EUROGRAPHICS 2003 / P. Brunet and D. Fellner
(Guest Editors)

Volume 22 (*2003*), *Number 3*

Ray Differentials and Multiresolution Geometry Caching for Distribution Ray Tracing in Complex Scenes

Per H. Christensen David M. Laur Julian Fong Wayne L. Wooten Dana Batali

Pixar Animation Studios

Abstract

When rendering only directly visible objects, ray tracing a few levels of specular reflection from large, low-curvature surfaces, and ray tracing shadows from point-like light sources, the accessed geometry is coherent and a geometry cache performs well. But in many other cases, the accessed geometry is incoherent and a standard geometry cache performs poorly: ray tracing of specular reflection from highly curved surfaces, tracing rays that are many reflection levels deep, and distribution ray tracing for wide glossy reflection, global illumination, wide soft shadows, and ambient occlusion. Fortunately, less geometric accuracy is necessary in the incoherent cases. This observation can be formalized by looking at the ray differentials for different types of scattering: coherent rays have small differentials, while incoherent rays have large differentials. We utilize this observation to obtain efficient multiresolution caching of geometry and textures (including displacement maps) for classic and distribution ray tracing in complex scenes. We use an existing multiresolution caching scheme (originally developed for scanline rendering) for textures and displacement maps, and introduce a multiresolution geometry caching scheme for tessellated surfaces. The multiresolution geometry caching scheme makes it possible to efficiently render scenes that, if fully tessellated, would use 100 times more memory than the geometry cache size.

1. Introduction

Our goal is to render ray tracing and global illumination effects in very complex scenes — scenes that are so complex that a finely tessellated representation of all objects would take up orders of magnitude more memory than is available. Professional users of rendering programs for movie production and special effects routinely render scenes with tens of thousands of objects whose full tessellation result in hundreds of millions of polygons; these scenes contain hundreds of light sources, surfaces with many texture maps each, and shader-specified surface displacements in arbitrary directions. We want to extend the "tool box" of these users to include ray tracing and global illumination without limiting scene complexity or shader generality.

This work is driven by current production demand for ray traced shadows and reflections, ambient occlusion, and color bleeding. Even though shadow maps, reflection maps, and "bounce lights" are appropriate in many cases, there are plenty of other cases where ray tracing and global illumination are the most cost-effective ways of obtaining a given effect. For example, due to the fixed resolution of shadow maps, they are not suitable for computing tiny, sharp shadows in large scenes. Reflection maps cannot deal with realistic self-interreflections, and bounce lights require a lot of painstaking trial and error to emulate color bleeding.

Our goals are very similar to those of the Toro[21] and Kilauea[14] renderers. In Toro, rays are reordered to increase the coherency of geometry cache accesses. This reordering makes it possible to render scenes that are too large to fit in memory, but unfortunately introduces shader limitations. We are able to render even more complex scenes than Toro, despite using a smaller geometry cache, and we do not reorder rays. Kilauea is a massively parallel renderer that uses a cluster of PCs to store the fine tessellations of all objects in the scene. We are able to render equally complex scenes on a single PC.

Here we focus on efficient distribution ray tracing in complex scenes. Distribution ray tracing is used for Radiance-style global illumination[36], final gathering of photon maps[12], one-bounce global illumination, and ambient occlusion[16, 38].

2. Related work

Our method builds on prior work, particularly in geometric coherency and simplification, ray tracing of complex scenes, ray differentials, and texture caching.

2.1. Geometric coherency

Scanline rendering methods (such as the REYES algorithm[2, 5]) can handle very complex scenes, but can only compute local illumination effects. These methods handle geometric complexity by rendering one image tile a time, and only tessellating the objects visible in that tile. This maximizes geometry coherency and minimizes the number of tessellated surfaces that need to be kept in memory.

It is fairly straightforward to extend scanline rendering with ray tracing[7] as long as the rays intersect geometry in a coherent fashion. We call such rays *coherent rays*. Specular reflection and refraction rays from flat or slightly curved surfaces are usually coherent; shadow rays from point lights, spot lights, directional lights, and small area lights are also coherent. With this constraint, it is possible to ray trace complex scenes: only the directly visible objects and the reflected, refracted, and shadow-casting objects need to be kept in tessellated form at any given time. This coherency has been exploited by Green and Paddon[8] for geometry caching on a multiprocessor, Pharr and Hanrahan[20] for caching of displacement mapped surfaces, and Wald et al.[33] by tracing four coherent rays at a time.

But if the scene contains surfaces with high curvature or high-frequency bumps or displacements, reflection and refraction rays will go in every-which direction: these rays are incoherent and a geometry cache of limited size will thrash.

Pharr et al.[21] noted that for general path tracing[13], there is much less geometric coherency than for classic ray tracing. They suggested overcoming this obstacle by reordering the rays, as implemented in their Toro renderer. Unshot rays are inserted into a pool of rays. The image contribution of each ray is computed before the ray is inserted in the pool, and this weight (and the ultimate pixel position of the ray color) is stored with the pending ray. When sufficiently many rays are waiting to be intersection tested against an object, the geometry is read in, tessellated if not already in tessellated form, and inserted into the cache. This way, they can render scenes that are ten times larger than their geometry cache. The drawback of ray reordering is that it relies on being able to precompute the contribution of a ray before it is traced. This is fine for shooting a fixed number of rays from a shader with a linear BRDF. But it makes adaptive sampling impossible, and it does not work with the "creative" shaders that are often desirable in production. Consider for example a surface that should be red if more than half of the reflection rays hit a certain object. In such cases, there is no way we can know a priori the contribution of each ray.

