution surfaces that are watertight for ray tracing? 2) How should an acceleration structure store multiresolution surfaces so that the overall design is efficient for dynamic geometry?

Support dynamic scenes by lazily building the acceleration structure each frame:

The most straightforward approach to supporting arbitrary dynamic scenes is to dispense with the idea of pre-building and maintaining an acceleration structure, and instead build the acceleration structure from scratch each frame. This directly addresses the worst (and not unreasonable) case of unrestricted dynamic motion of all of the geometry in the scene. To avoid unnecessary work, the acceleration structure is built lazily, so that only the portions of it needed for a particular frame are built. At the end of the frame, the acceleration structure is discarded.

This conceptually simple idea presents three major challenges: First, how do we efficiently find the subset of the scene geometry that we need to insert into the acceleration structure in any particular frame? Second, how does a system like this interface with the rest of an interactive graphics application? Third, how do we keep the cost of lazy kD-tree construction low enough to do it every frame?

Decouple shading from visibility to eliminate redundant shading computations:

In a system that uses super-sampling the desired rate for visibility computations is usually higher than that for shading computations. The obvious solution to this mismatch is to decouple the visibility computations from the shading computations in some manner.

This is exactly the approach used by the REYES system [Cook et al. 1987] and by the multi-sampling technique used in modern Z-buffer graphics systems [Akenine-Moller and Haines 2002]. However, both of these systems are designed exclusively for eye rays. A ray tracer cannot pre-shade for a single viewpoint as the REYES system does. A ray tracer also cannot assume a regular pattern for all rays as the multi-sampling technique does.

Worse yet, the goal of decoupling visibility from shading interacts in difficult ways with the goal of using multiresolution surfaces. We now have a situation where shading may need to be performed at multiple resolutions for any particular surface. This is straightforward when visibility is coupled to shading, but less so once we decouple them. How do we solve this problem?

4 System architecture

It is clear that these various individual strategies for building an efficient distribution ray tracing system interact in complex ways. We will show how to combine these strategies so that they are compatible with each other and form a single integrated system. While some pieces of our system adapt well-known approaches, other portions 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.

4.1 Multiresolution ray/surface intersection

The problem of managing geometric level of detail is considerably more challenging in a ray tracer than it is in systems such as a Z-buffer that only use eye rays or their equivalent. In a Z-buffer system, the level of detail required to achieve a desired visual accuracy can be calculated consistently over the entire scene based on distance from the eye. In a ray tracing system, in general, the required level of detail is a complicated continuous function of location along a ray and this function is different for each ray. This raises the question of how to generate and manage surface tessellations at different levels of detail such that each ray can be intersected with the unique representation that it requires in a robust and efficient fashion.

In order to cache and re-use tessellations and associated vertex data or shading computations, they must be generated at discrete levels of detail. Unfortunately, in a ray tracer, naive discrete LOD approaches suffer from what we call the *tunneling* problem. 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 simple 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", and differs from other typical ray tracing artifacts. Unlike patch "cracking", holes can appear even in the middle of individual triangles. Unlike numerical precision problems, the holes have significant geometric extent. A key challenge in ray tracing multiresolution surfaces is to design a technique that avoids tunneling while satisfying other system constraints.

Our solution is to use a hybrid scheme, with discrete levels of detail but with continuous interpolation ("geomorphing") 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* (constructed lazily of course), targeted at the given edge length threshold. 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. The only other technique that we are aware of that handles multiresolution tessellations for arbitrary rays and avoids tunneling is that described in [Christensen et al. 2003; Christensen et al. 2006] (see section 6 for comparison).

Figure 3 illustrates this scheme. We refer to the adjacent discrete levels of detail as the *fine* mesh and the *coarse* mesh. The meshes in our system are generated by subdivision, and each triangle in the fine mesh maps to a portion of a single triangle in the coarse mesh.

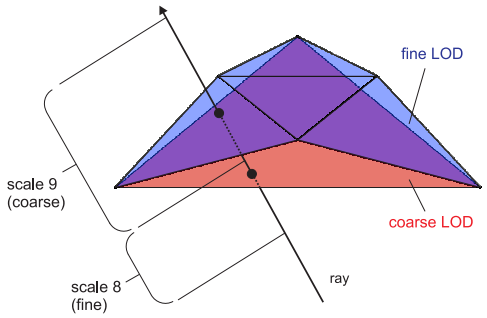

Figure 2: With discrete LODs, a ray may miss a surface completely if it changes the LOD that it is requesting at a point along the ray that is in between the surfaces produced by two discrete LODs.

The system is capable of corresponding each vertex of the finer triangle with a point on the corresponding triangle in the coarse mesh.

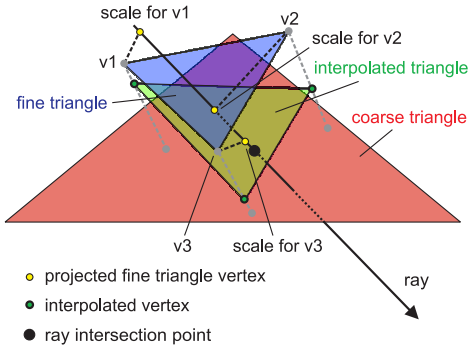

Figure 3: For each ray/triangle intersection test, the system generates a customized triangle that is specific to that ray. This customized triangle (shown in green) is generated by interpolating between triangles from two discrete levels of detail (shown in blue and in red). There is a separate interpolation weight for each vertex of the customized triangle. The weight for a vertex is determined by projecting the corresponding fine-triangle vertex (e.g. V1) onto the ray, and computing the weight from the scale value at that point on the ray (shown as yellow dots).

