Ray Tracing Large Distributed Datasets Using Ray Caches

by

Christopher Little

A thesis submitted in partial fulfillment

of the requirements for the degree of

Master of Science

in

The Faculty of Graduate Studies

Computer Science

University of Ontario Institute of Technology

Supervisors: Dr. Mark Green and Dr. Faisal Qureshi

November 2011

Abstract

Most large scale simulations now produce datasets that can be significantly larger than can typically be stored in memory on a visualization system. Visualization algorithms then become ineffective and stall since the data must be paged to disk. Recently, in-situ visualization has received renewed attention for visualizing large datasets that are distributed among many processors during a simulation. It takes advantage of the fact that the full dataset is already in main memory, distributed among multiple processors. Visualization in this environment then requires communication which can be more expensive than disk access. The goal of this thesis was to develop an in-situ visualization technique using ray tracing that employs ray caches to reduce communication overhead. Ray caches attempt to replace a communication operation with a less expensive cache search operation. A prototype implemented on Sharcnet shows ray caching can significantly improve overall performance at a small cost to image quality.

Acknowledgements

First I would like to thank my supervisors, Dr. Mark Green and Dr. Faisal Qureshi, for their guidance, insight and invaluable experience. I also would like to thank my fellow graduate students. I would not be where I am now without their support, encouragement and late night work and discussions. I want to thank Dr. Christopher Collins and Dr. Lennaert van Veen for evaluating and correcting my work. Lastly I want to thank my family, friends and my girlfriend Kayla for always encouraging me when I struggled.

Contents

List of Figures

List of Tables

Listings

Abbreviations

BVH	Bounding Volume Hierarchy
BSP	Binary Space Partition
SIMD	Single Instruction Multiple Data
GPU	Graphics Processing Unit
GPGPU	General Purpose computing on Graphics Processing Unit
RGB	Red Green Blue (In reference to a red, green and blue pixel value)
FIFO	First In First Out

Chapter 1

Introduction

Current high performance computers are capable of producing extremely large datasets through large scale simulations and real-world data recording. Traditionally scientific visualization of these datasets has been viewed as a post processing step where the data saved during computation is sent to an offline visualization system for rendering. Due to the increasing size of these datasets, it is difficult to transfer and store them for visualization. Researchers have been turning to alternate methods, such as in situ visualization where the visualization is produced in parallel with the simulation. Although this is not a particularly new approach, it has not received a lot of attention until recently [29].

Many scientific computations are performed on grids, where each processor within a cluster is responsible for a small portion of the dataset making up the grid. Typically a ray tracing algorithm is employed to render these kinds of datasets due to its high quality and accuracy. Ray tracing is also beneficial as it can directly sample the data structures that are used by most simulation code. However, since the dataset is distributed, it is difficult to facilitate ray intersection calculations without copying the dataset between processors. Most previous methods have taken the approach of caching portions of the dataset from each processor [5, 20], or rendering each portion of the dataset individually and then compositing [2, 29].

1.1 Contributions

This work describes a new technique for in-situ visualization based on distributed out-of-core ray tracing. Rather than copying data between processors, this approach instead sends and receives rays between the processors as the rays traverse each processor and search for an intersection. Communication of the rays is performed using a message passing paradigm supplemented by batching to alleviate network congestion. A prototype implementation is also described and demonstrated on a cluster of up to 125 processors.

Performance of this system is highly dependent on both the ray tracing algorithm and, more importantly, on the expense of communication. This work also describes the implementation of ray caches which are placed at the boundaries between neighbouring processors. Ray caches track rays that pass from one processor to another and store their colour values. This allows other rays that pass through the cache to retrieve previously computed ray colour values and thus prevent an expensive communication operation.

The current implementation of the distributed out-of-core ray tracing system supplemented with ray caches is evaluated for performance, image quality and scalability. The results show that overall performance can be improved significantly through the use of ray caches, and at a small cost to image quality, which demonstrates the validity of the techniques described in this work. Scalability, however, is clearly the most significant drawback of the current implementation.

1.2 Thesis Overview

This thesis begins by covering previous work into ray tracing, out-of-core rendering and large-scale visualization in Section 2. An overview of how the distributed out-of-core ray tracer is to function and how ray caches are implemented is discussed in Chapter 3. An initial implementation is described in detail, including test scene generation, ray intersection calculation, ray cache searching, and ray distribution, in Chapter 4. The results of the naive and ray cache supplemented rendering are discussed in Chapter 5.

Chapter 2

Literature Review

This chapter provides an overview of several areas of research that have stemmed from ray tracing and visualization. Each section discusses some of the major works that have formed the basis for the work presented in this thesis.

2.1 Ray Tracing Algorithm

Ray tracing can simply be described as an image rendering technique. It is based on the principles of ray optics where a ray of light can be followed from source to surface. The light ray may be absorbed, reflected to another surface or refracted as it travels through the surface. In real world situations these rays represent light energy or photons emanating from a light source. These photons can be captured at any point in space to estimate the intensity of light that reaches that point. Film and digital cameras take advantage of this by absorbing photons that impact the surface of the film or image sensor. The number of photons collected at the image sensor can be used to measure light intensity and determine pixel intensities for the final image. In much the same way, ray tracing seeks to simulate rays of light moving through a 3D scene and capturing or displaying those rays that intersect a planar grid of pixels.

A distinction exists between the notion of ray tracing and ray casting. The goal of ray tracing is to create an image by simulating light moving throughout a 3D scene. Ray casting on the other hand is used in many applications to calculate ray-surface intersections. The earliest use of this appears in the work by Appel [4] for the purpose of shading 3D wireframe models for printing via a digital plotter. His initial tests were based on generating randomized rays from a light source and checking for intersections with the planar surfaces of the wireframe mesh. Some techniques were developed to improve this process, such as limiting ray directions to those more likely to impact a specific surface. The shading method starts with a 2D projection of the wireframe model onto the image plane. Inside the boundaries of the projected vertices a set of points is selected for shading. A line of sight from the observer's eye position to each of these points is calculated. Each plane in the 3D model is then checked for intersections with the line of sight to determine the closest point that is visible to the observer. The line of sight, the location of the light source and the orientation of the surface can then be used to estimate the light intensity for the final image. Appel's work established the benefit of tracing rays originating from the eye that pass through the image plane rather than originating at the light source.

With advances in computer display technology throughout the 1970's alternative shading methods were developed. Two of the most notable methods were demonstrated by Gouraud [9] and Phong [16]. Their shading and illumination models provided good smooth surface shading at a relatively low computational cost. A shortcoming of these models is the lack of consideration for shadowing and light propagation due to reflection.

Unsatisfied with the lack of realism in these models, a hidden surface algorithm and a more accurate illumination model that could take advantage of global scene data was demonstrated by Whitted [26]. Whitted borrows from Appel's technique where a ray is traced from the observer's eye position. Where Appel had predefined points over the surface of a polygon that each ray points to, Whitted used a grid of pixels transformed into scene coordinates. Each ray cast through this grid now represents the visible light direction for the observer at that pixel. Each ray can now be checked for the closest surface intersection to perform hidden surface removal. Shading techniques such as Gouraud or Phong shading can now be easily applied at the surface intersection point.

This alone would not produce results much different from what was already possible, but the method also allows reflection and refraction ray directions to be calculated using the principles developed in conventional ray optics. A new reflection or refraction ray can be calculated and recursively traced to the next nearest intersection. Each subsequent intersection is shaded independently and contributes some illumination to the parent ray. The result of this method is very accurate simulation of specular reflection and refraction that is very difficult to achieve with other rendering methods.

Appel noted in his work that ray casting and surface intersection calculations are computationally very expensive. The added recursive complexity of Whitted's algorithm produces images with considerably higher quality but at a massive cost to performance. Whitted examined this impact in his performance measurements and found that between 75-95% of run time was spent performing ray intersection calculations. Ray tracer performance can therefore be improved significantly by any improvements in the performance of calculating intersections. For this Whitted offered the suggestion that simple bounding shapes could be placed around more complex sets of geometry. A ray could then be tested for an intersection with the bounding shape before finding an exact intersection with the contained geometry.

2.2 Ray Tracer Acceleration and Ray Coherence

Many strategies have been employed to accelerate ray tracing. A survey by Arvo and Kirk in [7] classifies these acceleration techniques into three categories: *faster intersections*, *generalized rays* and *fewer rays.* Faster intersections refers to methods that reduce the average cost of calculating an intersection or reducing the total number of intersection calculations per ray. Generalized rays refers to grouping rays to prevent repeating similar ray calculations. And lastly, fewer rays refers to techniques that reduce the total number of rays that are cast to produce an image.

The last category, "fewer rays", is less broad than the other two, however one example, subsampling, is used extensively in some applications. Subsampling operates similarly to the anti-aliasing method described by Whitted [26], however rays are cast at multiple pixel intervals rather than within a single pixel, for example a ray is cast through the four corners of each 4x4 block of pixels. If the four rays return colour values that are reasonably similar, then the 16 individual pixel colours are simply interpolated from the four corner samples, thereby reducing 16 rays to only 4. The drawback to this method occurs when some objects that appear in an area smaller than 4x4 pixels may be entirely missed by the four sample rays, thus omitting it from the final image.

2.2.1 Faster and Fewer Intersections

Whitted suggested the use of bounding volumes - shapes enclosing more complex objects - to prevent unneeded ray intersection computations [26]. He initially suggested surrounding each object with a sphere due to its simplicity. Substituting a large number of ray intersection calculations with a single sphere intersection test significantly reduces the overall number of computations required for each ray. However this method does not improve on the linear time complexity of exhaustive ray tracing and suffers linear performance degradation as scene size increases. Later work by Rubin and Whitted [18] introduced the concept of bounding volume hierarchies (BVH) for ray tracing. Rather than surrounding each object individually, groups of bounding volumes are surrounded by larger bounding volumes to form a tree data structure. Large sections of the scene can be ignored as a ray traverses deeper into the tree until reaching a set of leaf nodes that are likely to produce an intersection. This effectively reduces the computational complexity of ray tracing from linear to logarithmic with respect to scene size. Others have attempted to improve the BVH further by using different shapes such as the work by Kay [13] that uses arbitrary shapes to tightly surround a more complex object.

A similar technique demonstrated by Fuchs et. al. [6], known as binary space partition (BSP), groups primitives based on their position in scene space. The BSP tree structure is built by placing a bounding box around the extent of the model or scene and then splitting the box into two smaller bounding shapes. The plane that divides the space can be any orientation and it is that orientation that determines the properties of the resulting BSP tree. The "pure" BSP tree partitions space by placing division planes such that all objects are evenly distributed on either side of the partition. One particular BSP technique that is used extensively for sub-dividing a 3D scene is the octree, which hierarchically divides a region evenly into 8 sub-regions that are then further sub-divided based on the geometry found within the sub-region. Another similar structure is the kd-tree which uses axis aligned dividing planes. The scene is sub-divided along each axis in-turn, where one dividing plane placed on a particular axis creates two sub-regions on either side that contain an approximately equal amount of scene data. A new dividing plane is placed on a different axis for each sub-region, again ensuring that an equal amount of data is contained in each new sub-region. For each of these structures, a ray samples the scene by traversing the hierarchy either incrementally, progressing between neighbouring partitions, or top-down, traversing each child of the intersected parent tree node.

2.2.2 Generalized Rays

Typically in recursive ray tracers, ray directions will quickly diverge when reflected or refracted by curved surfaces. However there are several techniques that have been able to take advantage of ray coherence especially for primary rays. Ray coherence refers to the occurrence of multiple rays following a very similar path through a scene or, more specifically, encountering the same nodes in the resulting ray tree. Beam tracing [10] and pencil tracing [19] take advantage of this by replacing individual ray intersection calculations with fewer area surface calculations.

The beam or pencil can be viewed as a pyramid-shaped polygon that encompasses the frustum of the rays it contains. Reshetov et. al [17] showed that combining this technique with the BSP or BVH structures discussed above allows entire groups of rays to traverse the BSP or BVH as a single entity. Once the group encounters a bounding volume that is smaller than the bounds of the frustum of the ray group, or is a leaf node in the hierarchy, the group of rays can be split to more accurately capture the divergence of the individual rays. More recent work by Wald et. al. [22, 21] applies this technique in a realtime ray tracing implementation. In this case grouping rays is useful for performing ray intersections using SIMD operations. All the rays in the group then traverse the BSP or BVH simultaneously in parallel rather than incurring the same computation multiple times for each ray. Even as rays diverge, their work shows that there are still significant performance gains.

A technique described by Yoon et. al. in [28] attempts to reduce the complexity of calculating ray-object intersections by adapting the level of detail (LOD) technique often used to accelerate rasterization. Their technique, which they call R-LOD, also takes advantage of a kd-tree bounding volume hierarchy. Rather than simplifying the geometry of the entire model, the R-LOD generates a plane to represent the geometry contained in the bounding box of a node in the kd-tree, typically a leaf node. This plane is defined by a point and a normal vector, which are assigned based on the curvature of the contained geometry. Sections of the model previously made up of many individual polygons can now be represented as a single plane, and any ray passing through these sections only require a single plane intersection calculation, rather than many polygon intersection calculations. They show that building an R-LOD as part of a preprocess for a particular model can take minutes or hours depending on the model. However, they also show that using the R-LOD during ray tracing can significantly reduce render time as well as reduce overall memory requirements, replacing multiple polygons with a single plane, at the expense of surface discontinuity artifacts in the rendered image.

Ray coherence has also been shown to be useful in applications that require super sampling. There are many optical phenomena that ray tracing can emulate by super sampling an area with randomized techniques, such as anti-aliasing using distribution ray tracing or diffuse interreflection using Monte Carlo ray tracing. Although this results in a higher quality image, the number of rays that are cast grows exponentially. Taking advantage of ray coherence, this exponential growth of rays can be alleviated or avoided entirely by casting fewer generalized rays.

Igehy [11] uses this principle to demonstrate a technique for texture filtering. When a ray intersects a textured surface the intersection point is typically mapped to a position in a 2D texture. If only a single texture sample is acquired then the resulting textured surface often becomes blurry or "pixelated", while acquiring multiple texture samples would typically require casting additional rays. Instead, the derivative of the ray with respect to image coordinates, which Igehy calls ray differentials, can be employed to produce an estimate of the area on a surface between a ray and its neighbouring rays. This area is called the ray footprint, referring to an area of the surface that would likely contribute to the resulting colour value for the ray. This allows an area of the texture to be sampled and averaged, rather than sampling a single point, to produce a better colour approximation in the final image without the added expense of casting multiple rays.

A similar idea forms the basis for a global illumination sampling technique described by Ward and Heckbert [24]. Global illumination typically requires casting many randomized rays through a hemisphere centred at a point on a surface. In much the same way as Igehy, their method computes gradients across an area of the surface for which illumination samples should be interpolated. This produces much smoother global illumination effects, such as diffuse interreflection, without the need for casting additional rays.

2.3 Radiance Caching

There are many situations in graphics and rendering where storing previously calculated radiance values can be beneficial. The general idea is to store information about the scene such as light traversal or previously rendered pixels so that they can be reused. Rather than computing a new sample for each image pixel or ray intersection, the cached information can be reused to interpolate a new value at a much lower performance cost.

Storing rays and radiance values has been shown to be very useful for 3D image reconstruction by Gortler [8] and Levoy [15]. The primary goal of this work is to reconstruct a 3D scene from a series of real world or synthetic source images. The source images contain not only colour values, but also information about light direction and distance which are stored as rays. Aggregating all the rays from the series of source images together produces a light field. Since the storage size and complexity of this field is quite high, it is converted to a 4D plenoptic function. Each ray in this function is represented as two 2D points, or U-V coordinates, where it intersects two parallel planes. Inherently, the stored rays each correspond to a camera position and view direction, depending on the source image from which they were produced.

To generate an image of the scene from a new view direction or to create a smooth animated view of the scene, new rays must be interpolated based on the new camera position/direction and the rays stored in the plenoptic function. In short, to generate a new image, a ray is cast for each pixel, then the plenoptic function is sampled to find similar existing rays and interpolate or average their colour values. Since each new ray is entirely dependent on existing rays to produce a colour value, the resulting image quality is highly dependent on the quality and distribution of the source images. Therefore, if there are large gaps in camera position or view direction between source images, or the new camera position or view direction differ too greatly from the source images, a new image will become completely incoherent. However, when source image density is high enough, this method has been shown to produce very high quality images at real-time performance.

Walter et. al. demonstrate a similar technique for use in real time ray tracing of static scenes, which they called the render cache [23]. They initially use a brute force ray tracing algorithm to produce one complete image or frame of the animation. Along with the final pixel colour values, the render cache also stores the camera position, view direction and the distance along each ray where an intersection occurred for each pixel. In subsequent frames, when the camera position or view direction changes, the data stored from the previous frame can be reprojected, similar to the z-buffer method used in rasterization. Naturally this creates gaps where previous pixels are reprojected onto the same pixel in the new frame. In this case it is a simple matter of casting another ray to calculate the missing pixel colour. For situations where the camera position or view direction changes gradually very few new rays need to be cast for each new frame.

Larson and Simmons [14] demonstrate the holodeck data structure which they use to achieve real-time walk-through rendering of scenes with global illumination. The goal of the holodeck is to provide a reusable ray cache in place of complex geometry that can be sampled from an arbitrary view point. It is permanently stored externally to the rendering system on a server that handles caching and retrieving ray radiance values. It is built by placing a bounding box around the extent of the scene geometry and dividing the planes on each side of the box into a grid of faces. Any ray that passes through the bounding box must therefore pass through at least two of these faces. Accordingly, the holodeck stores an individual group of rays for each pair of faces that they pass through. A method similar to that described by Gortler [8] and Levoy [15] is then used to generate a new image from the holodeck. If the ray tracer determines that a ray passes through the holodeck bounds, a request is sent to the holodeck server to produce a colour value from previously cached rays. The server first determines which two faces the new ray passes through to determine the correct groups of rays that should be sampled. The colour values of each ray in the group are then averaged based on their similarity to the new ray and returned to the ray tracer for display.

As with Gortler and Levoy above, the resulting image quality when samples are taken from the holodeck is highly dependent on the distribution of rays that it has stored. When the holodeck is first created it contains no samples and therefore any ray passing through its bounds must perform a regular ray trace to produce a colour value that can be stored. However, since the holodeck is managed by an external server, cached samples can be stored between rendering sessions and reused as long as the contained geometry remains unchanged. This initially results in slow performance and poor image quality but, as demonstrated by Larson and Simmons, as more samples are added over time the holodeck is able to retrieve reasonable colour values significantly faster than performing the ray intersection calculations and therefore produces an image much faster.