2.2. Geometric simplification for rendering

A common method to speed up rendering is to simplify geometry that only covers a small part of the image. Level-of-detail for rendering is described by Apodaca and Gritz[2].

Rushmeier et al.[23] used a coarse geometry representation for computing an approximate radiosity solution. Clusters of complex geometry were substituted by boxes with similar reflective and transmissive properties. Then, during rendering, rays for computing diffuse interreflection were intersected with the original geometry near the ray origins and with the boxes further away. A user-defined distance threshold was used to switch between the two representations. Our approach is based on the same premise: rays for computation of some types of reflection need less accuracy than other rays. However, we use distribution ray tracing[6] while they used path tracing, and we use photon maps[12] for global illumination instead of radiosity. We also use our method for other purposes than computation of diffuse global illumination, and our use of ray differentials means that we have a better way of choosing the appropriate representation.

Smits et al.[28] and Christensen et al.[4] clustered geometry for efficient light transport between distant groups of objects. Instead of computing light transport between all pairs of objects, the far-field radiance of one cluster of objects is computed, the light is transported to the other cluster, and then "pushed down" to its individual surfaces.

2.3. Ray tracing without tessellation

"Conventional wisdom" says that it is more efficient to ray trace tessellated surfaces than their high-level representation. But there has been significant recent work on speeding up direct ray tracing without tessellation[15, 17, 24, 29]. A distinct advantage of these approaches is that only the high-level description of the objects (for example the control points of NURBS patches, or top-level subdivision meshes) and a spatial acceleration data structure need to fit in memory. So these methods seem ideal for ray tracing of complex scenes, at least from a memory usage standpoint. Unfortunately, according to our experiments, these methods are still significantly slower than ray tracing tessellated surfaces — as long as the tessellation fits in memory.

Another reason that makes tessellation desirable is that it makes ray tracing of surfaces with arbitrary displacements simple[20]. The method of Smits et al.[29] computes ray intersections with very complex displaced surfaces (without explicit tessellation), but is restricted to displacements along the surface normal and requires repeated evaluations of the displacement shader. The more complex the displacement shader is, the more advantageous tessellation is.

Given these constraints, we remain convinced that for efficient ray tracing, it is still necessary to tessellate surfaces. Our focus here is on a demand-driven caching technique which minimizes the storage cost of the tessellation.

2.4. Parallel ray tracing

The IRT ray tracer by Parker et al.[18] uses many processors on a shared memory computer to obtain interactive frame rates. For best performance, scenes must fit in the 4 MB on-chip cache on each processor. The RTRT system by Wald et al.[32] uses a cluster of PCs for interactive ray tracing of complex scenes (up to 50 million triangles). Ray coherency ensures that each PC only needs parts of the scene. Wald et al. also used RTRT to render indirect illumination in simpler scenes[31]. They shot 20–25 shadow rays pr. pixel to virtual point lights generated by random walks (similar to "instant radiosity"), and averaged indirect illumination between neighbor pixels to reduce noise. Although impressive, these interactive systems only have simple shaders and the images are too aliased for movie production.

The Kilauea renderer[14] handles complex scenes by dividing the objects between a cluster of PCs. Each surface is tessellated and assigned to a processor. Packets of rays to be tested for intersection are communicated between the processors. After all rays have been shot by a shader, the shader is inserted into a pool of suspended shaders. When all those rays have been traced and shaded, the shader is taken out of the pool and its computation continues.

We are not currently focused on a parallel implementation; we are more interested in efficient rendering of complex scenes on a single processor. That is, we want to make it feasible to render, on a single machine, complex scenes that would otherwise require multiple machines. But our observation about ray differentials and coherency could also improve the efficiency of parallel renderers.

2.5. Ray differentials

Beam tracing[10], cone tracing[1], and pencil tracing[25] trace bundles of light paths instead of infinitely thin rays. General intersection, reflection, and refraction calculations are difficult since each bounce can split the light beam/cone/pencil.

Igehy's ray differential method[11] traces single rays, but keeps track of the difference between each ray and two (real or imaginary) "neighbor" rays. These differences give an indication of the cone/beam size that each ray represents. The curvature at surface intersection points determines how those ray differentials are propagated at specular reflection and refraction. For example, if a ray hits a highly curved, convex surface, the specularly reflected ray will have a large differential (representing highly diverging neighbor rays). Ray differentials help in texture antialiasing since they indicate the best texture filter size, but they do not help in aliasing of ray hits (visibility): the ray either hits an object or not.

Suykens and Willems[30] generalized ray differentials to glossy and diffuse reflections. For distribution ray tracing of diffuse reflection, the ray differential corresponds to a fraction of the hemisphere. The more rays are traced from the same point, the smaller the fraction becomes. For path tracing of diffuse reflection, the "path differential" is a global value $\sqrt[2d]{N}$, where d is the ray depth and N is the total number of rays that reach that depth. Distribution ray tracing usually gives smaller, more accurate ray differentials than path tracing.

2.6. Multiresolution texture caching

Peachey[19] introduced a multiresolution texture caching scheme that caches 32×32 pixel texture tiles from MIP maps[37]. He found that texture accesses are highly coherent for REYES rendering, and that a cache size of 1% of the total texture size is sufficient. Thanks to our observation about ray differentials and coherence, we are able to use the same texture caching method for distribution ray tracing: incoherent texture lookups have large ray differentials, so high levels in the texture MIP maps suffice for these.

