Razor: An Architecture for Dynamic Multiresolution Ray Tracing

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

University of Texas at Austin Department of Computer Sciences
Technical Report #06-21
April 26, 2006

* = Intel Research, ** = University of Texas at Austin, *** = University of Virginia

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 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, although not in real time. We present results from this system demonstrating the effectiveness of its software architecture and algorithms.

Outline of this document

Pages 4-15 of this document constitute the paper submitted to the SIGGRAPH 2006 conference on January 25, 2006. We have not made any changes to the document since that date, other than to de-anonymize the author list and change the page header to indicate that it is now a technical report. The paper was not accepted to SIGGRAPH and so we expect to submit a future version of the work for publication, but we wanted to make this snapshot description of our work available to the research community now.

Pages 1-3 of this document provide some updated information that did not appear in the original document, including some missing references to previous work, a list of concurrent 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.

Concurrent work on ray tracing dynamic scenes

WALD, I., IZE, T., KENSLER, A., KNOLL, A., AND PARKER, S. Ray tracing animated scenes using coherent grid traversal. Technical Report, SCI Institute, University of Utah, No UUSCI-2005-014, 2006. (conditionally accepted to ACM SIGGRAPH 2006).

This paper uses a grid acceleration structure for ray tracing arbitrary dynamic scenes of moderate complexity. By adapting and extending packet-tracing and frustum culling techniques originally developed for kd-trees, the system achieves performance for primary rays and shadow rays that is reasonably close to that of a cost-optimized kd-tree. No results are reported for other types of secondary rays.

WALD, I., BOULOS, S., SHIRLEY, P. Ray tracing deformable scenes using dynamic bounding volume hierarchies. Technical Report, SCI Institute, University of Utah, No UUSCI-2005-014, 2006. (conditionally accepted to ACM Transactions on Graphics).

This paper uses a bounding-volume acceleration structure for ray tracing dynamic scenes, and more specifically deformable objects. To achieve high performance, the acceleration structure must be pre-built in a cost-optimized manner for the expected object deformations.

LAUTERBACH, C., YOON, S., TUFG, D., AND MANOCHA, D. RT-DEFORM: Interactive ray tracing of dynamic scenes using BVHs, 2006. Available online at http://gamma.cs.unc.edu/BVH.

This paper also directly uses a bounding-volume hierarchy as a ray tracing acceleration structure. The BVH is incrementally updated as objects deform. Since the quality of the BVH degrades with time due to the incremental updates, the system rebuilds the BVH from scratch once its efficiency drops below a pre-set threshold.

WALD, I. On building fast kd-trees for ray tracing, and on doing that in O(N log N). Technical Report, SCI Institute, University of Utah, No UUSCI-2006-009, 2006.

This paper presents a nice overview of algorithms and implementation details for constructing and traversing cost-optimized kd-tree acceleration structures. The paper also describes an algorithmic change to improve the efficiency of building cost-optimized kd-trees.

Concurrent work on ray tracing with multiple levels of detail

YOON, S. LAUTERBACH, C., AND MANOCHA, D. R-LODs: Fast LOD-based ray tracing of large models, University of North Carolina at Chapel Hill, Department of Computer Sciences Technical Report #TR06-009, 2006.

This paper describes a simple mechanism for supporting LOD in a ray tracer for static scenes. As in Razor, a single kd-tree is used to hold both original and simplified representations. The simplification used for LOD does not preserve topology, and the LOD transitions are

discrete. This approach has the advantages that the implementation is fast and that drastic simplification is possible, but the disadvantage that artifacts such as cracking and popping occur. The system provides some control over LOD artifacts by suppressing the LOD transition for a particular kd-tree node until the screen-space projection of the kd-node is smaller than a user-specific threshold measured in pixels.

Acknowledgements

Don Fussell participated in much of our early thinking about the system design and in particular about the importance of integrating the scene graph with the acceleration structure. Okan Arikan provided several useful suggestions and helped us compare Razor's approach to that of batch rendering systems. Jim Hurley, Bob Liang, and Stephen Junkins at Intel have strongly supported this research effort. This work was funded by Intel, Microsoft Research, and the University of Texas.

Razor: An Architecture for Dynamic Multiresolution Ray Tracing

Gordon Stoll* Intel Corporation
William R. Mark† UT Austin
Peter Djeu‡ UT Austin
Rui Wang§ U Virginia
Ikrima Elhassan¶ UT Austin

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 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, although not in real time. We present results from this system demonstrating the effectiveness of its software architecture and algorithms.

