Raytracing Dynamic Scenes on the GPU Using Grids

Sashidhar Guntury and P.J. Narayanan

Abstract—Raytracing dynamic scenes at interactive rates have received a lot of attention recently. We present a few strategies for high performance raytracing on a commodity GPU. The construction of grids needs sorting, which is fast on today's GPUs. The grid is thus the acceleration structure of choice for dynamic scenes as per-frame rebuilding is required. We advocate the use of appropriate data structures for each stage of raytracing, resulting in multiple structure building per frame. A perspective grid built for the camera achieves perfect coherence for primary rays. A perspective grid built with respect to each light source provides the best performance for shadow rays. Spherical grids handle lights positioned inside the model space and handle spotlights. Uniform grids are best for reflection and refraction rays with little coherence. We propose an Enforced Coherence method to bring coherence to them by rearranging the ray to voxel mapping using sorting. This gives the best performance on GPUs with only user-managed caches. We also propose a simple, Independent Voxel Walk method, which performs best by taking advantage of the L1 and L2 caches on recent GPUs. We achieve over 10 fps of total rendering on the Conference model with one light source and one reflection bounce, while rebuilding the data structure for each stage. Ideas presented here are likely to give high performance on the future GPUs as well as other manycore architectures.

Index Terms—Raytracing, grids, ray coherence, load balancing, GPU.

1 Introduction

Raytracing has been a method of choice for producing photorealistic images. Interactive and real-time raytracing using the Graphics Processor Unit (GPU) has received a lot of attention recently with the increase in their computation power. Raytracing has two major parts: the construction of a suitable acceleration data structure and computing the ray-geometry intersections. The data structure building is a preprocessing task for static scenes and only the intersections need to be performed at interactive rates. Efficient and hierarchical structures like Bounding Volume Hierarchies (BVH) and its variants, Kd-trees, and variations of grids have been used as acceleration structures. The structure needs to be rebuilt every frame when retracing dynamic scenes involving deformable geometry. The structure building and tracing need to be performed in real time then. The acceleration structure and the tracing method should be selected to minimize the total time.

High performance needs GPU-friendly structures and algorithms as a result. The GPU has a large number of cores and favors massively multithreaded algorithms with regular operations and memory access patterns. The manycore CPU architectures are likely to share these characteristics in the future. Regular structures like grids can be built and operated efficiently on such architectures compared to irregular structures like BVH and Kd-tree. We explore the use of grids as the acceleration structure for interactive raytracing in this paper. Grids have been explored for raytracing on the CPU [1], [2] but their advantages are more pronounced on a GPU-like architecture. Grids use a sorting-like step for construction and can be implemented efficiently on the GPU. This makes them particularly well suited for dynamic scenes. The quick building time makes them more attractive though they lack hierarchy to eliminate large portions of the scene quickly.

In this paper, we explore fast raytracing of dynamic scenes with primary, shadow, and secondary rays on a modern GPU using grids. We minimize the total time to build the structure and trace the rays. We advocate the use of appropriate acceleration structures for each stage, as different rays from different stages behave differently. Fast grid construction achieves high overall performance even if the data structure is built multiple times per frame.

We use a perspective grid in camera space for the primary rays and a perspective grid in light space for shadow rays, building on our earlier work [3], [4]. Camera-space perspective grid provides perfect coherence to primary rays. Frustum culling and back-face culling (BFC) reduce the geometry to be processed as the grid is built per frame. The user-managed shared memory of the GPU is used effectively for the tracing step [3]. We build a grid of smaller voxels to limit the number of triangles in each but trace multiple voxels together to better exploit the the large SIMD width on GPUs [4]. A spherical perspective grid with respect to each light source enables tracing shadow rays like primary rays. Coherence is increased using a load-balancing step.

We use a uniform grid for reflection and refraction rays, as they have no natural coherence. Grid construction using sorting is fast on the GPUs, allowing repeated construction for each frame and each stage. We describe two ways to

• *The authors are with the Center for Visual Information Technology (CVIT), International Institute of Information Technology, Gachibowli, Hyderabad 500 032, Andhra Pradesh, India.*
E-mail: sashidhar@research.iiit.ac.in, pjn@iiit.ac.in.

Manuscript received 22 Aug. 2010; revised 12 Jan. 2011; accepted 28 Jan. 2011; published online 2 Mar. 2011.
Recommended for acceptance by J. Ahrens and K. Debattista.
For information on obtaining reprints of this article, please send E-mail to: tvcg@computer.org, and reference IEEECS Log Number TVCGSI-2010-08-0190.
Digital Object Identifier no. 10.1109/TVCG.2011.46.

1077-2626/12/$31.00 © 2012 IEEE Published by the IEEE Computer Society

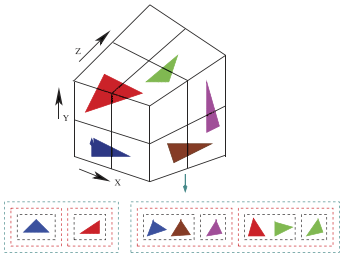

Fig. 1. The triangle storage layout. This kind of layout is achieved by keeping the X value in the MSB and Z extent in the LSB.

trace the reflection rays. The Enforced Coherence (EC) method brings all rays passing through a cell together using sorting. This enables simultaneous checking of intersections of all rays with triangles of a grid cell. The intersection step exploits the coherence by using the shared memory. This provides high performance on today's GPUs even if the sorting is an additional step. The Independent Voxel Walk (IVW) method traces each ray independently using a GPU thread loading the triangles in its way. The moderate caches on the current GPUs provide good performance for this simple method. The EC method works best on older generation GPUs with no caches, while the IVW method is slow on them. Both methods perform similarly on recent GPUs with a richer memory architecture. Future GPUs and manycore architectures are likely to favor the IVW method as the cores are more independent.

We raytrace the Conference model with a single light source and one bounce of reflection rays on an Nvidia GTX480 at 8 fps using the EC method and at 14 fps using the IVW method. Two perspective grids and one uniform grid are constructed in each frame. Our approach will provide good results on other GPUs of today and will be beneficial to the future GPUs and other manycore architectures as they are likely to share many of these characteristics.

2 Background and Previous Work

Raytracing has been widely studied recently. A survey of the current techniques for raytracing can be found in [1]. Here, we describe some of the recent work which is directly related to our own.

Considerable effort has gone into speeding up the process of building data structures, especially on GPUs. Zhou et al. [5] constructed Kd-trees and Lauterbach et al. [6] constructed BVHs on the GPU. Methods proposed by Dammertz and Keller [7] and Ernst and Greiner [8] recommend splitting triangles to get a better quality data structure. Stich et al. [9] and Popov et al. [10] proposed variations to BVHs to improve their quality. The construction of Spatial BVH (SBVH) proposed in [9] is considerably higher as an efficient building method is mapped onto GPU is still an issue. Such a data structure is really good for static scenes but not for scenes with dynamically changing geometry. Kd-trees and BVHs are still expensive to build on GPU, especially if they need to be built every frame. Patidar and Narayanan used a perspective grid to raycast dynamic scenes using a fast building of the grid in each frame [3]. Their triangle storage layout is shown in Fig. 1 which we use in our grid implementation as well. Their approach was sensitive to triangles distributions spanning arbitrary number of voxels. Kalojanov and Slusallek [11] solved this problem on uniform grids and also used an appropriate grid resolution to improve the quality. Our method uses perspective grids and uniform grids, incorporating ideas to make them insensitive to triangle distribution. Resolution of grids was analyzed by Ize et al. from an algorithmic point of view to predict the number of voxels given a model's characteristics [12].