The system produces the in-between surface by interpolating between vertex positions in the fine mesh, and the corresponding points on the coarse surface. 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 vertex onto the ray, and computing the weight from a continuous scale function defined on the ray. This projection and interpolation step reduces the problem to normal ray/triangle intersection, and is thus very efficient (various direct solution alternatives involve multiple cubic equations). The interpolation weights in this scheme are associated with vertices, not triangles, so if both the fine and the coarse meshes are watertight, the interpolated mesh is as well. This relates to a key larger 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 surface that is "seen" by a ray. Note that this guarantee is for 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.). There is some commonality between this approach and eye-ray LOD techniques for terrain [Luebke et al. 2003].

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. The continuous scale function is calculated using ray differentials [Igehy 1999].

4.1.1 Computing scale values for rays

Each ray in our system has an associated scale that varies continuously with position along the ray. As explained earlier, this scale is used to decide which surface resolution to use for intersection testing. In this section we explain briefly how this scale is computed.

Our approach builds on the concepts of ray differentials [Igehy 1999] and path differentials [Suykens and Willems 2001], which we will summarize here. Each ray carries information with it sufficient to compute the origin and direction of its immediate neighbor. For example, the image-plane differentials provide the origin and direction of ray that is one pixel to the right and one pixel down on the image. These differentials are propagated through events such as reflections so that they continue to indicate the behavior of the neighbor ray at that point in the ray tree. Additional differentials are introduced each time the ray tree forks; for example, the system generates an additional pair of differentials for a ray when an area light source is sampled.

Each ray is best thought of as a beam with a finite cross-section. At any point on the ray, the ray differentials specify the area and geometry of the beam cross section. Most systems project this cross section onto a hit surface to compute a texture footprint.

Our system uses the differentials in a different manner, to compute a *single, isotropic* world-space scale value at each point on the ray. The scale is computed such that it is proportional to the width of the beam footprint. In the case of an anisotropic beam cross-section, the minimum width is used. By choosing the minimum width we guarantee that we tessellate and shade at a rate in each dimension equal to or greater than the desired rate.

Our system currently simplifies the problem of computing footprints from arbitrary path differentials by retaining the just most important differential pair along with the scale value used at the last intersection point. 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, 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 effects, it might be necessary to track more differentials.

4.1.2 Subdivision implementation

The geometry for each discrete scale is generated by adaptive tessellation of subdivision patches. We have implemented two subdivision systems which both work with Razor's multiresolution framework: the Planar system and the Catmull Clark system.

The Planar system treats each base patch in the input geometry as the control points for a bilinear patch. Planar patches are subdivided 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 Planar system provides an interesting upper bound to the performance of our geometry system because it incurs much less overhead than a more complicated subdivision scheme. In addition, the objects represented in the Planar system are conducive to our split-phase shading model; however, they do not develop geometric complexity as they are subdivided.

The Catmull-Clark system, which is the second geometry system, treats each mesh from the input geometry as a Catmull-Clark subdivision surface [Catmull and Clark 1978]. Unlike the Planar system, Catmull-Clark patches develop addtional geometric detail as they are subdivided. Our implementation has support for normals, textures, geometric creases, and texture creases [Halstead et al. 1993; Biermann et al. 2000]. Following the design used by Derose et al. [DeRose et al. 1998], the Catmull-Clark patches are represented as a topology face in regions of the mesh that contain irregularities and as a uniform bicubic B-spline in regions of the mesh that are regular [Peterson 1994]. The latter representation is well suited for Razor because it can be split into two smaller bicubics or tesselated at a rate independently determined in each of its two parametric directions. This support for anisotropic tesselation mitigates the potential for high overtesselation caused by anisotropy in the base patches.

As in any adaptive tessellation system, there is the possibility of cracks forming between adjacent patches. In the Planar system, patches have linear borders so no explicit crack fixing is needed, regardless of a difference in tesselation rate across the border. Catmull-Clark patches, on the other hand, border each other along cubic B-splines which leads to potential cracks when neighboring patches are tesselated at different rates. Razor restricts the size of grids in the Catmull-Clark system to powers-of-two to guarantee correspondence for the vertices of two neighboring patches. In conjunction with a crack fixing algorithm that removes inter-patch cracks regardless of whether the patches exist in the bicubic or topology world, Razor is able to move vertices around to "stitch" the Catmull-Clark surface back together without introducing any new geometry. The details of this algorithm are unfortunately outside the scope of this paper.

4.2 Dynamic Multiresolution Acceleration Structure

The system utilizes two primary data structures: a scene graph and a multi-scale kD-tree acceleration structure. The upper levels of the scene graph contain hierarchy nodes and base patches comprising the scene and are generally persistent frame to frame, although they may be updated according to animation or interaction as with any typical scene graph system. The lower parts of the scene graph and all acceleration data structures in the system are rebuilt from scratch every frame. The lower levels of the scene graph contain split and tesselated patches that are built out during the course of rendering a frame subdividing the original base patches. Hierarchical bounding volumes are maintained throughout this extended scene graph.