Keywords: ray tracing, level of detail, rendering

1 Introduction

It has been a longstanding goal in computer graphics to synthesize images interactively that are indistinguishable from those we observe in the real world. 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 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].

Recently developed interactive ray tracing systems [Parker et al. 1999; Woop et al. 2005; Reshetov et al. 2005] 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. In particular, these systems perform poorly for large dynamic scenes, and especially for scenes containing deformable objects such as human characters. Furthermore, when these systems are running at interactive rates on practical hardware they typically implement classical Whitted ray tracing, 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 in testbed form. Although this system is not parallelized and performance-tuned as would be necessary to achieve interactive performance, it demonstrates the viability of the core ideas in our new rendering architecture.

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) or simply running on impractical hardware (large clusters) 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-

*e-mail: gordon.stoll@intel.com
†e-mail:billmark@cs.utexas.edu
‡e-mail:djeu@cs.utexas.edu
§e-mail:rui.wang@gmail.com
¶e-mail:ikrima@mail.utexas.edu

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 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 severely degrade overall system performance, since it is common for a renderer's surface shading costs to be greater than than that of all other rendering costs combined.

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

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 (Figure 1).

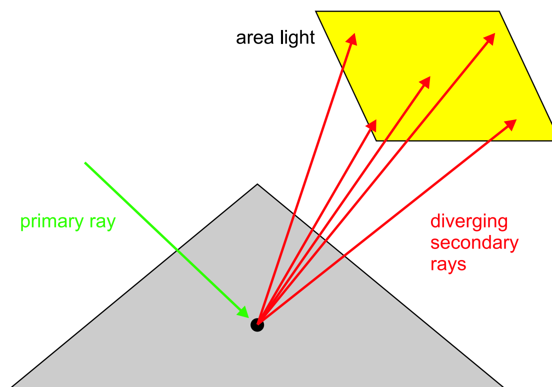

Figure 1: Distribution-sampled secondary rays diverge rapidly as they leave a surface. As Christensen *et al.* demonstrated, the ray tracing system must use a multiresolution surface representation to minimize the cost of tracing these secondary rays.

Thus, efficient distribution ray tracing for large scenes requires a multiresolution scene representation. Without this capability, the cost of generating and accessing the geometry touched by the secondary rays becomes prohibitive, particularly if this geometry is dynamic. 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. It is unclear how or whether the multiresolution ray tracing system described by [Christensen et al. 2003] solves this problem. In future interactive systems these guarantees 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 an acceleration structure, and instead build the acceleration structure each frame. 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

As summarized earlier, the problem of managing geometric level of detail [Luebke et al. 2003] 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. This difficulty is caused by the fact that it is no longer possible to choose a single level of detail for each object or surface region based on its distance from the eye point. We must switch from thinking about level-of-detail in an geometry-centric manner to thinking about it in a ray-centric manner. The level of detail required by each individual ray is a unique function of the location along that ray. Each surface region may be accessed at multiple levels of detail by different rays [Christensen et al. 2003]. 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 geometry that it requires in a robust and efficient fashion.

Our solution to this problem applies to adaptive surface tessellation, rather than more aggressive topology modifying LOD or non-surface primitives (volumes, point clouds, etc.) In other words, the question is reduced to one of how to robustly and efficiently intersect every ray in the system with surfaces tessellated to an appropriate level of detail. There are three important requirements that constrain the solution space. First, the technique should guarantee that there will be no cracks or pinholes in the surface. Second, the technique must be entirely local in nature. This second requirement is important because our system computes everything on demand in an unspecified order, and so we cannot rely on the availability of information about a large local neighborhood or about global surface topology. Third, the technique must allow the system to cache and reuse tessellations and shading computations at tessellation vertices.

In order to generate and cache tessellations, it seems necessary to discretize the levels of detail in the system. Conventional continuous-LOD tessellation would have to generate unique geometry for every ray and thus would not allow reuse of tessellations or associated shading computations.