Possibly the most prominent application of radiance caching is the irradiance cache introduced by Ward et. al. [25]. Even today, diffuse interreflection is extremely expensive to perform in ray tracing. A brute force method of approximating this would be to cast many randomized reflection rays each time a ray intersects a surface. Obviously this scale of ray propagation quickly becomes prohibitive for most ray tracing applications. Instead, Ward et. al. proposed that a smaller set of diffuse illuminance values could be computed across all surfaces and stored in a cache, and later reused for subsequent rays. The cache is initially empty when ray tracing starts and a diffuse interreflection value must initially be computed using the brute force method above for each ray. Once this value is computed it is stored in the irradiance cache as a 3D point, corresponding to the ray-surface intersection point, and an illuminance or colour value. If a subsequent ray is determined to have intersected a surface at a nearby point, the cached illuminance value can be retrieved rather than computing a new value using the brute force method. This work was later extended to include other complex phenomena, such as caustics, in the form of the photon map [12].

2.4 Large-Scale and Parallel Visualization

The ray tracing acceleration and radiance caching methods described previously in this chapter are only effective when the scene or dataset to be rendered is stored entirely in main system memory. If the model is too large, attempts by the ray tracer to access certain portions of the scene will cause a page fault. In these cases a typical ray tracer would be forced to stall while the required data is loaded from disk into main memory. Regardless of whether any of the ray intersection acceleration methods above are employed, any performance gain is eliminated since the ray tracer must wait for the geometry to be loaded before proceeding. This is often the case when rendering large CAD models or datasets from large-scale simulations that can be many gigabytes in size. To solve this researchers have concentrated on two main strategies; adaptive memory management and distributed parallel rendering.

One memory management technique that has become common-place is to treat main memory as a cache for loading chunks of a large scene stored on disk. This should not be confused with the type of caching described in the previous section where the cache stores ray and radiance values. An example of a large geometric scene is the schematic model of the Boeing 777 jet (see [20]). Wald et. al. demonstrate a system for rendering this model in real-time that takes advantage of 64-bit memory addressing and the extensive virtual memory capabilities of most operating systems [5, 20].

Initially a pre-process organizes the full model into a kd-tree structure in much the same way as discussed above [20]. The geometry contained at each leaf node of the tree are grouped together contiguously in memory, thereby creating pages in memory containing smaller chunks of the scene. Now, when a ray enters one or more leaf nodes, the ray tracer need only load the corresponding pages from disk once, rather than reading in each piece of geometry individually, i.e. each triangle. Once a page has been loaded it remains in main memory and can be accessed for subsequent rays without the need to access the disk. When main memory space becomes full, pages are purged based on how recently they have been accessed. However, this method does not entirely avoid the delay that accessing the disk incurs and requires a work management scheme in the ray tracer to mask it.

Ray casting is performed in small coherent groups, rather than individual rays, since a group will often pass through the same leaf nodes in the kd-tree. Once a leaf node is encountered, its corresponding page is loaded into memory once and reused for all the rays. To mask this inherent need to read from the disk the ray tracer initially loads a subset of the pages that will fit in main memory. When a ray group encounters a leaf node which has already been loaded then ray tracing proceeds as normal. However, if the leaf node page has not been loaded, the ray tracer suspends ray tracing for that group and requests the page from disk. The disk access routine is performed by a continuously looping thread that accepts a memory request, reads a page into memory and then notifies the ray tracer when the page is available through a shared page index table. It becomes unavoidable, at least shortly after startup or moving the camera, to mask all disk access latency within the time it takes to display a single frame in real-time. When the geometry is not available in memory to compute a particular ray, the corresponding colour is simply coloured red or interpolated from nearby pixel values. The end result is a real-time animation of the large scene at the expense of image accuracy for several frames after the camera is moved.

Distributed parallel rendering has been employed by many researchers, particularly when visualizing scientific data generated from simulations or recorded from real world sensors. In recent years super computer simulations have grown in scale significantly and are capable of generating massive datasets. As with super computing, datasets of this scale simply do not fit in main memory on a stand-alone workstation and require distributed processing for rendering. One relatively simple example of distributed rendering of large datasets, which takes advantage of general purpose computing on graphics processors (GPGPU), is demonstrated by Abraham and Waldemar [2]. Their distributed visualization system uses a cluster of commodity PCs equipped with GPUs to display black oil reservoir simulation data for industrial research.

One of the processors in the cluster performs a preprocess to partition the data into a kd-tree and then distributes a subset of the kd-tree nodes to each processor in the cluster. Each processor is then responsible for rendering an image of the subset it has been assigned, given a camera position and view angle that is static across all processors in the cluster. The actual rendering is offloaded to the GPUs which, in this case, perform a simple rasterization of polygonal data generated from the data subset. Upon completion, one image is produced for each processor in the cluster and subsequently composited together to form one final image of the full dataset. This method is capable of real-time performance, however, since a “master” process must pre-compute the kd-tree, this is not possible if the dataset changes.

In recent years, simulations have become capable of producing datasets so large that the time required to physically transfer the data to a visualization system is prohibitive, if not impossible. Researchers have typically solved this problem by reducing the resulting dataset, either by saving only a spatial subset or a coarse subset of all time steps of the computed data. Even when these techniques are employed the datasets can still be extremely large, in some cases reaching the petabyte scale. A more promising visualization technique for data of this scale is to perform in situ visualization as part of the simulation computation while the full dataset is available in memory.

In situ visualization is not a new technique, but it has not been widely adopted because researchers have preferred to devote their HPC resources to computation, rather than visualization. Recently, however, it has received renewed interest as exemplified by the work of Yu et. al. [29]. They demonstrate a technique that is capable of live, real-time visualization of a simulation in progress that incorporates visualization computations into the simulation code. A typical simulation computation operates on a dataset made up of a grid or lattice of scalar data. To distribute the data among the processors, they are partitioned or sliced at some predefined interval. Where Abraham and Waldemar's method required a partitioning preprocess, this property is inherent when a simulation is still running.

Yu et. al. do, however, share a similar parallel rendering method to Abraham and Waldemar. The primary goal of their work is to render combustion chamber simulations at real-time speeds. As such, they chose to use ray tracing to produce a volumetric rendering as well as render spheres generated from the dataset. After a processor has completed a time-step in the simulation it then ray traces an image of the data subset that it contains. As with Abraham and Waldemar, this results in a set of images, one for each processor, that are composited together using a parallel painter's algorithm. One of the drawbacks of this method is that a six-way communication is required among each spatial neighbour of each processor to complete the compositing step. Despite this, Yu et. al. show that rendering performance in most cases approaches real-time speeds.

Chapter 3

Ray Caches

This chapter will outline the design for an out-of-core ray tracer to be implemented on a distributed memory cluster, and the construction and use of ray caches to enhance its performance. Previous work in this domain has typically shown that the most significant performance bottleneck is the overhead of communication between processors [20, 28]. The primary goal of ray caches is to reduce the overall amount or number of these exchanges, or more accurately, to replace communication with a less expensive local cache lookup operation. The first section in this chapter, Section 3.1, describes a simple, naive out-of-core ray casting algorithm for rendering scenes that are distributed among many processors on a cluster. The remaining sections introduce ray caches to this algorithm, and describe a method for constructing, populating, and sampling these caches during rendering.

3.1 A Method For Distributed Out-of-Core Ray Casting

The literature review in Chapter 2 has already discussed several methods of "out-of-core" rendering for large datasets or scenes. Each of these systems employ one of two main strategies that are intended for a particular hardware configuration:

1. a single, shared memory multi-processor machine, or

2. a distributed memory system or cluster.

A single multi-processor rendering system typically employs memory caching or paging schemes for rendering scenes that do not fit entirely in main memory. However, this method becomes ineffective when the scene size grows large enough that the cached pages themselves become too large. In contrast, a distributed memory or cluster rendering system relies on distributing a scene among each separate processor that renders an image of the locally stored data; followed by a compositing algorithm to produce the final image. This method too can be limited because, as image size increases, the compositing algorithm can become more expensive than the rendering computation. Both methods appear to be susceptible to degraded image quality, either due to geometry that is not available for intersection calculations because its page has not been loaded into memory, or discontinuities created when a set of individually rendered images are composited.

To begin investigating the effectiveness of ray caches, a different distributed out-of-core rendering system is needed. This system must be able to store the full dataset or scene in memory distributed among the available processors, to avoid the cost of paging memory from disk. This system must also be able to cast rays through the scene only to their closest intersection, rather than casting all rays locally on all processors and then compositing.

Computer clusters have always been an effective way of increasing processing power by orders of magnitude. As processors have become increasingly powerful and more affordable, clusters made up of consumer PCs have become just as effective for increasing computing power. Clusters such as these can easily contain hundreds or even thousands of individual processors at a very low cost. They have been utilized to great effect for parallelizing large scale computations and simulations that would simply overwhelm a standalone computer system.

As eluded to previously in Section 2.4, most simulation computations operate on a dataset made up of a grid of scalar data points. To distribute these datasets equally across the cluster, the grid is subdivided into smaller "voxels" that contain a subset of the full grid. Each voxel can then be assigned to an available processor for local computations. To understand this visually, this subdivision process is analogous to slicing a solid cuboid into several smaller, equally sized cuboids; resembling the smaller cubes making up a Rubik's Cube (see Figure 3.1).

At each simulation time step, each processor independently operates on the voxel of data that it has been assigned and, when necessary, exchanges boundary data with the processors that are responsible for the neighbouring voxels. This data subdivision process is quite similar to a technique employed by Wald [21] to subdivide a scene comprised of triangles into a 3D grid of

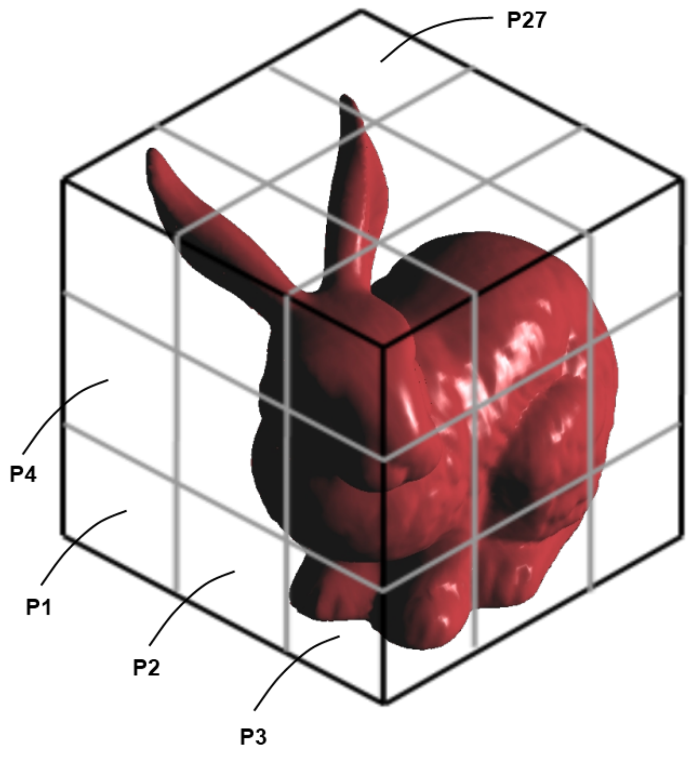


Figure 3.1: This figure shows the set of cuboids or voxels created by subdividing a triangle mesh. P1, P2, etc. refers to the processor that stores the geometry within the indicated voxel.

voxels that each contain a subset of the scene.

A ray cast into this subdivided scene then traverses the set of voxels that intersect its path. Upon entering the voxel, the contained triangles are checked for intersections with the ray before proceeding to the next voxel. Although Wald's ray tracing method is quite effective on a shared memory multi-processor machine where voxels can be loaded asynchronously from disk, it is hardly feasible if the voxels are stored on separate processors on a cluster.

In this case, when a ray travels from one voxel to the next it must access the triangles stored at a physically separate processor. One solution would be to copy the triangles between processors and to employ a paging scheme similar to that mentioned previously (see Section 2.4). However, this would lead to significant communication overhead and increased memory requirements for each processor. Instead, since each processor already has a particular voxel loaded in memory, the ray itself can be sent, and the receiving processor can calculate an intersection and colour value independently.

3.1.1 Scene and Voxel Bounding Boxes

One of the most basic ray tracing acceleration structures is the bounding box. Many efficient algorithms have already been developed to take advantage of them, such as triangle-box overlap tests by Möller [3], to determine if a triangle pierces an axis aligned bounding box; and ray-box intersections by Williams et. al. [27], to determine if and where a ray passes through an axis aligned bounding box. Using axis aligned bounding boxes significantly reduces the complexity of these computations compared to non-axis aligned boxes, and they can be simply defined by the minimum and maximum of their bounds on the x, y and z axes. These bounds are stored as two 3D points, $\mathbf{b}_{min} = \{x_{min}, y_{min}, z_{min}\}$ and $\mathbf{b}_{max} = \{x_{max}, y_{max}, z_{max}\}$. Traditionally one bounding box is built to surround the extent of the scene. A rectilinear grid of scalar data points inherently defines a bounding box since its minimum and maximum boundaries are known; however, this must be computed for a scene made up of triangles. This is accomplished by parsing each triangle and increasing the minimum and maximum bounds to encompass them. Sub-bounding boxes can be created by subdividing the scene bounding box at regular intervals on each axis to form voxels. Once the minimum and maximum bounds of each voxel are computed, Möller's triangle-box overlap algorithm can be used to determine the set of triangles it contains. This is discussed in greater detail in Section 4.2.

For a ray to travel through this subdivided scene and find an intersection it must first pass through the scene bounding box. Further, if the ray does pass through the scene bounding box, it must also pass through one or more voxels before it encounters an intersection or exits the scene. The method of traversing a ray between voxels is addressed further in Section 3.1.2, however, both the intersection point and the "face" of the bounding box where the ray enters the box must first be calculated. This is possible through a small modification of Williams's intersection algorithm. Their algorithm considers each axis of the interval between $\mathbf{b}_{min}$ and $\mathbf{b}_{max}$ and tests each for overlaps with the ray. Two floating point values are produced for each axis, t_{min} and t_{max}, which are the distances from the ray origin point to the point where the ray enters and exits the interval respectively. This results in three separate t_{min} values; one for each axis. The highest of these three values indicates the distance from the ray origin that the ray actually enters the bounding box, and can be used to calculate the 3D intersection point. This method has been extended to assign a "face" index that indicates which face the ray passes through to enter and exit the bounding box. To reference the face, each face of the bounding box is assigned an index value in the following order: $-x \Rightarrow$

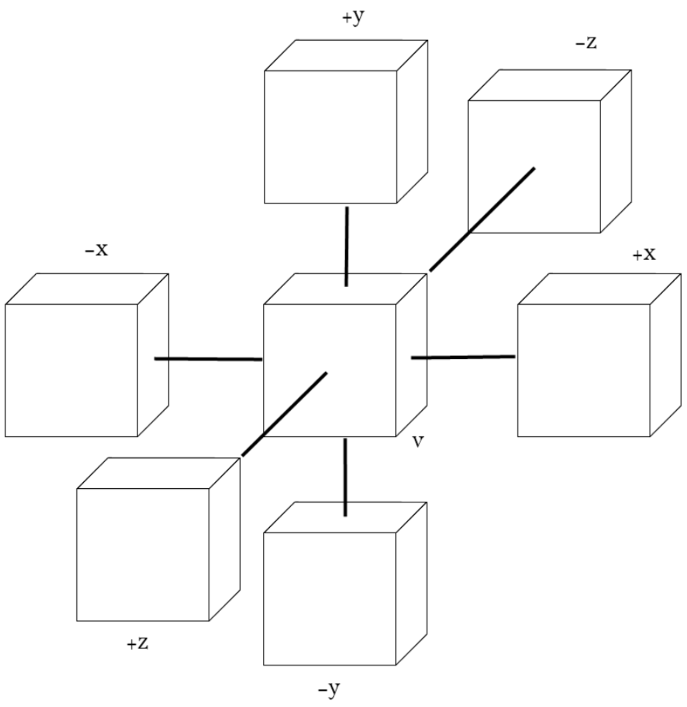


Figure 3.2: The voxel v has six neighbouring voxels. The processor responsible for v stores the rank of the processor responsible for each neighbour. For example *neighbour-ranks*$[-z]$ yields the processor rank responsible for the neighbouring voxel in the $-z$ direction.

0; $+x \Rightarrow 1$; $-y \Rightarrow 2$; $+y \Rightarrow 3$; $-z \Rightarrow 4$; $+z \Rightarrow 5$. For example, face 0 refers to the minimum boundary on the x axis, and face 3 refers to the maximum boundary on the y axis, etc. (see Figure 3.2). Since t_{min} and t_{max} are chosen based on the axis that yields the highest or lowest value, the face index is determined by that same axis.

3.1.2 Voxel Traversal and Ray Communication

Each ray cast into the scene must traverse the grid of voxels in the order that they are encountered until an intersection is found with the contained triangles. First each processor on the cluster must be assigned an integer rank to uniquely identify it. This rank is used to determine which processor the ray should be sent to when it moves between voxels. To simplify this, each processor stores a table, *neighbour-ranks*, which stores the six ranks that correspond to the processors responsible for the six neighbouring voxels (see Figure 3.2). Now when a ray passes through a particular voxel face, and must travel to a neighbouring voxel, the processor need only look up the processor rank in this table to determine where the ray should be sent in the cluster.

The voxel traversal algorithm operates in two phases; initialization and incrementation. Initialization begins by finding an intersection with the scene bounding box and then identifying the first voxel that the ray passes through. Given the scene bounding box intersection point, $\mathbf{p}$, the minimum and maximum boundary points of the scene, $\mathbf{b}_{min}$ and $\mathbf{b}_{max}$, and a set of three integers that indicate the number of voxels on each axis, G, the processor rank is then computed by Algorithm 3.1.

