Eurographics Symposium on Parallel Graphics and Visualization (2010)
J. Ahrens, K. Debattista, and R. Pajarola (Editors)

Ray Tracing Dynamic Scenes with Shadows on the GPU

Sashidhar Gunturу and P J Narayanan

Centre for Visual Information Processing
International Institute of Information Technology (IIIT)
Hyderabad India
{sashidhar@research.,pjn@}iiit.ac.in

Abstract

We present fast ray tracing of dynamic scenes in this paper with primary and shadow rays. We present a GPU-friendly strategy to bring coherency to shadow rays, based on previous work on grids as acceleration structures. We introduce indirect mapping of threads to rays to improve the performance of ray tracing on the GPU for the traversal and intersection steps. We also construct a light frustum in a spherical space for shadow rays. A grid structure is constructed each frame for the light frustum and traversed coherently. This involves careful mapping of the primary ray information to the light space and balancing the work load of the threads. Using the finegrained parallelism of GPU, we reorder the shadow rays to make them coherent and process multiple thread blocks to each cell to balance the work load. Spherical mapping is key to handling light sources placed anywhere in the scene by reducing the triangle count and improving performance in shadow checking. In addition it also allows us to introduce spotlights in raytracing. In practice, we attain interactive performance for moderately large models which change dynamically in the scene.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture I.3.6 [Computer Graphics]: Methodology and Techniques—Graphics data structures and data types I.3.1 [Computer Graphics]: Hardware Architecture—Graphics processors

1. Introduction

Raytracing can produce images with very high degree of visual realism, but at the expense much higher computation. Ray tracing consists of two tasks: building of the acceleration data structure and tracing of the rays. Tracing involves traversing the acceleration structure and primitive intersection. Fast raytracing on the GPU is receiving a lot of interest in recent times, given their high compute power. Ray tracing dynamic and deformable scenes is now possible at near-interactive rates on modern GPUs. Acceleration data structures play a central role in ray tracing. A uniform grid is perhaps the simplest data structure used for ray tracing. Bounding Volume Hierarchies (BVH) [WBS07, LeYM06] and KD-trees [PGSS07, SSK07] have also been used widely for raytracing. BVH and kd-trees can exploit ray coherence more efficiently than grids. Their hierarchy also helps eliminate large portions of primitives before actual intersection.

Grids are easiest to build, giving them an edge over other structures for scenes where acceleration structure has to be built frequently. The cost of building a kd-tree or a BVH is significantly higher. Ray specialized grids like a perspective grid take advantage of the common point of origin of the rays and have a high degree of coherence. This makes ray traversal efficient and fast. Primary rays are examples of highly coherent rays. They originate from the camera point and their traversal is always bounded by a frustum. Patidar and Narayanan demonstrated the utility of such grids for primary rays [PN08] for deformable models by sorting triangles to perspective cells in each frame. The grid construction took only 15-30% of the total time for them. The ray tracing step dominates the total running time as a result.

In this paper, we examine the performance of grids for raytracing dynamic scenes for both primary as well as shadow rays. We show the use of perspective grids for the primary as well as shadow rays. We reduce the problem of shadow rays to another round of spherical grid construction and tracing from the point of view of the light source. The shadow rays are traced exactly to provide correct shadows. We build the perspective grid for each light in a spherical space. This enables us to place the lights anywhere inside or outside the scene and also provide spotlight effects. We are able to construct and render fairy forest model with a moving light in about 80 ms and the Conference hall model with one light in about 62 ms.

2. Background and Previous Work

A survey of the current techniques for ray tracing can be found in [WMG*07]. Acceleration data structures are built to speed up the process of finding ray triangle intersection. Earlier, the entire process of raytracing was offline and the time to build the datastructure was relatively small. Multi-core CPUs and GPUs, make raytracing is possible at interactive rates [RSH05], with the data structure building being the bottleneck, especially on dynamic scenes. Considerable work has gone into speeding up this process, especially on parallelizing the build on GPUs. Zhou et al. [ZHWG08] and Lauterbach et al. [LGS*09] gave fast methods of constructing kd-trees and BVH respectively on GPU. Patidar and Narayanan built a perspective grid in parallel on the GPU [PN08]. The main drawback of their approach was that of triangles distributions and spanning arbitrary number of voxels. It was solved by Kalojanov and Slusallek on uniform grids [KS09]. They also used appropriate grid resolution to improve the quality of the grid.