Peachey's texture cache also inspired our multiresolution geometry cache. Pharr et al.[21] observed that their geometry cache has similarities with Peachey's texture cache in that data is only loaded on demand, and a limited amount is stored in memory. With our method, the similarity is even stronger: one can think of our multiresolution geometry cache as a tessellation MIP map cache.

2.7. Ray tracing and global illumination with the RenderMan interface

Our implementation is done within the framework of the RenderMan specification[22]. There have been several earlier implementations of ray tracing and global illumination within this framework, for example the Vision system[27] and BMRT[9]. However, we believe that no other renderer has taken advantage of the relationship between ray differentials and coherency, and that neither Vision nor BMRT would be able to render scenes as complex as the ones we test here.

3. Overview

We are faced with the following conundrum: we need to keep a tessellated version of the entire scene in memory (since many rays are incoherent), we also need a fine tessellation (since some rays require high accuracy), and we do not want to reorder the rays. How is this possible? Fortunately, a key insight about the relation between ray differentials and ray coherency makes it possible to overcome this obstacle and deal with classic and distribution ray tracing in complex scenes. The insight is that there are two types of rays:

1. Specular reflection and refraction rays from surfaces with low curvature and shadow rays to point-like light sources. These rays have small differentials, and require high accuracy and fine tessellation. These rays are usually coherent, so using a geometry cache with relatively small capacity (few entries) works well.

2. Specular reflection and refraction rays from highly curved surfaces and rays from wide distribution ray tracing. These rays have large differentials, do not require high accuracy, and can use a coarse tessellation. These rays are incoherent, so they require a cache with large capacity (many entries) and/or entries that are fast to recompute.

Utilizing this observation, we present a multiresolution geometry caching scheme with separate caches for coarsely, medium, and finely tessellated surfaces. This exploits the different coherencies of various types of rays, and their different accuracy requirements. It is interesting to note that this scheme results in an automatic level-of-detail representation of the tessellation — a tessellation MIP map[37]. In fact, as mentioned earlier, our multiresolution geometry cache is remarkably similar to Peachey's texture MIP map cache[19].

In our implementation, we assume that the high-level descriptions of all objects fits in memory. The same assumption is typically made by pure REYES renderers[5]. Fortunately, NURBS control points, top level subdivision meshes, etc. are typically orders of magnitude more compact than their fully tessellated representation. The only extra memory we use in addition to that used by a pure REYES scanline renderer is for the geometry cache (fixed size, typically 30 MB) and a spatial acceleration data structure (less than 50 MB even for very complex scenes).

We tessellate all geometry on demand. We choose the appropriate tessellation rate based on ray size: the quads should be approximately the same size as the ray beam cross-section. (Using smaller quads is a waste of time and memory; larger quads do not give adequate precision.) Tessellation makes ray tracing faster, simplifies displacement mapping, and allows for displacements in arbitrary directions. The ability to cache displaced tessellations ensures that we rarely need to run the displacement shader repeatedly for the same surface.

4. Coherent rays are narrow, incoherent rays are wide

In this section we analyze in detail the coherency and differential sizes for different types of rays.

4.1. Terminology and ray propagation

A ray consists of an origin, P, and a direction, D. The *ray differential* at P is $(\frac{\partial P}{\partial u}, \frac{\partial P}{\partial v}, \frac{\partial D}{\partial u}, \frac{\partial D}{\partial v})$. The ray's $\frac{\partial P'}{\partial u}$ and $\frac{\partial P'}{\partial v}$ at a point P' span a parallelogram. A *ray beam* is spanned by the parallelograms along the ray. The *ray footprint* at a ray intersection point is the projection of the ray parallelogram onto the surface tangent plane at that point. Please refer to figure 1 for an illustration. We call a ray *narrow* if its ray beam has a small cross-section, and *wide* if it has a large cross-section.

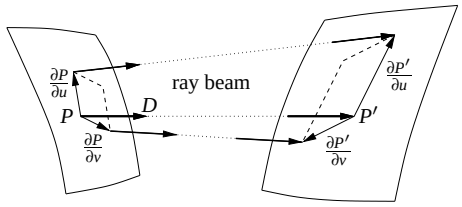

Figure 1: *Ray differentials and beam for a ray from P to P'.*

4.2. Specular reflection and refraction rays

Figure 2 shows parallel rays specularly reflected by flat, convex, and concave surfaces. Reflection from a flat surface gives coherent, narrow reflection rays. Conversely, reflection from a highly curved, bumped or displaced surface gives incoherent wide reflection rays: two adjacent rays are reflected in different directions, but also have large differentials. Note that even though the reflection rays from concave surfaces are initially narrow, after a certain distance, the ray differentials cross over and the rays get wider again.

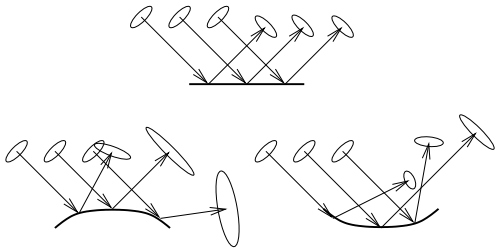

Figure 2: *Specular reflection (shown in 2D for clarity): a) flat surface; b) convex surface; c) concave surface.*

Specular refraction is very similar: refraction through a flat surface gives parallel, narrow refraction rays, while refraction through a highly curved surface gives diverging wide refraction rays.

It is tempting to conclude from these examples that all coherent specular rays have narrow beams and all incoherent specular rays have wide beams. But there is an unfortunate exception: surfaces with many tiny flat facets, as for example a disco ball. The small flat facets reflect rays with narrow beams, but in incoherent directions.

