Rendering Complex Scenes with Memory-Coherent Ray Tracing

Matt Pharr Craig Kolb Reid Gershbein Pat Hanrahan

Computer Science Department, Stanford University

Abstract

Simulating realistic lighting and rendering complex scenes are usually considered separate problems with incompatible solutions. Accurate lighting calculations are typically performed using ray tracing algorithms, which require that the entire scene database reside in memory to perform well. Conversely, most systems capable of rendering complex scenes use scan-conversion algorithms that access memory coherently, but are unable to incorporate sophisticated illumination. We have developed algorithms that use caching and lazy creation of texture and geometry to manage scene complexity. To improve cache performance, we increase locality of reference by dynamically reordering the rendering computation based on the contents of the cache. We have used these algorithms to compute images of scenes containing millions of primitives, while storing ten percent of the scene description in memory. Thus, a machine of a given memory capacity can render realistic scenes that are an order of magnitude more complex than was previously possible.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing

Keywords: scene data management, caching, computation reordering, coherence

1 Introduction

Rendering systems are challenged by three types of complexity: geometric, surface, and illumination. Geometric complexity is necessary to model detailed environments; many more primitives than can fit into memory may be necessary to model a scene accurately. Surface complexity is a result of programmable shading and many texture maps. Illumination complexity arises from realistic lighting models and the interreflection of light. Previous rendering algorithms have not been able to handle all of these types of complexity simultaneously. Generally, they either perform illumination computations assuming that the entire scene fits in main memory, or only store part of the scene in memory and simplify the lighting computation. In order to be able to use algorithms that compute accurate illumination with such complex scenes, the coherence of scene data reference patterns must be greatly improved.

Exploiting coherence to increase efficiency is a classic technique in computer graphics[19]. Increasing the coherence of a computation can reduce the amount of memory used, the time it requires, or both. For example, Z-buffer rendering algorithms operate on a single primitive at a time, which makes it possible to build rendering hardware that does not need access to the entire scene. More recently, the Talisman architecture was designed to exploit frame-to-frame coherence as a means of accelerating rendering and reducing memory bandwidth requirements[21].

Kajiya has written a whitepaper that proposes an architecture for Monte Carlo ray tracing systems that is designed to improve coherence across all levels of the memory hierarchy, from processor caches to disk storage[13]. The rendering computation is decomposed into parts—ray-object intersections, shading calculations, and calculating spawned rays—that are performed independently. The coherence of memory references is increased through careful management of the interaction of the computation and the memory that it references, reducing overall running time and facilitating parallelism and vectorization. However, no system based on this architecture has been implemented.

We have independently developed similar algorithms, based on two main ideas: caching and reordering. We cache a subset of large geometric and texture databases in main memory for fast access by the rendering system. Data is added to these caches on demand when needed for rendering computation. We ensure coherent access to the cache by statically reordering scene data, dynamically placing it in memory, and dynamically reordering ray intersection calculations. This reordering is critical for good performance with small caches. These algorithms have made it possible to efficiently compute images using global illumination algorithms with scenes containing roughly ten times as many primitives as can fit into memory. This marks a large increase in the complexity of scenes that can be rendered effectively using Monte Carlo methods.

In this paper, we describe the algorithms we have developed and the system we have built that uses them. We first discuss how previous rendering systems have managed complexity. We then introduce and describe our algorithms in detail and discuss their implementation. Finally, we present results from applying our algorithms to a variety of realistic complex scenes and discuss the performance of the algorithms.

2 Background

Previously developed techniques that address the problems of rendering complex scenes include culling algorithms, lazy evaluation and caching, and reordering independent parts of a computation to improve its memory coherence. In this section, we briefly describe some of this previous work and how our work draws from and builds upon it.

While most ray tracing systems do not explicity address scene memory management, several researchers have investigated this issue, particularly in the context of managing scene distribution on multiprocessors. Jansen and Chalmers have written a survey of past work in parallel rendering that investigated these issues[11], and in particular, Green used geometry caching techniques to manage scene distribution on a multiprocessor[7]. Pharr and Hanrahan later used geometry caching to manage large amounts of displacement-mapped geometry in a serial ray tracer[17]. Global illumination calculations were not performed in these geometry caching systems, so the rays that were traced passed through coherent regions of space

and the caches performed well. In general, however, Monte Carlo ray tracing systems that evaluate the trees of rays in depth-first order access scene data too incoherently for caching algorithms to be effective.

A number of techniques have been presented to increase the coherence of rays traveling through a scene. Fröhlich traversed ray trees iteratively in order to be able to gather rays into coherent bundles. The bundles were stored in the intersection acceleration data structure and voxels of rays and geometry were processed in order based on how many rays were contained within. Rays in voxels were processed as a group so that candidate objects for intersection tests could be found and so that the overhead of octree traversal could be reduced. In a manner similar to shaft culling[9], Reinhard and Jansen gathered rays with common origins into frustums that were traced together so that a set of the objects inside the frustum could be found to accelerate ray-object intersection tests[18]. Pharr and Hanrahan reordered eye rays using space-filling curves over the image plane to improve the coherence of spawned rays in a depth-first ray tracer, which in turn improved geometry cache performance[17].