Unlike on a BVH and a kd-tree, rays on grids can not be handled as packets easily. Wald et al. [WIK*06] presented an algorithm for traversing the grid in a slice-wise coherent manner. Due to the use of a frustum like grid, Hunt and Mark [HM08] and Patidar and Narayanan [PN08] treat the camera rays in a totally coherent manner, making traversal efficient. Hunt and Mark suggested the idea of rebuilding the data structure from the light point of view on a multicore CPU [HM08]. Our technique of building a DS from light point of view is similar to theirs but goes much further by building a spherical grid to increase efficiency as well as to support spotlights and lights within the scene.

2.1. GPU Computing Model

We implement our techniques using the CUDA [NBGS08] programming model on Nvidia GPUs. CUDA uses kernels, which are programs that run in parallel on all threads. A huge number of threads – upwards of 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.

Threads are batched into warps (of 32 threads), which run in a strictly SIMD mode. Warps are scheduled sequentially on available processor resources. Thus, the SIMD width of the 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 proximate threads access global memory locations that are close. Performance is best if data used by all threads of a block is loaded onto the shared memory.

3. Perspective Grids for Ray Tracing

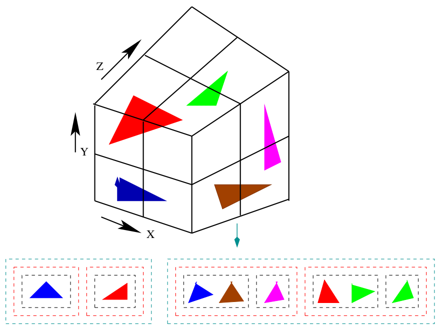


Figure 1: *The triangle storage layout. This kind of layout is achieved by keeping the X value in the MSB and Z extent in the LSB.*

We construct a perspective grid by dividing the view frustum into voxels. This scheme is very similar to [PN08, HM08]. Similar to [KS09] we determine the cells each triangle spans and build a list of triangles for each cell. This procedure is less sensitive to the triangle distribution in the scene. We create a list of (triangle, cell number) pairs, with one entry for each cell that each triangle maps to. We also can remove back-facing triangles to reduce the number of entries in the list. This leads to a faster build of the grid and fewer triangles to be checked during ray traversal. Since the number of cells is in the range of 8K to 1M, we use the scalable SplitSort [PN09] to sort this list with the cell-id as the key. We use scan primitives to build the list of triangles in each cell.

The perspective grid provides perfect coherence to primary rays. We process the rays of each tile together on the GPU using a CUDA block or a work group, with each pixel assigned to a thread or a work item. The triangle data is brought into the shared memory before intersection calculations. Since all threads need to process all triangles in the cell, 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 cell need not check for intersection in a later cell, as the cells are processed in a front-to-back order. We use the optimized routine for checking triangle intersection [MT05].

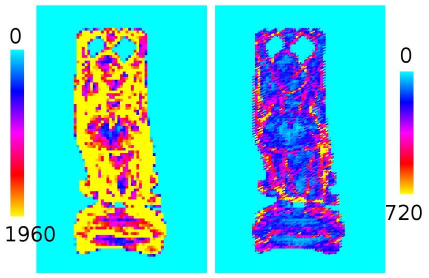


Figure 2: *Heat map of the number of triangles checked before declaring intersection. The left image corresponds to direct mapping while there is marked reduction in indirect mapping, shown in right. Number of triangles checked before declaring intersections increases from blue to pink and is highest in yellow regions.*

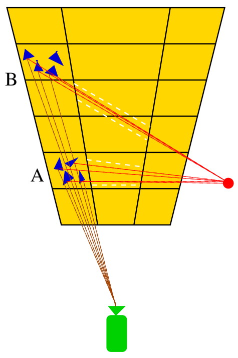


Figure 3: *Ray coherence is lost for shadow rays near silhouettes.*

The size of the image-space tiles and depth-space voxels can impact the performance. Tile is a coherent, rectangular cross section of rays. Larger tiles may exploit greater coherence than smaller ones. However, smaller tiles and cells 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 ray trace using larger number of threads, by mapping threads differently. In practice, we sort the trianges to $kN \times kN$ tiles in image space. For ray tracing, we divide the image into $N \times N$ tiles such that a $k \times k$ group of sorting tiles fit into each ray tracing tile. The work groups used while tracing have more threads. The available shared memory is partitioned equally among the sorting tiles during ray tracing. Triangles from each sorting tile is 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 ray tracing 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 30-50% speedup using indirect mapping as the maximum number of triangles checked dropped by more than half. Figure 2 shows this using a heatmap for the work done.