In order to support the interpolating intersection technique described earlier, we would conceptually like to build a separate kD-Tree for every pair of adjacent discrete levels. The geometric primitive at the leaf nodes in each such tree 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 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.

4.2.1 Merged kD-trees

Figure 4 illustrates our kD-tree. 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 multi-scale 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 multi-scale 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. If we set aside the laziness and use of hierarhcy for the building process for now, the algorithm for building the tree is as shown in Figure 5.

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 nearly identical SIMD packet traversal code and an eight-byte internal node record. Rays simply 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 discrete level pair (i.e. from step 6 above). In short, the addition of multi-resolution capabilities does not inhibit the use of current best practice approaches for fast traversal.

4.2.2 Fast Construction

We use a variety of synergistic algorithmic approaches in order to build our kD-tree structures quickly: lazy evaluation of scene graph nodes, use of the scene graph hierarchy for build acceleration and the fast scan approximation [Hunt et al. 2006] of the surface area heuristic [Havran and Bittner 2002]. A fourth technique we use is to build down to regular patches (igrids) instead of triangles. These patches use their own, more regular, acceleration structure.

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 2000] 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]). [Pharr and Hanrahan 1996] builds 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

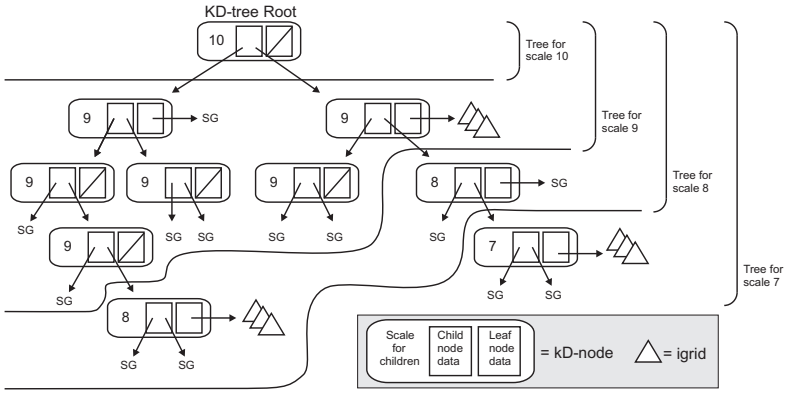

Figure 4: Multi-scale dynamic kD-tree. 'SG' designates a pointer into the scene graph.

```
1. Create a root node for the kD-tree with the
   scene bounding box and the scene graph root
   node.
2. Set the current node to be the root.
3. Set the current discrete LOD level to be the
   coarsest supported level.
4. Subdivide the geometry at the current node
   until it satisfies the current discrete LOD
   criteria.
5. Build out the kD-tree from this node until the
   tree termination criteria are satisfied.
6. Retain the current geometry (these nodes are
   effectively leaves for the current discrete
   LOD level).
7. Set the current discrete LOD level to the next
   finer level.
8. Goto 4.  (unless termination reached)
```

Figure 5: Pseudo-code for building the kd-tree without laziness

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 sub-patch 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 to build BSP trees for collision detection [Ar et al. 2002].

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. Since scene graph nodes may be used as proxies for large amounts of geometry, the kD-tree builder can be exposed to a significantly smaller set of 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 choose even the top level kD-tree splits and removes the "top heavy" property of 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.

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 doesn't need to be refined.

As shown in the results section, although the kD-trees split planes are chosen from a smaller set 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.

The use of the fast scan approximation kD-tree build algorithm provides a rapid baseline build which is much faster than a sorting build [Hunt et al. 2006]. This is particularly important if little or no hierarchy exists in a scene, or if much of the world is visible, reducing the effectiveness of the lazy build. This safety-net improves the performance robustness of the builder across a variety of scenes.

4.2.3 Low-Level Grid Intersection Structures

In addition to the fast/lazy kD-tree builder, we organize geometry into grids (small regular meshes) rather than individual triangles, and the system also performs lazy evaluation at this granularity. A kD-tree node that serves as a leaf node at a particular scale may have the associated geometry marked as "lazy". Such a node has a linked list of geometry (patches and sub-patches), but the final grid data structures have not been constructed yet. When such a node is intersected, all of the final vertex data is computed. In addition, a simple bounding volume hierarchy is constructed based on the internal structure of the tessellation. This low-level acceleration structure obviates the need to compute several levels of kD-tree splits at the bottom of the tree and takes advantage of the a-priori knowledge that triangles within a patch have known connectivity and are usually nearly co-planar. The grids are also important for our shading model, for enabling features such as displacement mapping, and they improve the performance of the subdivision systems.

4.2.4 A note on efficiency

This lazy kD-tree-building mechanism is extremely effective. As mentioned above, laziness is required in order to efficiently support multiresolution geometry. What is less obvious is the fact that multiresolution and hierarchical clustering 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" 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 (you always touch the root, obviously).