Unlike on a BVH and a Kd-tree, rays on grids cannot be handled as packets easily. Wald et al. [13] presented an algorithm to traverse the grid in a slice-wise coherent manner. Due to the use of a frustum like grid, Hunt and Mark [14] and Patidar and Narayanan [3] treat primary rays as coherent packets of rays. These packets of rays work together bringing optimal amount of data and reusing it. Hunt and Mark [14] suggested the idea of rebuilding the data structure from the light point of view on the CPU. Coherence is an even more critical factor for good performance on the GPU. We extend their ideas to the GPU but go much further by building a spherical grid to increase efficiency as well as to support spotlights and lights within the scene, building on our earlier work [4]. We also balance the workload among the many cores available as shadow rays tend to be concentrated.

There has been some work on enforcing coherence among secondary rays. Pharr et al. [15] and Navratil et al. [16] proposed reordering techniques on multicore CPUs. The ray reordering technique proposed by Pharr et al. [15] queues rays and schedules the processing of this queue in a way to minimize cache misses and I/O operations. Recently, Moon et al. [17] suggested the use of Hit Point Heuristic and Z-curve filling-based ray reordering to achieve cache oblivious coherence on multicore architectures. They concentrate on simplifying the model and using these simplified models for global illumination methods such as path tracing and photon mapping. There has been some work on secondary rays on the GPUs. Budge et al. [18] analyzed the bottlenecks during path tracing a complex scene and proposed a software system that splits up tasks and schedules them appropriately among CPU and GPU cores. Garanzha and Loop [19] demonstrated a method to treat shadow rays from point and area light sources. Their reordering scheme requires them to build virtual frustums and reorder rays according to these frustums. The reordering technique which we propose doesn't use queues. We don't need to construct a virtual or a scheduling grid to reorder the rays since our basic structure itself is a grid. Our method uses primary hit points from ray casting for reordering the rays. Aila and Karras [20] proposed possible extensions to hardware which can speed up secondary rays. Their treatment is from a hardware point of view studying the cache performance. We concentrate on speeding up the tracing of reflection rays.

2.1 GPU Computing Model

We implement our raytracing techniques using the CUDA [21] programming model on Nvidia GPUs. CUDA uses kernels, which are programs that run in parallel on all threads. A huge number of threads—up to tens of thousands—is launched for efficiency. The threads are

grouped into thread blocks or CUDA blocks. Threads within a single CUDA block can be synchronized with negligible overhead. They also have access to a small, fast, on-chip shared memory. Global memory is accessible to all threads, but is considerably slower. The new generation of Nvidia GPUs (Fermi) has L1 and L2 caches. The amount allocated to each of the L1 and shared memory can be configured for each kernel. The L2 cache is accessible to all threads of the kernel. The threads are batched into warps of 32 threads, and are scheduled sequentially on available processor resources. Thus, the SIMD width of GPU computing model is the size of the warp. Memory access patterns of threads of a warp also affect performance deeply. Memory performance is best if data used by all threads of a block are loaded onto the shared memory. Additionally, if nearby threads access global memory locations that are close, performance will receive a boost as the data in those locations will be cached in L1 and L2 cache. The GPUs from other manufacturers have similar architectures at the lowest level. Thus, our approach will scale well to those, given the use of a standard programming model like OpenCL.

3 Grid Data Structure

Constructing a grid data structure involves dividing the world into voxels and binning scene geometry in those voxels. On the GPU, grids have been constructed as both uniform grids [11] and perspective grids [3]. Unlike BVH or Kd-tree, grids do not possess hierarchy and therefore do not have the tree property which is extremely useful in eliminating triangles before the actual intersection checking. Also, the notion of packets in BVH and Kd-tree is straightforward while this notion is not so straightforward in uniform grids.

However, by building a perspective grid, one gets perfect coherence for primary rays. We can treat primary rays as packets. Similar to the technique used in our earlier work [3], we process the rays of each tile together on GPU using a CUDA block or a work group, with each pixel assigned to a thread or a work item. The triangle data are brought into the shared memory before intersection calculations. Since all threads need to process all triangles in the voxel, the overhead of bringing the triangles is amortized over the intersections. The threads alternate between loading a portion of the triangles into shared memory and computing intersections for them, with a synchronization between these two roles. A thread that has found an intersection at one voxel need not check for intersection in a later voxel, as the voxels are processed in a front-to-back order. We use an optimized routine for checking triangle intersection [22].

The perspective grid data structure that we build for primary rays is ray specialized [14] for primary rays and cannot efficiently handle other rays. The cost of building a grid is low as it reduces to a sorting of triangle-voxel pairs [11]. This is one of primary reasons why rebuilding the data structure multiple times is attractive. We also employ View Frustum Culling (VFC) and Back-Face Culling for eliminating triangles which are not going to be tested by camera rays. This removes a large part of the geometry in the scene and results in lower grid construction times. This advantage is not available for the uniform grid for general secondary rays.

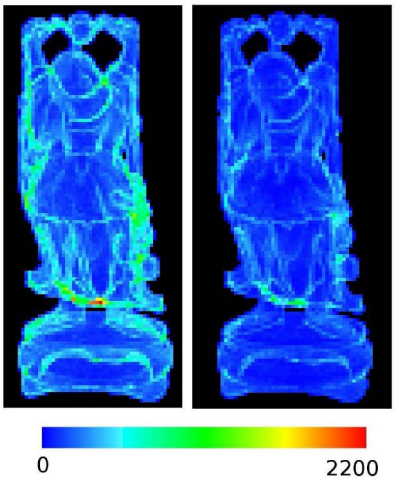

Fig. 2. Heat map showing the number of triangles checked before declaring intersection. Left image corresponds to direct mapping while there is marked reduction in indirect mapping (right). Number of triangles checked before declaring intersections increases from blue to pink and is highest in yellow regions.

The size of the image-space tiles and voxels in the grid can impact the performance. A tile is a coherent, rectangular cross section of rays. Larger tiles may exploit greater coherence than smaller ones. However, smaller tiles and voxels result in fewer overall ray-triangle intersection calculations due to a finer sorting. The SIMD width of the architecture also affects the performance, as the computing resources may be wasted if the number of threads used is below the SIMD width. We use an *indirect mapping* of threads to strike a balance between these conflicting demands.

We sort the triangles to smaller tiles, but raytrace using larger number of threads, by mapping threads differently. In practice, we sort the triangles to $kN \times kN$ tiles in image space. For raytracing, we divide the image into $N \times N$ tiles such that a $k \times k$ group of sorting tiles fits into each raytracing tile. The work groups used while tracing have more threads. The available shared memory is partitioned equally among the sorting tiles during raytracing. Triangles from each sorting tile are brought to the respective area of the shared memory and are intersected with the rays corresponding to the sorting tiles. The configuration of 2×2 sorting tile within each tracing tile provides the best results on current GPU hardware.

The most computationally intensive part of the entire raytracing routine is the triangle intersection part and that is where indirect mapping helps. Indirect mapping reduces the overall triangles to be checked. For the Happy Buddha benchmark, we got 10-30 percent speedup using indirect mapping as the maximum number of triangles checked dropped by more than half. Fig. 2 shows this using a heat map for the work done.

4 Spherical Light Grid for Shadows

Primary rays generate an intersection point for each pixel from which another set of rays are spawned. For secondary rays, methods to efficiently trace shadows were described by

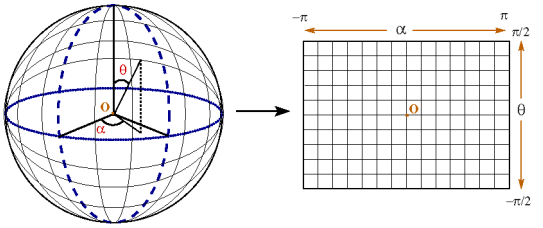

Fig. 3. Spherical space used for shadows.