Scanline-based rendering algorithms are able to render images of scenes that are too complex to fit into memory; the REYES architecture[4] is representative of these approaches. Sorting, culling, and lazy evaluation techniques further increase the efficiency of REYES. At startup time, geometric primitives are sorted into the screen-space buckets that they overlap. When a bucket is rendered, the primitives overlapping it are subdivided into grids of pixel-sized micropolygons that are shaded all at once and discarded as soon as they have been sampled in the image plane. As rendering progresses, most hidden geometry is culled before being shaded by an algorithm similar to the hierarchical Z-buffer[8, 1]. This sorting and culling process allows REYES to store in memory only a small fraction of the total number of potential micropolygons.

Two major features of REYES are programmable shading[3, 10] and support for large amounts of texture data. A texture caching scheme, described by Peachey[16], makes it possible to render scenes with much more texture than can fit into memory. Textures are pre-filtered into a set of multiresolution images (used for antialiasing) that are stored on disk in tiles of approximately 32 by 32 texels. A fixed number of texture tiles are cached in memory, and when the cache fills up, the least recently used tile is discarded. Since texture is only read into memory when needed, startup time is low and textures that do not contribute to the image do not affect performance. Furthermore, since texture is resampled from the prefiltered images, each shading calculation makes a small number of accesses to a local part of the texture, which further improves locality of reference and, thus, performance. The texture cache in REYES performs extremely well: Peachey found that less than 1% of the texture in a scene can typically be kept in memory without any degradation in performance. Our system uses texture caching in a similar manner, and extends these ideas to support efficient geometry caching.

Algorithms that explicitly take advantage of the dense occlusion present in large architectural models have been used to compute radiosity solutions in scenes that would otherwise be intractable[20, 6]. These algorithms break the computation into nearly independent sub-problems, based on sets of mutually-interacting objects. Computation is reordered so that only a spatially local part of the data is processed at a time, and computation is scheduled based on which parts of the scene are already in memory to minimize the time spent reading additional data from disk. These algorithms make it possible to compute radiosity solutions for models that would require enormous resources using traditional techniques. Our computation reordering techniques build on the reordering frameworks of these systems.

3 Overview

In the next two sections, we describe the techniques we have investigated for managing scene data and for finding and exploiting coherence in ray tracing based rendering algorithms. Figure 1 illustrates how the various parts of our system interact. Disk storage is used to manage texture, geometry, queued rays and image samples. First, the camera generates eye rays to form the image, and these are partitioned into coherent groups. The scheduler selects groups of rays to trace, based on information about which parts of the scene are already in memory and the degree to which processing the rays will advance the computation. Intersection tests are performed with the chosen rays, which causes geometry to be added to the cache as needed. As intersections are found, shading calculations are performed, and the texture maps used for shading are managed by the texture cache. Any new rays that are spawned during shading are returned to the scheduler to be added to the queues of waiting rays. Once all of the rays have terminated, the image samples are filtered and the image is reconstructed.

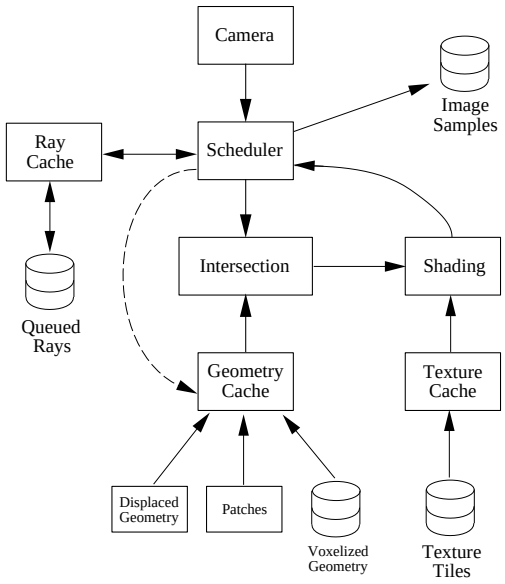

Figure 1: Block diagram of our system.

4 Caching Scene Data

Our system uses texture caching and geometry caching to manage scene complexity. Our texture cache is much like Peachey's[16], and texture access is driven by procedural shaders written in a language similar to the RenderMan shading language. Our geometry caching scheme draws upon ideas from the texture cache: a limited amount of geometry is stored in memory, and lazy loading limits the amount of data added to the cache to that which is needed for ray intersection tests. The only information needed by the geometry cache at startup time is the spatial bounds of each model in the scene. Both caches use a least recently used replacement policy.

4.1 Geometry Sources

A distinguishing feature of our ray tracer is that we cache a single type of geometric primitive: triangles. This has a number of advantages. Ray intersection tests can be optimized for a single case, and memory management for the geometry cache is easier, since there is less variation in the amount of space needed to store different types of primitives. It is also possible to optimize

Figure 2: Trees by a lake

many other parts of the renderer when only one type of primitive is supported. The REYES algorithm similarly uses a single internal primitive—micropolygons—to make shading and sampling more efficient[4]. Unlike REYES, we optimize the system for handling large databases of triangles; this allows our system to efficiently handle a wide variety of common sources of geometry, including scanned data, scientific data, and tessellated patches. A potential drawback of this single representation is that other types of primitives, such as spheres, require more space to store after they are tessellated. We have found that the advantages of a single representation outweigh this disadvantage.