Unfortunately, in a ray tracer, discrete level-of-detail approaches suffer from what we call the *tunneling* problem. Figure 2 illustrates this problem, in which a ray with a series of discrete scales passes through a surface without the intersection being detected, due to the abrupt transition from one discrete scale to another at a point along the ray. The result is cracking artifacts in the image. 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, in which tessellation and shading are performed at discrete levels of detail, but the system interpolates between adjacent discrete levels to produce a unique continuous surface for intersection testing against each ray. Figure 3 illustrates this scheme. We refer to the adjacent discrete levels

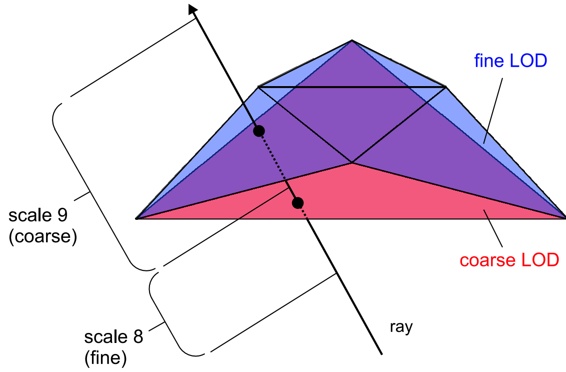

Figure 2: With discrete LODs, a ray may miss a surface completely if it changes the LOD that it is requesting at a point along the ray that is in between the surfaces produced by two discrete LODs.

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

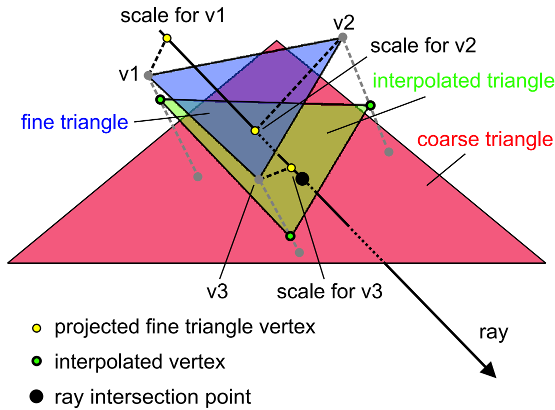

Figure 3: For each ray/triangle intersection test, the system generates a customized triangle that is specific to that ray. This customized triangle (shown in green) is generated by interpolating between triangles from two discrete levels of detail (shown in blue and in red). There is a separate interpolation 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 in yellow).

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). One minor alternative would be to use distance from the origin of the ray to the vertex rather than projection of the vertex onto the ray, which might have advantages when multiple rays share an origin (such as within a SIMD packet). 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. 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 remaining questions are how to compute the continuous scale function and how to manage transitions from using one pair of levels to using another. The continuous scale function is calculated using ray differentials [Igehy 1999], as described below. We manipulate this scale function so that the abrupt switch from using one pair of levels to using another pair occurs in a region of flat (constant) scale. These constant-scale regions are made (provably) large enough that any individual vertex will always be "seen" consistently by the ray. Space limitations prevent us from discussing this mechanism in detail, but we hope to report on it in a future publication that focuses on the LOD mechanism.

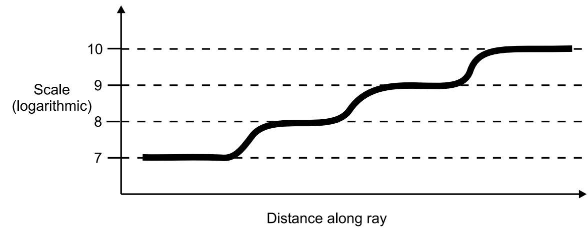

Figure 4: The system manipulates the scale values along the ray to insure that regions of varying scale are separated from each other by regions of constant scale. These regions of constant scale correspond exactly to one of the discrete levels of detail.

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 just the 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.

Before tracing rays, the system must partition each ray into a series of segments. Each segment represents the portion of the ray that can be intersected with a single pair of our discrete geometry levels. To determine each cut point between segments, the system must invert the equation that computes the scale value from the differentials as a function of position along the ray. In the general case this inversion requires solving a quadratic equation, although in common cases such as eye rays and area-light shadow rays the equation is linear. Our system uses division to solve the linear equation and otherwise uses the quadratic formula.

4.1.2 Subdivision implementation

The geometry for each discrete scale is generated by adaptive tessellation of subdivision patches. We currently use a very simple implementation of the Loop subdivision scheme for triangles [Loop 1987], with support for crease edges [Hoppe et al. 1994] and texture coordinates [DeRose et al. 1998]. Our implementation of subdivision operates on vertex grids formed from triangles pairs [Pulli and Segal 1996]. Vertex grids larger than a specified threshold are broken up into smaller grids to allow for adaptivity and lazy evaluation in both tessellation and shading. Currently, the target grid size is 5x5 vertices (32 triangles). Once subdivision has been applied twice to reach this 5x5 size, all further grids will be of this size (i.e. the vast majority of the grids in the system). The system computes bounds on the limit surface for each patch and sub-patch using the technique described by Kobbelt [Kobbelt 1998]. These bounds are used during the kD-tree construction.

