Ray Casting Deformable Models on the GPU

Suryakant Patidar and P. J. Narayanan
Center for Visual Information Technology, IIIT Hyderabad.
{skp@research., pjn@}iiit.ac.in

Abstract

The GPUs pack high computation power and a restricted architecture into easily available hardware today. They are now used as computation co-processors and come with programming models that treat them as standard parallel architectures. We explore the problem of real time ray casting of large deformable models (over a million triangles) on large displays (a million pixels) on an off-the-shelf GPU in this paper. Ray casting is an inherently parallel and highly compute intensive operation. We build a GPU-efficient three-dimensional data structure for this purpose and a corresponding algorithm that uses it for fast ray casting. We also present fast methods to build the data structure on the SIMD GPUs, including a fast multi-split operation. We achieve real-time ray-casting of a million triangle model onto a million pixels on current Nvidia GPUs using the CUDA model. Results are presented on the data structure building and ray casting on a number of models. The ideas presented here are likely to extend to later models and architectures of the GPU as well as to other multi core architectures.

1. Introduction

Ray casting heavy models is a highly compute intensive process. Ray casting has been performed for static models on the CPU [11, 16, 19] and on the GPU [2, 10, 20, 18]. A lot of effort has been put into ray casting and ray tracing of static models at real-time rates. A recent work [20] involving ray tracing of deformable objects draws small and medium sized models. They render models of 180K triangles at 6fps. Shvetsov et al. [14] ray trace dynamic models using a parallel and linearly scalable technique of kd-tree construction. They achieve 7 to 12 FPS for models with up to 200K triangles at a window resolution of 1024×1024.

We aim to ray cast a million triangle deformable model onto a million pixel window in real time. CPUs are not capable of performing ray casting of heavy deformable models at real time rates. Even with the introduction of multi core CPUs, it is difficult to achieve the above target. Ray casting is inherently parallel and maps well to architectures like FPGA, Cell Processors, GPU etc. Co-processors like FPGAs and GPUs are promising due to their scalable parallel architecture. The Cell processor also falls in this category but are not accessible widely. GPUs provide high computation power at low costs and are widely available as an integral part of the computer, making them strong candidate as a co-processor.

Figure 1: Dragon ray casted (870K triangles)

Earlier, general purpose computing on GPU was performed via the graphics pipeline, which provided limited features and a steep learning curve. With the introduction of general purpose programming model on modern GPUs such as CUDA (Nvidia) and CTM (ATI), their use has become more widespread to applications involving complex data structures. The data parallel architecture they embody is well suited for ray-casting. Techniques used on such general programming models are also likely to be applicable to other parallel processors such as the Cell, FPGA etc.

The main contribution of this paper is the algorithm for real-time ray casting of a deformable model on the modern GPU. We set a performance goal of real time rendering

of a million triangle deformable model to a million pixels. We describe a three-dimensional, screen-space data structure and the algorithm that are efficient on the GPU for ray tracing. We build the data structure from scratch in each frame and can handle deforming objects as easily as rigid ones. We describe how such a data structure can be built efficiently on the SIMD architecture of the GPU. Rendering results presented on a number of standard models demonstrate that we achieve our performance goals. The methods presented here may have applications on ray casting and other problems on the GPU and other multi core architectures.

1.1. Related Work

Ray tracing is a simple way of rendering the world by finding the color at each pixel of the image. The disadvantage of ray tracing is its computational cost. Due to its point sampling approach, any kind of spatial or primitive based coherence can not be exploited. Ray tracing has been attempted over years, across architectures like, CPUs, multicores, clusters, CellBE, FPGAs etc. Beam Tracing [7] was introduced to exploit the spatial coherence of polygonal environments. Rather than working with high number of rays per image, beam tracing sweeps areas of the scene to form beams.