A number of different sources can supply the geometry cache with triangles:

- *Secondary storage.* We store triangle meshes on disk presorted into voxels. All of the geometry in each voxel is stored contiguously, so that it may be quickly read from disk. The model's bounding box is stored in the file header for efficient access at startup time. Because geometry may be read from disk many times during rendering, it is stored in a compact format in order to minimize time spent parsing the file. As with tiled texture maps, the time to create these files is negligible: a mesh of slightly more than 1 million primitives stored in a traditional format can be read, parsed, sorted, and written out in the new format in well under a minute.
- *Tessellated patches and subdivision surfaces.* In our system, patches are tessellated into triangles for the geometry cache. The only information that must be stored in memory until these triangles are generated is the bounding box of the patch or group of patches and, possibly, the control points.
- *Displacement mapping.* Our system supports displacement mapping. We subdivide input geometry into small triangles, the vertices of which are perturbed by a displacement shader. This can result in enormous amounts of geometry to be stored, since the triangles created must be the size of a pixel if they are not to be individually visible.
- *Procedurally generated geometry.* Relatively simple programs can be used to describe complex objects; the geometry needed to represent these objects can be stored in the geometry cache. The program need only be run again to regenerate the geometry if it is discarded and later needed again.

4.2 Geometry Cache Properties

Our geometry cache is organized around regular voxel grids termed *geometry grids.* Each collection of geometric objects is stored in its own grid, which tightly encloses it. The volume of the grid's voxels (and hence the amount of geometry in each voxel) determines the granularity of the cache, since the cache fills and discards all of the geometry in a voxel as a block. We have found that a few thousand triangles per voxel is a good level of granularity for caching. However, this is too coarse a granularity for ray intersection acceleration, so we insert another voxel grid, the *acceleration grid,* inside geometry grid voxels that hold more than a few hundred triangles. This two-level intersection acceleration scheme is similar to a method described by Jevans and Wyvill[12].

By construction, all geometry in a voxel occupies a contiguous block of memory independent of geometry in other voxels. In particular, triangles that span multiple voxels are stored independently in each of them. Thus, spatial locality in the three-dimensional space of the scene is tied to spatial locality in memory. If these two types of spatial locality were not coupled in this way, the cache would almost always perform poorly, since coherent access in three dimensional space would not generate coherent access in memory.

Memory management in the geometry cache is more complicated than it is in the texture cache. Whereas all texture tiles are the same size, each geometry voxel may require a different amount of space. These differences lead to repeated allocation and freeing of different-sized blocks of memory, which causes heap fragmentation. Before memory management was addressed in our system, fragmentation would often cause our renderer's size to double or triple after a few hours of execution. After replacing the system library's allocation routines with our own (based on segregated storage with bitmapped usage tables and no coalescing[23]), heap fragmentation was negligible. More recently, preliminary experiments suggest that Lea's allocator[15] also eliminates growth due to fragmentation.

The geometry cache could be implemented with virtual memory, with some loss of direct control of and information about the contents of the cache. Furthermore, the data being cached must still be organized to ensure coupled spatial locality as described above, and computation must still be reordered if the cache is to perform well.

5 Reordering Rendering Computation

These geometry and texture caching algorithms provide a framework for rendering large scenes. However, since cache misses are orders of magnitude more expensive then cache hits, we must find a way to minimize misses if overall system performance is to be acceptable. In order to ensure coherent access of the caches, we dynamically reorder ray-object intersection tests. Rather than evaluating ray trees in a fixed order, such as depth-first or breadth-first, all rays are placed on ray queues. The system chooses rays from the queues, simultaneously trying to minimize cache misses and advance computation toward completion. The degree to which we can minimize the number of times geometry must be added to the cache determines how efficiently we make use of the cache, and how well the system performs in the face of extreme complexity.

In order to perform this reordering, we would like each queued ray not to depend on the results or state of other queued rays. This independence implies that we must store with each ray all of the information needed to compute its contribution to the image; furthermore, the space occupied by this information must be minimized given the potentially large number of queued rays. Both of these goals can be achieved by decomposing the computation of outgoing radiance into a simple sum of weighted incoming radiances. To our knowledge, this decomposition was first used by Cleary et al.[2] to reduce storage and communication demands in a parallel ray tracing system.

5.1 Computation Decomposition

We can take advantage of the structure of the rendering equation[14] to decompose the rendering computation into parts that can be scheduled independently. When we sample the rendering equation to computing outgoing radiance at a point x in the direction ω_r, the result is:

$$L_o(x,\omega_r) = L_e(x,\omega_r) + \frac{1}{N}\sum_N f_r(x,\omega_i,\omega_r)\, L_i(x,\omega_i)\, \cos\theta_i, \tag{1}$$

where L_o is the outgoing radiance, L_e is the emitted radiance, f_r is the bidirectional reflectance distribution function (BRDF) at x, L_i is the radiance incoming from direction ω_i, and θ_i is the angle between ω_i and the surface normal at x. We can make use of the separability of (1) to decompose the computation in such a way as to make each queued ray independent from the others, and to minimize the amount of state stored with each queued ray. Given a ray r to be traced in direction ω_i, the weight of its contribution to the outgoing radiance at x is given by

$$w(x,\omega_i) = \frac{1}{N} f_r(x,\omega_i,\omega_r)\, \cos\theta_i. \tag{2}$$

If r, in turn, intersects a reflective surface at a point x', additional rays will be spawned from this point to determine the total radiance traveling along r. The contribution that one of these spawned rays r' will make to the outgoing radiance at x is simply the product of $w(x,\omega_i)$ and $w(x',\omega_i')$.