To summarize, the vector $\mathbf{d}$ is the vector from the minimum scene boundary to the point where the ray enters the scene. The vector $\mathbf{v}$ is the vector from the minimum scene boundary to the maximum scene boundary. Each component of the vector $\mathbf{v}$ is divided by the corresponding component in G, which gives the corresponding dimensions of each voxel. Each component of the vector $\mathbf{d}$ is then divided by the corresponding component of the voxel

Algorithm 3.1 Compute primary ray destination rank.

```
$\mathbf{d} \leftarrow \mathbf{p} - \mathbf{b}_{min}$
$\mathbf{v} \leftarrow \mathbf{b}_{max} - \mathbf{b}_{min}$
$x \leftarrow floor(\mathbf{d}_x/(\mathbf{v}_x/G_x))$
$y \leftarrow floor(\mathbf{d}_y/(\mathbf{v}_y/G_y))$
$z \leftarrow floor(\mathbf{d}_z/(\mathbf{v}_z/G_z))$
$rank \leftarrow x + yG_x + zG_xG_y$
```

dimensions. The resulting floating point value is truncated to leave only an integer indicating the x, y and z grid coordinate of the first voxel that the ray passes through. The destination processor rank is then computed by converting the grid coordinate into a linear index value. Once this is computed, the ray is sent to the destination and the traversal algorithm moves on to the incrementation phase.

After receiving the ray that passes through its voxel, the processor traces the ray through the locally stored triangles. This computation is very similar to any standard ray tracing algorithm and can be supplemented by further acceleration techniques, such as a BVH. If an intersection is found, the processor computes a colour value for the ray. However, in many cases the ray may pass through and exit the voxel, and must be sent along to traverse a neighbouring voxel. When this occurs the processor calculates the ray intersection with its voxel bounding box to determine the face index that the ray exits through. The face index is then used to lookup the processor rank in the neighbouring rank array to determine where the ray should be sent. Each subsequent processor that receives this ray repeats these steps until either an intersection is found and a colour value is computed, or the ray exits the scene bounding box and a default colour value is assigned.

Every time a ray transitions between two voxels there is a communication delay while the ray is sent between the two processors in the cluster. Further, another communication delay occurs when a colour value is found and sent back to the processor that cast the ray initially. Using a naive method, the colour value could be propagated back to each of the processors that the ray previously traveled through until reaching its origin. This would double the number of communications as each ray sent between voxels would result in a colour value being sent back. Instead, each sent ray carries a reference to the processor rank that originally cast the ray. Upon finding an intersection, the colour value can be sent back directly, resulting in a single extra communication operation. Despite this, if a scene is divided into hundreds or more voxels, a ray may incur many hundreds of communication operations before an intersection is found, resulting in significant traffic passing through the cluster network. To alleviate this network congestion, each processor can maintain a set of six batches corresponding to each neighbouring processor. These batches then collect all outgoing rays or colour values that are being sent to the same destination. This reduces the overall density of traffic in the network but can cause delays while rays and colour values must wait until the batch is filled (see Section 4.4).

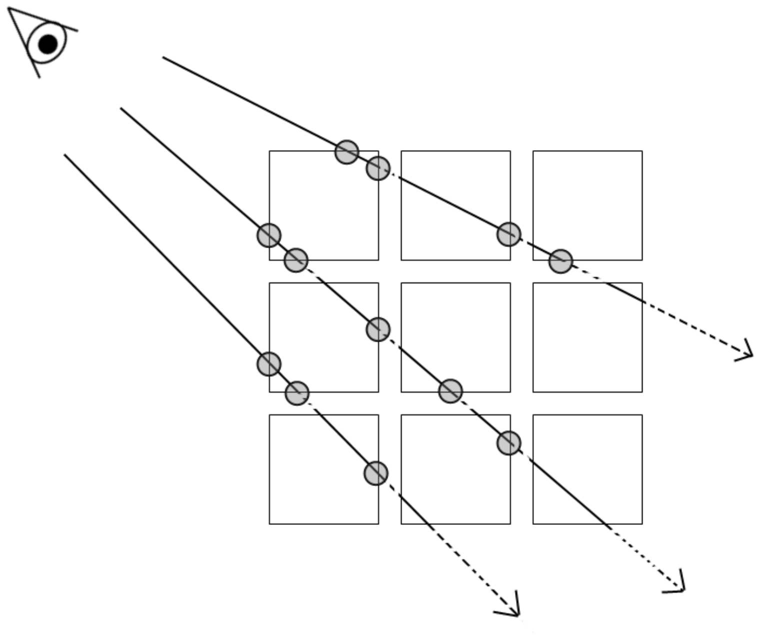


Figure 3.3: A ray communication operation occurs at each highlighted voxel boundary intersection.

3.2 Cache Construction

What is really needed to alleviate the overhead of ray and colour communication is a method to derive or approximate a ray colour value from previous rays that have already been computed. In the two dimensional example shown in Figure 3.3, each ray would incur a communication operation for each of the highlighted grid boundaries that it passes through. Due to the ray coherence property discussed previously, it is likely that several other rays will follow a similar path through the grid and cross many of the same voxel boundaries.

Further, Larson and Simmons [14] observed that the computed colour value for a ray is valid anywhere along its length, including positions behind its origin, if there are no obstructions. They use this observation in their holodeck data structure, which captures the point where a ray enters and exits the bounding box around a complex object along with its computed colour value. At the expense of image quality, any subsequent ray that enters and exits the bounding box at similar points can then reuse the captured colour values to produce a new value instead of performing an intersection calculation. There are two important conclusions to draw from this work which are useful for the out-of-core ray tracing method described here. First, for a ray that originates outside the scene bounding box and that pierces one of the bounding box faces, a ray intersection may only occur at some point further along the ray than the point where it entered the bounding box. Second, for a ray that originates inside or at the boundary of a voxel and does not find an intersection inside, it must exit the voxel and an intersection may only occur beyond the exit point. This means that any radiance value calculated for the ray is not only valid at the ray origin point but also for the point where the ray enters and exits a voxel.

Instead of viewing each voxel as a minimum and maximum boundary position, it can be thought of as a set of six planes or walls making up the six sides of its bounding box. For a ray to enter and exit these voxels it must pass through at least two of these walls; one to enter and another one to exit. Accordingly, a ray cache is created for each wall and is responsible for collecting colour values for every ray that passes through it. Each processor individually creates and maintains these caches locally. For any ray passing through its voxel wall, the processor provides the ray-wall intersection point, ray direction, and colour value to the appropriate cache, which can then store the ray or search for similar rays. More importantly, however, caches are also placed on the outer boundaries of the bounding box surrounding the extent of the entire scene. The is done to allow primary rays cast from the camera, i.e. outside of the scene bounding box, to sample a cache before it enters a voxel and thus requires a communication operation.

Although the principle is similar, the outer caches surrounding the scene are handled quite differently than the caches surrounding each voxel. A major concern is that the number of primary rays that pass through these outer caches is likely to be significantly larger than the number of rays that pass through any voxel. This is especially true under the assumption that most, if not all the voxels fall within the camera view frustum. Also consider that the rays will only ever be able to enter the scene bounding box through no more than three sides. This means only three of the six surrounding ray caches would be responsible for storing a significant number of rays, upwards of hundreds of thousands for even a relatively small 1024^2 pixel image. To reduce this load on a single cache, the outer bounding box walls are subdivided into a set of smaller caches. For simplicity the area of these smaller planes correspond to the wall surface area of the inner voxels, such that each smaller cache matches 1:1 with a voxel wall. Referring to Figure 3.1, the outer ray caches correspond to each square patch on the bounding box surrounding the scene, which each correspond to a voxel wall.

3.3 Cache Storage and Search

The purpose of the ray cache is to replace an expensive communication operation with a fast search operation to produce a colour value. To accomplish this the ray cache must be able to store, search for, and retrieve rays faster than sending a ray and computing an intersection and a colour value. This is challenging due to both the high number of rays that are cast as well as the potentially high number of caches that any given ray may pass through. When a cache stores a large number of rays it not only requires additional memory but also significantly increases search complexity. Also, in the case that the cache search fails to produce a suitable colour value, a communication operation becomes unavoidable. In this situation the cache search operation becomes a bottleneck to the ray tracing computation; therefore it is crucial that the cache search operation incurs a minimal cost to performance.

A ray is defined by its origin point and its direction vector. Thus to store a ray and to subsequently search for similar rays, the ray cache must store both an origin point and a direction for every ray. However, the ray origin point, i.e. the camera position, is not particularly useful to the cache. Instead, the cache records the point where the ray intersects the cache plane. This allows the cache to more easily compare the intersection points where any two rays pass through its plane. For an incoming ray that intersects the plane, the cache must then perform a nearest neighbour search across the surface of the plane for any stored rays that intersected at a similar point. Also, when no suitable cached value can be found and a new colour value must be calculated through ray tracing, the cache must be able to quickly insert the value without significantly delaying further ray tracing computations. Given these constraints, storing the cached rays in a linear data structure would be a poor choice due to its linear search complexity. One data structure that lends itself extremely well to these requirements is the kd-tree due to its ability to efficiently insert into and search through k-dimensional space. Insertion into the cache then requires $O(\log n)$ time compared to constant time for a linear array, however searching for nearby rays becomes significantly more efficient.

When a ray is found to intersect the cache plane and a suitable stored colour value can be extracted, this is referred to as a cache hit. Similarly, a cache miss occurs when a cached value cannot be extracted. The specific criteria needed to determine a cache hit is described by two tolerances; radius r and angle ϕ. The radius r refers to the maximum allowable distance between the point where a new ray intersects the cache and the point where a stored ray previously intersected the cache, see Figure 3.4. The angle ϕ refers to the maximum allowable angle between the direction of the new ray and the stored rays. The cache search routine takes r, ϕ, the cache plane intersection point $\mathbf{p}$ and the ray direction vector $\mathbf{d}$ as input. The search begins by querying the kd-tree for any stored ray intersection points $\mathbf{q}$ such that $\|\mathbf{p} - \mathbf{q}\| \leq r$. This search returns 0 or more stored rays that meet this tolerance. Next, each returned ray is further considered only if the direction of the stored ray $\mathbf{f}$ meets the requirement $\mathbf{d} \cdot \mathbf{f} \geq \cos\phi$. When one or more rays are found to satisfy both these tolerances, the ray tracer must determine the colour to be extracted. There are three potential ways to do this, and each have a potential trade-off between image quality and performance:

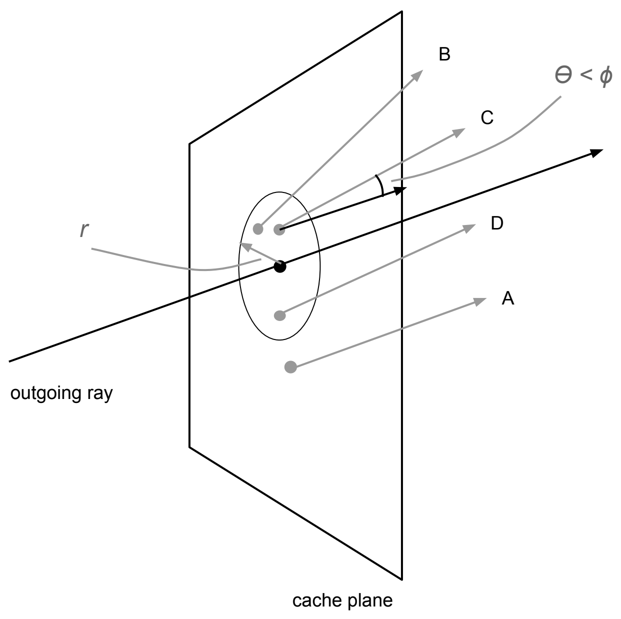


Figure 3.4: This figure shows which cached rays are likely to be selected for retrieval. Ray A is rejected because its intersection point is outside the search radius. Ray B is within range, but is also rejected since its direction differs too greatly. Ray C and D fall within both the range, r, and angle, ϕ, tolerances and may provide a suitable colour value.

1. Take the colour value of the first ray found within tolerance.
2. Take the colour value of the nearest or most similar ray found within tolerance.
3. Take the average colour value of all rays found within tolerance.

Retrieving the first colour found in the cache potentially allows the search routine to terminate more quickly at the expense of lower accuracy. Retrieving the nearest colour value, specifically for the ray that has a minimum intersection point distance, likely requires additional computation but ensures a more accurate colour value. In the event that only one ray is found within tolerance, or the nearest ray is very near maximum tolerance, the accuracy of the retrieved colour value will not significantly improve over the first result method. Lastly, retrieving the average colour value also requires additional computation and can alleviate loss of accuracy by taking multiple cached colours into account. These three methods are compared later in the results, see Section 5.3.

Chapter 4

Implementation

4.1 Sharcnet and MPI

The Shared Hierarchical Academic Research Computing Network (Sharcnet) is a consortium in Southern Ontario that has established many large clusters of high performance computers meant to facilitate academic, industrial and business research. The individual clusters are located and maintained at academic institutions and are connected via dedicated fiber optic links. Each cluster employs a different multi-processing scheme such as shared vs. distributed memory systems, or clusters comprised mainly of GPGPUs. Each processor in a cluster is connected through a specific hardware interconnect, such as basic Ethernet or InfiniBand. A typical cluster is a distributed memory system and supports multi-processor programming through the use of the Message Passing Interface (MPI). Due to the number of jobs submitted by students and faculty on these clusters, Sharcnet employs a job scheduling system to manage available resources and scheduling fairness. When a job is submitted to the queue the user must specify the number of required processors. The scheduler then assigns the job to a specific set of processors or *nodes* in the cluster to ensure the requested number of processors are available when the job starts.

MPI itself is a standard communication protocol for sending messages between parallel processes. Sharcnet provides implementations and bindings for MPI in C, C++ and Fortran, along with other parallel programming libraries such as pthreads and openMP. Each process spawned during the execution of an MPI program has access to a small set of basic functions to facilitate communication. The first of these is the ability for a process to determine its *rank* among all the processes and determine the *size* or total number of processes that have been spawned. This allows any process to be uniquely identified and targeted for communication. For example, a master-slave model can be implemented by choosing rank 0 to be the master process that manages dispensing work to all available processes by communicating data across the network. The *size* variable is particularly useful when writing an MPI program that is scalable to any arbitrary number of processes without having to change the source code. Again for the master-slave model, rank 0 could divide the total dataset that needs to be computed into *size* number of chunks and then communicate each chunk to the appropriate process. Thus the program can be executed on any number of processors and the work will be distributed approximately evenly.

MPI facilitates communication through two basic functions `MPI_Send()`

and `MPI_Recv()`. These functions form the basis of every communication operation; including collective communication and process coordination. `MPI_Send()` and `MPI_Recv()` both require four parameters; a reference to a data array, the data type stored in that array (int, double, etc.), the total number of items of that data type to be sent or received and lastly the target processor's rank. For a send, the sending process must initialize a data array that is at least one byte in size to contain the data that is being sent. In many cases this is simply a matter of specifying a memory address or pointer to an existing array of data. However some messages need to send data that is stored in non-contiguous memory locations. Here data must be copied into a contiguous array before it can be sent through MPI. The data type parameter is simply the MPI equivalent to the data type being sent, such as `MPI_INT` for integers and `MPI_BYTE` for basic byte arrays. The total number of items in the array to be sent is simply an integer value that has been precomputed or specified by the user. Lastly the target rank is an integer value specifying the rank of the process that the array will be sent to.

The parameters are used slightly differently for a receive operation. The receiving process must specify a memory address which has enough allocated space to receive the incoming array. If the size of the array being sent is not known beforehand or can be of variable size, efficiently allocating an array or allocating an array of sufficient size can be a difficult task. The receiving data type parameter is specified the same way as for the send operation, however it does not necessarily have to match the type that has been sent. The array length integer parameter specifies the upper bound for the number of items of the given data type that will fit in the allocated receiving array. As with allocating the receiving array size, this value must be chosen carefully. Lastly the target rank integer specifies the rank of the sending process. In some applications the rank of the sending process may not be known, in which case MPI provides the `MPI_ANY_SOURCE` value to allow the receiving process to collect data from any sending process.

It should be noted that these basic send and receive functions are *blocking* operations. For a sending process to complete a `MPI_Send()` operation, a receiving process must also perform or *post* a `MPI_Recv()` operation and vice versa, otherwise one process will block while it waits for its counterpart. MPI does provide a non-blocking alternative to each of these; `MPI_Isend()` and `MPI_Irecv()`. Rather than waiting for a matching send or receive to post, these functions both store the data array in a buffer and return immediately. This allows the process to continue while the communication operation completes in the background. However, in this case it becomes possible for the process to access and potentially alter the buffer before the operation has completed. It is then the programmer's responsibility to restrict access to the array or allocate and manage buffer space in memory to ensure it is not modified. For this purpose MPI provides the `MPI_Test()` and `MPI_Wait()` functions. `MPI_Test()` allows the process to check if a specific communication operation has completed and returns immediately. This is useful for periodically checking the status of a communication without interrupting computation. `MPI_Wait()` is similar but it will block the process until the communication has completed. This allows the process to continue computation for some time before blocking until the communication completes.

4.2 Test Scene Generation

To test the effectiveness of ray caches in the out-of-core ray tracer described in Section 3.1, it is necessary to generate a representative test scene and distribute it among the available processors. The implementation described here is restricted to polygonal scenes composed of triangle meshes. A common way of storing triangle meshes is in the PLY file format, developed for storing models in the Stanford 3D scanning repository. Recall from chapter 3 that each processor is intended to operate under the assumption that it will already contain some portion of the scene. To create a viable test case then, each processor must load a subset of the triangles making up a scene. This is decided through user defined parameters for the number of voxels on each axis of the voxel grid, and the number of processors. Since voxels are assigned 1:1 to processors, the total number of voxels must equal the number of available processors. For example, the user may specify a 3x3x3 grid resulting in 27 total voxels, therefore 27 processors must be available. The processor then must determine which of these voxels it is responsible for by mapping its rank to a voxel grid position; this is shown in Listing 4.1.

Essentially this algorithm is the reverse of a typical linear array mapping where 3D coordinates are converted to a linear array index. The linear array index, `rank` in this case, is instead mapped to a discrete x, y and z grid position based on the extents defined by `bounds`. Conceptually, as `rank` increments by

Listing 4.1: C++ code sample to map a processor rank to a voxel grid position.

```
/* Rank assigned to this processor */
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

/* Number of voxels on the x, y and z axes */
int bounds[] = {3, 3, 3};

/* Calculate x, y and z grid position of voxel */
int xVoxel, yVoxel, zVoxel;

if(rank >= bounds[0]*bounds[1]) {
        zVoxel = rank / (bounds[0]*bounds[1]);
} else {
        zVoxel = 0;
}
if(rank % (bounds[0]*bounds[1]) >= bounds[0]) {
        yVoxel = (rank % (bounds[0]*bounds[1]))
                / bounds[0];
} else {
        yVoxel = 0;
}
xVoxel = rank % bounds[0];
```

1 starting from the 0 position, the resulting grid position first increments along the x-axis until the according boundary is reached, in which case the position increments along the y axis and starts back at the 0 position on the x axis. Similarly, when the y axis boundary is reached, the z axis position is incremented and the x and y positions both start back at the 0 position.