Before the introduction of GPUs, ray tracing was performed on the CPU or on a cluster of CPUs. A single CPU works sequentially on the rays and finds closest intersections. With the increase in CPU cores and multi threaded architectures, ray tracing could be efficiently performed on a set of processors. MLRTA [11] performs fast ray tracing by allowing a group of rays to start traversing the tree data structure from a node deep inside the tree, saving unnecessary operations. RLOD [19] uses an LOD based scheme which integrates simplified LODs of the model into k-d tree, performing efficient ray-triangle intersections. Wald et al. [15] ray trace deformable objects on the CPU using a bounding volume hierarchy (BVH). They exploit the fact that the topology of the BVH is not changed over time so that only the bounding volumes need be re-fit per frame. In another work, Wald et al. [17, 16] ray trace animated scenes by rebuilding the grid data structure per frame. They use a new traversal scheme for grid-based acceleration structure that allows for traversing and intersecting packets of coherent ray using an MLRTA-inspired frustum-traversal scheme [11]. Ray tracing has also been performed on non-triangulated models like implicit surfaces [9] and geometry images [4, 3].

Programmable GPUs have been used for ray tracing even with their constrained programming model [2, 10]. Ray tracing was performed as a multi-pass operation due to insufficient capability of the fragment shaders. With growth in programmability of the GPU, more efficient methods have emerged which use the looping and conditional operations. Most of the work for ray tracing on GPU uses pre-built data structures, given that the cost of building parallel data structures may be high [8]. Recent work by Zhou et al. [20] builds and ray traces small and medium sized deformable models on the GPU using CUDA. Wei et al. [18] take an alternative approach of non-linear beam tracing on the GPU for deformable objects.

General purpose processing on the GPU provided efficient solution for parallel solutions. Sequential algorithms were efficiently parallelized for the use of GPU. Fast solutions are provided for audio and signal processing, computational geometry, data parallel algorithms, databases, data compression and data structures. With the introduction of CUDA architecture, GPGPU problems are not addressed with a much simpler API for GPU. Operations like sorting, searching and other data structure problems have been efficiently addressed for large data sets [5, 12]. Data structures have applications in multiple fields and efficient implementation of basic primitives have been addressed with their applications in various fields [6].

2. Ray Casting Deformable Models

Ray-casting is a highly parallel operation. In contrast to rasterization which maps the world on to the camera, ray-casting operates on every ray, yielding a highly parallel framework. In the process of ray-casting, each ray needs to process all triangles and identify the one which is closest, if any. For a considerable amount of geometry and large image size, it becomes a computationally heavy operation.

To speed up ray casting, we need to reduce the number of ray-triangle intersections per pixel/ray. This is achieved by organizing the triangles into a data structure such that a front-to-back scanning is possible. Data structures like k-d trees, grids, octrees, etc. , that organize the data spatially in the world space are used commonly. Rays traverse the data structure to find a valid subset of intersecting triangles. Cost of building the world space data structures is high. Thus, they are computed at the beginning as a preprocessing step, making them unsuitable for deformable models. Zhou et al. [20] report real-time k-d tree construction on graphics hardware for small and medium sized models. For a model with 178K triangles, the construction time of the k-d tree is 78msecs and consequent rendering achieves 6fps on the latest GPU. Shevtsov et al. [14] deliver $7-12$ fps on models consisting of $200K$ dynamic triangles with shadows and texture using a parallel fast k-d tree construction on the CPU. Beam tracing exploits the coherence of pixels belonging to the same primitive to reduce the computation. Nonlinear Beam Tracing on GPU [18] proposes a fast beam-tracing procedure. They render models with $80K$

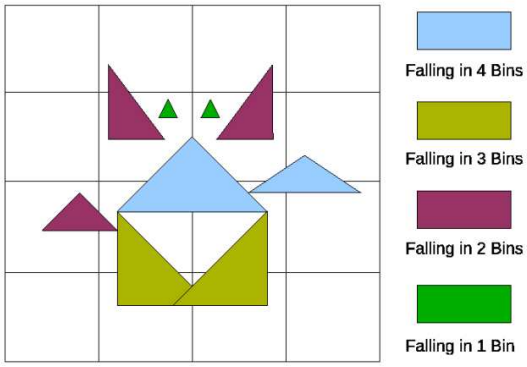

Figure 2: 2D view of the data structure for Ray Casting. Image-space is divided into Tiles.