4. Spherical Light Grid for Shadows

Primary rays generate an intersection point for each pixel. Secondary rays are generated from the intersection points in general ray tracing. Secondary rays could be shadow rays that go to each light source, reflection rays that reflect off the surface or refraction rays that enter the object. Secondary rays are inherently less coherent as they neither have a common starting point nor a direction. Methods like the DDA algorithm in a 3D voxel space [AW87] can be used to enumerate the cells traversed by each ray. A packet based traversal where a set of shadow ray packets are traced using frustum shaft culling and mailboxing was proposed for the CPU with SSE [WIK*06]. Coherence that is significant to GPU with its wide SIMD width cannot be achieved using this kind of traversal alone. Shadow rays are not always coherent [WIK*06] and need to be split into multiple packets around object silhouettes as shown in Figure 3. Splitting them into a number of packets would introduce divergence and impede full use of the resources. Nevertheless shadow rays can be made coherent as they converge at the light source. This is a role reversal from the point of view of primary rays, but similar techniques can be used for coherent tracing of shadow rays.

One way to exploit the aforementioned coherence is by listing the cells for each shadow ray and then merging them. This has been found to be expensive on CPU [WIK*06] ond will be expensive on GPU. However, building the grid data structure is cheap and 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. In the next few subsections, we present a method of raytracing shadow rays accurately and effectively. Our method works for different light types as positions of the light source.

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 set of pixels. This is not natural for light sources. Point light sources emit light in all directions. Spot lights have lighting volumes of impact. We use a spherical mapping to map light's world into a perspective grid. A light frustum is constructed in the α-β space where α and β are the azimuthal

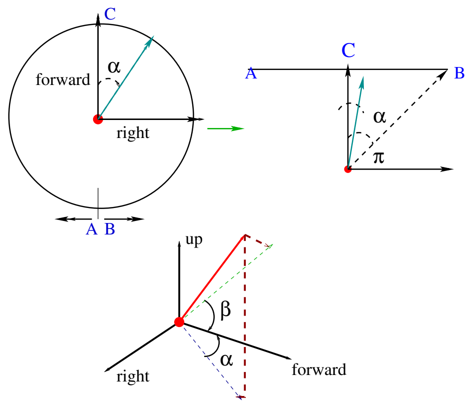


Figure 4: *Spherical space used for shadows.*

and elevation angles (see Figure 4). A rectangle in the α-β space defines the light frustum and plays the role of the image for primary rays. We define "tiles" on this rectangle to build cells of the grid using constant depth planes. Figure 4 shows the spherical space with respect to the forward, right and up directions. The angle α is measured from the forward direction in the forward-right plane and and the angle β 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.

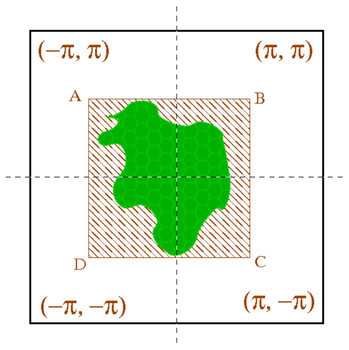


Figure 5: *Bounding rectangle of the geometry in spherical space defines the light frustum of interest.*

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. Figure 5 demonstrates this. This reduces the number 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 spot lights with light fall off. The spot can be marked as a bounding rectangle in the spherical space shown in Figure 5. A cubemap style of ray mapping to limit light space rays was used earlier [HM08]. They handle each frustum separately, resulting in a lot of extra work for the traversals. Furthermore, clamping a cubemap is very tedious. Spherical mapping is more convenient and provide exact shadows as we will see later.

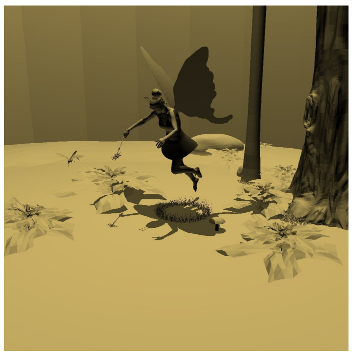


Figure 6: *By clamping the region to a userdefined value, we can get a spotlight like effect too. Only the shadow of fairy is seen while the bushes, trees and dragonfly don't have shadows.*