When an initial camera ray r_0 is spawned, we record a unit weight and its corresponding image plane location. When the intersection of r_0 is found and a secondary ray is spawned, we compute the total weight for the new ray as above, store this weight and the same image plane location, and repeat the process. The weight associated with a ray r thus represents the product of the weighted BRDF values of all the surfaces on the path from a point on r to the image-plane location. Once a ray intersects an emissive object, the emitted differential irradiance that it carries is multiplied by the ray's weight, and the result can be immediately added to the radiance stored at the ray's associated image-plane location.

This decomposition does introduce limitations. For example, because all of the information about the point being shaded is discarded once secondary rays are generated, adaptive sampling (of area light sources, for example) is not possible. Although more state could be stored with each ray, including information about the surface from which it originated, this would increase memory requirements. Furthermore, the scheduling algorithm's interest in deferring rays that will cause cache misses means that this state information might have to be stored for a great many points (until all of their spawned rays have been traced.)

For high resolution images with many samples per pixel, the storage needed to hold intermediate results for all of the image samples can be hundreds of megabytes. The result for each image sample is found in parts, and results are generated in an unknown order, so we write results to disk as they are computed rather than storing them in main memory. When rendering finishes, we make a number of passes through this file, accumulating sample values and filtering them into pixel values.

5.2 Ray Grouping

Given this decomposition of the illumination computation into pieces that can be easily reordered, we must find effective reordering techniques. A perfect scheduling system would cause each primitive to be added to the geometry cache only once. However, this is not generally possible for ray tracing algorithms, since there is a strict order relationship in ray trees: it is not possible to spawn secondary rays until we find the intersection positions of the rays that cause them to be spawned.

One early reordering approach we tried organizes rays with nearby origins into clusters. When a cluster of rays is traced, its rays are sorted by direction to increase their coherence, and each ray is then traced through the scene until an intersection is found. This technique is good at exploiting coherence in scenes where the majority of rays are spawned from a few locations, such as the eye and the light sources. However, it has two drawbacks. First, because each ray is traced to completion before starting another ray, this method fails to exploit coherence between rays in a beam as they move through the scene together. Secondly, this technique fails to exploit coherence that exists between rays that pass through the same region of space, but whose origins are not close together.

The approach we currently use is designed to account for the spatial coherence between all rays, including those whose origins are not close to each other. We divide the scene into another set of voxels, the *scheduling grid.* Associated with each scheduling voxel is a queue of the rays that are currently inside it and information about which of the geometry voxels overlap its extent. When a scheduling voxel is processed, each queued ray in it is tested for intersection with the geometry inside each overlapping geometry voxel. If an intersection is found, we perform shading calculations and calculate spawned rays, which are added to the ray queue. Otherwise, the ray is advanced to the next non-empty voxel it enters and is placed on that voxel's ray queue. Figure 4 summarizes the rendering process.

If the subset of the scene geometry that overlaps each scheduling voxel can fit into the geometry cache, the cache will not thrash while the rays in an individual voxel are being traced. Conversely, if a scheduling voxel contains too much geometry to fit into memory at once, the geometry cache will thrash (this is the analogue of trying to render a scene that cannot fit into memory without computation reordering). To account for variations in geometric complexity in different regions of the scene, the regular scheduling grid we use could be replaced with an adaptive spatial data structure, such as an octree.

To amortize the expense of cache misses over as much ray intersection work as possible, we attempt to defer tracing rays in voxels where there will be misses. We preferentially process scheduling voxels that have all of their corresponding geometry cached in