The processor is now able to determine the portion of the scene that it must store for rendering. There are two ways of creating a scene that are used in this implementation. One method is to assign static voxel dimensions and position to each processor which then individually load a single PLY mesh, such that it will fit within its assigned voxel. This allows any arbitrarily large scene to be generated by simply specifying a larger number of processors. Alternatively, a single mesh can be subdivided and the triangles distributed to the processors based on which voxel they reside in. This is useful for assessing the effect of changes in voxel grid density, or number of processors, with respect to a given model. Recall from Section 3.1.1 that each processor loads its portion of the model in three phases. In the first phase it parses the set of triangles and calculates the minimum and maximum bounds of the axis aligned bounding box that encompasses the entire scene. In the second phase it calculates the dimensions and position of its voxel. Finally, in the third phase it tests each triangle for an intersection with the voxel; if the triangle does intersect, it is stored for later rendering. Once each processor has loaded its portion of the scene, it proceeds to the render loop where it begins receiving rays, see Section 4.3.

4.3 The Render Loop

Once a scene is properly loaded and ray casting has begun, each processor enters the render loop where it will spend the majority of its time. The purpose of the render loop is to retrieve and service any incoming rays and returned colour values that are sent from another processor. To determine if a message has arrived and requires service, the render loop depends on the FIFO incoming message queue. To facilitate both incoming rays and colour values in this single

Listing 4.2: The PackedRayStruct struct contains either a ray or a colour value.

```
struct PackedRayStruct
{
        bool isColour;
        int rank;
        int ID;
        int originRank;
        int originID;
        float e[3];
        float d[3];
};
```

queue, the ray tracer stores incoming messages in a `PackedRayStruct` object, which is described further in Section 4.3.1.

4.3.1 Packing and Unpacking Rays

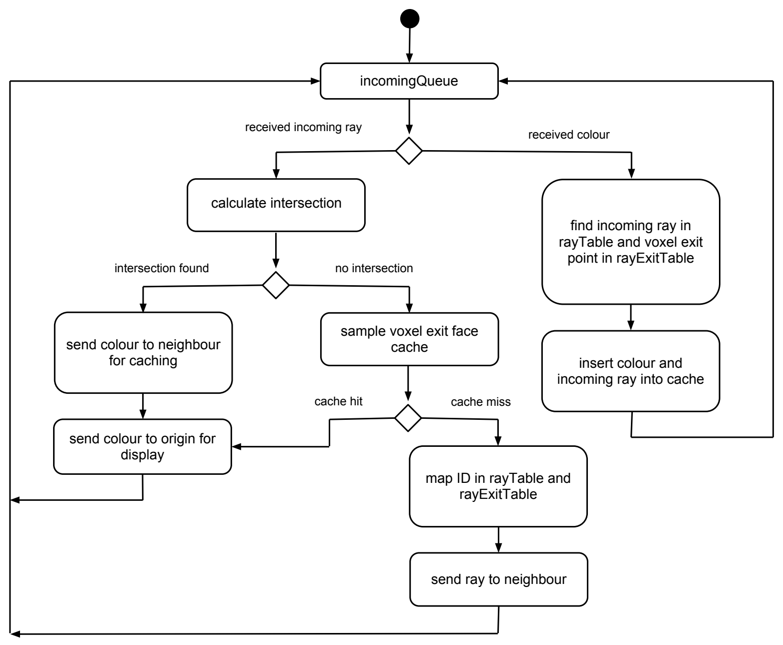
Listing 4.2 shows the definition for a packed ray object. Its primary purpose is to pack ray or colour data into a contiguous memory block that can be sent as an MPI message. An added benefit of this is the ability to group ray and colour objects together into a single array. This is not only useful for storing incoming data in the incoming message queue but is also useful for buffering communications (see Section 4.4).

The *isColour* member is true when the data contained is a colour value, otherwise it is assumed that a ray is stored. The *rank* member refers to the processor rank from which the object was sent. The *ID* member is a unique integer identifier generated by the processor that sent the object but is treated differently whether a ray or colour is stored. For a ray, *ID* is uniquely generated for that ray by the sending processor. For a colour, ID refers to the ID of the ray for which the colour value has been returned, see Section 4.3.2. The $originRank$ and $originID$ members are similar to $rank$ and ID, however they refer to the rank of the processor that originally cast the primary ray from the camera position, and the ID that the primary ray was originally assigned. The e and d members store the vector or colour components. For a ray, e stores the origin point, and d stores the direction vector. For a colour, e stores the RGB colour components, and d is simply ignored.

The integer ray ID of each ray sent by a given processor must be unique in order to match a returned colour value to a particular ray. To do this each processor initializes an ID generator to a value of `INT_MIN`, the minimum possible integer value. When sending a ray, the ID generator is incremented by 1 and the value is assigned to the ID member of the packed ray object. There is obviously a limit to the number of times the ID generator can be incremented before reaching the maximum integer value, however this number is far larger than the number of rays that a processor would send in this implementation. Another consideration is the potential conflict created when two processors generate the same ID and each send a ray to the same processor. The solution is to have the processor returning a colour value verify that both the ray ID and processor rank that sent the ray are matched.

4.3.2 Workflow

The overall workflow of the render loop is shown in Figure 4.1. The render loop begins each iteration by polling the incoming message queue, known from



incomingQueue
received incoming ray
received colour
calculate intersection
find incoming ray in rayTable and voxel exit point in rayExitTable
intersection found
no intersection
send colour to neighbour for caching
sample voxel exit face cache
insert colour and incoming ray into cache
cache hit
cache miss
send colour to origin for display
map ID in rayTable and rayExitTable
send ray to neighbour

Figure 4.1: This flow diagram shows the high level workflow of the render loop.

here on as simply the incoming queue. When no messages are available the render loop simply loops continuously or "busy waits" until a message arrives. Doing so ensures that any incoming messages are retrieved as quickly as possible. When a message arrives, the render loop pops it off the incoming queue and then branches depending on the type of packed ray object that has been received.

If a ray is received, the render loop passes it to the ray tracing routine which returns a boolean value, indicating if an intersection was found, and a colour value. Note that the internal operation of the ray tracing routine has no consequence on the render loop workflow, so long as it is capable of producing intersections and colour values. The ray tracing routine can therefore be supplemented with any applicable type of acceleration techniques. For instance, in this current implementation, each processor builds a grid of bounding boxes around the locally stored set of triangles to improve ray-intersection calculation efficiency. If the ray tracing routine finds an intersection, the render loop immediately packs the computed colour value into a new packed ray object and prepares it to be sent. However, if an intersection does not occur, the render loop must compute which voxel wall the ray exits through to determine which cache should be searched.

Recall from Section 3.1.2 that the ray-bounding box intersection method produces the intersection point and the integer face index of the voxel wall where the ray exits the voxel. Before searching the cache, the render loop first checks if the voxel wall is actually the boundary of the scene, in which case the cache is not searched, and the default ray colour is assigned and packed for sending. Otherwise the render loop performs a cache retrieve operation, which is discussed further in Section 4.6. If a suitable cached colour value can be found in the cache it is packed for sending. Once each of these options for producing a colour value - ray intersection, scene boundary or cache retrieval - have been exhausted, the render loop then repacks the ray into a new packed ray object, which generates a new ray ID, and prepares it to be sent to the neighbouring voxel.

When the processor sends a ray it must maintain a reference to the ray in order to match it with an incoming colour value, such that both the ray and the colour value can be added to the ray cache. A simple way to solve this is to map the ray ID to the packed ray object containing the outgoing ray. This is stored in a standard map structure, `std::map<int, PackedRayStruct> rayTable`, referred to simply as the ray table. When the render loop eventually receives a colour value for the ray, the packed colour ID is used as the search key to the ray table to retrieve the corresponding ray. Next the render loop must determine the correct cache to populate, which requires the cache face index and the intersection point to be added. Rather than recalculating this, the render loop maintains another map, `std::map<int, RayExitPointStruct> rayExitTable`, which maps the outgoing ray ID to a ray exit point object that contains the face index and the intersection point where the ray exits the voxel. Listing 4.3 shows how a ray ID is mapped to a packed ray object and to an exit point object. Listing 4.4 then shows how these maps are searched when a colour value is received.

Up to this point, the difference between returning a ray colour value for caching and returning a ray colour value for display has not been addressed. Section 3.1.2 suggested that a processor should send back a colour value immediately to the processor that cast the ray, rather than propagating it through each processor that the ray previously traversed. This is good for producing a colour value for display as quickly as possible, however it prevents the intermediate processors from adding that colour to their respective caches. There are two options for solving this; 1) the colour value can be sent back only to the last voxel that the ray passed through; or 2) the colour value can be propagated back to each voxel the ray passed through. Option 1) requires only one additional communication operation but only one voxel will be able to retrieve the computed colour from its cache. Option 2), however, requires an additional

Listing 4.3: This listing shows how an outgoing ray ID is mapped to an incoming ray in the ray table, and mapped to a ray exit point object.

```
PackedRayStruct incomingRay =
                incomingQueue.pop_front();

/* ... ray is found to exit the voxel ... */

/* Pack the ray and generate a unique ray ID */
PackedRayStruct rayToSend = packRay(...);

/* Map the outgoing ray ID to the incoming ray */
rayTable.insert(rayToSend.ID, incomingRay);

/* Voxel face and point where ray exits the voxel */
int exitFace; Vector p;
boundBoxIntersect(voxel_dimensions, &exitFace, &p);

/* Map outgoing ray ID to the ray exit point */
RayExitPointStruct exitPoint =
        makeExitPointStruct(exitFace, pFace);
rayExitTable.insert(rayToSend.ID, exitPoint);

send(rayToSend);
```

communication operation for each voxel that the ray passed through but each voxel will have access to the colour value in its cache. In practice, however, these two options have little performance difference. Although propagating causes the number of cached rays and thus overall cache hit rate to increase slightly, the added communication overhead appears to negate any potential improvement.

Further to this, the render loop must also consider whether a colour value should be sent back for caching at all. Consider, for instance, when a proces-

Listing 4.4: This pseudo-code snippet shows how an incoming colour object is used to search the rayTable and rayExitTable.

```
PackedRayStruct incomingColour =
                incomingQueue.pop_front();
Colour returnedColour = incomingColour.e;

/* Retrieve original incoming ray */
PackedRayStruct incomingRay =
        rayTable.find(incomingColour.ID);

/* Retrieve the ray exit point */
RayExitPointStruct exitPoint =
        rayExitTable.find(incomingColour.ID);

/* Populate the cache at face faceIndex, with point */
/* pFace, ray direction d and colour returnedColour */
cachePopulate(exitPoint.faceIndex, exitPoint.pFace,
        incomingRay.d, returnedColour);

/* Repack the colour to send the received colour */
/* along to subsequent neighbour */
PackedRayStruct colourToSend =
        packColour(incomingRay.ID, returnedColour);
sendToNeighbour(colourToSend);
```

sor produces a colour value by sampling a cache. The colour value may be appropriate for display but may not be suitable for insertion into the cache in a neighbouring voxel. In other words, the render loop assumes an incoming colour value is an accurate value corresponding to the ray that was sent out. If a cached colour value is retrieved and returned, then adding that colour value to a neighbouring voxel's cache will likely introduce unwanted noise to future cache sampling. The render loop must then decide if a computed colour value should be returned for caching based on whether the colour value was calculated by the ray tracer or retrieved from a cache.

4.4 Threaded Communication

To ensure optimal performance of the ray casting and ray cache search routines, each processor must put the majority of its computing power into the render loop and avoid interruptions that may be caused by managing communications. Recall from Section 4.1 that MPI supports blocking and non-blocking communications. Certainly blocking communication operations would not be suitable for use in the render loop since this would cause the processor to pause until the matching processor is ready. It would seem then that non-blocking operations would be the solution, however this would force the processor to allocate and manage a potentially large number of memory buffers. This is difficult since during an iteration of the render loop, the processor is unable to determine the number of messages that it may receive and therefore cannot predict the number of message buffers that would need to be allocated. One solution would be to allocate some number of message buffers and post a non-blocking receive operation for each of them. Then, at the beginning of each iteration, the render loop would query each buffer for a message, extract the received object and post another receive operation. This forces the render loop to spend time checking and operating on each buffer, and, if too few buffers have been allocated, prevents some communications from completing until a subsequent iteration.

The receive method employed in this implementation attempts to alleviate these issues by performing receive operations in a thread running in parallel to the render loop. The thread is created using the pthread library and scheduling is left to the operating system since the thread function is quite simple. Before initiating the render loop, each processor first spawns the receive thread to ensure that any sent rays will be immediately received and available for ray intersection, or received colours will be immediately available for display. Once created, the thread allocates a reusable message buffer to store an incoming packed ray object. The thread then calls `MPI_Recv()`, which causes the thread to block until a message arrives. Upon receiving a message, the packed ray object stored in the message buffer is copied and appended to the incoming queue. Since the incoming queue is the only means of communication between the receive thread and the render loop, it is protected by a mutex lock. Threading communication in this way minimizes buffer management by receiving one message at a time, as they are received, and also minimizes delay in the render loop by handling communication in parallel.

Another major consideration for communication is the potentially large number of communication operations across the cluster that may occur at a given time. Not only does this create congestion in the interconnect network, it also causes the processor to perform frequent send or receive operations. In Section 3.1.2 batching was suggested as a potential solution to this kind of network congestion, at the expense of potentially delayed ray tracing computations. In practice, however, the resulting delay is insignificant since the incoming queue will, quite often, already contain rays and colour values that need to be computed. In other words, a ray delayed by batching does not cause significant delays in the render loop. To implement batching, considerations must be made for how large the batches should be and how often they should be sent and cleared. If a batch is too small then the number of communications will not be greatly reduced since the batch will fill and must be sent quite often. Conversely, if a batch is too large then the delay in sending rays will begin to delay computations by the render loop. To balance these two extremes, a maximum batch size of 16 was chosen and each processor maintains a total of six batches, one for each neighbouring processor that rays and colour values may be sent to. Also, delays may occur if the render loop is not able to fill a batch quickly, therefore causing some rays not to be sent while they sit in an idle batch. To combat this, the render loop *flushes* the batches every 256 iterations to ensure that partially filled batches are sent regularly.

Batching requires a small modification to both the render loop send routine and the receive thread. The batch itself is an `std::vector<PackedRayStruct>` which contains a set of packed ray objects (outgoing rays or colour values) contiguously in memory. The render loop allocates six of these batches in an array, `sendBatches[]`, where `sendBatches[r]` corresponds to the batch that will be sent to processor rank r. Recall, each iteration of the render loop produces one packed ray object that contains either a ray or colour value that will be sent to one of the neighbouring voxels. Rather than initiating the send operation immediately, the render loop appends the object to the appropriate batch before moving on to the next iteration. When a batch reaches its maximum size or a sufficient number of iterations have completed, the render loop initiates the send routine. The render loop uses the asynchronous `MPI_Isend()`

function to minimize delays to subsequent iterations by performing the actual communication in the background. If a single batch has reached its maximum size then only that batch is sent, and if the maximum number of iterations have completed then all non-empty batches are sent. Accordingly, the receive thread supports batching by allocating a message buffer large enough to contain 16 packed ray objects rather than just one and, upon receiving a batch, appends the objects to the incoming queue.

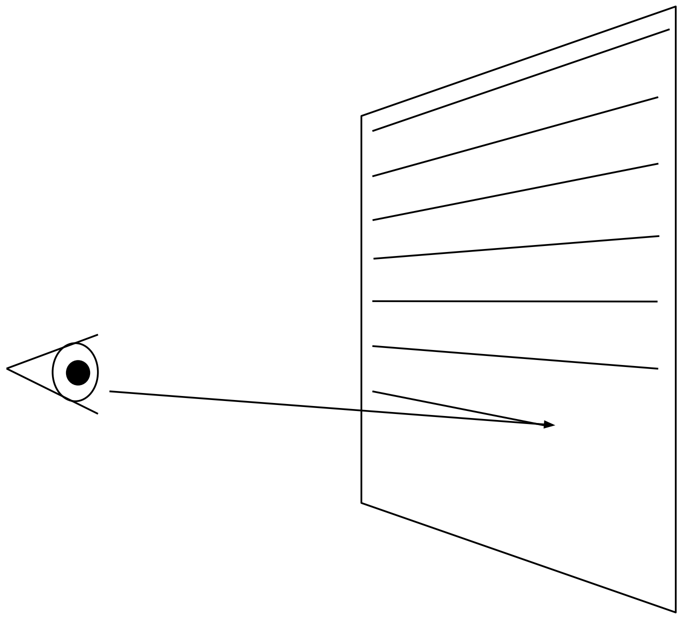
4.5 Ray Casting and Load Balancing

At this point each processor in the out-of-core ray tracer is capable of receiving rays, tracing them locally and sending out further rays or colour values to neighbouring processors. This only functions under the assumption that rays are initially sent to a processor from a position outside of its voxel, such as a camera that is positioned outside the bounds of the scene. As a result, each processor requires another processor, possibly itself, to cast primary rays into its voxel and therefore send it rays to begin ray tracing. For the current implementation, ray casting is handled by a single processor, which requires that one processor take on the dual role of intersecting rays in one of the voxels as well as casting and collecting primary rays to render a final image. This task is arbitrarily assigned to the processor rank 0 at run time. Before entering the render loop, this processor performs all computations necessary to cast primary rays through the image plane, calculate which voxel a given ray first penetrates and finally send the rays to the appropriate processors.