4.2. Ray Sorting and Reordering

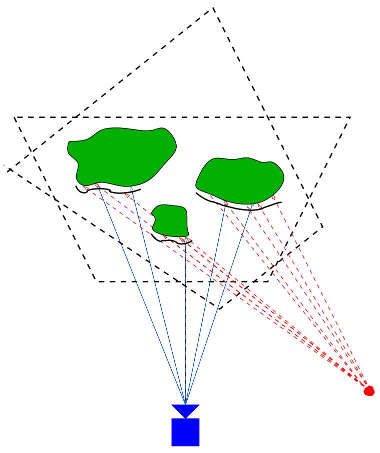


Figure 7: *Points of primary ray intersection are mapped to the light frustum.*

The shadow rays emanate from the intersection points of primary rays and travel towards the light source. Rays in the primary space that are distant may follow similar paths to the light source, as shown in Figure 7. 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 for each. Thus a pair of (primary ray, 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 the GPU in parallel using scan primitives [SHZO07] from the CUDPP library and the SplitSort primitive, which can handle arbitrary length keys [PN09].

4.3. Load Balancing

For tracing primary rays, blocks of threads are assigned to tiles directly or indirectly. This is efficient for them as the number of rays in each tile is a 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 work loads. 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.

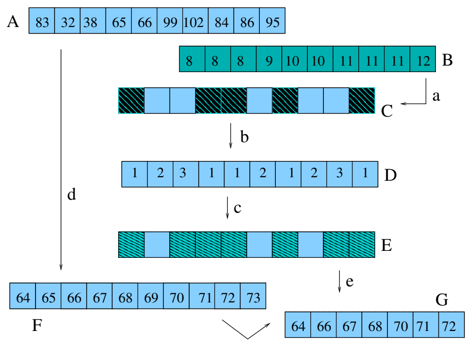


Figure 8: *Mapping frustum blocks to CUDA blocks.*

The ray ordering and load balancing are illustrated in Figure 8. The array of (primary ray, light tile number) pairs for each primary intersections is sorted with tile number as the key. The pixel number array is shown as **A** and the tiles array as **B**. An array that holds thread numbers (or index numbers) is shown as **F**. A simple kernel (step **a**) marks the boundaries of each light tile of **B** into an array **C**. The hatched cells signify the starting of a new tile. We call this a hard boundary. A segmented scan of an array of all 1's with **C** defining the segments, gives us the number of shadow rays for each tile in **D** (step b). A tile with more than a threshold r rays are mapped to multiple blocks. This is marked in array **E** where the grained cell signifies the starting of a new block (step e). We call this a soft boundary. Every hard boundary (different tile) is also a soft boundary (mapped to a different block.) To keep track of the first ray in each block, we do a stream compaction step and shrink the number of cells to C_{total}. In the Fig. 8, $r = 2$ is used. An array G, whose size is equal to C_{total} contains the location of the cells in the list of triangle-cell pairs. The first ray of each block b_i is the ray whose index in **A** is referenced by the value in location b_i in G. The total number of rays in a block b_i is the difference between the values in locations b_i and b_{i+1} in G. This completes the load-balanced mapping of shadow rays to thread blocks.

Algorithm 1 Rays to CUDA Block Mapping

```
 1: $TOTALPIXELS \leftarrow imagesize$
 2: $pseudoArr \leftarrow$ array of TOTALPIXELS zeroes
 3: $scratchArr \leftarrow$ array of TOTALPIXELS ones
 4: $validArr \leftarrow$ array of TOTALPIXELS zeroes
 5: $threadArr \leftarrow$ array of TOTALPIXELS zeroes
 6: $outArr \leftarrow$ array of TOTALPIXELS zeroes
 7: $gr \leftarrow dim3(BLK_X, BLK_Y, 1)$
 8: $bk \leftarrow dim3(THD_X, THD_Y, 1)$
 9: $tagThread <<< gr, bk >>> (pixelIDArr, threadArr)$
10: $sort(tileIDArr, pixelIDArr)$
11: $flagScan <<< gr, bk >>> (tileIDArr, pseudoArr)$
12: $segScan(tileIDArr, pseudoArr, scratchArr)$
13: $getChunk <<< gr, bk >>> (scratchArr, validArr)$
14: $numCBlocks \leftarrow strCompact(threadArr, validArr, outArr)$
```

4.4. Shadow Ray Tracing

The shadow tracing is similar to primary ray tracing. Each thread block knows the shadow rays it traces from the mapping described above. Triangles are loaded into shared memory for each cell in order before checking for intersection. Each thread knows the identity of the primary ray whose shadow ray it is tracing as well as the shadow ray origin. Shadow ray tracing only checks if an intersection exists or not. If one is found, the shadow bit corresponding the primary ray is set. Shadow bits are initially reset. By keeping the shadow information in the primary ray space, we avoid the need for costly synchronization steps that may be needed when tiles and cells are processed by multiple thread blocks simultaneously. This cannot work for primary rays, as the point of the closest intersection is needed, not a yes/no answer.