As in any adaptive tessellation system, there is the possibility of cracks forming between adjacent patches. In our system, it is easiest to consider the patch cracking problem for the case of a single discrete scale applied to every patch on a surface. It turns out that solving the patch cracking problem for this single-scale case is sufficient to solve the problem for the general case as well, since our multiresolution geometry-interpolation scheme will work correctly if the geometry for each discrete scale is watertight. We use a simple local crack fixing technique [Owens et al. 2002] to insure that each discrete scale is watertight.

Our current subdivision system has serious shortcomings for our application in that it cannot actively target a specific edge length (our world space scale threshold) and it cannot actively control patch aspect ratios. It simply subdivides each patch into four pieces, roughly evenly in each parametric direction. We initially chose explicit Loop subdivision for its simplicity and to allow the system to be tested with existing triangle-mesh content. Using Catmull-Clark patches instead [Catmull and Clark 1978; DeRose et al. 1998] would facilitate independent and variable subdivision in both parametric directions, be a better match for modern animated content, and generally be a better long-term choice.

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 the original geometric primitives comprising the scene (subdivision surface patches) and are relatively persistent, updated from frame to frame according to animation or interaction as with any typical scene graph system. All other data in the system is rebuilt from scratch every frame. The lower levels of the scene graph are built out during the course of rendering a frame using the results of subdivision operations applied to the original patches. Hierarchical bounding volumes are maintained throughout this extended scene graph.

The multi-scale kD-tree acceleration structure must support the interpolating intersection technique described earlier. This technique breaks individual rays into segments, each of which is intersected against geometry generated from adjacent discrete levels of detail. Conceptually, we could 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 elaborate on this basic scheme in three ways: 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, 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 5 illustrates our kD-tree. The multiresolution capability is provided 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 of the building process for now, the algorithm for building the tree is as follows:

```
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
```

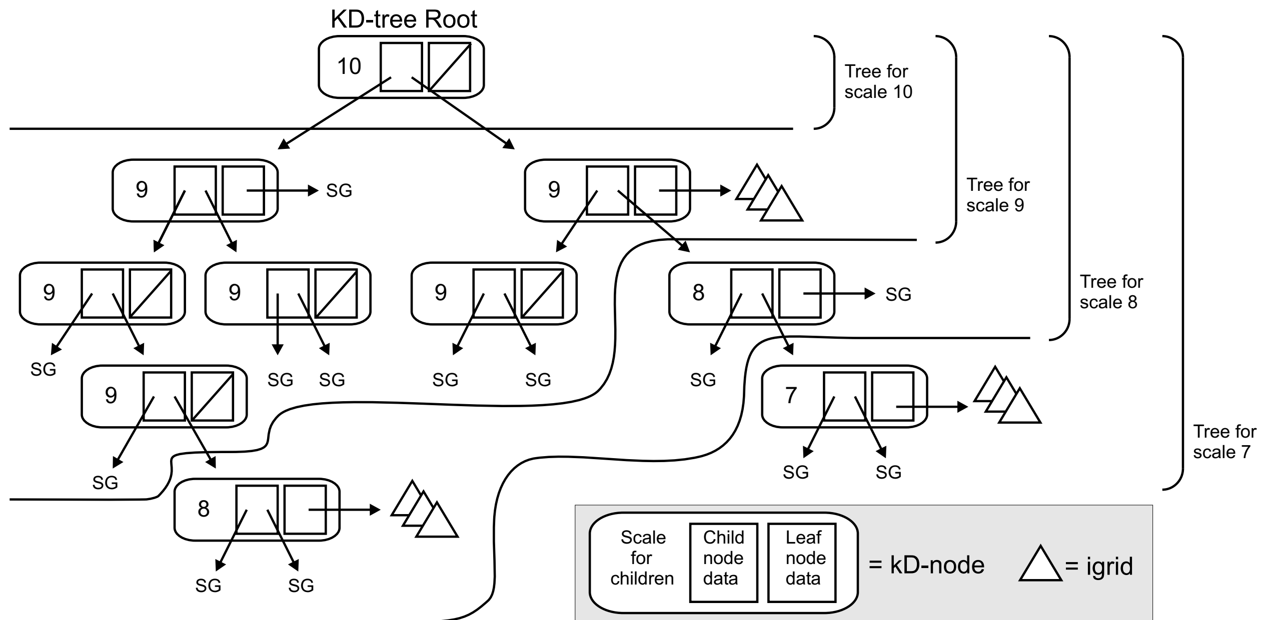

Figure 5: Multi-scale dynamic kD-tree. 'SG' designates a pointer into the scene graph.

```
   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.
```

As mentioned, we perform traversal for a single ray segment (and thus a single discrete level pair) at a time. Traversal is very nearly identical to normal kD-tree traversal, and thus is similarly efficient. 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). Note that we have not yet attempted to merge the traversal of the individual segments of the rays into a single continuous traversal operation. We simply break rays up into segments and then process each segment against the merged data structure in order along the rays. This is a significant inefficiency which we intend to address in the future.