4.3 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 eye 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 (arbitrary differential calculations, displacement shading) 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 cheaper-to-evaluate but ill-behaved visibility function is super-sampled.

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.

Altogether, there are four sources of performance improvement in this shading system. First, redundant shading computations caused by visibility super-sampling 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 executing the shader three times for each hit point as is standard in ray tracers [Gritz and Hahn 1996]. Third, the grid structure of phase one shading makes it amenable to acceleration by SIMD mechanisms like x86 SSE. Grid-based shading also improves memory-access locality. Fourth, the scheme 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.

5 Results

We have evaluated our prototype system using several scenes, and various rendering configurations which are described in Figure 6.

5.1 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 Intel Xeon X5355 processors (2.66 GHz with 1333 MHz FSB) and 16 GB of DRAM. Note that most of our demos use only about 4GB of the DRAM.

Figure 6 summarizes performance for our scenes under various rendering configurations. Rendering times at 1024x1024 take significantly less time per pixel than the 512x512 images. This is primarily because the costs of building the acceleration structure are amortized over more rays for larger images, but also because for a given scene, rays become more coherent as image resolution increases.

These results show that Razor delivers near-interactive performance for scenes consisting of millions of visible micropolygons and hundreds of thousands of base patches, even when many secondary rays are cast.

Comparsion to Grid: We are not aware of any published performance results for dynamic-scene ray tracing systems with a combination of performance and functionality similar to Razor's, so is difficult to make precise performance comparisions with other systems. The closest comparable system is Wald'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 1024x1024 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.[2] 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 3.2 GHz Pentium 4's. For ray tracing code, one Xeon X5355 core is approximately equal to two 3.2 GHz 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. Razor's parallel scalability also seems to be better than that of the grid system (see [Ize et al. 2006], end of Section 6).

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

							
Title	Courtyard1	Courtyard64	Forest	Courtyard64	Forest	Courtyard64	Forest
Quality	Fast	Fast	Fast	Balanced	Balanced	High	High
Base patches	16,690	605,308	27,341	605,308	27,341	605,308	27,341
Tessellation Type	Planar	Planar	CC	Planar	CC	Planar	CC
Lights	2	2	2	2	2	2	2
Image Size	1024x1024	1024x1024	512x512	1024x1024	1024x1024	1024x1024	1024x1024
Primary Rays/pixel	1	1	1	4	4	8	8
Shadow Rays/light	1	1	1	4	4	4	4
Total Rays/pixel	3	3	3	36	36	72	72
Max Micropolygon Area	16 pixels	16 pixels	8 pixels	4 pixels	4 pixels	1 pixel	1 pixel
Micropolygons	1,062,454	1,349,978	669,517	4,481,824	4,300,698	16,129,880	15,815,587
Shaded Microvertices	496,867	517,194	386,378	1,721,281	2,300,821	6,072,569	7,899,393
Max Memory Usage	0.8GB	1.3GB	1.8GB	2.0GB	2.8GB	3.8GB	5.8GB
Render Time (this frame)	0.58s	1.24s	0.916s	3.83s	6.94s	9.26s	15.0s
Render Time (movie avg)	0.558s	1.26s	0.916s	3.85s	6.76s	10.3s	14.9s

Figure 6: Scenes used for evaluation and for the video. Unless specified otherwise, the specific frames illustrated by the thumbnails are used to report results in other tables. The rendering times reported in this table are all on our dual Xeon X5355 2.66 GHz machine (8 cores total). (*) The micropolygon and microvertex numbers are measured by rendering on a single core, since some duplication of micropolygons occurs in multicore rendering.

We believe this comparsion actually understates the performance potential of Razor. For rendering configurations with many secondary rays Razor spends most of its time in ray traversal, but Razor's traverser is not yet as well optimized as those in other systems. Most importantly, Razor does not yet use frustum or interval techniques [Reshetov et al. 2005], but we are confident that such techniques could be integrated successfully.

5.2 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. Figure 7 compares the performance of Razor to that of Mental Ray, a fast batch renderer integrated with the Maya 8.0 modelling package. Both rendering systems are configured for 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 in Mental Ray than ray tracing the primaries), and ray tracing for all secondary rays. For these comparisons we used a PC with a dual-core Pentium D 3.2 GHz processor, with hyperthreading disabled, and 4.0 GB of DRAM.

These experiments show that Razor is 3.6x to 7.3x 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 incorporates years of performance tuning.

Comparisions between systems that are as different as these are fraught with difficulties, but the key point is that the performance of our experimental system already exceeds the performance of the highly optimized rendering architectures used for batch rendering. We do not claim that Razor's rendering architecture should be adopted for batch rendering; rather, we believe that the requirements for interactive ray tracing systems are different from those of batch renderering, and that these results show that Razor's architecture is particularly appropriate for future interactive rendering systems.

Scene	Razor frame time	Mental Ray frame time	Razor speedup
Courtyard64	179s	1310s	731%
Forest	239s	856s	358%

Figure 7: Performance comparision between Razor and a batch renderer (Mental Ray) at high quality settings on a dual core Pentium D PC. Both renderers are configured for 1024x1024 images, 16x 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 planar subdivision in Razor and no subdivision in Mental Ray while the Forest scene uses Catmull-Clark subdivision in both renderers.