5. Results and Discussion

We evaluated the performance of our techniques on an NVIDIA 280 GTX card on a 32-bit linux machine. Our test-cases included static as well dynamic models. Our default resolution was 1024x1024. In all cases, we don't include the

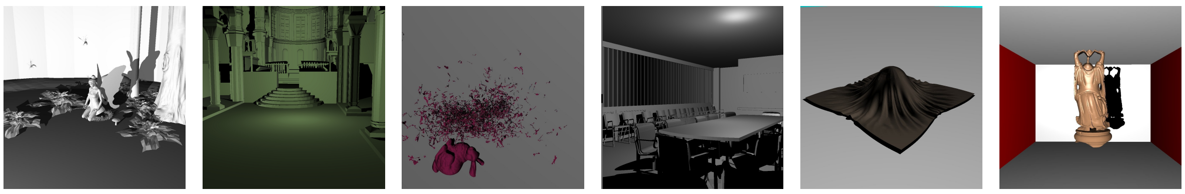
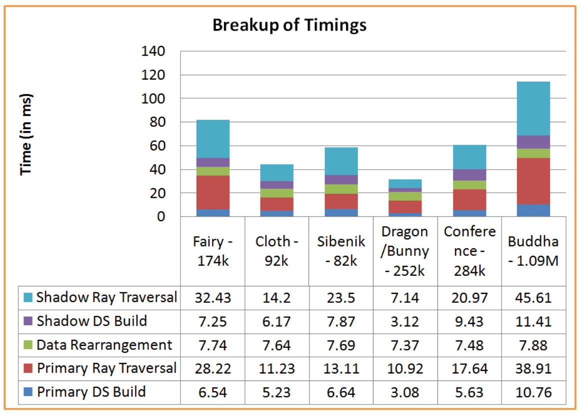


Figure 9: *Some of the test scenes : Fairy Forest(174k), Sibenik Cathedral(82k), Bunny/Dragon Model(252k), Conference Room(284k), Cloth Folding(92k), Buddha(1.09M)*

Model	Tris	Primary Rays				With Shadows
		HBVH	kd-tree	Grid	Our Method	Our Method
Fairy	174k	124 / 33.11	65 / $125^{\#}$	24 / 285.7	8.73 / 32.43	82.18 ms
Sibenik	82k	30 / 45.32	n/a	n/a	8.86 / 13.11	58.81 ms
Bunny/Dragon	252k	66 / 128.94	93 / 25	13 / 129.87	3.71 / 10.92	31.63 ms
Conference	284k	105 / 37.11	n/a	27 / 142.85	7.51 / 17.64	61.15 ms
Cloth	92k	39 / n/a*	n/a	n/a	5.23 / 11.23	44.47 ms
Buddha	1.09M	n/a	n/a	n/a	13.76 / 38.91	114.57 ms

Table 1: *Results are as noted on our raytracer at* 1024×1024 *resolution. First three columns give the primary rays performance. Next column gives the performance of our system on shadow rays. In each of the entry, the first value corresponds to the DS build time and the second value corresponds to the time taken to traverse and trace the rays. However in the shadow rays column, the entry corresponds to time taken for the all steps till shadow computation. Our times don't include time to shade or the time taken for the costly transfers of data from host to GPU. All times are in milliseconds, ms. HBVH is timings of [LGS*09], kd-tree numbers are from [HSZ*09], Grid values are from [KS09]. All on NVIDIA GTX 280 hardware. Also, Fairy in kd-tree was rendered with two lights and shadows. Cloth model in BVH was rendered at 14 fps with shadows.*

time required to transfer the model data from host to device. We build the datastructure for every frame. Table 1 shows the performance of our system with the models shown in Fig. 9.