4.3. Shadow rays from point-like sources

Shadow rays to a point, spot, or directional light source are very narrow and very coherent. In this respect, they behave as specular reflection rays from a flat surface. If there are several light sources, only the rays to each light are coherent with each other; rays to different light sources are not coherent. Fortunately, even if there are thousands of light sources in a scene, usually only a small fraction of them illuminate a given point by a significant amount — the rest of

the lights can be probabilistically sampled, computed without shadows, or skipped entirely[26, 34]. So for each part of a surface, only a few light sources require narrow shadow rays.

4.4. Glossy reflection and refraction rays

Distribution ray tracing is used to render glossy reflection and refraction. Assume that glossiness is specified as a solid angle over which the reflection rays can be reflected. The directional differential $(\frac{\partial D}{\partial u}, \frac{\partial D}{\partial v})$ of each ray corresponds to the glossy cone angle divided by the number of rays — see figure 3.

Figure 3: *Glossy reflection from flat surfaces.*

For glossy reflection from curved surfaces, we use the maximum of the ray differential for specular reflection from a curved surface and the differential for glossy reflection from a flat surface. This is a heuristic that seems to work well in practice.

Rays from narrow glossy reflection and refraction have small differentials and are coherent. Conversely, rays from wide glossy reflection and refraction have larger differentials and are incoherent.

4.5. Shadow rays from area light sources

Distribution ray tracing is also used to compute soft shadows from area light sources. The directional differential of a shadow ray to an area light source is computed by taking the relative size of the light source divided by the number of shadow rays. Shadow rays to a small area light source are narrow and coherent, while shadow rays to a large area light source are wide and incoherent.

4.6. Hemisphere sampling

Distribution ray tracing over a hemisphere is used to compute diffuse reflection and transmission (translucency), ambient occlusion, one-bounce color bleeding, and final gathering of global illumination.

For such hemisphere sampling, the directional ray differential corresponds to the fraction of the hemisphere that is covered by that ray. If the hemisphere is sampled in a cosine-weighted fashion, the rays are more dense near the pole than near the equator, and the rays near the pole have smaller directional differentials than rays near the equator. See figure 4 for an illustration.

The hemisphere sampling rays are quite wide, except near their origin. But the rays are hit-tested with coherent geometry near their origin even though their directions diverge.

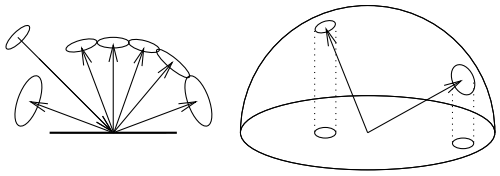

Figure 4: *Diffuse reflection from flat surface.*

This approach breaks down if there are only a few hemisphere sampling rays since the ray differentials get very large. The worst case is if there is only one ray (as in path tracing); in that case the directional ray differential would correspond to the entire hemisphere. One would then have to resort to a global value based on the total number of rays at that depth[30]. But to get hemisphere sampling results without excessive noise, we need to shoot many hemisphere samples anyway — typically at least 256.

4.7. Summary

From this analysis, we conclude that in most cases, *coherent rays have narrow beams*, and *incoherent rays have wide beams*.

5. Implementation

We used this observation as part of our recent extension of Pixar's RenderMan renderer (PRMan) to support ray tracing and global illumination. PRMan is a widely used commercial renderer that adheres to the RenderMan specification[22] and is based on the REYES architecture[2, 5].

5.1. REYES and rays

In a REYES renderer, all visible geometry is tessellated into micropolygon grids and the vertices of the grids are shaded. With the addition of ray tracing, shading at these vertices can cause rays to be shot to compute reflections, shadows, etc. This hybrid rendering technique means that, in contrast to "pure" ray tracing, there are no camera rays.

In our current implementation, the tessellated surfaces used for ray tracing are not identical to the REYES micropolygon grids, so each surface patch has two representations. It may be possible to merge the two, although it would require the tessellation cache to be able to deal with general tessellation rates (such as 5×13) instead of only the fixed rates currently handled (as described in the following).

Rendering with REYES and ray tracing inevitably takes longer than pure REYES rendering, not only because of the time spent calculating ray intersections, but also due to the additional shading required at the ray hit points.

To shade a ray hit point, we create a small shading grid of three points (similar to BMRT[9]). The two extra points are necessary for shaders that use derivatives and also to generate new ray differentials. The two points are created based on the ray footprint, and their normals are created based on the local curvature of the object. New rays are only traced from the real hit point; the rays from the two other points are only used to set up ray differentials.

To avoid hemisphere sampling from all shading points, we use interpolation of irradiance using irradiance gradients[35] and, similarly, interpolation of ambient occlusion using occlusion gradients. It is simpler to interpolate between values on a grid of REYES shading points (that are known to be on the same surface) than between disconnected points in 3D space.

We select the appropriate geometric representation on-the-fly based on the ray beam sizes. If the initial subpatch is too large for a sufficient tessellation, for example because the object is seen through a magnifying glass or reflected by a concave mirror, we subdivide it into even smaller subpatches. By contrast, rendering algorithms that pre-tessellate based on screen size will sometimes be wrong; in the presence of magnifying glasses, concave mirrors, etc., it is impossible to know the necessary tessellation rate before rendering starts.

5.2. Multiresolution geometry cache

We use a caching scheme with separate caches for coarse, medium, and fine tessellations to exploit the different coherency and accuracy needed:

1. A fine tessellation cache with large elements (17×17 vertices = 16×16 quads) only needs to hold relatively few entries since narrow rays are coherent.
2. A medium tessellation cache (5×5 vertices = 4×4 quads) in between for rays that are neither very coherent nor very narrow.
3. A coarse tessellation cache with small elements (2×2 vertices = 1 quad) can hold many entries. It is also cheap to recompute the entries if they have been swapped out since they consist of only four vertices.