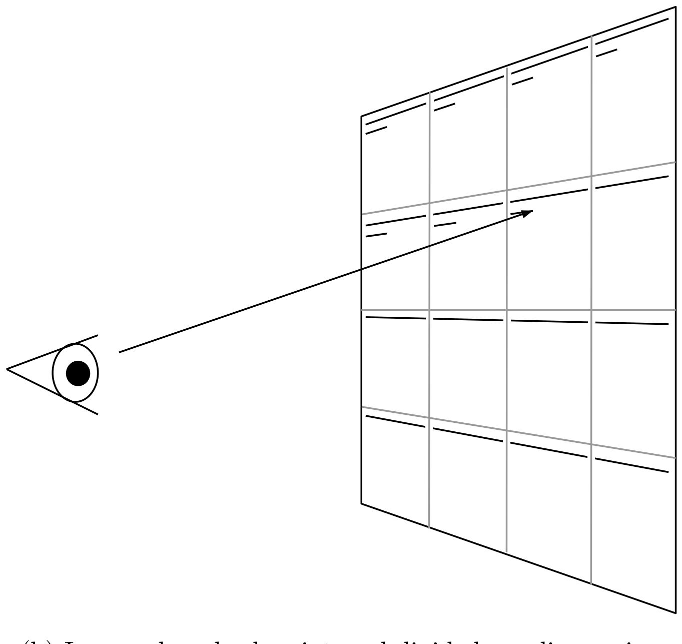
One of the primary concerns of the ray casting function is ray distribution. A basic ray tracing algorithm will typically cast a ray through each pixel using a scan-line pattern. Subsampling techniques are also often employed to speed up ray tracing for areas of little change across the image plane but they too often follow a scan-line-like pattern. This is not a particularly good way to cast rays for this out-of-core renderer since the scan-line pattern does not evenly distribute rays among the voxels. If a scan-line pattern is used, only a small number of the available processors begin ray tracing while the remainder are left idle until a later scan-line is reached, see Figure 4.2a. To ensure an approximately even distribution, the ray casting function *breaks up* the scan-lines into chunks across the image plane. This is done by simply dividing the image plane into a set of smaller sub-planes and also dividing each scan-line in these sub-planes into a set of pixel chunks, see Figure 4.2b. The ray casting function then proceeds sequentially through the set of sub-planes and generates rays for the next chunk of pixels. Although this accomplishes a sufficient distribution of work among the processors, it does not necessarily provide a very good distribution of cached colour values among the ray caches. This is addressed by a coarse subsampling process, discussed in Section 4.5.1.

4.5.1 Coarse Sampling

Before casting a ray through each pixel, the ray casting function first performs a coarse subsampling across the image plane. The purpose of this step is to populate the ray caches such that a representative set of stored rays are available for sampling once the primary rays begin searching the caches. The



(a) Regular scanline pattern.



(b) Image plane broken into subdivided scanline regions.

Figure 4.2: In the regular scanline pattern, most rays are initially cast into the voxels toward the "top" of the image plane. With subdivided scanlines, rays are distributed more evenly among all the voxels.

coarse sampling function operates similarly to the ray casting method discussed in Section 4.5; however, the rays do not correspond to image pixels, and the rays are computed only for storage in the ray caches, not for display in the final image.

The number of coarse samples to be taken must be carefully selected such that the rays are sufficiently distributed throughout the caches and that the coarse sampling process does not significantly degrade performance. For example, when rendering a 1024^2 pixel image, casting 256^2 coarse samples provides a sufficient distribution of cached rays without causing significant delay (see results in Section 5.2). A ray cast during coarse sampling does not necessarily pass through the same pixel position as a ray cast during rendering. The reason for this is that coarse sampled rays should be available for reuse for as many rays as possible, within the allowable tolerances outlined in Section 3.3. If a sample were to correspond with a particular pixel, that sample may be biased to provide accuracy for that one pixel while surrounding pixels would result in reduced colour accuracy, whereas a sample not corresponding to a pixel may provide a more reasonable approximation to multiple nearby pixels. This also allows the coarse sampled rays to be randomly distributed across the image plane, similar to the method used in distributed ray tracing. A random distribution does not necessarily affect performance or numerical measures of image quality, however it can produce a more visually appealing image by reducing aliasing in the cached samples; the results of this are shown in Section 5.2.1. To understand this visually, Figure 4.3 shows the resulting distribution of sample rays cast in regular and randomized patterns, while Figure 4.4

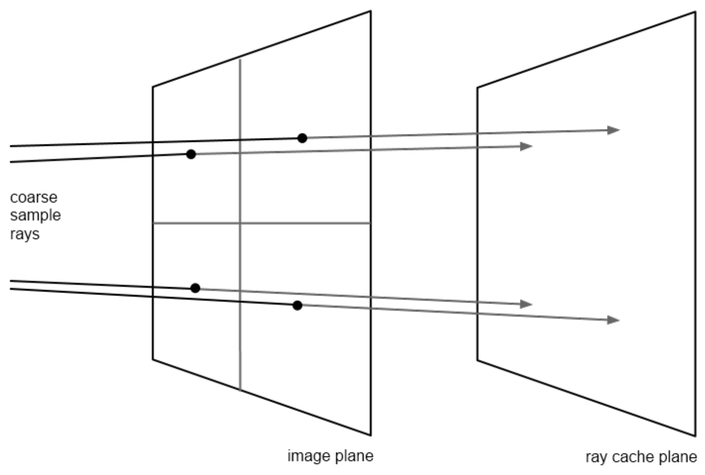
shows the potential effects of searching a cache that has stored regular and randomized samples.

4.6 Cache Retrieval and Population

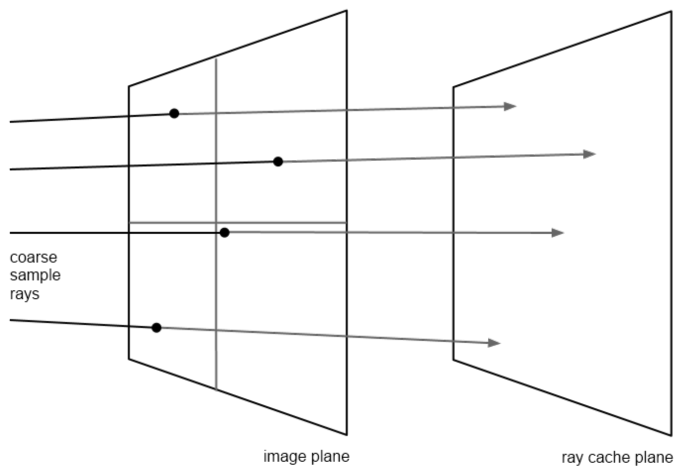
The ray caches are stored in a kd-tree structure using the open source kd-tree implementation library found at [1]. Using this library, a `kdtree` object is created with the `kd_create(int k)` function which creates a k-dimensional kd-tree. It is then possible to insert into and search the tree with the following two functions:

- `kd_insert(kdtree *tree, float[k] p, void *data)`
- `kd_nearest_range(kdtree *tree, float[k] p, float r)`

The parameters to the `kd_insert()` function are a reference to the kd-tree, an array of length k which contains the k-dimensional point to be inserted into the tree and an optional pointer to a data array that is to be stored along with the point. The parameters to `kd_nearest_range()` are similar, however `p` in this case is the k-dimensional point for which the tree is being searched and `r` is the allowable range from `p`. Also, note that `kd_nearest_range()` returns a `kd_res` object, which acts as an iterator that references the set of all points returned by the search. The points can then be retrieved using the function `kd_res_item(kd_res *points, float[k] pos)`, which accesses the next point found in the result list `points`, stores the retrieved k-dimensional point in `pos`, and returns a reference to the `data` array that was stored with the point.

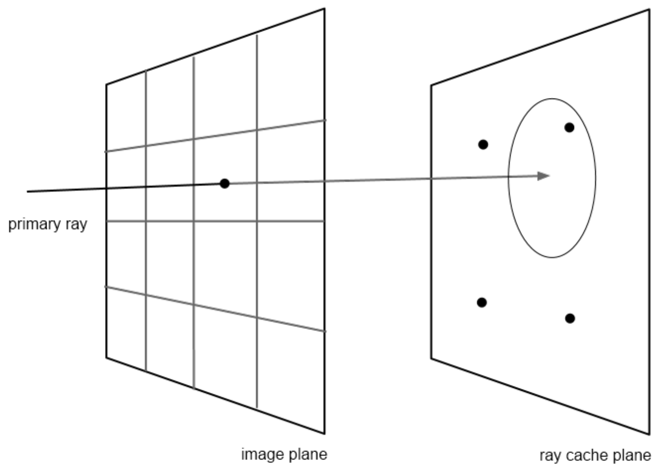


(a) Coarse sampling using a regular pattern.

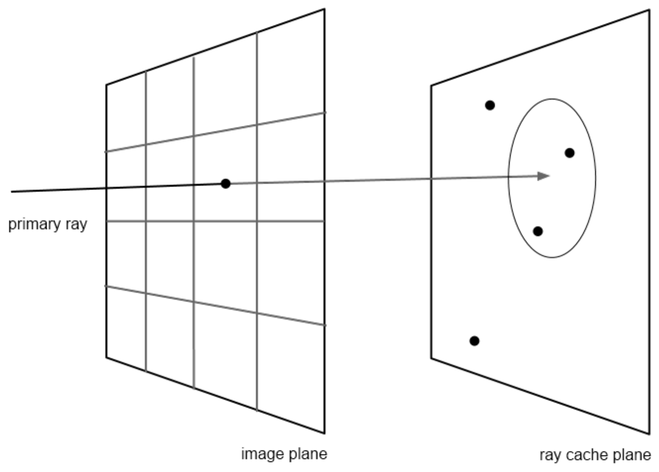


(b) Coarse sampling using a randomized pattern.

Figure 4.3: In the regular coarse sampling pattern, each sample ray is equally spaced within each subsample region on the image plane. In the randomized pattern sample rays are distributed randomly among the subsample regions, which produces a randomized distribution of samples on the ray cache plane.



(a) Searching a cache that has stored regular coarse samples.



(b) Searching a cache that has stored randomized coarse samples.

Figure 4.4: Searching a cache containing regular samples may yield a more even distribution but is susceptible to aliasing due to the regular grid pattern. Searching a cache containing randomized samples trades aliasing for randomized noise, which, in the case shown here, may result in a more accurate colour value. Conversely, a ray passing through the lower right of the image plane may not find any cached samples and must therefore be ray traced through the voxel.

These functions are incorporated into the ray cache implementation by wrapping them into the `cachePopulate()` and `cacheRetrieve()` functions. Recall from Section 3.3 that the ray cache must be able to store the cache plane intersection point, the ray direction and the resulting colour value for a given ray. The `kdtree` object will inherently store the intersection point since it is used as the kd-tree position index. However, the ray direction and colour value must be stored together such that they can be referenced by the `data` pointer. This is accomplished by storing them in the `CacheData` object shown in Listing 4.5. Now, each time a ray and colour value is added to the cache by the `cachePopulate()` function, a new `CacheData` object is created and passed in as the `data` parameter to the `kd_insert()` function. This subsequently allows the `cacheRetrieve()` function to extract the ray direction and colour by calling `kd_res_item()`, which now returns a pointer to a `CacheData` object. The `cacheRetrieve()` function takes in a ray origin and direction, and a `kdtree` pointer parameter and returns a colour value if one is found. It first searches the tree and then iterates through the `kd_res` search results and extracts the `CacheData` pointer for each. Now it can compare the ray directions and accumulate the colour values to produce a colour value for display, using one of the colour retrieval methods described in Section 3.3.

One final consideration for cache storage is the dimension k of the kd-tree. Recall the ray-voxel intersection calculation produces a 3D intersection point where the ray crosses the cache plane. For simplicity, each ray cache could simply store a 3D kd-tree such that the intersection point can be directly inserted. However, since each cache is represented by a 2D plane, only a

Listing 4.5: The CacheData struct contains a ray direction and rgb colour value for storage in the ray cache kd-tree.

```
class CacheData
{
        Vector direction;
        Colour colour;
};
```

2D point is needed, which reduces the storage requirements and potentially reduces search complexity. Converting the 3D intersection point to a 2D cache coordinate is quite simple since the voxels are all axis aligned. For example, if a ray intersects a cache on one of the bounds of the x-axis, the intersection point, p, can be converted to 2D by storing only the p_y and p_z components. After having implemented this 3D to 2D conversion, it appears to effect little difference in overall performance, however each cache now need only store two components for each stored point rather than three.

Chapter 5

Results

This chapter discusses the impact of ray caches on the performance and image quality of the out-of-core distributed ray tracer. Baseline performance and image quality are determined by rendering a scene using a naive ray casting method without the assistance of ray caches. When caches are subsequently employed, performance can be measured by the change in total render time. Image quality is then measured in two ways; visually and numerically. Visually the image quality difference is displayed in a difference image, which simply displays the difference in the red, green and blue component for each pixel; numerically the difference is calculated as a single value, root mean square error (RMS-error). Note that this chapter does not investigate the effect of changes in the cache hit angle parameter ϕ. Since a ray casting algorithm is used for rendering and secondary rays are not cast, ray directions will not significantly differ unless the cache hit radius parameter r increases. If r is small then the rays found in range will inherently travel in similar directions and increasing ϕ will have a negligible effect. Conversely, if r is increased to consider a broader range of rays, increasing ϕ will typically cause image quality to slightly degrade further without a meaningful improvement in cache hit rates or overall performance. In cases where secondary rays or randomized super-sampling are used, this parameter would be worth further investigation.

5.1 Cache Hit Radius

The initial tests extensively compare the overall effects of varying the cache hit radius parameter r. Determining a good value for r is highly dependent on the size of the overall scene space as well as the total number and dimensions of each voxel. The first set of tests, shown in Table 5.1, compares the render performance, image quality and cache hit success rates for rendering a 3x3x3 grid of voxels that each contain a Stanford Bunny model. The Bunny model consists of 69,451 triangles, therefore the overall scene contains 69,451 $\times$ 27 = 1,875,177 triangles. In this case each voxel has dimensions of (0.3, 0.3, 0.3) which produces 2D cache planes with dimensions (0.3, 0.3) between each pair of neighbouring voxels. Note that these dimensions are specified in world space coordinates, therefore if the overall scene size, image plane and voxel dimensions are increased by a factor of 10 then increasing the value of r by a factor of 10 would produce similar image quality results. A selection of the resulting rendered images are shown in Figure 5.1.

To summarize, Table 5.1a shows the rendering performance and resulting image quality error for varying values of r. *Coarse time* shows the total time

r	Coarse Time (s)	Ray Cast Time (s)	Render Time (s)	RMS error
w/o caching	-	3.91	13.12	-
0.001	1.33	6.77	13.43	0.0165
0.002	1.36	6.65	11.37	0.0593
0.003	1.35	6.47	7.73	0.0887
0.004	1.34	6.53	6.81	0.105
0.005	1.35	6.54	6.62	0.111
0.007	1.36	6.55	6.56	0.113
0.009	1.36	7.59	7.59	0.119

(a) Render time performance and RMS-error.

r	Inner Searches	Inner Hits	Inner Ratio	Outer Searches	Outer Hits	Outer Ratio	Total Ratio
0.001	1277203	10315	0.00808	837686	60764	0.0725	0.0849
0.002	955419	81201	0.085	837686	230323	0.275	0.372
0.003	541227	146297	0.270	837686	457472	0.546	0.721
0.004	216882	76183	0.351	837686	655858	0.783	0.874
0.005	59760	27348	0.458	837686	772322	0.922	0.955
0.007	2241	1435	0.640	837686	833497	0.995	0.997
0.009	131	88	0.672	837686	837429	0.999	0.999

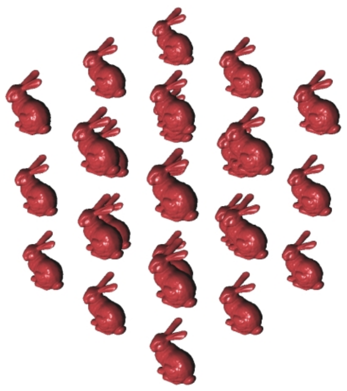
(b) Cache search and hit statistics.

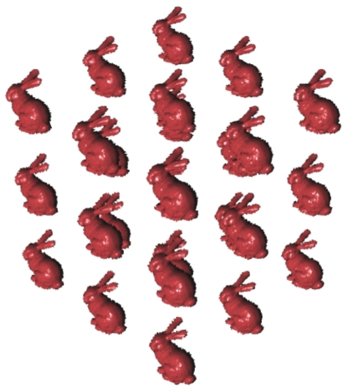
Table 5.1: Rendering performance for a 3x3x3 voxel grid on 27 processors, each containing a Stanford Bunny model. Voxel dimensions are (0.3, 0.3, 0.3) in world space coordinates. All other parameters are constant: Cache hit angle $\cos\phi = 0.005$; 256^2 coarse samples, 1024^2 pixels.

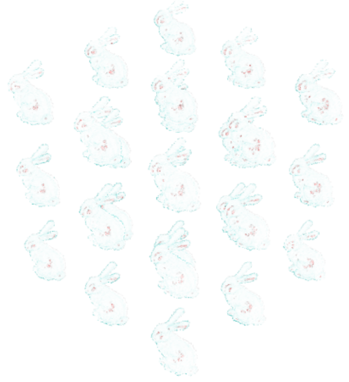
to cast the set of coarse rays and store their results in the ray caches; coarse sampling is investigated further in Section 5.2. *Ray cast time* shows the time needed to cast the primary rays into the scene and send them to the appropriate processor for rendering. Note that the ray cast time also includes the coarse sampling time. *Render time* shows the total time, including coarse sampling and primary ray casting, to render and store a colour for each pixel. Lastly, *RMS-error* is the image quality error between an image rendered with caching enabled and an image rendered naively without caching.

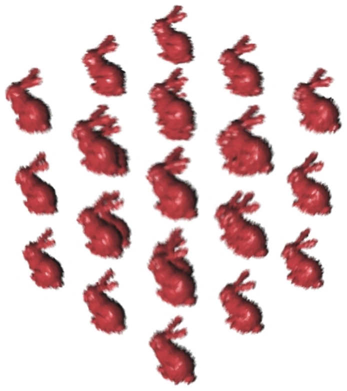
Table 5.1b shows the number of times the caches are searched, produce hits and the resulting hit ratio. This table is split between inner and outer caches to differentiate between the caches that are stored within each voxel (inner caches), and the caches between the scene boundary and the individual voxels (outer caches). This is meant to distinguish between rays that produce a colour value before entering the scene, thus preventing a communication operation, and rays that must traverse the voxels and search subsequent caches, requiring one or more communication operations. Keep in mind that any ray that produces a cache hit at an outer ray cache is not sent through any of the voxels and thus does not sample any of the inner caches. Lastly, *total ratio* shows the proportion of all primary rays for which any cache hit occurs. Also note, for the inner caches, the number of samples is the *total* number of times that the ray caches are searched; for example a single ray may search several inner caches as it travels through several voxels.