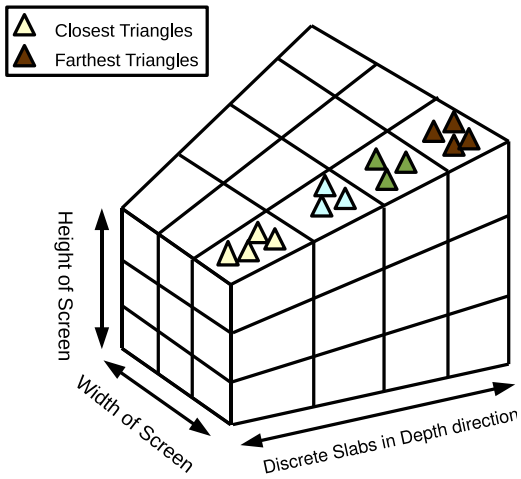

Figure 3: 3D view of the data structure. Tiles in the image-space are divided into frustum shaped slabs in z direction.

triangles at real time rates on the GPU. Above approaches work only for small and medium sized models.

2.1. Data Structure for Ray Casting

To ray cast a million triangle model onto a million pixel window, we need a data structure that can be built and processed at real-time rates. we propose a 3D data structure for this purpose as explained next. Processing image-space tiles exploits ray-coherence and has a flavor of beam tracing.

We divide the rendering area into regular *tiles* which represent a set of rays/pixels (Fig. 2). We sort the triangles to the tiles and limit the rays of each tile to intersect with the triangles that project into it. This produces batches of rays and triangles which can be independently processed on fine grained parallel machines like the GPU.

The number of triangles falling into each tile can be excessive to perform ray-triangle intersection with all the rays of the tile. If triangles in each tile are sorted in depth order the intersection can stop at the first occurrence. Sorting triangles of a tile on z completely is costly. We use a middle approach and divide the z-extent into discrete bins called *slabs* (Fig. 3). Each triangle is put into a slab based on its nearest z value. Triangles of a slab have no ordering with respect to each other, but triangles from different slabs do have a front to back ordering. For small tiles, this has the potential to exploit the spatial coherence of ray-triangle intersection.

The data structure is a grid of tiles in the image space which are further divided into discrete slabs in the depth direction. Each triangle is projected onto the screen and intersecting tiles are recorded against the triangle. Triangle is inserted into one or more slabs based on its tiles and distance from the camera. Thus, the data structure holds triangles ordered by tiles and slabs within each tile.

2.2 Ray Casting Algorithm

The CUDA algorithm for ray-casting is given in Algorithm 1. In ray-casting, all rays of a tile operate in parallel. Each ray intersects with all triangles of the next slab. The closest intersection point for each ray is kept track of. Rays which find a valid intersection drop out when a slab is completely processed. The computation ends, if all rays drop out. Otherwise the computation proceeds with the triangles of the next slab. Computation terminates when all slabs are done for all tiles.

We map a tile to a thread-block and each ray to a thread in it. The triangles reside in global memory, which is much slower to access than local shared memory. Since the triangles of a slab are all needed by all threads of a block, we bring the triangles to the shared memory before ray-triangle intersection are computed. The shared memory available to a block is limited on the GPU. All triangles of a slab may not, thus, fit into the available shared memory. We, therefore, treat triangles of a slab to be made up of *batches* which can fit into the shared memory. Triangles are loaded in units of batches. The threads of a block share the loading task equally among themselves when each batch is loaded (Line 4).

The ray-triangle intersection starts after all triangles of a batch are loaded. Each thread computes the intersection of its ray with each triangle of the current batch and stores the closest intersection in the shared memory. The next batch of the slab is loaded when all threads have processed all triangles in the current batch. This repeats till the slab ends. Each thread determines if its ray found a valid intersection with the slab and sets a local flag, *rayDone* (Alg. 1). The