Split planes in our tree are chosen using a simple surface area cost metric [Havran and Bittner 2002], using bounding boxes for split candidate determination (as opposed to more exact geometry).

4.2.2 Lazy Construction

The basic 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.

But 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 prohibitive.

Thus, our system builds its kD-tree 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. Conceptually 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. However, our current implementation only uses the last two cases. 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 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.

4.2.3 Low-Level Grid Intersection Structures

The geometry in the system is managed in grids (small regular meshes) rather than individual triangles, and the system also performs lazy evaluation at the granularity of a grid. 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, 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 (the "igrid" in figures 8 and 9) avoids computation of several levels of kD-tree splits

at the bottom of the tree and likely has better computational regularity and coherence properties as well. This data structure is traversed with a non-recursive fixed order ("flattened") traversal scheme as per [Smits 1998]. As described above in Section 4.1.2, the target grid size (and in fact the size of the vast majority of grids in the system)is 5x5 vertices or 32 triangles.

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 geometry, or some other form of hierarchical clustering, makes 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. The kD-tree is heavily "top-loaded" in computational cost, greatly impairing the benefits of lazy evaluation (you always touch the root, obviously).

Building a merged multiresolution tree as described above makes the cost of the root node split proportional to the amount of geometry at the coarsest supported level of detail, and similarly removes the top-loading of computational cost from the entire build process. Our results for tree building performance clearly demonstrate the advantages of this technique. We currently only utilize the natural "clustering" provided by repeatedly subdividing and breaking up patches. Further efficiency could be achieved by utilizing the clustering information inherent in a well-structured scene graph. We expect that the observed performance of kD-tree building for a well-structured scene graph using these techniques (including lazy evaluation) will be linear in the amount of geometry actually intersected by rays.

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. Variants on irradiance caching based on casting rays in phase one are certainly possible.

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] BRDF evaluation. It remains to be seen how well programmable shading can be adapted to this shading scheme.

5 Results

We have evaluated our prototype implementation using a courtyard scene with several animated skinned characters and two area lights, as shown in the accompanying video. Our rendering and timings were performed using single-threaded code running on a single 3.2GHz Intel Pentium 4 Processor with 2GBytes of memory.

The courtyard scene contains over 31,000 Loop subdivision patches, with 2,150 patches in each of the characters. Figure 6 shows a single frame from the animation, rendered at 512x512 resolution with 4x image-space super-sampling and 4x sampling of each area light from each of the four image-space samples. Figure 7 shows the elapsed time for rendering this frame with a range of tessellation rate settings. At the coarsest setting, the maximum on-screen area of a triangle is 37.5 pixels, and roughly 9,500 32-triangle grids are actually hit by rays and shaded. At the finest

Figure 6: Courtyard Scene

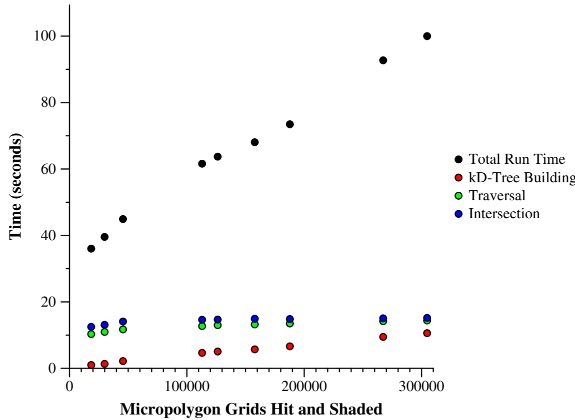

Figure 7: Performance on the Courtyard Scene. The scene was rendered at a range of tessellation rate settings, resulting in 9,500 to 300,000 visible micropolygon grids (each containing 32 triangles).

setting the maximum triangle area is one pixel, and over 300,000 grids are hit and shaded.