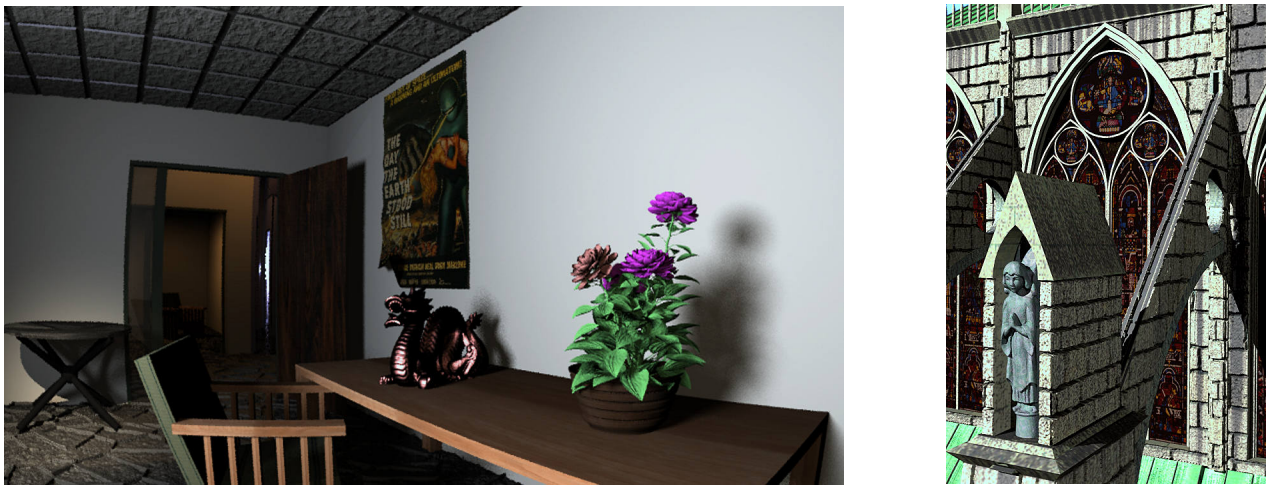

Figure 3: Indoor scene, with dense occlusion, and Cathedral scene.

```
Generate eye rays and place them in queues
While there are queued rays
        Choose a voxel to process
        For each ray in voxel
            Intersect the ray with the voxel's geometry
            If there is an intersection
                Run the surface shader and compute the BRDF
                Insert spawned rays into the voxel's queue
                If the surface is emissive
                    Store radiance contribution to the image
                Terminate the ray
            Else
                Advance the ray to the next voxel queue
```

Figure 4: Basic reordering algorithm.

memory, and accumulate rays in the queues of voxels that will incur geometry cache misses. This can lead to prohibitively many rays to store in memory, so we use a simple mechanism to manage the untraced rays. We store a limited number of rays in memory in a *ray cache*; excess rays are written to disk until needed for intersection calculations. When we trace the rays in a given voxel, those in the ray cache are traced first, and then those on disk are read and processed.

5.3 Voxel Scheduling

Given a set of scheduling voxels with rays in their queues, the scheduler must choose an order in which to process the voxels. It could simply loop through all of the voxels, processing the rays that are waiting in each of them until all rays have terminated. However, we can reduce cache misses and improve efficiency by more carefully choosing when to process each voxel.

We associate a cost value and a benefit value with each voxel. The cost is computed using a function that estimates how expensive it will be to process the rays in the voxel (for example, the cost should be high if the voxel encompasses a large amount of geometry, none of which is currently in memory). The benefit function estimates how much progress toward the completion of the computation will be made as a result of processing the voxel. For example, voxels with many rays in their queues have a higher benefit. Furthermore, voxels full of rays with large weights have a greater benefit than voxels full of rays with low weights, since the rays with large weights are likely to spawn a larger number new rays. The scheduler uses these values to choose voxels to work on by selecting the voxel with the highest ratio of benefit to cost.

Both of these functions are only approximations to the true cost and benefit of processing the voxel. It is difficult to estimate *a priori* how many cache misses will be caused by a group of rays in a voxel since the geometry caching algorithms add geometry to the cache lazily. If the rays in a voxel don't access some geometry, that geometry is never added to the cache. It is also difficult to estimate how many new rays will be spawned by a group of rays before they are actually traced, since the number and weights of the spawned rays depend on the reflectances and orientations of the surfaces the rays hit.

In our implementation, the cost of processing a voxel is based on an estimate of how much of the geometry in a voxel that will be accessed is already present in memory. If some but not all of the geometry in a voxel is in memory, we reduce its expected cost by 90% if we have already traced rays inside the voxel and no geometry that overlaps the voxel has been removed from the geometry cache in the meantime. This reduction keeps voxels with geometry that cannot be accessed by the rays currently passing through them from seeming to be more expensive than they actually are. The benefit is the product of the the number of rays in the voxel and the sum of their weighted contributions to the final image. Other possible cost and benefit functions could account for a user interested in seeing results for one part of an image as soon as possible, or for minimizing the number of queued rays.

6 Results

We have implemented our caching and reordering algorithms as part of a ray tracing based rendering system. Like most ray tracers, our renderer supports several mechanisms for accurately simulating light transport. However, because it was specifically designed to manage scene complexity, our system also supports features usually only found in scanline-based systems, including displacement mapping, the ability to render large geometric databases and NURBS, and programmable shaders that use large numbers of texture maps.

We conducted a number of experiments to determine how our algorithms performed on complex scenes that would be difficult to renderer using traditional approaches. Our experiments were performed on a lightly loaded 190 MHz MIPS R10000 processor with 1GB of memory. So that our results would be indicative of perfor-

mance on machines with less memory (where there wouldn't be excess memory that the operating system could use to buffer I/O), we disabled I/O buffering for our tests. In practice, performance would be improved if buffering was permitted. Running time was measured in wall clock time, which ensured that time spent waiting for I/O was included. Heap fragmentation caused by the large amount of memory allocation and freeing necessary to manage the geometry cache was accounted for by basing memory use statistics on the total size of the process. Although our algorithms make use of disk I/O in a number of ways, all of our tests were performed using a single disk; using more disks, a RAID system, or asynchronous prefetching of data would further improve performance.

Each of our test scenes would be considered complex by current standards, requiring between 431MB and 1.9GB of memory to store in their entirety. The scenes cover a variety of common situations, including an indoor scene that is densely occluded and an outdoor scene with little occlusion. We use no instantiation of geometry in these scenes; all objects are distinct and managed individually.