These three co-existing approximations of a surface subpatch are shown in figure 5. The fine tessellation cache also stores 4×4 bounding boxes (each for 4×4 quads) for efficient intersection tests.

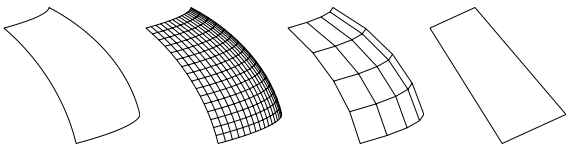

Figure 5: *A surface subpatch and its tessellations (left to right): original subpatch, 16×16 quads, 4×4 quads, 1 quad.*

We use a least-recently-used (LRU) cache replacement scheme. The size of the geometry cache can be specified by the user. By default, the size is 10 MB for each of the three caches, so with a vertex taking up 12 bytes, the coarse cache has a capacity of 220,000 entries, the medium cache has 35,000 entries, and the fine cache has 3,000 entries. For comparison, a single fine-resolution cache of 30 MB can hold only 9,100 entries. It would be interesting to investigate other choices for the number of caches and their relative sizes, or to use a single multiresolution cache for all tessellations.

We choose the appropriate cache such that the quads are approximately the same size as the ray beam. If the ray beam size is in-between cache levels, we lookup in the nearest finer cache and merge each set of 2×2 quads into one quad for faster intersection testing. This, in effect, gives us five different tessellation resolutions while only storing three.

6. Results

The following tests were performed on a standard PC with a 900 MHz Pentium III processor and 512 MB of memory. The rendered images are 1024 pixels wide and have micropolygons that are at most one pixel large.

We used a geometry cache size of 30 MB (both for single-resolution and multiresolution geometry caches), except where noted, and a texture cache size of 10 MB.

6.1. Terminology

When a ray hits the bounding box of a subpatch that has never been tessellated at the appropriate resolution before, the subpatch is tessellated at that resolution and the tessellation inserted into the cache. We call this a *cold tessellation*. If the subpatch has been tessellated at the desired resolution before, the tessellation is looked up in the cache. A *cache hit* means that the tessellation was in the cache; the opposite is a *cache miss*. The *cache hit rate* is cache hits/cache lookups. When a cache miss occurs, we have to retessellate the patch. We measure *cache cost* as the number of vertices that are recomputed due to retessellations.

6.2. Parking lot

The first test scene consists of fifteen cars on a plane, as shown in figure 6. Each car consists of 2155 NURBS patches, many of which have trimming curves. The cars are explicitly copied, not instanced. During rendering, the NURBS patches are split into 940,000 subpatches. The spatial acceleration data structure (a Kay-Kajiya tree[7]) uses around 35 MB. A full tessellation would result in $940,000 \times 17^2 \approx 270$ million vertices (240 million quads or 480 million triangles), consuming around 3.3 GB. This is 110 times the size of the geometry cache.

6.2.1. Shiny cars on diffuse ground

For this test, the ground plane is purely diffuse. The car paint and chrome shaders shoot specular reflection rays. The scene is illuminated by a directional light with sharp ray traced shadows. Since the rays in this version of the scene are specular reflection rays from mostly smooth objects and shadow rays to a directional light source, most rays are coherent and narrow. So this is a case where we expect a lot of benefit from caching, but little from multiresolution.

Figure 6: *Shiny cars on diffuse ground.*

Rendering the image in figure 6 causes the tracing of 4.1 million specular rays and 4.0 million shadow rays. These rays result in 100 million ray-subpatch intersection tests.

Our tessellation scheme employs two distinct mechanisms to achieve efficiency: a multiresolution representation and a cache of reusable tessellations. To distinguish the contributions of each mechanism, let's first consider single-resolution tessellation, with and without caching.

With a single-resolution cache, there are 380,000 cold tessellations (producing 110 million vertices), 100 million cache lookups, and 1.3 million cache misses, corresponding to a hit rate of 98.7% and 360 million recomputed vertices. The run time is 79 minutes. Without caching, the 100 million intersection tests would require computing 100 million $\times$ $17^2 \approx 29$ billion vertices; this 80 times more than the vertices recomputed due to cache misses.

With a multiresolution cache, there are 400,000, 100,000, and 30,000 cold tessellations (13 million vertices) and 35, 23, and 41 million cache lookups in the coarse, medium, and fine caches. There are 7,100, 3,300, and 14,000 cache misses, corresponding to hit rates of 99.97-99.99% and 6.2 million recomputed vertices. The run time is 62 minutes. Without caching, there would have been $35 \times 2^2 + 23 \times 5^2 + 41 \times 17^2$ million ≈ 12 billion computed vertices — around 2000 times more than the 6.2 million vertices with caching.

6.2.2. Shiny cars on ambient occlusion ground

For this test, the ground plane is shaded with purely ambient occlusion, as shown in figure 7. This means that there is a mix of coherent and incoherent rays: 163 million occlusion rays, 4.1 million specular rays, and 3.6 million shadow rays. These rays cause 1.2 billion ray-subpatch intersection tests.

Figure 7: *Shiny cars on ambient occlusion ground.*

With a single-resolution cache, there are 650,000 cold tessellations (190 million vertices), 1.2 billion cache lookups, and 30 million cache misses — corresponding to a hit rate of 97.5% and 8.7 billion recomputed vertices. The runtime is 32 hours. Without caching, the 1.2 billion intersection tests would cause the computation of 350 billion vertices.