	Fairy - 174k	Cloth - 92k	Sibenik - 82k	Dragon /Bunny - 252k	Confere nce - 284k	Buddha - 1.09M
🟦 Shadow Ray Traversal	32.43	14.2	23.5	7.14	20.97	45.61
🟪 Shadow DS Build	7.25	6.17	7.87	3.12	9.43	11.41
🟩 Data Rearrangement	7.74	7.64	7.69	7.37	7.48	7.88
🟥 Primary Ray Traversal	28.22	11.23	13.11	10.92	17.64	38.91
🟦 Primary DS Build	6.54	5.23	6.64	3.08	5.63	10.76

Figure 10: *Time spent in each of the broadly classified stages. Data Rearrangement includes the time taken to map the ray, sort them, scan them, segment scan them into chunks and finally stream compact too. Refer to the algorithm*

Figure 10 shows the detailed breakup of various stages for a frame. Unlike primary rays, shadow rays are not equally distributed among all tiles. Therefore shadow ray traversal is almost always more time consuming than primary ray traversal. Ray reordering consists of mapping the shadow rays to the spherical map, clamping the spherical map to get a tight frustum. Sorting the shadow rays, binning them into the right tiles and finally stream compacting these tiles to eliminate empty tiles is also included in this step. The time needed to do all this is more or less the same for all models as it doesn't depend on the model. Instead it depends on the resolution at which we are rendering the images. In our case, we need to sort 1024x1024 shadow rays, bin them and perform stream compaction.

The performance of our technique shows vast improvement of grid based raytracer for architectural and room models like Conference and Sibenik Cathedral. However, large sized triangles that span across many voxels is still a problem of grid. Presence of large number of such triangles hampers the performance. This problem is less for BVH or kd-tree as they split the triangle [SFD09].

Also shown is the performance of other GPU based implementations. As of writing this paper, there is little work dealing with shadow rays on GPU. The numbers have been given to show the difference and as such it would be unfair to compare our shadow numbers with that of [LGS*09] and [ZHWG08] as they mainly concentrate on building the datastructure. Our grid construction is similar to that of [KS09] but due to the use of a much faster SplitSort [PN09] routine, we build the grid very fast. We take advantage of the fact that the number of cells is less than 32 bits (in our case it is 18) for splitting.

The performance of our approach is much better in all testcases except that of Fairy where we are only slightly faster. This is due to the inherent problem of grids in handling teapot in stadium kind of scenarios. [HSZ*09] report their Fairy Forest timings with two lights as 5.26 fps while we are able to clock on an average 7.6 fps with two lights. By building the datastructure from scratch and data rearrangement, we are able to ensure a coherent traversal. Fig. 11 shows the change in time taken for traversal as the light moves further away from the model. This dependence of shadow rays' performance on the position of light is compensated by load balancing. This fact is illustrated in the same figure where we show for different threshold values. Shadow rays which would have got piled up in a few frustum blocks are broken into chunks before mapping them to CUDA blocks. This factor along with coherent traversal is the biggest reason why we are faster than kd-tree inspite of it being more efficient for such models.

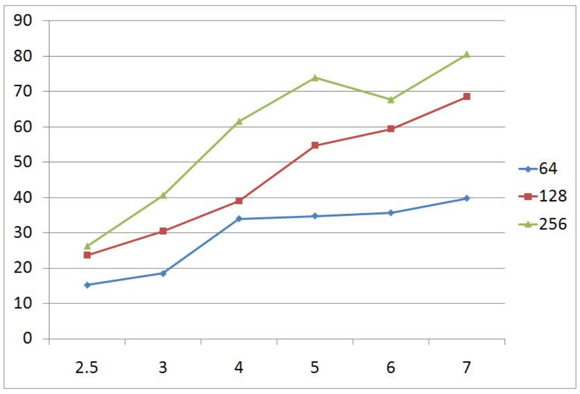


Figure 11: *Time in ms versus the distance of the light from the center of the model for different number of rays per CUDA block as the light moves further away from the model.*

6. Conclusion and Future Work

In this paper we demonstrated a strategy that gives significant improvements in performance of tracing rays on GPU. To the best of our knowledge there is no discussion of tracing shadows efficiently on GPU. Our implementation at every stage maintains coherence among rays to make full use of the frustum traversal technique. We get this coherence by first mapping the rays to the frustum and then sorting them to each tile. In order to cover all the points in the space we use spherical mapping system. Depending on the minimum and maximum extents of the shadow rays, we also clamp the frustum in order to devote more blocks to the geometry against which the rays will check. And finally in order to make the ray checking fast, we divide the ray packets into smaller, easily manageable chunks and distribute the load almost equally among all CUDA blocks.