- The first test scene is a Cathedral model that was used to demonstrate algorithms simulating the effects of weathering due to water flowing over surfaces[5] (Figure 3). The base Cathedral model itself consists of only 11,000 triangles, but displacement mapping is used to add a great deal more geometric detail. There is also a statue modeled using 36,000 polygons and three gargoyle models comprised of roughly 20,000 polygons each. The surface shaders access four or five different texture maps at each point being shaded, and the displacement shader accesses one texture map. After displacement mapping, total geometric complexity is 5.1 million primitives when the image is rendered at a resolution of 576 by 864 pixels. All tests were performed at this resolution, using four samples per pixel[1]. A total of 1,495 texture maps that use 116MB of disk space are used to store the displacement maps and the results of the flow simulations over the surface (wetness, the thickness of dirt deposited on it, and so on). The illumination in this scene is simple, consisting of one shadow-casting light source and one fill light.
- Next, we modeled a room in a small office building (Figure 3). The building has two floors, each with four offices; the floors are connected by a staircase. The floors and ceilings are modeled using procedural displacement shaders, and each of the rooms is filled with complex models, including dense meshes from Cyberware scans and plant models from an organic modeling program. Total geometric complexity is 46.4 million primitives, which would require approximately 1.9GB of memory to store in memory at once. Most of the light in the main room is due to to sunlight shining through a window, modeled as a large area light source. The hallway and nearby offices are further illuminated by light sources in the ceiling. The reflective column in the hallway causes light from rooms not directly visible to the camera to strike the image, and forces additional scene geometry to be be brought into memory. We rendered this scene at a resolution of 672 by 384 pixels.
- Lastly, we constructed an outdoor test scene consisting of group of trees by a lake (Figure 2). There are four tree models comprised of 400,000 to 3.3 million triangles. Even the individual leaves of the trees are modeled explicitly; nothing is instantiated. The terrain and lake were created using displacement mapping, resulting in a total scene complexity of over 9.6 million primitives, which would require approximately 440MB to store in memory in their entirety. Almost all of the geometry in this scene is visible from the eye. The only direct lighting in the scene comes from the sun, which is back-lighting the trees. Most of the illumination is due to indirect lighting from the sky and indirect reflections from other surfaces in the scene. For our tests, we rendered the scene at 677 by 288 pixels.

[1]The test scenes were rendered at higher resolution with more samples per pixel for the color plates. Our scheduling and reordering techniques performed as well when rendering the plates as when used for the test cases.

6.1 Caching and Lazy Evaluation

We first tested the impact of lazy loading of scene data on both running time and memory use. We started with the Cathedral, a scene where most of the modeled geometry and texture was visible. The scene was rendered twice: once with caches of unlimited size that were filled lazily, and once with all texture and geometric data read into memory at startup time (Figure 5). As shown in the table, both running time and memory use decreased when scene data is brought into memory lazily. Memory use was reduced by 22% using lazy loading, indicating that approximately 22% of the scene data was never accessed. Given that this test scene was chosen as a case where we would expect little improvement, this was a particularly encouraging result. To investigate further, we rendered the office scene to see how well lazy loading of data worked in a scene exhibiting dense occlusion. For this test, we computed only direct illumination from the light sources. We were unable to read the entire database for this scene into memory at startup time, due to its large size. However, when we lazily read the data into memory only 18% of the memory that would have been necessary to store the entire scene was used.

	Cathedral	Cathedral Lazy	Indoor Lazy
Running Time	163.4 min	156.9 min	35.3 min
Memory Use	431MB	337MB	316MB
% Texture Accessed	100%	49.6%	100%
% Geometry Accessed	100%	81.6%	17.8%

Figure 5: The overhead introduced by lazy loading of data is very small, and doing so improves performance even for scenes with little occlusion. For the scene with dense occlusion, lazy evaluation makes it possible to render a scene that could not fit into memory otherwise.

Using the Cathedral scene, we investigated how well the geometry cache performs if standard depth-first ray tracing is used in a scene that spawns a limited number of illumination rays (Figure 6). We rendered the scene using different geometry cache sizes, and recorded running time and geometry cache hit rates. We found that when geometry caching was used we could efficiently render the scene using only 90MB of memory, which is 21% of the space needed to store the scene in its entirety, or 27% of the space needed to store all of the data that was ever accessed.

Next, we investigated the performance of the texture cache using the Cathedral scene. We rendered the scene a number of times, each with a different cache size, in order to determine how hit rate and running time were affected. To increase the number of texture accesses for displacement map lookups, we used a moderately sized geometry cache for this test, ensuring that some of the displacement mapped geometry would be discarded from the cache and later recreated. We found that the texture cache performed extremely well. Even with only 32 tiles in memory, the cache hit rate was was 99.90%, and running time was 179 minutes, compared to 177 minutes when using a cache of unlimited size. A cache of 32 tiles uses 128kB of memory, which is 0.1% of all of the texture data in the scene.

Maximum Memory Use	Running Time	% of time with unlimited cache
90MB	184.3 min	117%
100MB	177.7 min	113%
300MB	156.9 min	100%

Figure 6: Time to render the Cathedral scene with varying limits on memory use. Although performance degraded slowly as maximum memory use was reduced from 300MB to 90MB, performance degrades catastrophically when it is smaller than 90MB. The working set for this scene is between 80 and 90MB–we were unable to complete any runs with maximum memory use of 80MB or below.