Looking now at the results in Table 5.1a, when ray caching is enabled, the ray cast time increases compared to the naive ray tracer without caches due to the added overhead of coarse sampling and searching the ray caches as primary rays enter the scene. Also notice that ray casting time is approximately constant among the variations in r. This is due to the constant number of coarse samples that are stored in the *outer* ray caches during the coarse sampling step. Therefore complexity of searching the outer caches for each primary ray is approximately constant. However, the ray cast time begins to increase as r continues to increase to 0.009 despite the greater probability that a suitable


(a) Rendered without caches


(b) $r = 0.003$


(c) RMS-error = 0.0887


(d) $r = 0.007$

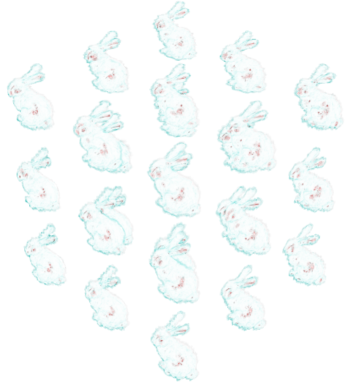

(e) RMS-error = 0.113

Figure 5.1: Comparison of image quality with variation of r when rendering a 3x3x3 voxel grid; each voxel containing a Stanford bunny model.

ray may be found. Looking at the cache statistics shown in Table 5.1b, when $r = 0.009$, the ray casting step appears to retrieve a cached ray from the outer caches for almost 100% of all primary rays that intersect the scene bounding box. Essentially this means that the single processor responsible for casting the primary rays must shoulder the majority of the rendering load, including searching the ray caches, while very little work is assigned to the remaining processors.

Looking now at the overall render time, as r approaches 0, overall performance is actually worse than the naive ray tracer since the vast majority of rays still traverse the voxels and are intersected normally, but with the added overhead of searching multiple ray caches along each ray's path. Once a more reasonable value for r is found, in this case ~ 0.003 to ~ 0.005, overall performance improves quite significantly. At $r = 0.003$, 54.6% of the rays cast into the scene produce a cache hit at the scene boundary and do not require a communication operation to compute a colour value. Adding this to the number of rays that produce a cache hit at an inner voxel, the resulting overall cache hit ratio increases to 72.1%. This produces an image in 7.73s and with rms-error of 0.0887, which is quite reasonable compared to the naive render time. Repeating this for $r = 0.005$ causes the outer hit ratio to increase significantly to 92.2%, while the overall hit ratio increases to 95.6%. Note that, although the inner hit ratio increases, the total number of rays passing through the inner caches decreases quite significantly due to the higher number of outer cache hits. This produces an image in 6.62s with rms-error of 0.111; a slight performance improvement over $r = 0.003$ at the expense of further reduced image accuracy. This highlights the importance of the number of cache hits, specifically cache hits at the outer scene boundaries, in achieving improved performance.

5.1.1 Increasing Scene Size

The next set of results are primarily concerned with the effect of increasing the scene size and increasing the number of voxels to match. Similar to the previous tests, each voxel contains a Stanford Bunny model, however the number of voxels is increased to a 5x5x5 grid distributed among 125 processors. Table 5.2 shows the performance and rms-error results, and Figure 5.3 shows a selection of the rendered images and the resulting difference images.

As expected, looking at Table 5.2a, coarse sampling and ray cast time both increase compared to the 3x3x3 voxel scene. This is partially due to the increased number of voxels that a given ray may pass through, but is also affected by the overall number of coarse sample rays and primary rays that penetrate the scene bounding box; see Figure 5.2. The scene size in this case is 69,451 $\times$ 125 = 8,681,375 triangles, approximately 5 times larger than the 3x3x3 voxel scene. Overall render time, however, only increases by approximately 2.25-2.75 times. This is an encouraging result as it suggests that the distributed out-of-core ray tracer render time will increase at a proportionally lower rate than the increase in scene size. Of course these render times are also highly dependent on the camera position and voxel size with respect to the image plane. For example, in Figure 5.2b, some voxels of the 5x5x5 grid appear to reside just outside the view frustum and potentially don't receive

r	Coarse Time (s)	Ray Cast Time (s)	Render Time (s)	RMS error
w/o caching	-	4.33	32.92	-
0.001	7.23	14.08	37.02	0.027
0.002	7.45	12.71	27.03	0.0829
0.003	7.08	12.68	18.09	0.139
0.004	7.16	12.35	14.68	0.175
0.005	6.78	12.77	14.72	0.19
0.007	6.47	12.71	14.32	0.194
0.009	6.80	13.88	15.46	0.201

(a) Render time performance and RMS-error.

r	Inner Searches	Inner Hits	Inner Ratio	Outer Searches	Outer Hits	Outer Ratio	Total Ratio
0.001	4048369	15881	0.00392	1047664	70297	0.0671	0.0823
0.002	3158712	87526	0.0277	1047664	267296	0.255	0.339
0.003	1972687	140975	0.0715	1047664	535113	0.511	0.645
0.004	948663	119243	0.126	1047664	778988	0.743	0.857
0.005	354439	67338	0.19	1047664	934154	0.892	0.956
0.007	29599	9681	0.327	1047664	1034698	0.988	0.997
0.009	2295	935	0.407	1047664	1046482	0.999	0.999

(b) Cache search and hit statistics.

Table 5.2: Rendering performance for a 5x5x5 voxel grid on 125 processors; each voxel containing a Stanford bunny model. Voxel dimensions are (0.3, 0.3, 0.3) in world space coordinates. All other parameters are constant: Cache hit angle $\cos\phi = 0.005$; 256^2 coarse samples, 1024^2 pixels.

many rays, whereas in the 3x3x3 grid all voxels are visible in the view frustum.

Performance improvement for the varying values of r follow a similar pattern to the smaller scene, reaching a peak performance improvement of $\sim$ 56.5% around a value for r of $\sim$ 0.003 to $\sim$ 0.005. Once again, a smaller r value causes the ray tracer to perform worse than the naive method, while larger values begin to overburden the processor responsible for casting primary rays. The RMS-error is greater overall for the larger scene since the scene now

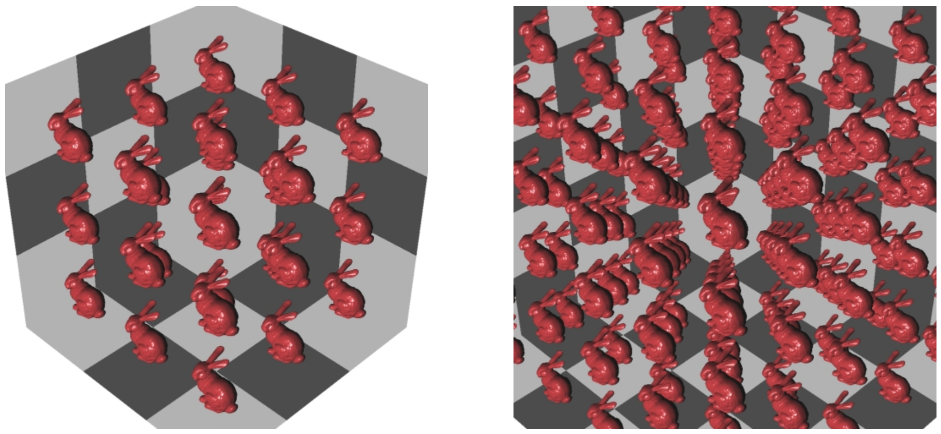
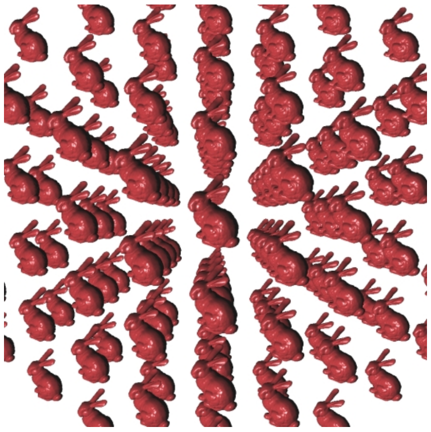

(a) 3x3x3 voxel scene. (b) 5x5x5 voxel scene.

Figure 5.2: The above images show the extents of the voxels when rendering scenes of varying size. Each processor assigns an alternating default colour to any ray that exits the scene, giving the checkered pattern which corresponds to the boundaries of each voxel. Notice the larger screen area covered by the voxel backgrounds for the larger 5x5x5 voxel scene. This means that a greater number of primary rays penetrate the scene bounding box and traverse the voxels.

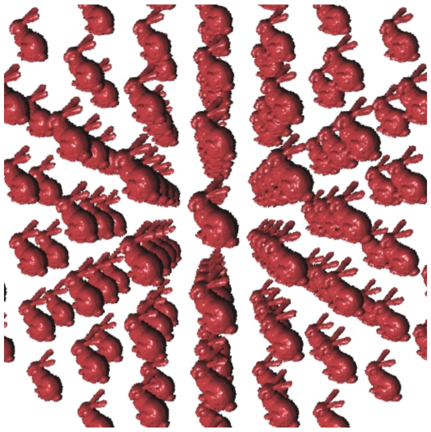
covers a much larger area of the screen space. This causes a larger number of the primary rays to search and retrieve cached colour values, rather than return a default scene colour. Contrasting this with the difference images shown in Figure 5.3, the visual changes in the local areas around the edges of each bunny remain very similar to those of the smaller scene in Figure 5.1.

5.1.2 Subdivision and Irregular Distribution Results

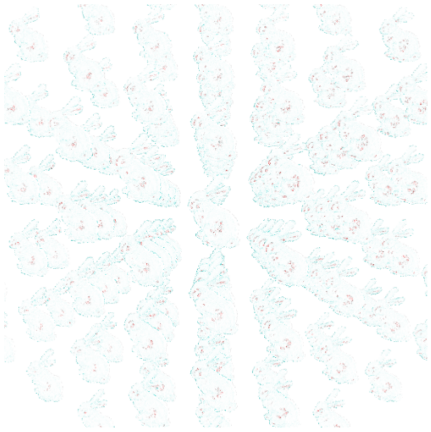
The next set of tests are to compare variations in r for scenes that are unevenly distributed among the grid of voxels. The test scene used for these tests is a set of polygonal stream tubes constructed from streamline paths through a real-world 3D vector dataset from a hurricane. In total, the scene consists of



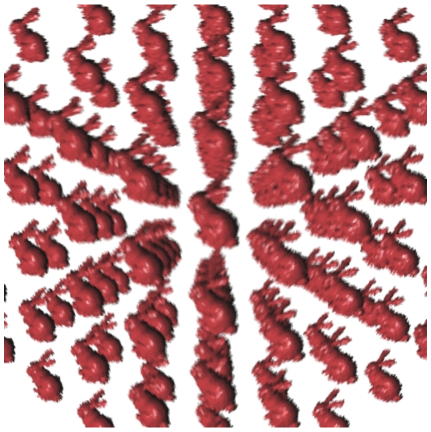
(a) Rendered without caches



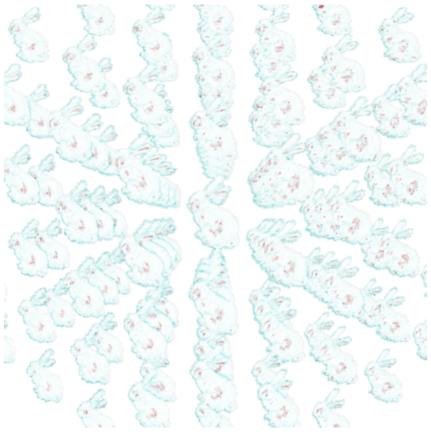
(b) $r = 0.003$



(c) RMS-error = 0.139



(d) $r = 0.007$



(e) RMS-error = 0.194

Figure 5.3: Comparison of image quality with variation of r when rendering a 5x5x5 voxel grid; each voxel containing a Stanford Bunny model.

r	Coarse Time (s)	Ray Cast Time (s)	Render Time (s)	RMS error
w/o caching	-	3.55	59.27	-
0.00075	4.16	11.57	52.33	0.0762
0.00100	4.15	11.61	44.21	0.130
0.00125	4.14	11.66	35.25	0.167
0.00150	4.18	11.29	26.95	0.201
0.00200	4.16	11.50	18.21	0.252
0.00250	4.18	11.70	13.87	0.265
0.00300	4.21	11.98	12.44	0.264

(a) Render time performance and RMS-error.

r	Inner Searches	Inner Hits	Inner Ratio	Outer Searches	Outer Hits	Outer Ratio	Total Ratio
0.00075	2038036	23171	0.0114	1016741	161472	0.159	0.182
0.00100	1736018	120197	0.0692	1016741	278064	0.273	0.392
0.00125	1392919	133769	0.0960	1016741	411679	0.405	0.536
0.00150	1050261	118313	0.113	1016741	546251	0.537	0.654
0.00200	485881	122636	0.252	1016741	771546	0.759	0.879
0.00250	197790	74080	0.375	1016741	897866	0.883	0.956
0.00300	87022	42028	0.483	1016741	956005	0.940	0.982

(b) Cache search and hit statistics.

Table 5.3: Rendering performance for hurricane stream tube scene subdivided by a 3x3x3 voxel grid on 27 processors. Voxel dimensions are (0.315607, 0.275606, 0.112471) in world space coordinates. All other parameters are constant: Cache hit angle $\cos\phi = 0.005$; 256^2 coarse samples, 1024^2 pixels.

355,248 triangles. This is actually a typical example of a worst case test for this implementation. In contrast to the scene containing the Stanford Bunny models, the voxel dimensions do not make a perfect cube and the number of polygons they contain will vary between voxels. This may produce voxels that contain no triangles at all. Also, the relative size of the individual tubes is significantly smaller than that of each individual bunny model. This can cause many rays to be cached that do not intersect the geometry at all, causing some portions of the model to be lost in the final image. Lastly, a more subtle difference from the bunny scene, the bounding grid used to accelerate ray tracing locally on each processor covers the entire area of the voxel, rather than only covering the area of the contained triangles. This means each ray that enters a voxel is much more likely to traverse a bounding hierarchy, instead of passing through the voxel without interruption.

Table 5.3 shows the performance, image quality and cache statistics for rendering the hurricane tubes scene subdivided among a 3x3x3 voxel grid on 27 processors. In this case the voxel dimensions are (0.316, 0.276, 0.112). The x and y components are very similar to those of the bunny scene (0.3, 0.3), resulting in 2D ray cache planes with dimensions (0.316, 0.276), only for those caches perpendicular to the z-axis. However for the remaining caches, the z component is much smaller, resulting in ray caches in the shape of elongated rectangles with dimensions (0.316, 0.112) and (0.276, 0.112), see Figure 5.4. This smaller size does not inherently reduce or otherwise affect the cache effectiveness overall since the search area is still significantly smaller than the area of the cache.

These tests show similar performance improvements to those found for the bunny scene, where a very small r value can potentially worsen performance compared to the naive method, and performance reaches a maximum improvement when bottlenecked by the ray casting step. Looking at the rendered images in Figure 5.5, image quality is still quite good for r=0.001 with rms-error of 0.130 and overall performance improvement of $\sim$15s or 25.4%. For r=0.002 overall performance improvement increases to $\sim$41s or 69.3%. However image

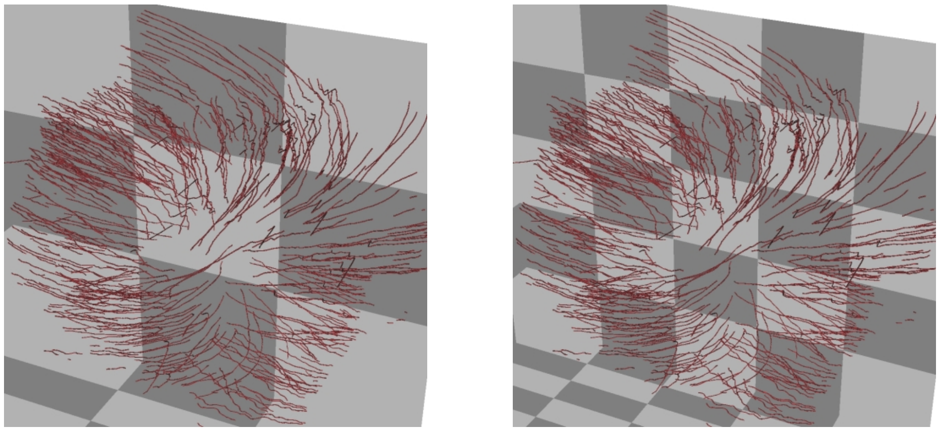
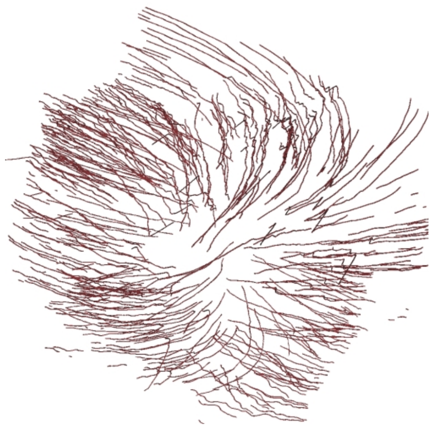
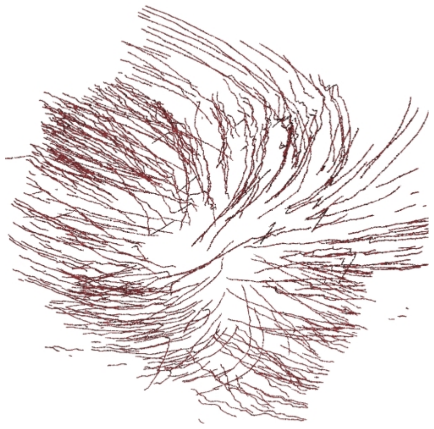

(a) Subdivided by 3x3x3 voxel grid. (b) Subdivided by 5x5x5 voxel grid.

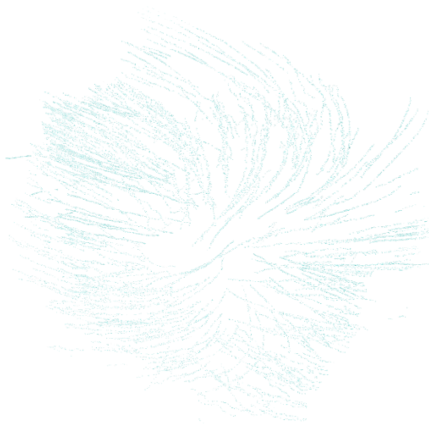
Figure 5.4: The above image show the extents of each voxel when rendering the hurricane stream tubes scene. Notice in this case the extents of the scene bounding box do not change. Also notice the elongated rectangular shape of some of the voxel walls caused by the relative size of the scene on the z axis.