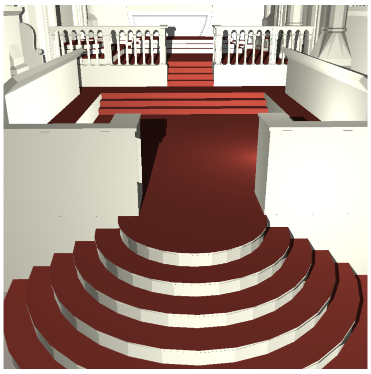
While many of the techniques we discussed are grid specific, ray sorting and load balancing can be applied to kd-tree and BVH based implementations. Any data intensive, work efficient GPU kernel takes advantage of the way data is organized and that is exactly what ray sorting and load balancing do. They lay the data in way such that the intersection checking kernels spend little time getting data and check for intersections faster.

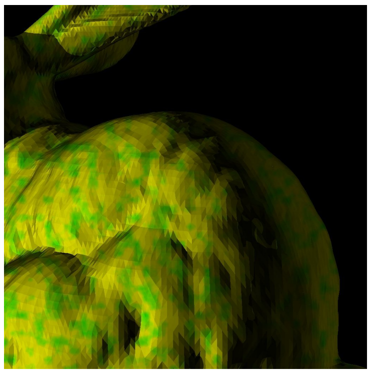
Although grids are not the best choice for raytracing on CPU, their structure intuitively seems more suited to GPU model. Also the use of grids to cheaply build datastructure helps us shift computational demand almost entirely to the actual raytracing activity. Since the method is a pure frustum traversal based method, the effectivenss of our strategies is only to a point till coherence exists but till that point, grid outperforms other datastructures by a wide margin. The time gained can thus be devoted for true secondary ray traversals. Thus we believe grids might still be as competitive as kd-tree and BVH. Our method is esspecially good for scenes with deformable models. Faster traversal of rays is good for interactive applications where there is a demand for higher framerates while at the same time raytracing standards are moderate, i.e have some basic lighting and shadows.

In the paper we demonstrated ways to load balance a scenario where there are too many rays in a frustum block. We are also looking at ways to load balance a scenario when too many triangles fall in a single frustum block. This can be achieved by dividing triangles in a frustum into chunks and parallely checking them. This might require some synchronization through atomic operation or some reduction. We are looking at this possibility.

We present the following conclusions based on our work on ray tracing dynamic models. These points are likely to hold for all high performance hardware platforms with some SIMD width, though they rose out of our experience with Nvidia GPUs.

- Coherence is the most important factor that determines the performance, especially when SIMD width is large. Processing coherent rays together can achieve high performance.
- Grids are the easiest structures to build as fast sorting is usually available on most platforms. The acceleration structure needs to be rebuilt in each frame when dealing with dynamic or deformable scenes. Adaptive hierarchies like kd-Trees and BVH trees are expensive to build. Rigid structures like grids and octrees are better.
- Perspective grids are most efficient for primary rays due to their maximum coherence. Coherence can be brought to shadow rays for point light sources by building a perspective frustum for each light.





- Grids are inefficient for secondary rays from reflection and refraction due to their incoherence. Tree-based data structures can do better on them, but only when the SIMD width is low. Divergence increases with SIMD width, making irregular access no better than grids.
- Sorting is usually fast, thus favouring data structures with smaller grid cells. Fewer triangles will be in each cell, reducing the ray-triangle intersections. 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.

7. Acknowledgements

We thank Marko Dabrovic for the Sibenik Cathedral Model, University of Utah for the Fairy Forest scene, Stanford 3D Scanning Repository for the Bunny, Dragon and Buddha models, UNC Dynamic Scene Benchmarks for Exploding Dragon and Cloth models and Anat Grynberg and Greg Ward for the Conference Room Model. We thank Nvidia for generous equipment donations and partial financial support through CUDA Fellowship.

References

[AL09] AILA T., LAINE S.: Understanding the efficiency of ray traversal on gpus. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009* (2009), pp. 145–149.

[AW87] AMANATIDES J., WOO A.: A fast voxel traversal algorithm for ray tracing. In *In Eurographics '87* (1987), pp. 3–10.

[BOA09] BILLETER M., OLSSON O., ASSARSSON U.: Efficient stream compaction on wide simd many-core architectures. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009* (2009), pp. 159–166.

[HM08] HUNT W., MARK W.: Ray-specialized acceleration structures for ray tracing. In *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on* (Aug. 2008), pp. 3–10.

[HSZ*09] HOU Q., SUN X., ZHOU K., LAUTERBACH C., MANOCHA D., GUO B.: *Memory-Scalable GPU Spatial Hierarchy Construction*. Tech. rep., MSR Asia, UNC, Chapel Hill, 2009.