When multiresolution caching is used, there are 730,000, 110,000, and 30,000 cold tessellations (producing 14 million vertices) and 950 million, 190 million, and 120 million cache lookups, respectively. There are 1.6 million, 4,500, and 13,000 cache misses, corresponding to hit rates of 99.8–99.99% and a cost of 10 million recomputed vertices. This is only 0.11% of the 8.7 billion recomputed vertices for the single-resolution cache. The runtime is 11.5 hours. Without caching, the intersection tests would cause the computation of 43 billion vertices.

6.3. Psychedelic dragons

This scene contains 94 dragons modeled as subdivision surfaces. The dragons are modeled individually, not instanced. Behind them there are several procedurally displaced NURBS spheres. The scene consists of 4,815 geometric primitives; these are subdivided during rendering into 183,000 subpatches. If fully tessellated, the scene would require 53 million vertices (630 MB of memory) corresponding to 47 million quads or 94 million triangles. The textures in the scene are a mix of images and procedural textures. The scene is illuminated by a directional light source, and shadows are computed by ray tracing.

6.3.1. Chrome and direct illumination of matte

For this test, half of the dragons are reflective chrome, half of them matte, as shown in figure 8. All rays (specular reflection and shadow rays) are coherent.

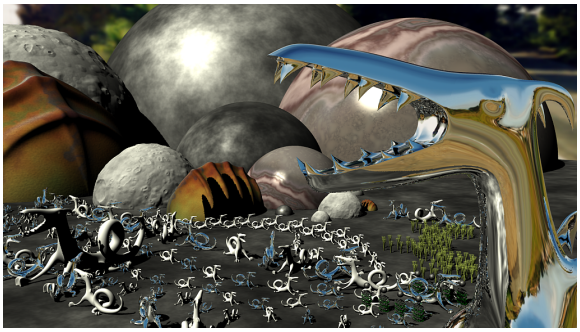

Figure 8: *Dragon scene with ray traced reflection and shadows.*

With a single-resolution cache, the cache hit rate is 98.5%, and $140,000 \times 17^2 \approx 40$ million vertices are recomputed due to cache misses. The run time is 21 minutes.

With a multiresolution cache, the cache hit rates are 99.96–100%, and $0 \times 2^2 + 1,200 \times 5^2 + 250 \times 17^2 \approx 100,000$ vertices are recomputed due to cache misses. The run time is 19 minutes. Even though all lookups are coherent, the multiresolution cache reduces the number of recomputed vertices by a factor of 400.

6.3.2. Chrome and color bleeding

For this test, half of the dragons are chrome again, while the other half have a material that computes color bleeding (direct illumination plus single bounce soft indirect illumination). See figure 9. Notice the color bleeding from the ground and sky onto the matte dragons. The run time with a 30 MB multiresolution cache is 2 hours 1 minute.

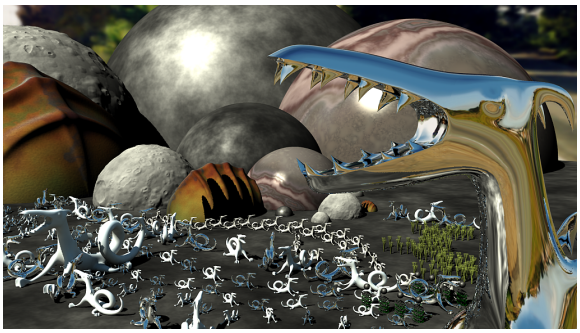

Figure 9: *Dragon scene with ray traced reflection, shadows, and soft indirect illumination.*

With a single-resolution cache, there are 120,000 cold tessellations (producing 35 million vertices) and 18 million cache lookups. The cache capacity, cache misses, cache hit rate, and cache cost for varying cache sizes are listed in the table below.

size	capacity	misses	hit rate	cost
100MB	30k	100k	99.4%	29Mvtx
30MB	9k	350k	98.0%	100Mvtx
10MB	3k	700k	96.1%	200Mvtx
3MB	900	1.2M	93.3%	350Mvtx
1MB	300	1.7M	90.5%	490Mvtx

With the multiresolution representation, there are 120,000, 40,000, and 7,500 cold tessellations (producing 3.6 million vertices) and 6.0 million, 5.4 million, and 6.3 million lookups in the coarse, medium, and fine caches. The results for varying cache sizes are tabulated below.

size	cap.	misses	hit rate	cost
100MB	840k	0+0+0	100%	0
30MB	260k	0+84+180	99.9–100%	54kvtx
10MB	84k	21k+770+2.3k	99.6–99.9%	770kvtx
3MB	26k	180k+16k+7.3k	97.2–99.9%	3.2Mvtx
1MB	8.4k	470k+39k+20k	92.7–99.7%	8.6Mvtx

Comparing these two tables leads to several interesting observations. For example, with a multiresolution geometry cache of only 1 MB, 8.6 million vertices are recomputed. This is significantly less than the number of vertices that are recomputed using a single-resolution cache, even if that cache uses 100 MB. With a multiresolution cache size of 3 MB, the cost of the cache misses (3.2 million vertices) is about the same as the cold tessellations (3.6 million vertices). This means that the 3 MB cache performs well despite its small size. 3 MB is less than 1/200th of the 630 MB this scene would consume in fully tessellated form.

6.3.3. Texture caching

We have not tested texture cache performance as thoroughly as the results above for geometry caching. However, in a separate test of the dragon scene from the previous section, we observed that when all texture lookups are at the most detailed MIP map level, the default 10 MB texture cache thrashed heavily, and half of the run time was spent (re)reading textures from disk. In contrast, when ray differentials are used to determine the appropriate MIP map level, texture reads do not contribute measurably to run time.