5.3 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. Nor-

mally 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 re-rendering the same frame without deleting the on-demand data structures. Build time is the difference between these two measurements.

Figure 8 shows that Razor's three build techniques for the kd-tree combine synergistically to reduce build times. We report results for a viewpoint with high-depth complexity and fewer visibible polygons as well as results for a viewpoint with low depth complexity and many visible polygons.

Lazy build skips portions of the scene graph that are not visible. Hierarchical build uses scene graph bounding boxes as aggregate stand-ins for the geometry they contain; eventually they are split to insure an adequate number of candidate split planes. Scan build uses an approach described in [Hunt 2006] to rapidly choose split planes. This table reports single-threaded Xeon X5355 build and trace times for two different frames in the Courtyard64 scene. To focus on kd-tree build time, the "fast" rendering settings are used, but with patch tesselation essentially disabled.

			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

Figure 8: Courtyard 64 frame 230 is the viewpoint used in the first teaser image. Courtyard 64 frame 460 is the viewpoint used in the second teaser image.

5.4 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 an LOD that is substantially coarser than that requested by primary rays hitting the same surface. This improves the coherence of shadow rays, which in turn increases packet occupancy and reduces the size of the memory working set.

Figure 9 illustrates a case where adaptive LOD makes a huge 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 non-adaptive 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 for non-adaptive 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 will have a huge advantage. Christensen et al [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.

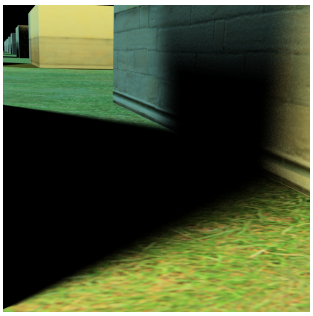

Figure 9: For this scene and viewpoint, per-ray adaptive LOD drastically improves performance compared to a uniform tessellation of all surfaces hit by rays. Adaptive LOD takes 92 seconds, uses 247 MB of memory, and has 39,081 VTune L2 cache-miss events. Uniform LOD takes 139 seconds, uses 3,694 MB of memory, and has 298,377 cache-miss events.

5.5 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 6 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. Although additional analysis would be useful to evaluate this particular design decision, we believe that these results already demonstrate that the Reyes micropolygon approach can be usefully combined with a ray tracing visibility engine. But for those who are skeptical of this approach, it is important to realize that Razor could easily be 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.

5.6 Parallel speedup

Figure 10 shows that for the balanced and high quality settings that are Razor's primary design point, Razor achieves parallel speedups between 6.21 and 7.17 on an eight-core machine. At the "fast" quality settings, speedup is somewhat lower because a large fraction of the total processing time at this quality setting is spent on building the upper levels of the kd-tree (this work is done redundantly on each core).

6 Related work

Our work builds on five major foundations: 1) The basic principles of ray tracing and distribution ray tracing [Appel ; Whitted 1980;

Courtyard64

Quality	1 Core	8 Cores	Speedup
High	59.3s	8.26s	7.17x
Balanced	27.0s	3.83s	7.05x
Fast	2.80s	0.58s	4.37x

Forest

Quality	1 Core	8 Cores	Speedup
High	103s	15.0s	6.87x
Balanced	43.1s	6.94s	6.21x
Fast	2.82s	0.916s	3.08x

Figure 10: Razor achives parallel speedup of up to 7.17x on an eight-core machine, with the best speedups at higher image quality settings.

Cook et al. 1984; Igehy 1999], summarized nicely in [Pharr and Humpreys 2004]; 2) The REYES system for efficient, high-quality rendering of eye rays [Cook et al. 1987]; 3) Work on multiresolution ray tracing [Christensen et al. 2003; Christensen et al. 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]; 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 various aspects of our system design to alternative approaches.

6.1 Multiresolution Ray Tracing

There is relatively little previous work on multiresolution and LOD techniques in ray tracing. The underlying technology to drive such a system was laid down relatively recently [Igehy 1999; Suykens and Willems 2001]. The most similar previous work is that described in [Christensen et al. 2006]. This system is targeted at production rendering, and in particular at avoiding virtual memory thrashing during ray tracing. The ray tracing system operates as an extension to a REYES scanline rendering system. The surface patches generated by REYES "splitting" provide a fixed partition of the scene. Tessellation resolution for intersection of a given ray with a patch from the partition is selected by comparing the ray's differentials to the size of the patch. Because the patch partition is fixed, and the tessellation resolution for a patch is fixed for a given ray, the tessellations that neighbor each other for a given ray can be stitched together. This technique avoids the tunneling problem, but the LOD selection for any given surface region and ray is dependent on the partition of the scene produced by REYES, which may not be of an appropriate granularity in the general case.

[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.

6.2 Caching schemes for 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], and we have already begun to experiment with this capability.

Our caching and interpolation mechanism was inspired by REYES [Cook et al. 1987]. REYES assumes a single viewing-ray direction, and thus can evaluate, cache, and interpolate the *entire* shading computation rather than just the BRDF. Both Razor and REYES cache 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 optimizations. REYES explicitly generates and caches results for just a single resolution of each surface, whereas Razor can cache results for several several different resolutions of a single surface. In both systems, each cached sample is associated with a particular resolution and may thus be pre-filtered.