6.2 Scheduling and Reordering

We used the lake scene to test the performance of the reordering algorithms. The scene was rendered with Monte Carlo path tracing and no limit to the length of each path; instead, rays with low contributions were terminated probabilistically with Russian Roulette. A ray cache of 100,000 rays was used, representing 6% of the total number of rays traced. As shown in Figure 7, rendering the lake scene with global illumination algorithms and a small geometry cache was feasible only if computation reordering was performed. If reordering was not used, storing any less than 80% of the scene in memory caused running time to increase rapidly, as rendering was dominated by cache misses. Using reordering, we were able to render the lake scene using a cache of only 10% of the total size of the scene.

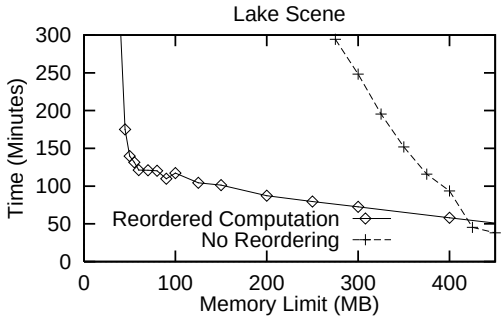

Figure 7: Running time for rendering the lake scene as the maximum amount of memory used is varied. When the cache is large enough to hold the entire scene database, the difference in time between the reordered and depth-first computation represents the overhead introduced by reordering. As can be seen, this overhead is not excessive. When maximum memory use is reduced to a small percentage of scene complexity, the reordered algorithm's performance allows us to efficiently render the scene using far less memory.

I/O performance had a greater impact on running time as we decreased cache sizes, but only significantly affected running time when the cache was thrashing. When the lake scene was rendered with all of the geometry stored in memory, the renderer performed 120MB of I/O to read models from disk. CPU utilization was 96%, indicating that little time was spent waiting for I/O. When we limited the total rendering memory to 50MB, the renderer performed 938MB of I/O managing models on disk, though CPU utilization was still 93%. For both of these tests, less than 70MB of I/O for the ray queues was done. Without computation reordering, much more I/O is performed: when we rendered the lake scene without reordering using a 325MB geometry cache, 2.1GB of data was read from disk to satisfy misses in the thrashing geometry cache.

Finally, we gathered statistics to gain insight into the interplay between our scheduling algorithm and the geometry cache. Figure

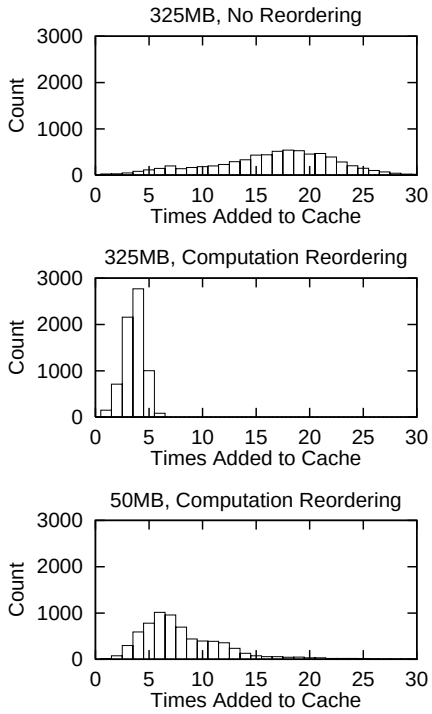

Figure 8: Histogram of the number of times each voxel of geometry was added to the geometry cache for the lake scene. When the scene is rendered without computation reordering with a cache of 325MB (left), geometry is added to the cache many more times than when computation is reordered with a cache of the same size (middle). When the cache size is limited to just 50MB and computation is reordered (right), cache performance is significantly better than the 325MB cache without reordering.

8 is a histogram of how many times each voxel of geometry was added to the geometry cache when the lake scene was rendered. Depth-first ray tracing led to poor performance with a medium-sized cache, as illustrated by the fact that most voxels were added to the cache between fifteen and twenty times. However, if computation reordering was used, the number of times each voxel was added to the cache was greatly reduced. With a very small cache of 50MB and computation reordering, voxels were inserted into the cache an average of approximately eight times. This compares well to the three complete passes through the database that systems such as REYES would make in rendering the shadow map, water reflection map, and final image for this scene. Of course, REYES would be unable to accurately reproduce the reflections in the water, or other effects due to indirect illumination.

7 Summary and Conclusion

We have presented a set of algorithms that improve locality of data storage and locality of data reference for rendering algorithms that use ray tracing. We have implemented these algorithms as part of a rendering system that supports programmable shading, large amounts of texture, displacement mapping and global illumination. These algorithms have enabled us to render scenes with far greater geometric, surface, and illumination complexity than previously possible on a machine of given capacity.

Our algorithms decompose the rendering computation into separate parts that can be worked on in any order, while minimizing the amount of state that needs to be stored for the rays waiting to be traced. Our decomposition of rendering computation is based on forward ray tracing; determining how to incorporate algorithms

such as bidirectional path tracing[22] is an interesting future challenge. Other questions to examine include how performance would be affected by other scheduling algorithms, and whether it is possible to greatly reduce the number of queued rays without causing performance to degrade.