Wald et al. [13]. Their technique involves a slice-wise coherent traversal. Though their method works well on CPU, it's not well suited on wide SIMD architecture like GPU. Furthermore, near the silhouettes of the object, shadow rays are not always coherent which would introduce divergence in the ray packet. The resulting packet would have to include a large number of triangles and thus lots of wasteful intersection checks. Shadow rays share many aspects of primary rays. In the case of point-based lights, shadow rays converge to a point. In many ways, it's like a role reversal from the point of view of lights and therefore shadow rays can be treated similar to primary rays.

One way to exploit the aforementioned coherence is by listing the voxels for each shadow ray and then merging them. This has been found to be expensive on the CPU [13] and merging is prone to perform poorly on the GPU too. Since building the grid data structure is cheap, building it again from the point of view of the light source is feasible on GPU. The process of tracing the shadow ray is then similar to that of tracing the primary rays.

4.1 Building Light Grids and Ray Mapping

We build a perspective grid with the light source taking the role of the camera. The camera has an intrinsic direction and a field of view. This is not natural for light sources. Point light sources emit light in all directions. We use a spherical mapping to map light's world into a perspective grid. A light frustum is constructed in the α-θ space where α and θ are the azimuthal and elevation angles (latitude-longitude scheme). Fig. 3 shows the spherical space with respect to the forward, right, and up directions. A rectangle in the α-θ space defines the light frustum and plays the role of the image for primary rays. We define "tiles" on this rectangle to build voxels of the grid using constant depth planes. The angle α is measured from the forward direction in the forward-right plane and the angle θ is measured from the forward direction in the forward-up plane. Lower and upper limits on the distance from the light source play the role of near and far planes. This method however suffers from the demerit of pole singularity. All triangles lying in the right-forward plane (θ is $\frac{\pi}{2}$), will be duplicated along all the tiles near the pole. This loss in performance can be mitigated by choosing a "good" forward direction. Choosing the line joining the light position to the centroid of the model helps us limit the number of triangles along the pole.

Fig. 5. Clamping the region to simulate spotlights.

Spherical mapping of this kind treats all directions equally. We would ideally want to handle only the geometry which is visible to the camera. For this, we limit the angular extents of the light's frustum to the bounding box of projection of the camera's view frustum. Fig. 4 demonstrates this reduction of triangles participating in the grid building and ray-triangle checking. Furthermore, it also devotes the grid tiles to a smaller area thus dividing the area more finely. This technique also points a way to implement proper spotlights with light falloff. The spot can be marked as a bounding rectangle in the spherical space shown in Fig. 4. Fig. 5 demonstrates the spotlight achieved using this technique. A cubemap style of ray mapping to limit light space rays was used earlier [14]. They handle each frustum separately, resulting in a lot of extra work for the traversals. Furthermore, clamping a cubemap is very tedious when it has to identify the grid which a ray has to check. In contrast, spherical mapping provides a more unified framework to compute shadows.

4.2 Ray Reordering

The shadow rays emanate from the intersection points of primary rays and travel toward the light source. Rays in the primary space that are distant may follow similar paths to the light source, as shown in Fig. 6. The primary intersection points are recorded against each ray at the end of the primary step. We map the starting points of the shadow rays to spherical space of the light source and store the tile number

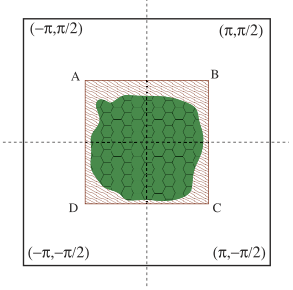

Fig. 4. Bounding rectangle of the geometry in spherical space defines the light frustum of interest.

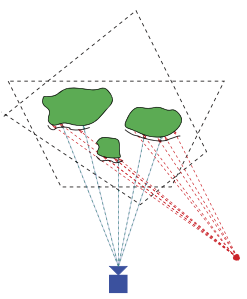

Fig. 6. Points of primary ray intersection are mapped to the light frustum.

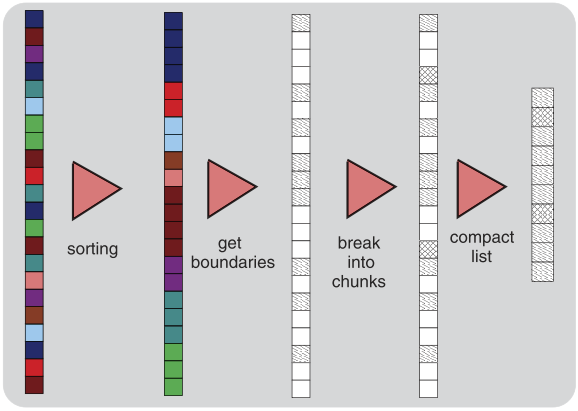

Fig. 7. Reordering illustrated. Different colors correspond to different voxels. Sorting results in all colors coming together. *Get Boundaries* gets the locations where enumeration of a new voxel starts. Based on threshold value (3 here), rays are divided into chunks and compacted in a tight array.

for each. Thus, a pair of primary ray and light tile number is created for each shadow ray. This list is sorted with the tile number as the key to bring shadow rays that belong to the light tile together. This brings similar coherence to secondary rays as the primary ones, with the information about each shadow ray that passes through the tile available. Shadow ray generation and reordering are performed on GPU in parallel using scan primitives [23] from the CUDPP library.

4.3 Load Balancing

For tracing primary rays, blocks of threads are assigned to tiles directly or indirectly. This is efficient as the number of rays in each tile is constant. For shadow rays, however, the number per tile can vary widely. The above thread mapping strategy can be inefficient due to the imbalance in workloads. We try to keep the number of rays handled by each thread block below a maximum value. This needs assigning multiple thread blocks to excessively populated light tiles. We do this by splitting tiles with more than a maximum number of shadow rays into multiple logical tiles. Sparsely populated tiles, however, cannot be merged as they work on different triangle data.

Suppose a light tile has $R > r$ rays mapping to it, where r is the number that a thread block can handle efficiently. We assign $\lceil R/r \rceil$ blocks in the CUDA program to this tile. Other tiles are mapped to one thread block each, after eliminating empty ones. The total number of thread blocks needed is $C_{total} = \sum_{j=1}^{N} \lceil R_j/r \rceil$, where R_j is the number of rays in tile j.

Ray Reordering and *Load Balancing* are illustrated in Fig. 7. The array of (primary ray, light tile number) pairs for each primary intersections is sorted with tile number as the key. A kernel marks the boundaries of different tiles and marks their positions. We call these boundaries hard boundaries. A segmented scan on it gives us the number of values having the same key. This allows one to break this (possibly huge) packet into multiple sizeable chunks. This is done by marking the boundaries on the existing array which had the hard boundaries. We call these new boundaries as soft boundaries. Every hard boundary (different key) is also a soft boundary (mapped to a different CUDA block.) To keep track of the first ray in each block, we do a stream compaction step and shrink the number of voxels (in the spherical grid space) to C_{total}. In Fig. 7, $r = 3$ is used. Thus, we have a list of locations to the values each of which belongs to a different CUDA block. The difference between two adjacent terms in the array gives us the number of rays belonging to that CUDA block. This completes our load-balancing step and we can invoke as many CUDA blocks as the number of elements in the compacted array.