Irradiance caching [Ward et al. 1988; Ward and Heckbert 1992; Tabellion and Lamorlette 2004] and radiance caching [Arikan et al. 2005] systems cache just irradiance or radiance, rather than caching the results of the full shading computation. Photon mapping systems [Wann Jensen 2001] behave similarly. All of these systems typically cache data as individual points in a global 3-D data structure such as an octree or kD-tree, and thus do not explicitly associate cached points with a particular 2-D 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 does.

6.3 Ray tracing dynamic scenes

A variety of techniques have been proposed for ray tracing dynamic scenes. We discuss these techniques in turn and compare them to our approach.

For the special case of rigid objects, it is possible to pre-build an acceleration structure for each object and transform rays into the object coordinate system during ray tracing [Lext and Akenine-Moller 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 non-rigid motion). Several systems rely on building a complete kD-tree for these objects [Wald et al. 2003], 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 a kD-tree for ray tracing acceleration.

Several systems [Torres 1990; Chrysanthou and Slater 1992; Reinhard et al. 2000; Luque et al. 2005; Wald et al. 2007; Lauterbach et al. 2006; Woop et al. 2006] dynamically update an acceleration structure rather than lazily rebuilding it each frame as we do. However, we believe that it is simpler and more efficient to lazily rebuild the tree, especially since it is difficult to guarantee that a kD-tree remains optimized for traversal cost [Havran and Bittner 2002] when it is incrementally modified. This restriction is less important if it is known a-priori that motion will be restricted to known motions, such as certain motions of deformable characters [Wald et al. 2007].

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

Two groups have concurrently developed similar techniques for rapidly building a well-optimized kd-tree by making approximations that avoid a full sort of geometry [Popov et al. 2006; Hunt et al. 2006]. We use this technique in our system, but augment it with several other techniques. Similar fast-build techniques for 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 partially lazy build for a bounding interval hierarchy, which is a close relative of the SKD-tree.

Also concurrently with our work, [Boulos et al. 2006] present an interactive distribution ray tracing system. They present improved sampling patterns for distribution ray tracing, showing that it is possible to achieve reasonable coherence for distribution secondary rays. They do not provide performance measurements that are sufficiently detailed to make a careful comparison with our system.

7 Discussion and Future Work

Razor's high-level system architecture and algorithms are *explicitly designed* for future interactive use, even though the performance of our current implementation is not quite interactive at our target image quality. Over the past year, we have improved Razor's performance by more than a factor of 50 through a combination of algorithmic improvements, parallelization, and use of newer hardware. Many of Razor's subsystems are still not fully tuned, and we expect to make substantial additional performance improvements over the next six months.

Our experimental implementation current lacks several features that the overall system architecture would easily support. Depth-of-field would be easy to add and virtually free, 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.

Our experimental system also lacks some useful features that would require more effort to support, including motion blur and more aggressive topology-modifying LOD.

Working within our system feels qualitatively different from working within any other ray tracing framework we've used. In particular, the notion that almost all operations are performed with respect to a specific spatial scale is very powerful. For example, most "epsilon" values within our system are set relative to the current scale, rather than to fixed global values.

8 Conclusion

We have presented a new software architecture for a dynamic-scene ray tracer. The architecture represents surfaces at multiple resolutions, integrates scene management with ray tracing, builds most of its per-frame data structures lazily, and partially decouples shading computations from visibility computations. The architecture is designed to efficiently support the needs of distribution ray tracing, including future interactive systems.

We believe that the goal of building an efficient distribution ray tracer for dynamic scenes leads almost inevitably to a design using principles similar to ours. Efficient support for distribution-sampled secondary rays requires multiresolution surfaces, and efficient support for multiresolution surfaces requires a lazily-built acceleration structure. Allowing shading operations to be performed on surface neighborhoods is in many respects more natural than performing them at intersection points.

The experimental system that we have built is not a product-quality system, and in its current form leaves some questions partially unanswered. However, our implementation clearly demonstrates the potential of our system architecture by successfully integrating a complex set of ideas into a single high-performance system.

We believe that many of the principles used in our system will be important to the design of future interactive rendering systems, and we hope that others in the graphics community can benefit from learning about our ideas and the results from our experimental system.

References

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

APPEL, A. Some techniques for shading machine renderings of solids. In *AFIPS 1968 spring joint computer conf.*, vol. 32, 37–45.

AR, S., MONTAG, G., AND TAL, A. 2002. Deferred, self-organizing bsp trees. In *Eurographics 2002*.

ARIKAN, O., FORSYTH, D. A., AND O'BRIEN, J. F. 2005. Fast and detailed approximate global illumination by irradiance decomposition. *ACM Trans. Graph. 24*, 3, 1108–1114.

ARVO, J., AND KIRK, D. 1987. Fast raytracing by ray classification. *SIGGRAPH 87 21*, 4 (July), 55–64.

BIERMANN, H., LEVIN, A., AND ZORIN, D. 2000. Piecewise smooth subdivision surfaces with normal control. In *Siggraph 2000, Computer Graphics Proceedings*, ACM Press / ACM SIGGRAPH / Addison Wesley Longman, K. Akeley, Ed., 113–120.