As can be seen in the figure, traversal and intersection consume from 10-15 seconds each, and are fairly insensitive to the amount of geometry. The time spent on calculating kD-tree splits is roughly linear with respect to the amount of geometry and is remarkably small, at roughly 10 seconds for over 300,000 grids (containing over nine million triangles). Because of the merged kD-tree and build algorithm, the maximum number of candidates considered for any single split (the split at the root) is only proportional to the number of grids at the coarsest scale, rather than the finest. Lazy building provides additional efficiency. As a result, on-the-fly conversion from the scene graph into a kD-tree is clearly not a bottleneck.

A large fraction of the run time is not being spent in any of these three fundamental ray tracing operations. The bulk of the rest of the time is being consumed by subdivision surface calculations, ray differential calculations, and MIPmap filtering, all of which are completely unoptimized in the prototype. The subdivision calculation and MIPmapping costs in particular are exacerbated by over-tessellation, addressed below.

The ambient occlusion sequence in the accompanying video was rendered with 6x image-space super-sampling and 26x hemisphere occlusion sampling at each image-space sample for a total of 162 rays cast per pixel. On a similar machine, these frames are roughly three times as expensive as the multiple-area-light frames, averaging 292 seconds each.

5.1 Over-Tessellation

The prototype implementation suffers from severe over-tessellation, producing approximately thirty times the number of micropolygons that would be expected in the ideal case (i.e. simply the screen area divided by the requested micropolygon area). There are four primary factors that cause this over-tessellation:

Non-Uniform Edge Lengths in a Single Grid Our simple Loop subdivision system cannot adapt to varying edge lengths within a single grid. Large variation in edge lengths can be caused by highly elongated triangles in the initial mesh, or by pairing a small triangle with a large triangle in the base grid. As a result, a single grid may have many edges that are much shorter than the maximum length edge which drives tessellation.

Subdivision Occurs in Discrete Steps Each iteration of our subdivision scheme reduces edge lengths by about a factor of two and triangle area by about a factor of four. This discretization is too coarse to precisely target a desired maximum edge length.

Shading-Grid Scales are Discrete For the two-level intersection scheme, a ray requires geometry grids for the two discrete scales that bracket the continuous scale that the ray actually wants. One of these geometry grids is tessellated at a finer scale than is strictly necessary for the continuous scale wanted by the ray.

Viewing Angle We use an isotropic world-space scale metric to control tessellation. Rays that strike surfaces at shallow angles may request geometry that is over-tessellated with respect to projected area.

The first two causes could be largely eliminated with a more sophisticated subdivision surface system capable of tessellating at varying rates in each parametric direction. Such a system could also ameliorate the third cause. If the subdivision system can consistently generate discrete scale levels which differ by a factor of two in area rather than four, then the system's discrete scale values can be set correspondingly, and the finer-level geometry needed by the ray will similarly be off by a factor between one and two rather than between one and four. We believe that this third cause can also be ameliorated by adjustments to the mechanism for breaking rays into segments. The fourth and final cause (viewing angle) is significantly more difficult to address, as the isotropic world-space scale metric is a basic component of the architecture.

We have measured the separate impact of each of these causes for the courtyard scene at a requested tessellation rate of one pixel per triangle. The breakdown is as follows: non-uniform edges = 3.47x, subdivision discretization = 2.26x, grid-scale discretization = 2.19x, and off-axis viewing = 1.84x. Combining these four measurements yields 31.6x over-tessellation, which closely matches our observed total deviation from the ideal. For this scene, at least, the breakdown of the various causes of over-tessellation indicates that

we can eliminate much of the tessellation problem with more implementation work – the first three causes combined account for 17.2x over-tessellation and can be largely eliminated.

5.2 Memory Consumption

Because of the over-tessellation the prototype implementation suffers from high memory consumption at the desired shading rates. As an interim measure, we use a workaround that takes advantage of the fact that the system computes most data structures on demand. We set a maximum memory consumption level and when that level is hit all data structures other than the persistent top-level scene graph are simply discarded. Rendering continues, with needed portions of the data structure being built or re-built on demand. This scheme is a primitive version of a caching scheme that we plan to implement later; the discard operation is equivalent to a complete cache flush.

For the courtyard scene rendering described above, at the finest tessellation setting, this memory flush operation occurs 11 times during the course of the frame. This cost is included in the times shown in figure 7; i.e. the total rendering time at the finest setting was approximately 100 seconds, and the aggregate kD-tree build time was approximately 10 seconds, even though the kD-tree and all tesselated and/or shaded geometry was simply thrown away eleven times during the course of the frame. It is difficult to precisely measure the impact of this flushing, but our best estimate is that the impact on total run time is less than 10%. This estimate is based on experiments where we varied both the tessellation settings and the memory flush thresholds.