Algorithm 1. Reordering Load Balancing Shadow Rays

```
totalrays ← image size

for ray < totalrays in parallel do
  sphericalMap(tileIDArr[ray], rayIDArr[ray])
  pseudoArr[ray] ← 0
  scratchArr[ray] ← 1
  oArr[ray] ← 0
end for

sort(tileIDArr, rayIDArr)
getBoundaries(tileIDArr, pseudoArr)
segScan(pseudoArr, scratchArr)
getChunks(scratchArr, validArr)
numBlocks ← compact(validArr, oArr)
```

5 Reflection Rays

Reflection rays exhibit very little coherence for general scene geometry. That is, the reflection rays go off in all directions and need to check against different voxels of the grid. Since there is no preferred point or direction unlike primary and shadow rays, perspective grids are not especially useful for reflection rays. There is no natural direction to create a perspective space for such rays. However, the construction of a grid structure is inexpensive and should be used for reflection rays also. The perspective grids built for primary and shadow passes cannot be used directly as they use BFC and VFC to eliminate unnecessary geometry for efficiency. Since there is no general direction of reflection rays, we cannot eliminate triangles. We, therefore, construct a uniform grid in object space to handle reflection rays, on the lines described by Kalojanov and Slusallek in [11]. Rays can traverse a uniform grid as outlined by Amanatides and Woo in [24], by walking from one voxel to another using a *voxel walk* procedure. We describe two methods to trace reflection rays using a uniform grid on the GPU.

Independent voxel walk. Each ray can walk along the voxels it encounters by computing the next voxel, starting with the starting point. Each ray checks intersection by loading the triangles of the voxel it encounters. This continues until the first intersection is found, when the ray terminates. In this method, traversal and intersection checking are tightly knit. This method is a simple one and assumes no coherence between rays. The lack of coherence can incur heavy penalty on the older GPUs with large SIMD width and no caching to exploit access locality that may be present across threads over a period of time. The newer architecture (Fermi) has a moderate L1 cache for a group of processors and a larger L2 cache for all processors. The L1 cache can be shared by threads of a CUDA block and the L2 cache can be used by all threads. The independent voxel walk method can benefit from these caches if multiple rays are checking intersection for the same voxel simultaneously or close together in time. Pseudocode for the independent voxel

walk method is presented in Algorithm 2. As the future GPUs and manycore architectures are likely to have even more flexible caching mechanisms, this type of an approach will continue to benefit from the architectural improvements.

Algorithm 2. Independent Voxel Walk

```
totalrays ← image size

for ray < totalrays in parallel do
  while ray has not found intersection do
    voxel ← determineVoxel(ray)
    for all triangles in voxel in serial do
      checkIntersection()
      if found intersection then
        break
      end if
    end for
  end while
end for
```

Enforced coherence method. Tracing primary rays is totally coherent because we can identify groups of rays passing through each voxel easily. The triangles of the voxel are brought to shared memory and the intersection calculations can be performed from the shared memory. We can enforce coherence by processing all rays that pass through a voxel together.

Enforced coherence method involves reordering the rays to force coherence. To do this, we first determine the voxels which each ray passes through to get a list of (ray, voxel) pairs. We sort this list using the voxelID as the key to bring all rays that traverse each voxel together. We can now use a procedure similar to primary raytracing by allocating a CUDA block to each voxel. Load is balanced by allocating multiple CUDA blocks, each processing a packet of rays, to voxels that have large numbers of rays passing through them, similar to the handling of shadow rays. The start-to-finish ordering of each ray is, however, lost in this process and rays cannot terminate on finding the first intersection. Thus, if a ray passes through multiple voxels, it has to be processed as part of each packet.

The first step in intersection checking is loading the triangle data to shared memory. This is done by the threads in the CUDA block, each of which brings data of one triangle from global memory. For voxels which have more triangles than the number of threads, triangles are brought in batches. Each batch is completely used before loading the next batch of triangles. As already mentioned, the ordering of rays is lost, due to which one cannot terminate the process of checking the rays by finding the first intersection. Therefore, one has to find all the intersections and then get the closest one among them.

Reordering is computationally intensive (sorting, compaction, etc.) and memory intensive. The number of (rayID, voxelID) pairs for a typical conference room scene is about 10 million, as each ray passes through 10 voxels on an average. This number only indirectly depends on the geometry of the scene through the reflection ray origins and directions. The overheads incurred in reordering and minimum finding can be offset by the coherence we obtain using this method. The pseudocode of the enforced coherence method is presented in Algorithm 3.

Algorithm 3. Enforced Coherence Method

```
totalrays ← image size

for ray < totalrays in parallel do
  countArr[ray] ← 0
  countArr[ray] ← DetermineVoxels(ray)
end for

totalvoxels ← 0
for ray < totalrays in parallel do
  totalvoxels ← totalvoxels + countArr[ray]
end for

allocate memory(tileIDArr, rayIDArr)

for ray < totalrays in parallel do
  DumpVoxels(rayIDArr[ray], tileIDArr[ray])
end for

for i < totalvoxels in parallel do
  pseudoArr[i] ← 0
  scratchArr[i] ← 1
  oArr[i] ← 0
end for

sort(tileIDArr, rayIDArr)
getBoundaries(tileIDArr, pseudoArr)
segScan(pseudoArr, scratchArr)
getChunks(scratchArr, validArr)
numBlocks ← compact(validArr, oArr)

for all blocks in numBlocks do
  while all rays in block do
    load triangles to shared memory
    check for intersection
  end while
end for

for rays < totalrays in parallel do
  get minimum of all intersections
end for
```

6 Results and Analysis

We implemented the techniques described in the earlier sections on an NVIDIA 280 GTX and NVIDIA 480 GTX card using an Intel Core2Duo Q6600 processor running a 32-bit Linux system. The system rendered each scene at a resolution of $1{,}024 \times 1{,}024$. Our analysis is divided into two sections—primary, shadow rays; and reflection rays. Our method during each frame computes a matrix inverse for each triangle and stores it for faster intersection checking [22]. The cost for computing these matrices is quite low as the task can be easily parallelized and GPU architecture is optimized for vector and matrix operations.

6.1 Primary and Shadow Rays

We tested primary and shadow rays on a number of scenes and models, shown in Fig. 8. We build the grid data

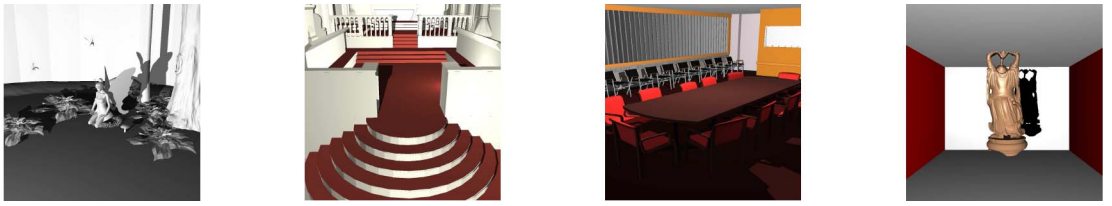

Fig. 8. Some of the test scenes: Fairy Forest (174 k), Sibenik Cathedral (82 k), Conference Room (284 k), and Buddha (1.09 M).

TABLE 1
Time in Milliseconds for Primary and Shadow Rays for Different Stages for Our Method and an Implementation of Kalojanov and Slusallek [11]

		Fairy (174K)		Sibenik (82K)		Conference (284K)		Happy (1.09M)	
		Our	Kal 09	Our	Kal 09	Our	Kal 09	Our	Kal 09
Stages	Primary DS Build	3.92	16.65	3.11	9.22	4.11	13.47	9.04	12.04
	Primary Ray Traversal	5.88	72.26	3.18	54.27	2.98	44.25	6.70	40.10
	Shadow DS Build	4.19	0.0	3.58	0.0	5.15	0.0	9.08	0.0
	Data Rearrangement	3.78	0.0	3.69	0.0	3.72	0.0	3.69	0.0
	Shadow Ray Traversal	6.09	122.73	5.69	43.73	4.52	46.64	8.43	81.18
	Total	23.86	211.68	19.25	107.22	20.48	104.36	36.94	133.32
	FPS	41.9	4.72	51.94	9.32	48.83	9.58	27.07	7.50

They use a uniform grid structure for primary and shadow rays. Times are on a GTX480 GPU.

structure every frame. Table 1 gives a detailed breakup of various stages for a frame. Unlike primary rays, shadow rays are not equally distributed among all tiles making shadow ray traversal (including shadow DS build and rearrangement) more expensive than primary ray traversal. Data rearrangement consists of mapping the shadow rays to the spherical map, clamping the spherical map, sorting shadow rays, binning them, and performing a stream compaction. These steps work on the rays in the image space and are independent of the scene to be rendered. The time taken is more or less the same for all scenes. We compare the performance of perspective grids for primary and shadow rays for one light source with the method by Kalojanov and Slusallek that uses uniform grids [11]. Table 1 shows that perspective grids are three to five times faster.