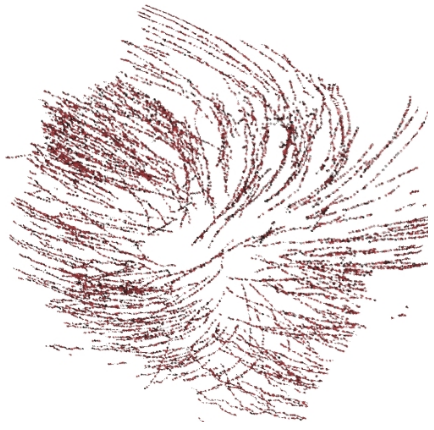
quality does appear to suffer much more significantly, primarily due to the relatively small size of the tubes which reduces the effect of ray coherence. This causes the stream tubes to appear more like a series of blobs, however the overall direction and length of a given tube is still reasonably visible and can provide useful visual information.

To examine the effect of further subdividing the scene, Table 5.4 shows the rendering results of the same hurricane scene, this time subdivided among a 5x5x5 voxel grid on 125 processors. The ray casting time increases quite significantly compared to the 3x3x3 grid, specifically the coarse sampling step takes considerably longer due to the increased number of voxels that each ray must traverse. Overall render time, however, transitions from showing improved performance over the 3x3x3 grid to performance that is slightly worse. The improvement seen in the naive ray tracer and for smaller values of


(a) Rendered without caches


(b) $r = 0.001$


(c) RMS-error = 0.130


(d) $r = 0.002$

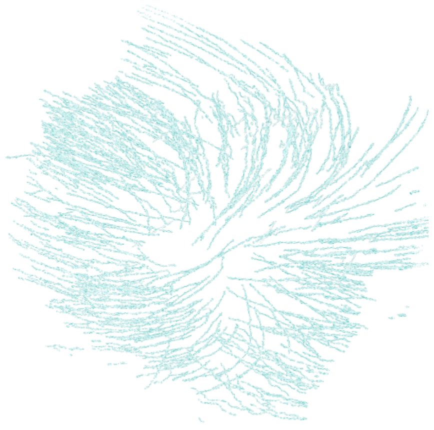

(e) RMS-error = 0.252

Figure 5.5: Comparison of image quality with variation of r when rendering hurricane stream tubes scene subdivided by 3x3x3 voxel grid.

r	Coarse Time (s)	Ray Cast Time (s)	Render Time (s)	RMS error
w/o caching	-	7.38	46.51	-
0.00075	6.98	15.60	44.75	0.0761
0.00100	6.80	13.78	36.52	0.129
0.00125	7.01	14.34	31.19	0.167
0.00150	7.13	14.14	27.45	0.201
0.00200	6.79	13.43	19.69	0.250
0.00250	7.10	13.20	16.29	0.265
0.00300	6.81	13.41	14.76	0.266

(a) Render time performance and RMS-error.

r	Inner Searches	Inner Hits	Inner Ratio	Outer Searches	Outer Hits	Outer Ratio	Total Ratio
0.00075	3983606	26162	0.00657	1016741	161243	0.159	0.184
0.00100	3393093	144536	0.0426	1016741	277532	0.273	0.415
0.00125	2722359	167320	0.0615	1016741	410754	0.404	0.569
0.00150	2053422	150086	0.0731	1016741	544963	0.536	0.684
0.00200	960274	111696	0.116	1016741	769505	0.757	0.867
0.00250	395318	75460	0.191	1016741	895817	0.881	0.955
0.00300	174784	48385	0.277	1016741	954286	0.939	0.986

(b) Cache search and hit statistics.

Table 5.4: Rendering performance for hurricane stream tube scene subdivided by a 5x5x5 voxel grid on 125 processors. Voxel dimensions are (0.189364, 0.165363, 0.0674826) in world space coordinates. All other parameters are constant: Cache hit angle $\cos\phi = 0.005$; 256^2 coarse samples, 1024^2 pixels.

r can be attributed to the increased number of voxels, since a smaller set of geometry is stored at each processor, thereby reducing complexity of finding an intersection within. For larger values of r, performance degrades compared to the 3x3x3 grid due to the increased ray casting time, which causes a bottleneck to the overall render performance. Note that images rendered with the 5x5x5 voxel grid are not shown since they are very similar to the images shown in Figure 5.5.

Coarse Samples	Coarse Cast (s)	Coarse Receive (s)	Ray Cast Time (s)	Render Time (s)	RMS error
w/o caching	-	-	7.38	46.51	-
128^2	2.08	5.73	13.36	44.91	0.115
256^2	2.40	7.01	14.34	31.19	0.167
384^2	3.41	10.12	17.70	24.29	0.195
512^2	4.32	16.43	23.19	24.84	0.187

(a) Render time performance and RMS-error.

Coarse Samples	Outer Stored	Outer Searches	Outer Hits	Outer Ratio	Total Ratio
128^2	15952	1016741	112916	0.111	0.334
256^2	63602	1016741	410754	0.404	0.569
384^2	143036	1016741	723447	0.712	0.811
512^2	254222	1016741	902976	0.888	0.956

(b) Cache search and hit statistics.

Table 5.5: Performance measurement with variation of number of coarse samples when rendering hurricane stream tube scene subdivided by a 5x5x5 voxel grid on 125 processors. All other parameters are constant: Cache hit radius r=0.00125, Cache hit angle $\cos\phi = 0.005$, 1024^2 pixels.

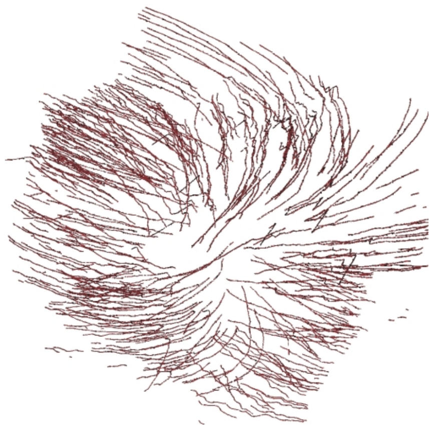
5.2 Coarse Sample Density

The coarse sampling step is performed to provide a certain minimum number of samples that can be searched and retrieved from the ray caches as primary rays are cast into the scene. The density of these coarse samples has a significant impact on performance and image quality. An increased number of samples requires more time for the coarse sampling step to complete but provides a higher number of samples that can potentially be retrieved from the caches. Table 5.5 shows the performance, image quality and cache hit rates when rendering the 5x5x5 voxel hurricane tubes scene with a varying number of coarse samples. Recall the coarse samples are cast in a similar manner to casting the primary rays for the naive ray tracing method where caches are not employed. For example, coarse sample rays are cast through a 128x128 grid across the image plane and fully traverse the scene until an intersection is found or the ray exits the scene. The resulting colour value is then inserted into the appropriate cache at the scene boundary where the ray entered; it is also inserted into one or more of the inner ray caches.

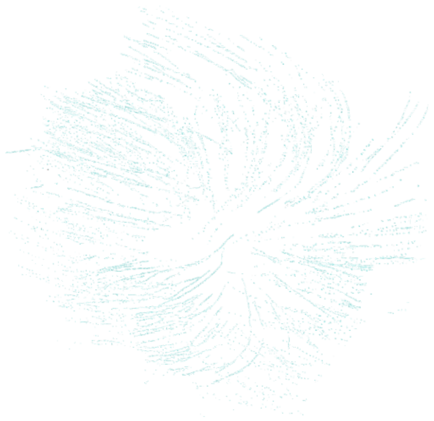
Looking at Table 5.5a, coarse sampling time has been split into *casting* and *receiving*. Casting refers to the time to send the sample rays to the respective voxels, while receiving refers to the total time (including casting) to receive and store the returned colour values in the ray caches. For Table 5.5b, *Outer Stored* refers to the total number of coarse samples that are stored in the outer ray caches before the primary rays are cast. As expected, the increased number of coarse samples causes the coarse sampling time to increase, thereby increasing the overall time to cast the primary rays. This also increases the number of samples that are stored in the ray caches when primary ray casting begins. The effect this has on the rendered image quality is actually quite interesting. When fewer samples are taken, 128^2 for instance, render performance does not significantly improve due to the relatively low number of samples that are stored in the cache, resulting in an overall hit rate of 33.4%. Increasing the number of samples then increases the chance for a primary ray to find a suitable colour value, thereby improving performance while degrading image quality. An interesting result occurs when 512^2 samples are taken, where rms-error actually decreases slightly. This sample density provides the caches with a significantly higher number of samples which inherently increases the accuracy of any colour values that are retrieved. Unfortunately this increases coarse sampling time by such a large amount that overall render performance does not continue to improve. See Figure 5.6 for a visual comparison of image quality and Figure 5.7 for a closer look at the less apparent differences between images rendered using 384^2 and 512^2 coarse samples.

5.2.1 Randomized Coarse Sampling

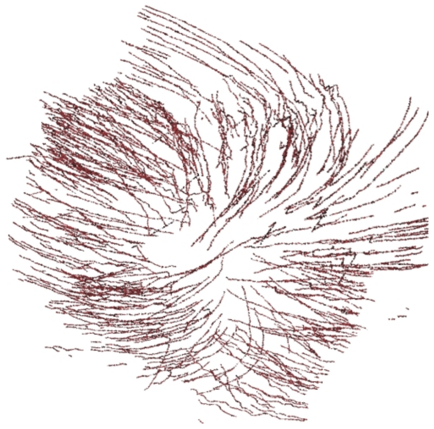
One other aspect of the coarse sampling method is the distribution of the sample rays. As eluded to in Section 4.5.1 casting only a single ray for each coarse sample can cause aliasing similar to that which occurs when a ray tracer uses only one ray per pixel to render an image. Typically this is solved by casting additional rays, but this is not useful here since coarse sampling time must be kept to a minimum. Instead, the use of randomized rays within the area of each coarse sample on the image plane can replace aliasing with randomized noise when rendering the final image. Figure 5.8 shows the difference between the regular and randomized coarse sampling techniques when rendering both the Stanford bunny scene and the hurricane stream tube scene. Recall that the overall cache hit ratio and render performance are not significantly affected by the randomized distribution of coarse samples. However, randomized sampling clearly results in a much more visually appealing image. For the Stanford Bunny scene, since the models are fairly large contiguous objects, the edges of the model and the edges created by specular highlights show signs of aliasing when regular coarse sampling is performed. The hurricane tubes scene suffers similar aliasing artifacts, however they are much more pronounced due to the



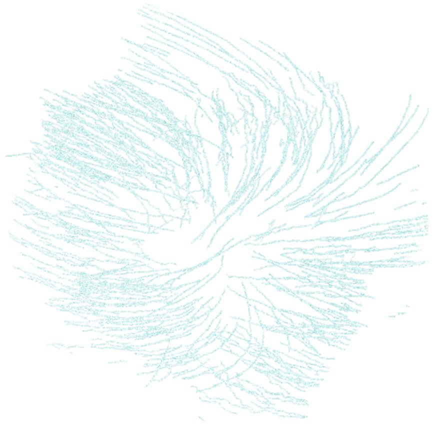
(a) 128^2 samples



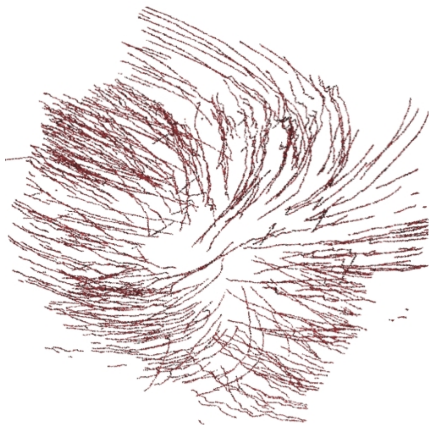
(b) RMS-error = 0.115



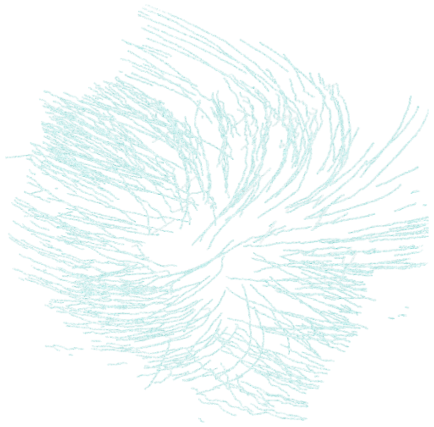
(c) 384^2 samples



(d) RMS-error = 0.195

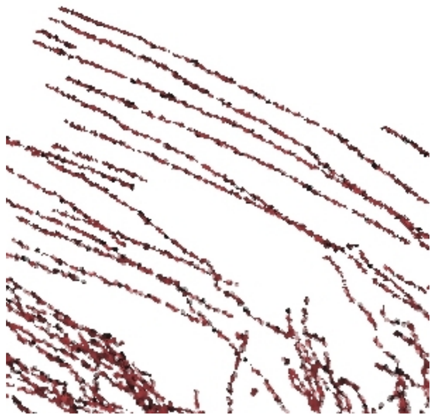


(e) 512^2 samples



(f) RMS-error = 0.187

Figure 5.6: Comparison of image quality with variation of total coarse samples when rendering hurricane stream tubes scene subdivided by 5x5x5 voxel grid.


(a) 384^2 samples; rms-error 0.187

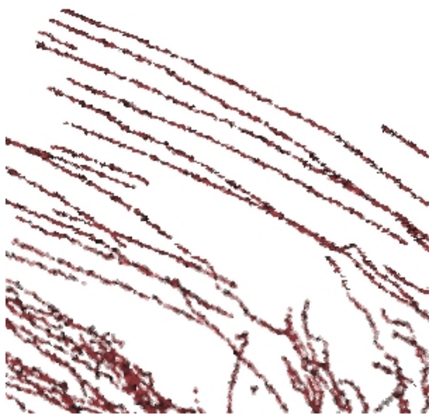
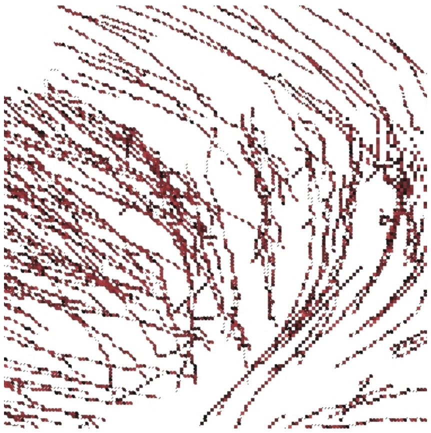

(b) 512^2 samples; rms-error 0.195

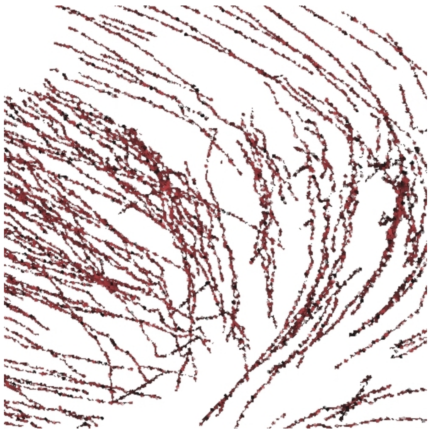
Figure 5.7: Closer comparison of 1024^2 pixel images rendered with 384^2 and 512^2 coarse samples. Notice the slightly more coherent and less "blotchy" tubes in the 512^2 sample image.

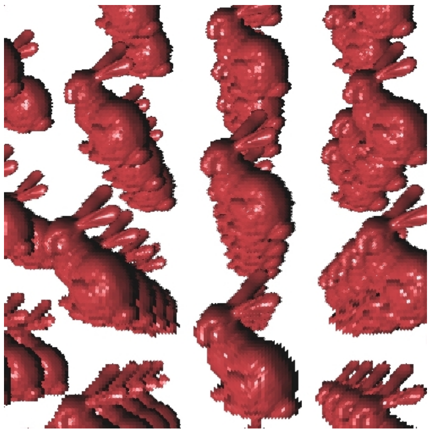
very small thickness of the tubes, which causes discontinuities at some points in the image. With randomized sampling, the edges of the bunny models and continuity of the stream tubes appear much smoother.

5.3 Cache Retrieval and Colour Reproduction

This set of results examines the different methods of extracting a colour from the cache, as described in Section 3.3. This is again measured by rendering the hurricane stream tubes scene, subdivided by a 5x5x5 voxel grid on 125 processors. Recall, the three retrieval methods are 1) find the *first* cached ray that meets r and ϕ tolerance and immediately return its colour value for display, 2) search through all rays within tolerance and retrieve the ray with minimum distance between cache intersection points, and 3) average the result of all rays within tolerance. Table 5.6 shows the performance and RMS-error


(a) Regular coarse sampling


(b) Randomized coarse sampling


(c) Regular coarse sampling

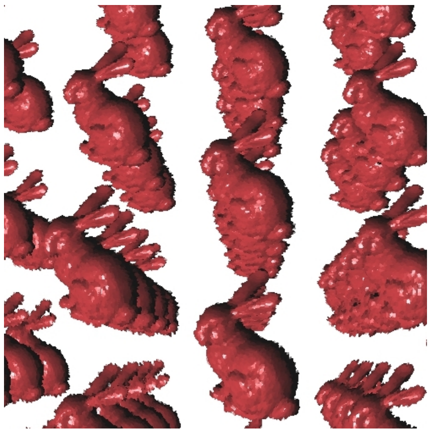

(d) Randomized coarse sampling

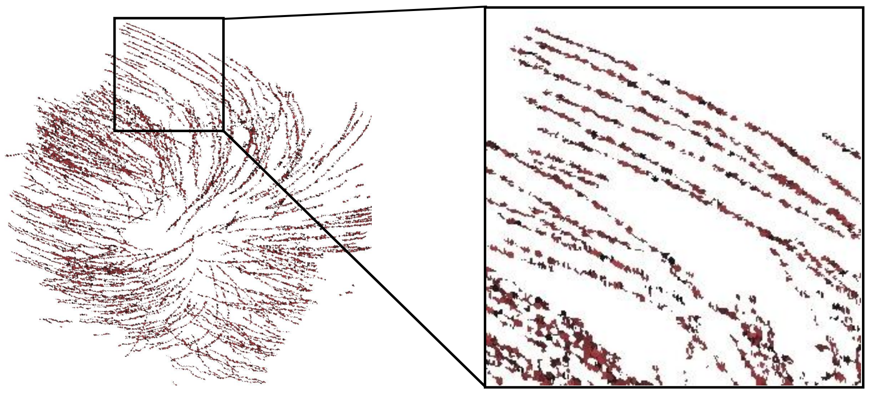
Figure 5.8: These images show the difference between coarse sampling the image plane in a regular grid pattern and randomized sampling. For the hurricane stream tubes, aliasing is quite extreme with regular coarse sampling. For the bunny scene the aliasing effect is most noticable at the edges of each bunny model and the edges of specular highlights.