The minimal performance impact of this crude mechanism indicates that a more sophisticated software caching scheme is likely to be very effective. A variant of this mechanism would also provide a simple but efficient coarse-grained parallelism technique. Rather than dealing with the synchronization issues inherent in the lazy construction of the data structures, each thread would simply build its own data structures.

6 Related work

Our work builds on five major foundations: 1) The basic principles of ray tracing and distribution ray tracing [Appel ; Whitted 1980; 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] 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 [Loop 1987; Hoppe et al. 1994; DeRose et al. 1998].

In this section we compare various aspects of our system design to alternative approaches.

6.1 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 plan to implement this capability in the near future.

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.

6.2 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] 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 re-build the tree, especially since it appears to be difficult to guarantee that a kD-tree remains optimized for traversal cost [Havran and Bittner 2002] when it is incrementally modified.

6.3 Interface between scene graph and ray tracer

Our system closely couples the scene graph to the ray tracing acceleration structure, as proposed by [Mark and Fussell 2005]. This system organization enables the system to lazily build the acceleration structure every frame. This organization is very different from the classical one in which the two data structures are separated by an API layer such as OpenGL [OpenGL Architectural Review Board 2003] (for Z-buffers) or OpenRT [Dietrich et al. 2003] (for ray tracers), and has implications for the design of ray tracing hardware which are discussed in [Mark and Fussell 2005].

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 multiple orders of magnitude away from interactive performance for our target imagery. As with any complex new system design, we expect a rapid ramp in performance as we address issues that we have identified in the first working implementation. As our performance results show, most of our execution time is spent in parts of our system that are unoptimized and whose execution time grows linearly with micropolygon count. By addressing issues with over-tessellation and by aggressively tuning all aspects of system performance, we believe that we can improve performance by 10-20x. An additional 5x or more in performance should be possible by parallelizing our system for multi-threaded, multi-core processors, even using the simple scheme mentioned above. Thus, we believe that our system will soon be 50-100x faster on commonplace desktop hardware without any fundamental changes to the system architecture.

Our experimental implementation current lacks several features that the overall system architecture would easily support. Displacement mapping and depth-of-field would be easy to add and virtually free, just as they are 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 and will likely prove to be more efficient in an optimized implementation.

The experimental system that we have built is not a product-quality system, and in its current form leaves some important questions unanswered. However, our implementation clearly illustrates the potential of our system architecture by successfully integrating a complex set of ideas into a working system with powerful new capabilities.

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.

9 Acknowledgments

Removed for review.

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.

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*.

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.

DIETRICH, A., WALD, I., BENTHIN, C., AND SLUSALLEK, P. 2003. The OpenRT application programming interface – towards a common API for interactive ray tracing.

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

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

HOPPE, H., DEROSE, T., DUCHAMP, T., HALSTEAD, M., JIN, H., MCDONALD, J., SCHWEITZER, J., AND STUETZLE, W. 1994. Piecewise smooth surface reconstruction. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA, 295–302.

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

KOBBELT, L. 1998. Tight bounding volumes for subdivision surfaces. In *PG '98: Proceedings of the 6th Pacific Conference on Computer Graphics and Applications*, IEEE Computer Society, Washington, DC, USA, 17.

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

LOOP, C. T., 1987. Smooth subdivision surfaces based on triangles.

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*.

MARK, W. R., AND FUSSELL, D. 2005. Real-time rendering systems in 2010. *UT-Austin Computer Sciences Technical Report TR-05-18* (May).

OPENGL ARCHITECTURAL REVIEW BOARD. 2003. OpenGL 1.5 specification.

OWENS, J. D., KHAILANY, B., TOWLES, B., AND DALLY, W. J. 2002. Comparing Reyes and OpenGL on a stream architecture. In *2002 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 *Symposium on interactive 3D graphics*.

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.

PULLI, K., AND SEGAL, M. 1996. Fast rendering of subdivision surfaces. In *Proc. of Eurographics Rendering Workshop*.

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

SMITS, B. 1998. Efficiency issues for ray tracing. *J. Graph. Tools 3*, 2, 1–14.

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*.

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*.

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.