Fig. 11 compares our grid building with state-of-the-art BVH building procedure proposed by Pantaleoni and Leubke [26] for building HLBVH and HLBVH + SAH. Grid building is fast due to its simplicity. A typical HLBVH with SAH build is about 10 times more expensive. In the construction of a grid, the only costly operation is sorting of the triangles to the voxels. Perspective grids admit BFC and VFC to reduce the number of triangle-voxel pairs to sort.

Spatial BVH is a better quality BVH than others, providing more efficient traversal for intersections [9]. It takes advantage of splitting triangles to improve the quality of the BVH [7], [8]. It makes these splitting decisions during the tree construction (on a per node basis) after evaluating a cost function. This makes SBVH a really good data structure for traversal of static scenes. For dynamic scenes, the overhead of building the data structure or refitting it might outweigh the performance improvements. Its build time on GPU would be greater than the time required for the construction of HLBVH + SAH whose timings are shown in Fig. 11. A simple implementation of SBVH on CPU takes 72 seconds to construct an SBVH for Happy Buddha Model which has about 1.09 M Triangles. Fig. 10 compares the tracing performance of our method with one using the SBVH with a speculative while-while scheme [25]. Fig. 10 compares the

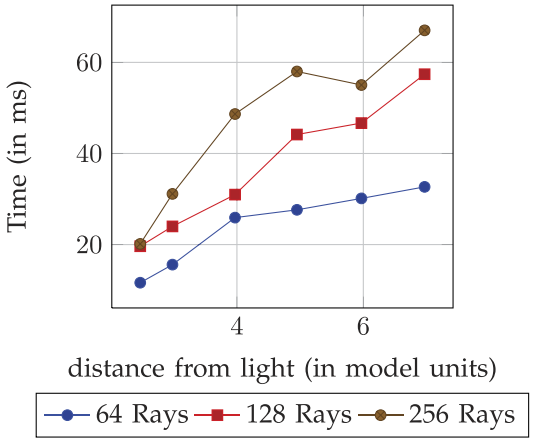

Fig. 9. Time taken to compute shadows as a function of light distance from Fairy in the model. Light moves away in the direction of the line joining the light to the center of the fairy model. Times were taken for chunks of three different sizes—64, 128, and 256. Values are as measured on GTX 280.

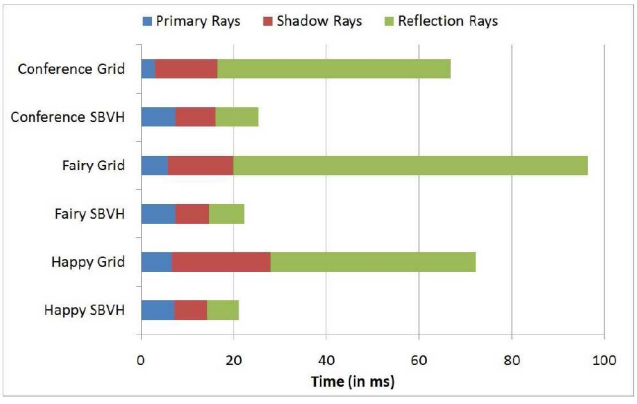

Fig. 10. Comparison of traversal times between our method (Grid) and SBVH traversal (SBVH) [25] for various passes in a frame, viz. Primary, Shadow and Reflection rays. For shadow and secondary, time taken to rebuild the data structure and rearranging the data is also included. Numbers are as noted on NVIDIA GTX 480.

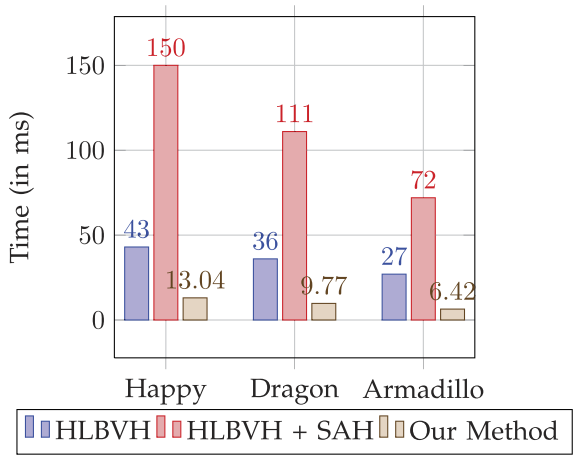

Fig. 11. Comparison of HLBVH construction [26] with grid construction of our method. Numbers are as noted on NVIDIA GTX 280. Happy is 1.09 M tris, Dragon is 871 K tris, and Armadillo is 345 K tris.

tracing times using perspective grids and SBVH on a GTX480. Our method is very good for primary raytracing, but loses out on shadow rays and reflection rays. The additional traversal time of grids in the case of shadows will be more than offset by the savings in build times as seen earlier.

Our method constructs a frustum grid from the point of view of light to trace the shadow rays. This method is sensitive to the distance of light from the model as a lot of rays can be bunched in a tile leading to unequal distribution of work among CUDA threads. Load balancing alleviates this problem. Fig. 9 shows the performance of our shadow tracer as light moves away from the centroid of the fairy model. Binning 64 rays to one CUDA block provides consistent performance.

Our perspective grid-based method processes rays that go through a voxel in a CUDA block. The corresponding threads check intersection with the same set of triangles. They are brought to the shared memory and managed explicitly. On a cache-less architecture like GTX280, the shared memory speeds up the computation considerably. The performance on the GTX480 with L1 and L2 caches is different. The shared-memory-based method is only about 5 percent faster than an independent voxel walk method in which ray-triangle intersections are processed independently from the global memory.

6.2 Reflection Rays

We compare the performance of the IVW and EC methods on different GPUs. For this analysis, we used a $128 \times 128 \times 128$ voxel resolution for all scenes. We used the radixsort from CUDPP [27] to sort the ray-voxel pairs. We focus on three representative models for this analysis.

Conference model. This model has a room with a table, a few chairs, and walls. This model has triangles reasonably uniformly distributed in the scene space and has large horizontal or vertical triangles. As a result, the reflection rays behave well and may have a high degree of coherence.

Fairy in the forest model. This model is mostly sparse with dense geometry at a few locations in space. The normals vary considerably which makes the reflection rays quite incoherent.

Buddha model. This is a scanned model with all the geometry bunched in a tight space. The model is finely tessellated because of which the normals vary considerably in nearby areas. Since the number of triangles is high, intersection checking might dominate the tracing time. For this study, we render the model by itself, with reflections only from itself.

Table 2 summarizes the results on the three models from the viewpoints given in Fig. 13. The enforced coherence method is slower than the independent voxel walk method on the GTX480, as the latter can exploit the caches well. In contrast, the EC method is much faster on the GTX280 on Fairy and Happy Buddha models. They perform similarly on the Conference model, perhaps due to the moderate coherence of the reflection rays on this model. The reordering time of the EC method is avoided by the IVW method. Table 2 also shows the number of ray-voxel pairs created during the enumeration step. The number is large on models with a lot of empty space and affects the performance of the EC method, as it needs more data movement for sorting.