[ISP07a] IZE T., SHIRLEY P., PARKER S.: Grid creation strategies for efficient ray tracing. In *RT '07: Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 27–32.

[ISP07b] IZE T., SHIRLEY P., PARKER S.: Grid creation strategies for efficient ray tracing. In *RT '07: Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing* (2007), IEEE Computer Society, pp. 27–32.

[IWRP06] IZE T., WALD I., ROBERTSON C., PARKER S.: An evaluation of parallel grid construction for ray tracing dynamic scenes. In *Interactive Ray Tracing 2006, IEEE Symposium on* (Sept. 2006), pp. 47–55.

[KS09] KALOJANOV J., SLUSALLEK P.: A parallel algorithm for construction of uniform grids. In *HPG '09: Proceedings of the 1st ACM conference on High Performance Graphics* (2009), pp. 23–28.

[LD08] LAGAE A., DUTRÉ P.: Compact, fast and robust grids for ray tracing. *Comput. Graph. Forum 27*, 4 (2008), 1235–1244.

[LeYM06] LAUTERBACH C., EUI YOON S., MANOCHA D.: Rt-deform: Interactive ray tracing of dynamic scenes using bvhs. In *In Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (2006), pp. 39–45.

[LGS*09] LAUTERBACH C., GARLAND M., SENGUPTA S., LUEBKE D., MANOCHA D.: Fast bvh construction on gpus. *Comput. Graph. Forum 28*, 2 (2009), 375–384.

[MT05] MÖLLER T., TRUMBORE B.: Fast, minimum storage ray/triangle intersection. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses* (2005), p. 7.

[NBGS08] NICKOLLS J., BUCK I., GARLAND M., SKADRON K.: Scalable parallel programming with cuda. *Queue 6*, 2 (2008), 40–53.

[PBMH02] PURCELL T. J., BUCK I., MARK W. R., HANRAHAN P.: Ray tracing on programmable graphics hardware. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques* (2002), ACM, pp. 703–712.

[PGSS07] POPOV S., GÜNTHER J., SEIDEL H.-P., SLUSALLEK P.: Stackless kd-tree traversal for high performance gpu ray tracing. *Computer Graphics Forum 26*, 3 (2007).

[PN08] PATIDAR S., NARAYANAN P. J.: Ray casting deformable models on the gpu. In *ICVGIP* (2008), pp. 481–488.

[PN09] PATIDAR S., NARAYANAN P.: *Scalable Split and Gather Primitives on the GPU, IIIT/TR/2009/99*. Tech. Rep. IIIT/TR/2009/99, 2009.

[RSH05] RESHETOV A., SOUPIKOV A., HURLEY J.: Multi-level ray tracing algorithm. *ACM Trans. Graph. 24*, 3 (2005), 1176–1185.

[SFD09] STICH M., FRIEDRICH H., DIETRICH A.: Spatial splits in bounding volume hierarchies. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009* (2009), pp. 7–13.

[SHG09] SATISH N., HARRIS M., GARLAND M.: Designing efficient sorting algorithms for manycore gpus. In *IPDPS* (2009), pp. 1–10.

[SHZO07] SENGUPTA S., HARRIS M., ZHANG Y., OWENS J. D.: Scan primitives for gpu computing. In *Graphics Hardware* (2007), pp. 97–106.

[SSK07] SHEVTSOV M., SOUPIKOV A., KAPUSTIN E.: Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes. *Computer Graphics Forum 26*, 3 (2007).

[WBS07] WALD I., BOULOS S., SHIRLEY P.: Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph. 26*, 1 (2007).

[WGBK07] WALD I., GRIBBLE P. C., BOULOS S., KENSLER A.: *SIMD Ray Stream Tracing - SIMD Ray Traversal with Generalized Ray Packets and On-the-fly Re-Ordering*. Tech. Rep. UUSCI-2007-012, 2007.

[WIK*06] WALD I., IZE T., KENSLER A., KNOLL A., PARKER S. G.: Ray tracing animated scenes using coherent grid traversal. *ACM Trans. Graph. 25*, 3 (2006), 485–493.

[WMG*07] WALD I., MARK W. R., GÃIJNTHER J., BOULOS S., IZE T., HUNT W., PARKER S. G., SHIRLEY P.: State of the Art in Ray Tracing Animated Scenes . pp. 89–116.

[ZHWG08] ZHOU K., HOU Q., WANG R., GUO B.: Real-time kd-tree construction on graphics hardware. *ACM Trans. Graph. 27*, 5 (2008).