6.4. City street

The last test scene is a city street modeled with NURBS patches and subdivision surfaces. It consists of around 46,000 top-level primitives. During rendering, these primitives are divided into 970,000 subpatches. Fully tessellating the entire scene would give 280 million vertices (3.4 GB of memory) corresponding to nearly 250 million quads or 500

million triangles. We shade all objects with short-range ambient occlusion (meaning that the occlusion rays have a short cut-off distance) multiplied by surface color. The resulting image is shown in figure 10. Computing the image causes tracing of 210 million occlusion rays.

Figure 10: *City street with short-range ambient occlusion.*

With a single-resolution cache, there are 710,000 cold tessellations, 1.1 billion cache lookups, and 2.1 million cache misses. This corresponds to a hit rate of 99.8% and 610 million recomputed vertices.

With the multiresolution cache, there are 710,000, 300,000, and 93,000 cold tessellations and 300, 230, and 600 million lookups in the coarse, medium, and fine caches. There are 2,700, 6,500, and 29,000 cache misses, corresponding to hit rates above 99.99%. The cache misses cause recomputation of 8.6 million vertices — 71 times fewer than with the single-resolution cache. The run time is 7.1 hours.

This shows that even when occlusion rays are only traced over short distances and the accessed geometry is coherent, multiresolution tessellation and caching still pays off. For long-range ambient occlusion, where the rays intersect the geometry in a less coherent manner, the multiresolution cache would be even more beneficial.

7. Discussion and future work

In this section, we discuss some of the limitations of our current implementation and list suggestions for future work.

7.1. Geometric inconsistency

Geometric inconsistencies can occur where the tessellation rate changes — typically cracks in the geometry due to T-vertices. There are several ways to reduce or eliminate this problem, as discussed by for example Pharr and Hanrahan[20]. We have, perhaps surprisingly, not seen any artifacts from this. This may be because soft diffuse illumination and ambient occlusion are such low-frequency functions that a single ray slipping through a crack does not contribute much error.

To avoid potential "popping" in animations due to sudden change of tessellation rate, we could stochastically choose the tessellation rate (rounding up or down) for each ray-subpatch intersection test.

7.2. The disco ball problem

As mentioned already, disco balls with small mirror facets are a problem. The reflected rays have narrow beams (since each little facet is flat), but the rays are incoherent since all the facets reflect in different directions. This destroys cache coherency. Possible work-arounds may be to artificially increase the beam size of the reflection rays, use a simplified scene for reflections in the disco ball, or use a reflection map. If the facets are large, many coherent rays are traced from each facet, and the problem goes away.

7.3. Importance

It is not clear how a ray's image contribution (aka. importance[3]) should affect the tessellation rate. Shooting more rays to sample the hemisphere above a point makes each ray narrower, but also reduces its contribution. The tessellation rate should depend on beam size, importance, and ray type, but exactly how? It would probably be safe to use coarser tessellations than the beam size when the importance is low.

8. Conclusion

In this paper, we have introduced and exploited the observation that coherent rays are narrow, while incoherent rays are wide. By careful analysis of the requirements of a geometry cache, we get the benefits of tessellation (speed and flexible displacement) without the excessive memory overhead. This makes it possible to render very complex scenes with ray tracing — both classic ray tracing for specular reflection, refraction, and sharp shadows, and wide distribution ray tracing for global illumination, ambient occlusion, etc. With the multiresolution geometry cache, it is possible to render scenes of nearly the same complexity as with pure REYES scanline rendering.

Acknowledgements

We would like to thank the other members of Pixar's RenderMan Products group, including Dylan Sisson for creating the dragon scene and Jamie Hecker for writing ptviewer. Many thanks also to Tony Apodaca for explaining Opal's shading system, Brian Smits for discussions about raytracing and displacements, Mitch Prater for a car reflection test model and much encouragement, Guido Quaroni for test scenes, Mark VandeWettering for tests on production shots, and Piero Foscari for helpful comments to this paper.

References

1. J. Amanatides. Ray tracing with cones. In *Computer Graphics (Proc. SIGGRAPH 84)*, pages 129–135, 1984. 3

2. A. Apodaca and L. Gritz. *Advanced RenderMan — Creating CGI for Motion Pictures*. Morgan Kaufmann, 2000. 2, 2, 5

3. P. Christensen. Importance for ray tracing. *Ray Tracing News*, 12(2), 1999. (www.acm.org/tog/resources/RTNews/html). 9

4. P. Christensen, D. Lischinski, E. Stollnitz, and D. Salesin. Clustering for glossy global illumination. *ACM Transactions on Graphics*, 16(1):3–33, 1997. 2

5. R. Cook, L. Carpenter, and E. Catmull. The Reyes image rendering architecture. In *Computer Graphics (Proc. SIGGRAPH 87)*, pages 95–102, 1987. 2, 4, 5

6. R. Cook, T. Porter, and L. Carpenter. Distributed ray tracing. In *Computer Graphics (Proc. SIGGRAPH 84)*, pages 137–145, 1984. 2

7. A. Glassner. *An Introduction to Ray Tracing*. Academic Press, 1989. 2, 6

8. S. Green and D. Paddon. Exploiting coherency for multiprocessor ray tracing. *IEEE Computer Graphics and Applications*, 9(6):12–26, 1989. 2