We analyze the performance of reflection rays on these models. Fig. 12 shows the percentage of rays that find their intersections as IVW iteration proceeds. An iteration for a ray is the processing of a single voxel, beginning with the starting voxel. The Buddha model starts slow but behaves the best overall with 80 percent of the rays terminating in fewer than 80 iterations. This is because all reflections are self-reflections which need only a few iterations. Other rays terminate when they cross the bounding box of the model. The Conference model starts well, but the progress is slower after 60 iterations. The Fairy model starts and progresses slowly, needing over 450 iterations for completion. The timing performance (Table 2) mirrors this directly with Buddha model attaining the best reflection performance.

TABLE 2
Time in Milliseconds for Reflection Rays in Each of the Broadly Classified Stages

Model	GPU	DS Build	(Ray,voxel) Pairs	CUDA Blocks	EC			IVW Trace	speedup IVW / EC
					Reorder	Trace	Total		
Conference	280	31.16	10.46 M	262 K	98.46	158.62	257.03	247.61	0.96
	480	13.47			57.21	40.12	97.33	36.92	0.37
Fairy	280	37.31	15.57 M	191 K	152.23	252.51	404.74	679.76	1.67
	480	16.65			87.54	37.22	124.76	59.77	0.47
Happy	280	25.43	2.61 M	190 K	28.61	101.18	129.79	263.62	2.03
	480	12.04			15.56	43.71	59.27	32.43	0.54

The fourth column gives the number of ray-voxel pairs created during the enumeration of rays and the fifth column gives the number of blocks assigned after compaction step. The last column gives the relative performance of the EC and IVW methods.

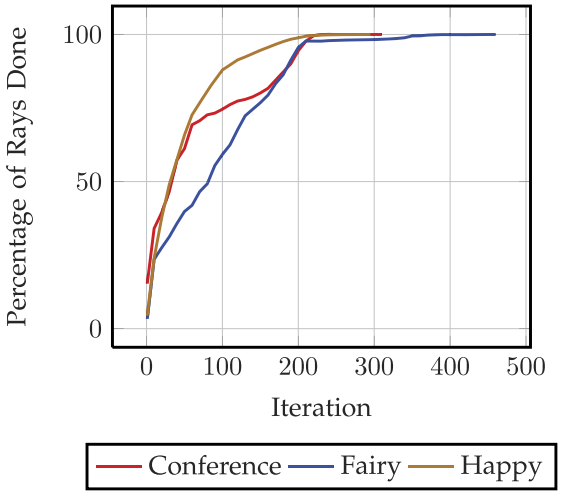

Fig. 12. Percentage of rays declaring intersection at each step of iteration. Fairy grows very slowly, taking 454 iterations to check reflections. In contrast, conference takes 306 iterations. Happy Buddha takes just 294 iterations before declaring the status of the reflected rays.

We study how the reflection rays are distributed among the voxels. The top left of Fig. 14 shows the ray concentration by voxels for the first iteration (or set of voxels explored) of the IVW method for the three models. Most voxels of the Buddha model have fewer than 50 rays passing through them, while the other models have a few hundred voxels with over 400 rays in them. Rays are processed in parallel by different CUDA threads. If there are more rays in the voxel, the corresponding threads check the same set of triangles for intersection and reuse the same data. This is a situation that can make good use of the L2 cache shared by all threads of the GPU (as the threads processing these rays may come from different streaming multiprocessors). Buddha performs the worst in exploiting the L2 cache, but its overall performance is best due to early termination seen before. Top right of Fig. 14 zooms into the tail of the ray distribution plot. The Conference model outscores the Fairy model with a larger number of dense voxels. The relatively bad performance on Fairy can be explained partly by this.

The bottom left of Fig. 14 shows the divergence present within each primary tile or packet of rays processing the reflection rays. During IVW, the reflection rays are still processed as packets corresponding to the tiles of the primary rays. If the number of voxels in a packet or a tile is low, the IVW method will have more rays of the CUDA block accessing the same triangles. This will efficiently use the L1 cache. Most tiles have low divergence in both Conference and Fairy models not on the Buddha model. The early part of the plot (bottom right of Fig. 14) shows that the Conference model exhibits lower divergence than the Fairy model and performs better, as is confirmed by the tracing times we obtained.

Ray distribution and tile divergence are thus good predictors of reflection performance. If the triangle normals are mostly parallel (as with the Conference model), the reflection rays will be largely coherent, if a coherent packet of rays hits it. This will reduce the tile divergence and improves the performance with the use of L1 cache. If the triangle distribution is sparse and the triangles have widely varying normals (as with the Fairy model), the reflection rays emanate at few places and travel in all directions. This reduces the number of rays per voxel and diminishes the overall performance.

Figs. 15 and 16 show results for some other scenes like Fairy, Sibenik, and Conference with a simulation in midair. Fig. 16 places the Dragon-Bunny collision in the Conference Room model. These frames take 115 to 200 ms per frame to render, depending on the distribution of the fractured dragon triangles.

Fig. 10 already presented a comparison of the tracing times of our method and an SBVH-based method on a GTX480 for reflection rays. Grids offer no advantages to largely incoherent reflection rays whereas SBVH treats reflection rays in a nearly same fashion as other rays. Thus, reflection rays are much slower in our method, but if we take into account the time needed for SBVH construction, we gain significantly. However, if rendering a scene requires several reflection or refraction passes, a BVH-based method can catch up with ours even if the acceleration structure is built every frame. Referring to Fig. 10, we can see that SBVH gains 25 ms per reflection pass over our method for Happy Buddha model. If SBVH build time on GPU is 40 times faster than that of CPU, it will take about 80 passes for SBVH to catch up. Similarly, it will take 35 passes if the implementation is 100 times faster.

7 Conclusions

In this paper, we presented methods to raytrace dynamic scenes on the GPU in near real time. This requires building the acceleration structure each frame. We showed the better performance of our method over structures like BVH that

Fig. 13. The models and viewpoints used for evaluation of the performance of reflection rays. The models are Conference Room (284 k), Happy Buddha (1.09 M), and Fairy Forest (174 k). The Buddha model has the reflection parts colored white.

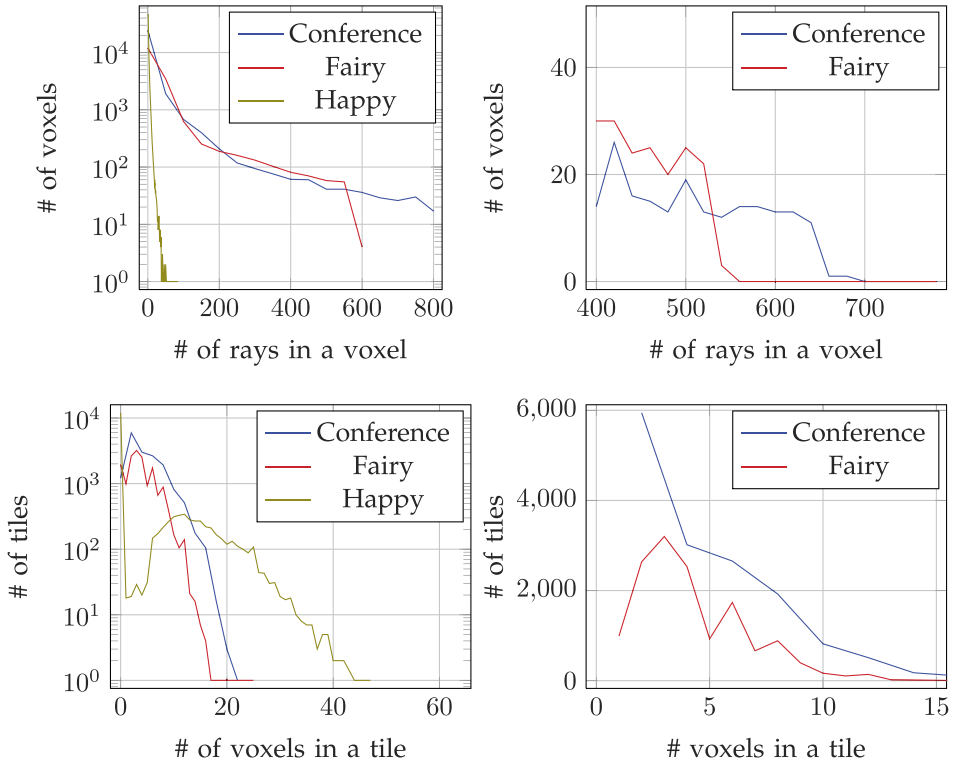

Fig. 14. Study of triangle and voxel distributions affecting reflection performance. Top left plot shows the concentration of rays in each voxel. Top right examines the tail of the plot. Longer tail with larger number of voxels is better for performance. Bottom left shows voxel divergence in each tile. Bottom right examines the front. Higher number of tiles with less divergence is good for performance. The plots in the top are both binned, i.e., rays which lie in the interval 500 to 550 are all counted in 500.