	Sampling Method	Render Time (s)	RMS-error
w/o caching		46.51	-
$r = 0.001$	first	36.17	0.130
	nearest	36.15	0.129
	average	36.52	0.129
$r = 0.0015$	first	27.27	0.210
	nearest	27.74	0.204
	average	27.45	0.201
$r = 0.002$	first	20.33	0.273
	nearest	20.16	0.258
	average	19.69	0.249

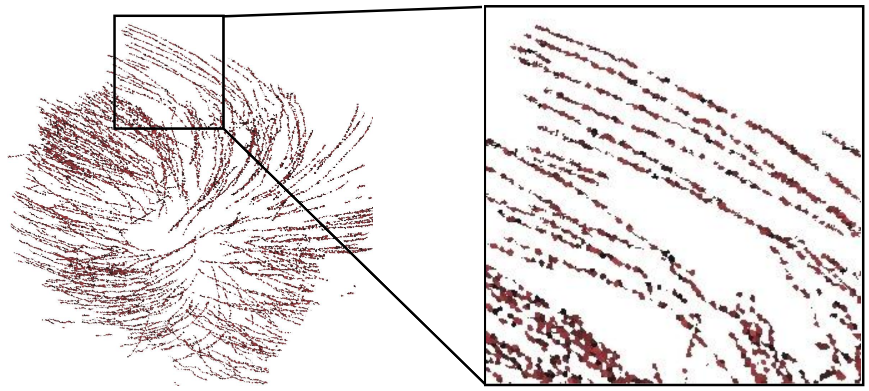
Table 5.6: Performance and image quality comparison for different cache retrieval methods when rendering hurricane stream tubes subdivided by 5x5x5 voxel grid. “First” means first cache result is used; “nearest” means best or most similar result is used; “average” means average of all results is used. Cache hit angle $\cos \phi = 0.005$, 256^2 coarse samples, 1024^2 pixels.

for these retrieval methods when rendering with differing cache search radius, r, values.

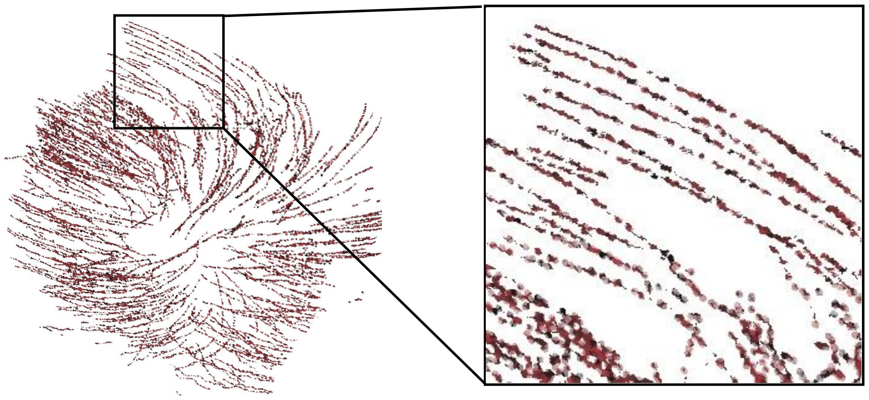
Somewhat surprisingly this has virtually no overall effect on render time performance despite the added computation required by the nearest and average methods. Note that the nearest and average retrieval methods are highly dependent on the number of rays that are found within tolerance. For example, if only one suitable ray is found in the cache, then the nearest and average methods will retrieve the same colour value as the first result method. Because of this, the difference in rms-error between the three retrieval methods increases as the search radius increases, which allows a higher number of samples to be considered by the nearest and average methods. This is most evident for r=0.002 which is shown in Figure 5.9. Retrieving the first re-



(a) First result.



(b) Nearest result.



(c) Average result.

Figure 5.9: Comparison of image quality between first, nearest and average cache retrieval methods when r=0.002.

sult found in the cache appears to cause more severe discontinuities due to its lower accuracy, and since it does not provide any meaningful performance improvement this method should be avoided. The difference between nearest and average results is somewhat superficial. Selecting the nearest result will tend to provide a more accurate colour value, however, if the nearest result is nearer the maximum tolerance this may not provide a better image than the first result method. The average result method can compensate for this situation by considering multiple rays, rather than just one, which trades accuracy for a slightly more blurred image.

5.4 Memory Usage

The previous results have shown that several hundred thousand rays rays may traverse the voxels that contain the scene. Each processor then has to accommodate many thousands of rays and colour values that may be received in its incoming queue, as well as many thousands of rays and colours that may be stored in its ray caches. Due to the parallel nature of this system it is quite difficult to display the number of items in the incoming queue at any given time and for all processors. Instead, memory is assessed by measuring the maximum memory usage attained throughout the entire run time for one particular processor that exhibits the highest memory usage. Table 5.7 shows typical highest memory usage for a processor that is responsible for a single voxel.

Highest memory usage is typically attained for voxels that are located

	Maximum Queue Size	Total Rays Cached	Total Memory
3x3x3 bunny scene	5636	2497	484KB
5x5x5 bunny scene	1729	2565	212KB
3x3x3 tubes scene	38269	3435	2.75MB
5x5x5 tubes scene	11458	1158	846KB

Table 5.7: Overview of memory usage when rendering bunny and hurricane stream tubes scenes. Bunny scene parameters are r=0.003, $\cos\phi$=0.005, 256^2 coarse samples, 1024^2 pixels. Tubes scene parameters are r=0.0015, $\cos\phi$=0.005, 256^2 coarse samples, 1024^2 pixels.

adjacent to the scene boundary and are the first voxel traversed by primary rays entering the scene. Voxels located further into the scene will typically receive fewer rays in its incoming queue since many rays will be terminated at a previous voxel, and thus the processor will store fewer rays in its caches. Notice the incoming queue size of the tubes scene is considerably larger than that of the bunny scene. This occurs because many rays passing through the bunny scene will find an intersection and immediately return a colour, while many rays passing through the tubes scene will pass straight through a given voxel. Most rays will pass through the tubes scene voxel due to the smaller relative size of the individual tubes. This causes many rays to be sent to a neighbouring processor, which will eventually send a colour value back through the incoming queue, thus increasing its size. The total number of rays stored in the caches stays approximately constant for the bunny scene since the voxel size remains the same. However, for the hurricane tubes scene, the increased voxel grid dimensions causes the voxel size to decrease and thus fewer rays intersect it.

Memory usage is not measured directly, rather it is calculated from the number of objects stored multiplied by the size of their respective containers. This gives a good gauge for the amount of memory that is required for each processor. The memory requirements for a single voxel appears to peak at ∼2.75MB. Considering that the cluster processors have upwards of 8GB-16GB of memory available, overall memory usage for a voxel is negligible.

The ray casting processor, rank 0, has the added responsibility to store all the outer caches around the scene bounding box as well as its own voxel caches. For this processor, maximum incoming queue size appears to peak at approximately 400,000 in the worst case while the total number of rays stored in the outer caches is dependent on the number of coarse samples that are cast. When 256^2 coarse samples are stored, memory usage for rank 0 peaks at ∼28.2MB, while if the number of coarse samples is increased to 512^2, memory peaks at ∼34.8MB. Although this is a significant increase over the individual voxel memory requirements, overall memory usage is still quite minimal.

Chapter 6

Conclusions and Future Work

This work has presented a novel approach to distributed out-of-core ray tracing and demonstrated a prototype implementation that leverages the Sharcnet cluster super computing environment. There is a growing need within the Sharcnet community to integrate visulization with simulation computation to avoid the pitfall of offline visuzliation. This work contributes to this need by demonstrating a method of reducing the bottleneck of communication through the use of ray caches in order to quickly produce a visualization without significantly delaying a simulation in progress.

Ray caches certainly show potential in providing a significant performance boost to distributed out-of-core ray tracing. A ray cache placed at the boundary between the datasets stored on two processors can collect and store any ray that passes through it. When a new ray is cast through this boundary, it can search the cache and retrieve a stored ray where appropriate, rather than incurring an expensive communication and intersection calculation. The cost of searching the ray cache is kept to a minimum by storing the rays in a kd-tree structure. The implementation presented here demonstrates that the ray cache can have a drastic effect on overall rendering performance, even when a fairly simplistic search method is used. Also, under suitable conditions, ray caches do not severely reduce image quality. This can be extremely valuable in applications such as viewing or steering a live simulation where a reasonably good quality image, generated quickly, can yield important information. However, there are clearly several aspects of ray caches that would benefit from further examination and research.

As with most rendering acceleration methods, there are some instances where the out-of-core ray tracer and ray caches perform quite poorly. For instance if the camera is located within the bounds of a scene or its view is concentrated on one particular voxel, then all primary rays will have to pass through one processor before reaching other processors in the cluster. This prevents an even distribution of work among the processors, which are forced to idle for extended periods, and will adversely affect performance.

Another shortcoming of the current implementation is the static nature of the cache search criteria. Rays are compared based on the distance between their cache plane intersection points followed by the similarity in their direction of travel. This method can become ineffective when the ray cache plane is at an oblique angle in relation to the ray (see Figure 6.1). When this occurs, two rays that follow a similar path may intersect the cache plane at very distant points. To compare the ray intersection points with the static radius parameter, r, would then yield no results despite a similar ray that is present

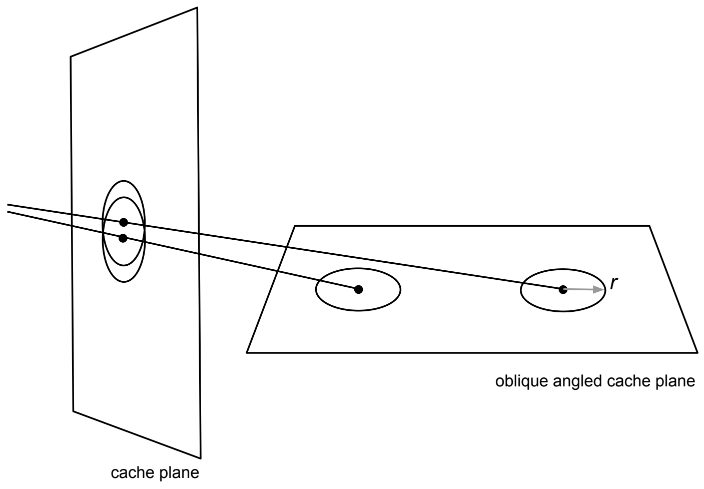


Figure 6.1: Notice the two rays follow a similar path and would produce a cache hit in the first cache plane on the left. However, in the cache oriented at an oblique angle, the rays would not intersect at a close enough position to produce a hit.

in the cache. Another more subtle issue can occur when two rays that appear similar at a cache plane may travel large distances and find intersections in very different areas of the scene. Ray caching would then benefit from a more sophisticated method of cache search and retrieval that takes cache orientation and ray intersection distance into account.

Currently the distributed out-of-core ray tracer only performs a ray casting algorithm to render an image. This is intentional since the system is primarily focused on scientific visualization rather than photo realistic image synthesis. However, the ray caches themselves are agnostic to the ray tracing algorithm that each processor performs. Therefore reflections, shadows and global illumination could be supported relatively easily. One major consideration for implementing these effects is the added volume of rays and colour values that would be sent and received, which would potentially overwhelm the incoming queue in the render loop.

Casting the primary rays into the scene is also quite limited. Currently, ray casting is front loaded since *all* rays are cast by one processor before it proceeds with rendering in its local voxel. This could be improved by casting rays in parallel where each processor is responsible for a subset of the image pixels to distribute the burden currently placed on a single processor. This could also be improved by casting only a subset of the rays periodically rather than all rays at once. This allows one set of computed rays to be returned and stored in the caches where a subsequent set of rays can then retrieve them. This is already performed to a limited extent in this implementation through the coarse sampling step, but it is limited to only casting two sets of rays: coarse samples and primary rays for each pixel. Casting rays periodically would also be beneficial if the out-of-core ray tracer were extended to support animation, i.e. camera movement. The ray caches would also provide a significant benefit to animation since small camera movements would not significantly change the direction of most primary rays. This means rays added to the caches in one frame can be retrieved for display in a subsequent frame.

Ray caches show encouraging results for distributed out-of-core visualization, but much work is still needed to apply them in more realistic applications.

Bibliography

[1] Open-source kdtree library implementation for c. `http://code.google.com/p/kdtree/`.

[2] Abraham, F., and Filho, W. C. Distributed visualization of complex black oil reservoir models. In *EGPGV* (2009), pp. 87–94.

[3] Akenine-Möller, T. Fast 3d triangle-box overlap testing. *journal of graphics, gpu, and game tools 6*, 1 (2001), 29–33.

[4] Appel, A. Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference* (New York, NY, USA, 1968), AFIPS '68 (Spring), ACM, pp. 37–45.

[5] Dietrich, A., and Slusallek, P. Massive model visualization using realtime ray tracing. In *ACM SIGGRAPH 2008 classes* (New York, NY, USA, 2008), SIGGRAPH '08, ACM, pp. 41:1–41:33.

[6] Fuchs, H., Kedem, Z. M., and Naylor, B. F. On visible surface generation by a priori tree structures. In *Proceedings of the 7th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1980), SIGGRAPH '80, ACM, pp. 124–133.

[7] GLASSNER, A. S., Ed. *An introduction to ray tracing.* Academic Press Ltd., London, UK, UK, 1989.

[8] GORTLER, S. J., GRZESZCZUK, R., SZELISKI, R., AND COHEN, M. F. The lumigraph. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1996), SIGGRAPH '96, ACM, pp. 43–54.

[9] GOURAUD, H. Continuous shading of curved surfaces. *IEEE Trans. Comput. 20* (June 1971), 623–629.

[10] HECKBERT, P. S., AND HANRAHAN, P. Beam tracing polygonal objects. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1984), SIGGRAPH '84, ACM, pp. 119–127.

[11] IGEHY, H. Tracing ray differentials. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1999), SIGGRAPH '99, ACM Press/Addison-Wesley Publishing Co., pp. 179–186.

[12] JENSEN, H. W., AND CHRISTENSEN, N. J. Photon maps in bidirectional monte carlo ray tracing of complex objects. *Computers & Graphics 19*, 2 (1995), 215–224.

[13] KAY, T. L., AND KAJIYA, J. T. Ray tracing complex scenes. In *Proceedings of the 13th annual conference on Computer graphics and interactive*

techniques (New York, NY, USA, 1986), SIGGRAPH '86, ACM, pp. 269–278.

[14] Larson, G. W., and Simmons, M. The holodeck interactive ray cache. In *ACM SIGGRAPH 99 Conference abstracts and applications* (New York, NY, USA, 1999), SIGGRAPH '99, ACM, pp. 246–.

[15] Levoy, M., and Hanrahan, P. Light field rendering. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1996), SIGGRAPH '96, ACM, pp. 31–42.

[16] Phong, B. T. Illumination for computer generated pictures. *Commun. ACM 18* (June 1975), 311–317.

[17] Reshetov, A., Soupikov, A., and Hurley, J. Multi-level ray tracing algorithm. In *ACM SIGGRAPH 2005 Papers* (New York, NY, USA, 2005), SIGGRAPH '05, ACM, pp. 1176–1185.

[18] Rubin, S. M., and Whitted, T. A 3-dimensional representation for fast rendering of complex scenes. In *Proceedings of the 7th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1980), SIGGRAPH '80, ACM, pp. 110–116.

[19] Shinya, M., Takahashi, T., and Naito, S. Principles and applications of pencil tracing. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1987), SIGGRAPH '87, ACM, pp. 45–54.

[20] Wald, I., Dietrich, A., and Slusallek, P. An interactive out-of-core rendering framework for visualizing massively complex models. In *ACM SIGGRAPH 2005 Courses* (New York, NY, USA, 2005), SIGGRAPH '05, ACM.

[21] Wald, I., Ize, T., Kensler, A., Knoll, A., and Parker, S. G. Ray tracing animated scenes using coherent grid traversal. In *ACM SIGGRAPH 2006 Papers* (New York, NY, USA, 2006), SIGGRAPH '06, ACM, pp. 485–493.

[22] Wald, I., Slusallek, P., Benthin, C., and Wagner, M. Interactive rendering with coherent ray tracing. In *Computer Graphics Forum* (2001), pp. 153–164.

[23] Walter, B., Drettakis, G., and Parker, S. Interactive rendering using the render cache. In *Rendering techniques '99 (Proceedings of the 10th Eurographics Workshop on Rendering)* (New York, NY, Jun 1999), D. Lischinski and G. Larson, Eds., vol. 10, Springer-Verlag/Wien, pp. 235–246.

[24] Ward, G., and Heckbert, P. Irradiance gradients. In *Eurographics Rendering Workshop* (May 1992), pp. 85–98.

[25] Ward, G. J., Rubinstein, F. M., and Clear, R. D. A ray tracing solution for diffuse interreflection. In *Proceedings of the 15th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1988), SIGGRAPH '88, ACM, pp. 85–92.

[26] Whitted, T. An improved illumination model for shaded display. In *Proceedings of the 6th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1979), SIGGRAPH '79, ACM, pp. 14–.

[27] Williams, A., Barrus, S., Morley, R. K., and Shirley, P. An efficient and robust ray-box intersection algorithm. In *ACM SIGGRAPH 2005 Courses* (New York, NY, USA, 2005), SIGGRAPH '05, ACM.

[28] Yoon, S.-E., Lauterbach, C., and Manocha, D. R-lods: fast lod-based ray tracing of massive models. *Vis. Comput. 22* (September 2006), 772–784.

[29] Yu, H., Wang, C., Grout, R., Chen, J., and Ma, K.-L. In situ visualization for large-scale combustion simulations. *Computer Graphics and Applications, IEEE 30*, 3 (2010), 45 –57.