9. L. Gritz and J. Hahn. BMRT: A global illumination implementation of the RenderMan standard. *Journal of Graphics Tools*, 1(3):29–47, 1996. 3, 6

10. P. Heckbert and P. Hanrahan. Beam tracing polygonal objects. In *Computer Graphics (Proc. SIGGRAPH 84)*, pages 119–127, 1984. 3

11. H. Igehy. Tracing ray differentials. In *Computer Graphics (Proc. SIGGRAPH 99)*, pages 179–186, 1999. 3

12. H. Jensen. *Realistic Image Synthesis using Photon Mapping*. A K Peters, 2001. 1, 2

13. J. Kajiya. The rendering equation. In *Computer Graphics (Proc. SIGGRAPH 86)*, pages 143–150, 1986. 2

14. T. Kato. The Kilauea massively parallel ray tracer. In A. Chalmers, T. Davis, and E. Reinhard, editors, *Practical Parallel Rendering*, chapter 8. A K Peters, 2002. 1, 3

15. L. Kobbelt, K. Daubert, and H.-P. Seidel. Ray tracing of subdivision surfaces. In *Rendering Techniques '98 (Proc. 9th Eurographics Workshop on Rendering)*, pages 69–80, 1998. 2

16. H. Landis. Production-ready global illumination. In *SIGGRAPH 2002 course note #16*, pages 87–102, 2002. 1

17. W. Martin, E. Cohen, R. Fish, and P. Shirley. Practical ray tracing of trimmed NURBS surfaces. *Journal of Graphics Tools*, 5(1):27–52, 2000. 2

18. S. Parker, W. Martin, P.-P. Sloan, P. Shirley, B. Smits, and C. Hansen. Interactive ray tracing. In *Symposium on Interactive 3D Graphics*, pages 119–126, 1999. 3

19. D. Peachey. Texture on demand. Pixar technical memo 217 (unpublished manuscript), 1990. 3, 4

20. M. Pharr and P. Hanrahan. Geometry caching for ray-tracing displacement maps. In *Rendering Techniques '96 (Proc. 7th Eurographics Workshop on Rendering)*, 1996. 2, 2, 9

21. M. Pharr, C. Kolb, R. Gershbein, and P. Hanrahan. Rendering complex scenes with memory-coherent ray tracing. In *Computer Graphics (Proc. SIGGRAPH 97)*, pages 101–108, 1997. 1, 2, 3

22. Pixar Animation Studios. RenderMan Interface Specification version 3.2, 2000. 3, 5

23. H. Rushmeier, C. Patterson, and A. Veerasamy. Geometric simplification for indirect illumination calculations. In *Proc. Graphics Interface '93*, pages 227–236, 1993. 2

24. A. Sherstyuk. Fast ray tracing of implicit surfaces. *Computer Graphics Forum*, 18(2):139–147, 1999. 2

25. M. Shinya, T. Takahashi, and S. Naito. Principles and applications of pencil tracing. In *Computer Graphics (Proc. SIGGRAPH 87)*, pages 45–54, 1987. 3

26. P. Shirley, C. Wang, and K. Zimmerman. Monte Carlo techniques for direct lighting calculations. *ACM Transactions on Graphics*, 15(1):1–36, 1996. 5

27. P. Slusallek, T. Pflaum, and H.-P. Seidel. Using procedural RenderMan shaders for global illumination. In *Computer Graphics Forum (Proc. Eurographics '95)*, pages 311–324, 1995. 3

28. B. Smits, J. Arvo, and D. Greenberg. A clustering algorithm for radiosity in complex environments. In *Computer Graphics (Proc. SIGGRAPH 94)*, pages 435–442, 1994. 2

29. B. Smits, P. Shirley, and M. Stark. Direct ray tracing of displacement mapped triangles. In *Rendering Techniques 2000 (Proc. 11th Eurographics Workshop on Rendering)*, pages 307–318, 2000. 2, 2

30. F. Suykens and Y. Willems. Path differentials and applications. In *Rendering Techniques 2001 (Proc. 12th Eurographics Workshop on Rendering)*, pages 257–268, 2001. 3, 5

31. I. Wald, T. Kollig, C. Benthin, A. Keller, and P. Slusallek. Interactive global illumination using fast ray tracing. In *Rendering Techniques 2002 (Proc. 13th Eurographics Workshop on Rendering)*, pages 9–19, 2002. 3

32. I. Wald, P. Slusallek, C. Benthin, and M. Wagner. Interactive distributed ray tracing of highly complex models. In *Rendering Techniques 2001 (Proc. 12th Eurographics Workshop on Rendering)*, pages 277–288, 2001. 3

33. I. Wald, P. Slusallek, C. Benthin, and M. Wagner. Interactive rendering with coherent raytracing. In *Computer Graphics Forum (Proc. Eurographics 2001)*, pages 153–164, 2001. 2

34. G. Ward. Adaptive shadow testing for ray tracing. In *Proc. 2nd Eurographics Workshop on Rendering*, pages 11–20, 1991. 5

35. G. Ward and P. Heckbert. Irradiance gradients. In *Proc. 3rd Eurographics Workshop on Rendering*, pages 85–98, 1992. 6

36. G. Ward Larson and R. Shakespeare. *Rendering with Radiance*. Morgan Kaufmann, 1998. 1

37. L. Williams. Pyramidal parametrics. In *Computer Graphics (Proc. SIGGRAPH 83)*, pages 1–11, 1983. 3, 4

38. S. Zhukov, A. Iones, and G. Kronin. An ambient light illumination model. In *Rendering Techniques '98 (Proc. 9th Eurographics Workshop on Rendering)*, pages 45–55, 1998. 1