are constructed only once for the scene, but are expensive to construct. With fast construction of grids, we can rebuild the data structure depending on the characteristics of the tracing pass. Perspective grids with respect to the camera provide perfect coherence and high performance for primary rays. For shadow rays, building a perspective grid with respect to each light source and following a similar tracing strategy provides good performance.

Uniform grids are best suited for reflection and refraction rays with little coherence. We tried two methods suited for the two recent GPU architectures. In one, we enforce coherence by sorting rays that pass through each grid voxel. These rays can then check for intersection with geometry in the voxel using the shared memory effectively. The other processes each ray independently using a thread, avoiding the sorting to enforce coherence. Enforcing coherence wins on older cache-less GPUs. The performance can improve as sorting gets faster [28]. The moderate amounts of L1 and L2 caches available on the latest GPUs tip the balance in favor of the second method.

The performance of our grid-based approach deteriorates as the number of bounces increases. The savings made in per-frame structure building can be offset by slow reflection ray performance if multiple passes are involved, compared to good BVH structures. Also, if a parallel SBVH construction is available, handling multiple bounces will be cheaper using BVH. Grids will continue to be faster for raytracing dynamic scenes with limited bounces.

Fig. 15. In Fairy and Sibenik, only the floor is reflective. In the case of Bunny floating in Conference Room, the wooden table and the wooden frame of the red chairs is a highly polished reflective surface.

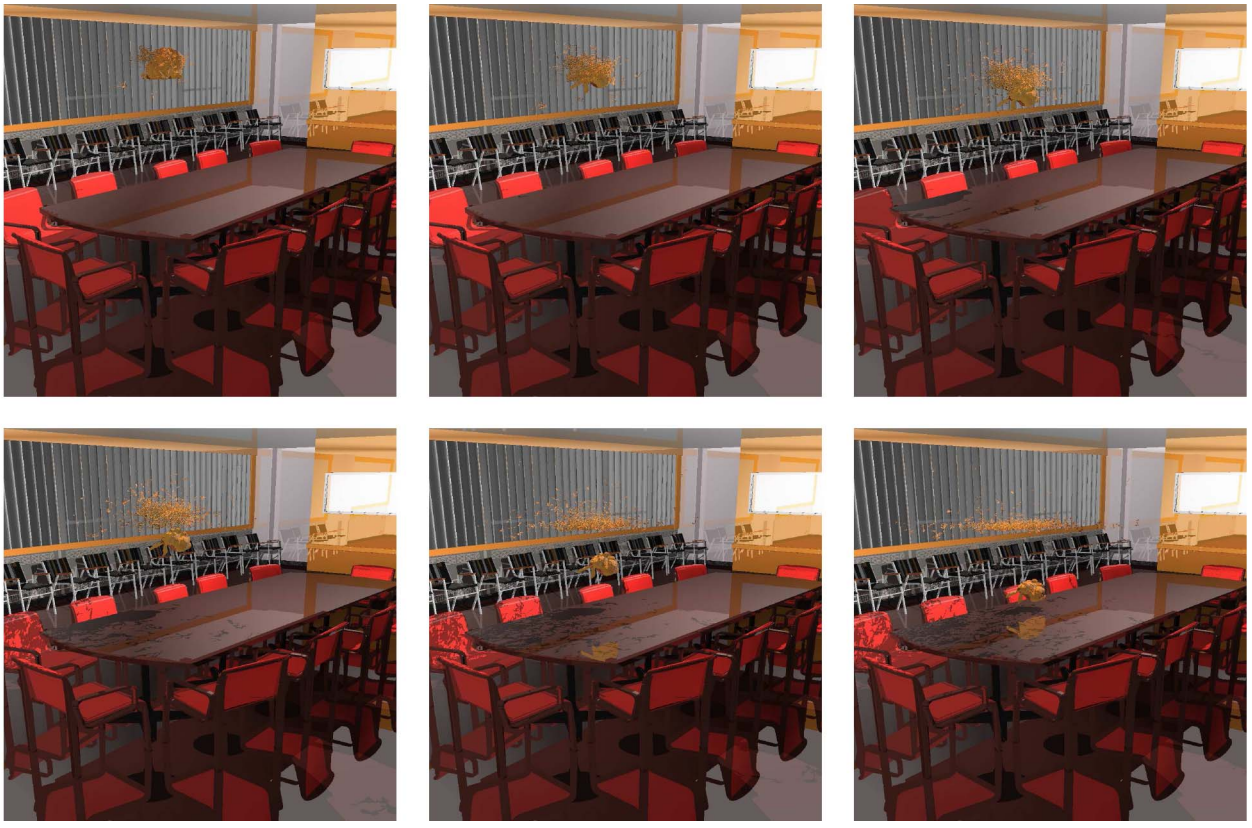

Fig. 16. Dragon, Bunny collision in a conference room.

We present the following conclusions based on our work on raytracing dynamic models. These are derived from our experience with Nvidia GPUs, but are likely to hold for most manycore architectures currently being planned.

- One acceleration structure may not work best for all passes. Building appropriate structure for each pass can save the overall time since data structure building is usually fast on these architectures. This is a significant departure from CPU raytracing, where data structure building is expensive. Perspective grids work best for primary and shadow rays, but uniform grids may be best for others.
- Grids are the easiest structures to build as fast sorting is available on most platforms. The acceleration structure needs to be rebuilt in each frame when dealing with dynamic or deformable scenes. An oncoming paper carries this idea further and builds a two-level grid structure, riding on fast sorting times [29]. Adaptive hierarchies like kd-trees and BVH trees are expensive to build which more than offsets any gains on traversal they may have.
- Smaller grid cells are preferred since fewer triangles will be in each cell. However, processing very small packets of rays may be inefficient if the SIMD width is high. The indirect mapping approach strikes a balance between these by processing multiple small cells together.
- Coherence is critical to high raytracing performance, especially when SIMD width is large. Processing coherent rays together can achieve high performance. Coherence can be enforced using perspective grids when possible or using sorting. Coherence may be discovered automatically through the use of caches, if they are present.
- As GPUs and manycore processors have richer memory hierarchies, enforcing coherence may not be advantageous as the inherent coherence will be exploited by the cache hierarchy. Simple algorithms like independent voxel walking win over attempts to enforce coherence as we reported in this paper. The margin of improvement between approaches that are deeply aware of the architecture and approaches that aren't should come down. This will be a critical factor in the wide adaptation of raytracing techniques in games, engineering applications, etc.
- A hierarchical structure like the BVH will eventually outperform simple grids if several bounces are needed, even when the BVH is constructed every frame. The crossover point depends on the architectural features and, to some extent, the scene type. Our current calculation indicates that our method will be ahead until the number of reflection passes per frame is 50 or more.