BOULOS, S., EDWARDS, D., LACEWELL, J. D., KNISS, J., KAUTZ, J., SHIRLEY, P., AND WALD, I. 2006. Interactive Distribution Ray Tracing. Tech. Rep. UUSCI-2006-022.

CATMULL, E., AND CLARK, J., 1978. Recursively generated B-spline surfaces on arbitrary topological meshes.

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 *Eurographics 2003*.

CHRISTENSEN, P. H., FONG, J., LAUR, D. M., AND BATALI, D. 2006. Ray tracing for the movie 'cars'.

CHRYSANTHOU, Y., AND SLATER, M. 1992. Computing dynamic changes to BSP trees. In *Proc. of Eurographics 1992*.

COOK, R. L., PORTER, T., AND CARPENTER, L. 1984. Distributed ray tracing. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA, 137–145.

COOK, R. L., CARPENTER, L., AND CATMULL, E. 1987. The REYES image rendering architecture. *SIGGRAPH 87 21*, 4 (July), 95–102.

DEROSE, T., KASS, M., AND TRUONG, T. 1998. Subdivision surfaces in character animation. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA, 85–94.

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

HALSTEAD, M., KASS, M., AND DEROSE, T. 1993. Efficient, fair interpolation using Catmull-Clark surfaces. *Computer Graphics 27*, Annual Conference Series, 35–44.

HAVRAN, V., AND BITTNER, J. 2002. On improving KD-trees for ray shooting. In *Proc. of WSCG 2002 Conference*.

HAVRAN, V., HERZOG, R., AND SEIDEL, H.-P. 2006. On the fast construction of spatial hierarchies for ray tracing.

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

IGEHY, H. 1999. Tracing ray differentials. In *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, 179–186.

IZE, T., WALD, I., ROBERTSON, C., AND PARKER, S. G. 2006. An Evaluation of Parallel Grid Construction for Ray Tracing Dynamic Scenes. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, 27–55.

LAUTERBACH, C., YOON, S.-E., TUFT, D., AND MANOCHA, D. 2006. Rt-deform: Interactive ray tracing of dynamic scenes using bvhs.

LEXT, J., AND AKENINE-MOLLER, T. 2001. Towards rapid reconstruction for animated ray tracing. In *Eurographics 2001*.

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 *Proc. of 2005 Conf. on Interactive 3D graphics*.

PARKER, S., MARTIN, W., SLOAN, P.-P. J., SHIRLEY, P., SMITS, B., AND HANSEN, C. 1999. Interactive ray tracing. In *Symposium on interactive 3D graphics*.

PETERSON, J. W. 1994. *Graphics Gems IV*. AP Professional (Academic Press), ch. Tessellation of NURB Surfaces, 286–320.

PHARR, M., AND HANRAHAN, P. 1996. Geometry caching for ray-tracing displacement maps. In *1996 Eurographics workshop on rendering*.

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

PIXAR. 2000. *The RenderMan interface version 3.2*, July.

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

REINHARD, E., SMITS, B., AND HANSEN, C. 2000. Dynamic acceleration structures for interactive ray tracing. In *Proceedings of the 11th Eurographics Workshop on Rendering*, Eurographics Association, 299–306.

RESHETOV, A., SOUPIKOV, A., AND HURLEY, J. 2005. Multi-level ray tracing algorithm. In *SIGGRAPH '05: Proceedings of the 32nd annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA.

RUBIN, S. M., AND WHITTED, T. 1980. A 3-dimensional representation for fast rendering of complex scenes. In *SIGGRAPH '80: Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA, 110–116.

SCHLICK, C. 1994. An inexpensive BRDF model for physically-based rendering. *Computer graphics forum 13*, 3, 233–246.

SUYKENS, F., AND WILLEMS, Y. 2001. Path differentials and applications. In *Rendering Techniques 2001: 12th Eurographics Workshop on Rendering*, 257–268.

TABELLION, E., AND LAMORLETTE, A. 2004. An approximate global illumination system for computer generated films. *ACM Transactions on Graphics 23*, 3, 469–476.

TORRES, E. 1990. Optimization of the binary space partition algorithm (BSP) for the visualization of dynamic scenes. In *Proc. of Eurographics 1990*.

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

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 2006 IEEE Symposium on Interactive Ray Tracing*, 61–69.

WALD, I., SLUSALLEK, P., BENTHIN, C., AND WAGNER, M. 2001. Interactive rendering with coherent ray tracing. In *Proc. of Eurographics 2001*.

WALD, I., BENTHIN, C., AND SLUSALLEK, P. 2003. Distributed interactive ray tracing of dynamic scenes. In *Proc. IEEE symp. on parallel and large-data visualization and graphics*.

WALD, I., IZE, T., KENSLER, A., KNOLL, A., AND PARKER, S. G. 2006. Ray tracing animated scenes using coherent grid traversal. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, ACM Press, New York, NY, USA, 485–493.

WALD, I., BOULOS, S., AND SHIRLEY, P. 2007. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics 1*, 26.

WANN JENSEN, H. 2001. *Realistic image synthesis using photon mapping*. AK Peters.