Another area of future work is applying these techniques to parallel and hardware accelerated ray tracing. Management of shared texture and geometry caches on distributed shared memory architectures offers challenges of coordination of cache access and modification, scheduling rays over groups of processors, and effective management of the increased I/O demands that multiple processors would generate. Furthermore, by presenting a framework for ray tracing without holding the entire scene in memory (as Z-buffering does for scan-conversion), this works suggests a new approach to the long-elusive goal of effective hardware accelerated ray tracing.

Improving performance by gathering related data together with related computation is a powerful technique, as evidenced by its central position in computer architecture today; we believe that it will lead to further benefits in other areas of computer graphics.

8 Acknowledgments

Special thanks to Jim Kajiya for sharing a copy of his whitepaper, and to Radomír Měch and Przemyslaw Prusinkiewicz for sharing their wonderfully complex Chestnut tree model which was used in Figure 2. John Gerth was a source of valuable discussions and suggestions. Tony Apodaca, Tom Duff, Larry Gritz, and Mark VanDerWettering in the RenderMan group at Pixar explained the intricacies of REYES. Julie Dorsey and Hans Pedersen kindly shared the Cathedral model used in Figure 3. James Davis wrote the code to tessellate NURBS for the geometry cache, and Chase Garfinkle provided much-needed editing help at the last minute. This research was supported by Silicon Graphics, Inc., NSF contract number CCR-9508579-001, and DARPA contracts DABT63-96-C-0084-P00002 and DABT63-95-C-0085-P00006.

References

[1] Tony Apodaca. Personal communication. 1996.

[2] J. G. Cleary, B. M. Wyvill, G. M. Birtwistle, and R. Vatti. Multiprocessor ray tracing. *Computer Graphics Forum*, 5(1):3–12, March 1986.

[3] Robert L. Cook. Shade trees. In Hank Christiansen, editor, *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 223–231, July 1984.

[4] Robert L. Cook, Loren Carpenter, and Edwin Catmull. The Reyes image rendering architecture. In Maureen C. Stone, editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, pages 95–102, July 1987.

[5] Julie Dorsey, Hans Køhling Pedersen, and Pat Hanrahan. Flow and changes in appearance. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, pages 411–420. Addison Wesley, August 1996.

[6] Thomas A. Funkhouser. Coarse-grained parallelism for hierarchical radiosity using group iterative methods. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, pages 343–352. Addison Wesley, August 1996.

[7] Stuart Green. *Parallel Processing for Computer Graphics*. Research Monographics in Parallel and Distributed Computing. The MIT Press, 1991.

[8] Ned Greene and Michael Kass. Hierarchical Z-buffer visibility. In *Computer Graphics (SIGGRAPH '93 Proceedings)*, pages 231–240, August 1993.

[9] Eric Haines and John Wallace. Shaft culling for efficient ray-traced radiosity. In *Eurographics Workshop on Rendering*, 1991.

[10] Pat Hanrahan and Jim Lawson. A language for shading and lighting calculations. In Forest Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 289–298, August 1990.

[11] Frederik W. Jansen and Alan Chalmers. Realism in real time? In Michael F. Cohen, Claude Puech, and Francois Sillion, editors, *Fourth Eurographics Workshop on Rendering*, pages 27–46. Eurographics, June 1993.

[12] David Jevans and Brian Wyvill. Adaptive voxel subdivision for ray tracing. In *Proceedings of Graphics Interface '89*, pages 164–72, Toronto, Ontario, June 1989. Canadian Information Processing Society.

[13] James Kajiya. A ray tracing arthitecture. unpublished manuscript, 1991.

[14] James T. Kajiya. The rendering equation. In David C. Evans and Russell J. Athay, editors, *Computer Graphics (SIGGRAPH '86 Proceedings)*, volume 20, pages 143–150, August 1986.

[15] Doug Lea. A memory allocator. Available on the web at http://g.oswego.edu/dl/html/malloc.html, 1996.

[16] Darwyn R. Peachey. Texture on demand. unpublished manuscript, 1990.

[17] Matt Pharr and Pat Hanrahan. Geometry caching for ray tracing displacement maps. In Xavier Pueyo and Peter Schröder, editors, *Eurographics Workshop on Rendering*, pages 31–40, 1996.

[18] E. Reinhard and F. W. Jansen. Rendering large scenes using parallel ray tracing. In A. G. Chalmers and F. W. Jansen, editors, *First Eurographics Workshop of Parallel Graphics and Visualization*, pages 67–80, September 1996.

[19] Ivan E. Sutherland, Robert F. Sproull, and R. A. Schumacker. A characterization of ten hidden-surface algorithms. *Computing Surveys*, 6(1):1–55, March 1974.

[20] Seth Teller, Celeste Fowler, Thomas Funkhouser, and Pat Hanrahan. Partitioning and ordering large radiosity computations. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94*, pages 443–450, July 1994.

[21] Jay Torborg and Jim Kajiya. Talisman: Commodity Realtime 3D graphics for the PC. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, pages 353–364. Addison Wesley, August 1996.

[22] Eric Veach and Leonidas Guibas. Bidirectional estimators for light transport. In *Fifth Eurographics Workshop on Rendering*, pages 147–162, Darmstadt, Germany, June 1994.

[23] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *International Workshop on Memory Management*, September 1995. held in Kinross, Scotland, UK.