ACKNOWLEDGMENTS

The authors thank Marko Dabrovic for the Sibenik Cathedral Model, University of Utah for the Fairy Forest scene, Stanford 3D Scanning Repository for the Buddha model, UNC Dynamic Benchmarks for the Cloth model, and Anat Grynberg and Greg Ward for the Conference Room Model. They thank Timo Aila and Tero Karras for sharing their BVH

traversal code with them and also Nvidia for generous equipment donations and partial financial support through faculty award.

REFERENCES

[1] I. Wald, W.R. Mark, J. GntherBoulos, S. Boulos, T. Ize, W. Hunt, S.G. Parker, and P. Shirley, "State of the Art in Ray Tracing Animated Scenes," *Computer Graphics Forum,* vol. 28, no. 6, pp. 1691-1722, 2009.

[2] A. Reshetov, A. Soupikov, and J. Hurley, "Multi-Level Ray Tracing Algorithm," *ACM Trans. Graphics,* vol. 24, no. 3, pp. 1176-1185, 2005.

[3] S. Patidar and P.J. Narayanan, "Ray Casting Deformable Models on the GPU," *Proc. Indian Conf. Computer Vision, Graphics and Image Processing,* pp. 481-488, 2008.

[4] S. Guntury and P.J. Narayanan, "Ray Tracing Dynamic Scenes with Shadows on GPU," *Proc. Eurographics Symp. Parallel Graphics and Visualization,* pp. 27-34, 2010.

[5] K. Zhou, Q. Hou, R. Wang, and B. Guo, "Real-Time KD-Tree Construction on Graphics Hardware," *ACM Trans. Graphics,* vol. 27, no. 5, 2008.

[6] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, "Fast BVH Construction on GPUs," *Computer Graphics Forum,* vol. 28, no. 2, pp. 375-384, 2009.

[7] H. Dammertz and A. Keller, "The Edge Volume Heuristic - Robust Triangle Subdivision for Improved BVH Performance," *Proc. IEEE Symp. Interactive Ray Tracing,* pp. 155 -158, 2008.

[8] M. Ernst and G. Greiner, "Early Split Clipping for Bounding Volume Hierarchies," *Proc. IEEE Symp. Interactive Ray Tracing,* pp. 73-78, 2007.

[9] M. Stich, H. Friedrich, and A. Dietrich, "Spatial Splits in Bounding Volume Hierarchies," *Proc. Conf. High Performance Graphics,* pp. 7-13, 2009.

[10] S. Popov, I. Georgiev, R. Dimov, and P. Slusallek, "Object Partitioning Considered Harmful: Space Subdivision for BVHs," *Proc. Conf. High Performance Graphics,* pp. 15-22, 2009.

[11] J. Kalojanov and P. Slusallek, "A Parallel Algorithm for Construction of Uniform Grids," *Proc. Conf. High Performance Graphics,* pp. 23-28, 2009.

[12] T. Ize, P. Shirley, and S. Parker, "Grid Creation Strategies for Efficient Ray Tracing," *Proc. IEEE Symp. Interactive Ray Tracing,* pp. 27-32, 2007.

[13] I. Wald, T. Ize, A. Kensler, A. Knoll, and S.G. Parker, "Ray Tracing Animated Scenes Using Coherent Grid Traversal," *ACM Trans. Graphics,* vol. 25, no. 3, pp. 485-493, 2006.

[14] W. Hunt and W. Mark, "Ray Specialized Acceleration Structures for Ray Tracing," *Proc. IEEE Symp. Interactive Ray Tracing,* pp. 3-10, 2008.

[15] M. Pharr, C. Kolb, R. Gershbein, and P. Hanrahan, "Rendering Complex Scenes with Memory-Coherent Ray Tracing," *Proc. ACM SIGGRAPH,* pp. 101-108, 1997.

[16] P.A. Navratil, D.S. Fussell, C. Lin, and W.R. Mark, "Dynamic Ray Scheduling to Improve Ray Coherence and Bandwidth Utilization," *Proc. IEEE Symp. Interactive Ray Tracing,* pp. 95-104, 2007.

[17] B. Moon, Y. Byun, T.-J. Kim, P. Claudio, H.-S. Kim, Y.-J. Ban, S.W. Nam, and S.-E. Yoon, "Cache-Oblivious Ray Reordering," *ACM Trans. Graphics,* vol. 29, no. 3, pp. 1-10, 2010.

[18] B.C. Budge, T. Bernardin, J.A. Stuart, S. Sengupta, K.I. Joy, and J.D. Owens, "Out-of-Core Data Management for Path Tracing on Hybrid Resources," *Computer Graphics Forum,* vol. 28, no. 2, pp. 385-396, 2009.

[19] K. Garanzha and C. Loop, "Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing," *Computer Graphics Forum,* vol. 29, no. 2, pp. 289-298, 2010.

[20] T. Aila and T. Karras, "Architecture Considerations for Tracing Incoherent Rays," *Proc. Conf. High Performance Graphics,* pp. 113-122, 2010.

[21] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable Parallel Programming with CUDA," *ACM Queue,* vol. 6, pp. 40-53, 2008.

[22] J. Schmittler, S. Woop, D. Wagner, W.J. Paul, and P. Slusallek, "Realtime Ray Tracing of Dynamic Scenes on an FPGA Chip," *Proc. ACM SIGGRAPH/EUROGRAPHICS Conf. Graphics Hardware,* pp. 95-106, 2004.

[23] S. Sengupta, M. Harris, Y. Zhang, and J.D. Owens, "Scan Primitives for GPU Computing," *Proc. ACM SIGGRAPH/EUROGRAPHICS Symp. Graphics Hardware,* pp. 97-106, 2007.

[24] J. Amanatides and A. Woo, "A Fast Traversal Algorithm for Ray Tracing," *Proc. Eurographics,* pp. 3-10, 1987.

[25] T. Aila and S. Laine, "Understanding the Efficiency of Ray Traversal on GPUs," *Proc. Conf. High Performance Graphics,* pp. 145-149, 2009.

[26] J. Pantaleoni and D. Luebke, "HLBVH: Hierarchical LBVH Construction for Real-Time Ray Tracing," *Proc. Conf. High Performance Graphics,* pp. 87-95, 2010.

[27] N. Satish, M. Harris, and M. Garland, "Designing Efficient Sorting Algorithms for Manycore GPUs," *Proc. IEEE Int'l Symp. Parallel and Distributed Processing (IPDPS),* pp. 1-10, 2009.

[28] D.G. Merrill and A.S. Grimshaw, "Revisiting Sorting for GPGPU Stream Architectures," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques (PACT),* pp. 545-546, 2010.

[29] J. Kalojanov, B. Markus, and P. Slusallek, "Two-Level Grids for Ray Tracing on GPUs," *Proc. Eurographics,* 2011.

Sashidhar Guntury received the bachelor's degree in computer science and engineering from IIIT Hyderabad in 2009. He is working toward the master's degree at IIIT Hyderabad. In IIIT Hyderabad, he is a part of the Center for Visual Information Technology (CVIT), where he works on Raytracing on GPU. He is interested in GPU programming and using it to speedup algorithms in Raytracing.

P.J. Narayanan received the bachelor's degree from IIT Kharagpur and the PhD degree from the University of Maryland. He is a professor and the dean of research at the IIIT Hyderabad. He was a research faculty member at the Robotics Institute of Carnegie Mellon University from 1992 to 1996 and a scientist at the Centre for Artificial Intelligence and Robotics, Bengaluru, till 2000. His research interests include computer vision, computer graphics, and GPU computing. He was made a CUDA Fellow in 2008. He was the general chair of ICVGIP '00 and the program cochair of ACCV '06 and ICVGIP '10. He currently cochairs the ACM India Council.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.