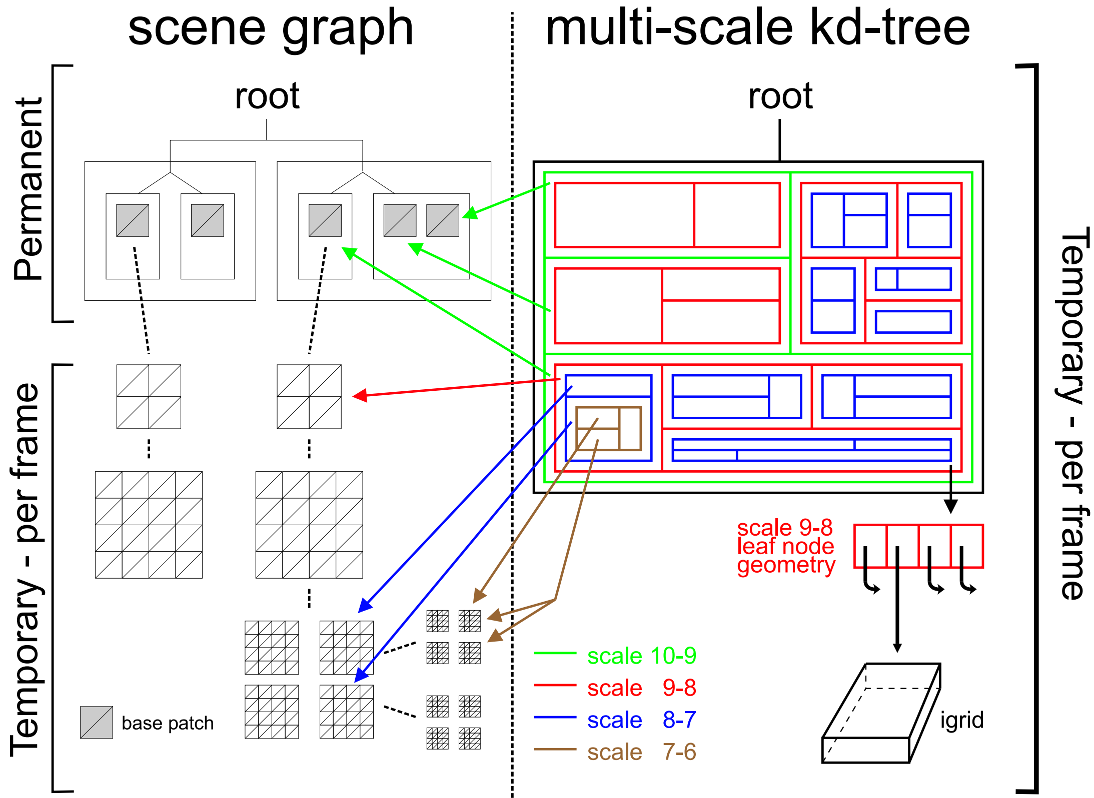

Figure 8: The key data structures in our system. The multi-scale kD-tree is closely coupled to the scene graph by “lazy” pointers. Regular (non-lazy) leaf nodes of the kD-tree point to a grid of geometry called an igrid.

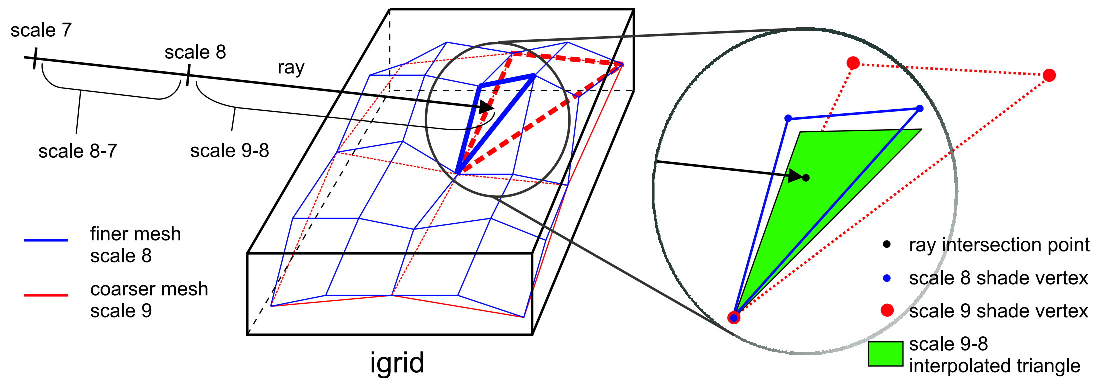

Figure 9: An igrid holds vertices for a pair of discrete scales. One set of vertices comes from a finer scale of geometry and the other set of vertices comes from a coarser scale of geometry. The igrid contains information associating each fine-scale vertex with a point on a coarse-scale triangle. The information in the igrid is used to generate interpolated triangles (shown in green) that are customized for particular rays. The igrid also contains (not pictured) a simple bounding volume acceleration structure based on the structure of the tessellation.