WARD, G. J., AND HECKBERT, P. 1992. irradiance gradients. In *Proc. 3rd 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 *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA, 85–92.

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

WILEY, C., A. T. CAMPBELL, I., SZYGENDA, S., FUSSELL, D., AND HUDSON, F. 1997. Multiresolution bsp trees applied to terrain, transparency, and general objects. In *Proceedings of the conference on Graphics interface '97*, Canadian Information Processing Society, Toronto, Ont., Canada, Canada, 88–96.

WOOP, S., SCHMITTLER, J., AND SLUSALLEK, P. 2005. RPU: a programmable ray processing engine. In *SIGGRAPH '05: Proceedings of the 32nd annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA.

WOOP, S., MARMITT, G., AND SLUSALLEK, P. 2006. B-KD Trees for Hardware Accelerated Ray Tracing of Dynamic Scenes. In *Proceedings of Graphics Hardware*.

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.

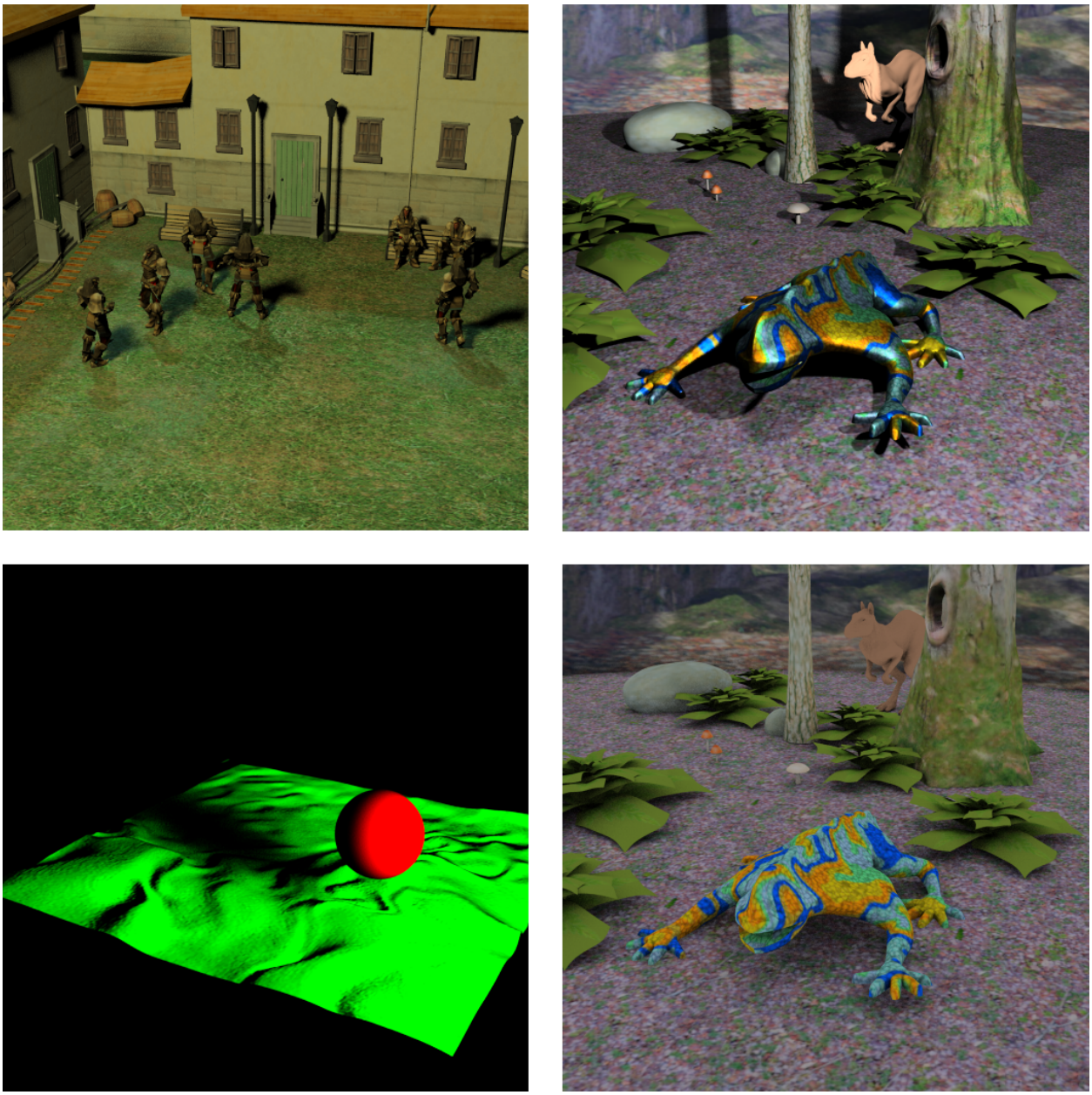

Figure 11: Four images produced by Razor. Top-left: Courtyard64 balanced quality. Top-right: Forest balanced quality. Lower-left: Our use of micropolygon grids enables displacement mapping. Lower-right: Forest ambient occlusion image, balanced quality with 8 occlusions rays instead of 8 shadow rays per primary ray, 14.76 seconds per frame.