#Bins	Global Memory Atomic			Hist/Thread[1] Approach			Multi Level Shared								
	1*M*	4*M*	16*M*	1*M*	4*M*	16*M*	1*M*			4*M*			16*M*		
							1-L	2-L	3-L	1-L	2-L	3-L	1-L	2-L	3-L
32	238	950	3815	2.9	25	164	2.5	-	-	11	-	-	90	-	-
64	170	673	2684	3.6	28	180	2.73	-	-	25	-	-	109	-	-
128	85	340	1370	14	46	213	2.78	-	-	25	-	-	148	-	-
256	44	177	710	27	88	343	2.87	-	-	26	-	-	168	-	-
512	33	132	520	53	173	652	2.9	6.5	-	25	39	-	169	186	-
1024	23	101	393	104	340	1284	3.38	6.4	10	25	39	44	172	183	215
2048	21	80	319	205	666	2514	4.3	5.5	8.3	26	37	41	158	225	205
4096	17	68	266	x	x	x	x	5.4	7.5	x	44	37	x	249	184
8192	15	58	273	x	x	x	x	5.7	6.8	x	40	40	x	270	198
16K	14	57	279	x	x	x	x	5.6	6.3	x	39	37	x	288	209
32K	15	59	286	x	x	x	x	5.5	6.4	x	37	35	x	241	217
64K	15.5	64	305	x	x	x	x	5.7	6.4	x	37	36	x	234	225
128K	15.3	62	303	x	x	x	x	6.0	6.2	x	32	37	x	216	222
256K	15.7	67	303	x	x	x	x	9.0	6.2	x	38	38	x	221	221
512K	15.5	64	301	x	x	x	x	12	6.1	x	39	38	x	222	225

Table 1: Comparison of the three Split operation for range of #bins and #elements. Histogram / Thread approach is similar to He's work [6]. Due to over-use of shared memory, maximum number of bins are limited to 64, thus we need multiple passes for bins greater than 64. We perform 2 *level* and 3 *level* splits where ever possible. '-' denotes the configuration which is not of interest.'x' denotes a configuration which is not possible.

whole block drops out from the rest of operations, if all the rays are done. This is evaluated using a logical AND of all local flags of the block in a procedure described later. If any ray is not yet done, computation in the block continues with the next slab of triangles. All threads of the block take part in loading the triangles of subsequent batches, but the threads with rayDone set do not participate in the intersection computation. Thus, threads which have found an intersection might diverge (Alg. 1 Line 6).

The threads of a block operate independently. Evaluating aggregate information of data stored in different threads, such as the logical AND of a bit, is difficult and slow. We, however, use a fast technique to compute the logical AND of the individual local ray flags. First, a common memory location in the shared memory is initialized to 1. Every thread that is not done writes a 0 to it and others abstain. CUDA architecture does not guarantee any specific order of writing when multiple threads write to the same shared location simultaneously. It, however, guarantees that one of the threads will succeed. That is sufficient for the above procedure to compute a logical AND in one instruction.

Our ray casting algorithm requires a 3D data structure which has triangles sorted to tiles in the image space. Triangles in each tile are arranged in z-slabs which are ordered from front to back from the camera. Considering triangles to be elements which can go to more than one tile in the image space, the problem of building the required data structure is similar to performing a multi-split. Building a compact list of triangles which are arranged by tiles and slabs is not straight forward on a parallel hardware. We propose a fast implementation of split and multi-split operation on GPU which can keep up with real-time rates for fast rendering of heavy deformable models.

3. Split Operation on the GPU

Split is a widely used operation for building data structures and performing database operations. Split can be defined as $append(x, List[category(x)])$, where each x is an input element and *List* holds all the categories x belongs to. Split is a function which divides an input relation into a number of partitions. Compact function makes sure that the output of split is a contiguous single list. Multi-split refers to the case when each element from the input relation can map to multiple categories. Thus, the partitions created are not disjoint. This increases the size of the output relation above the input relation. Split can be described as a 3 step procedure as below.

1. Count the number of elements falling into each bin.
2. Find the starting index for each bin using a prefix sum.
3. Assign each element to the output within its position, incrementally.

[1] Our implementation of He et al. [6]. For number of bins greater than 64 we implemented a multi pass approach which iterates over data multiple times.

Algorithm 1 CUDA_RAYCASTING :: Ray casting by the GPU using 3-D data structure

```
1: {Each Block executes the following in parallel on the GPU}
2: for each slab of this tile do
3:    for batch = 1 to maxBatch(slab) do
4:       Load triangles of the current batch from global memory
5:       SyncThreads
6:       if ( !rayDone ) then
7:          Perform ray-triangle intersections with the triangles of current batch
8:          Keep track of closest triangle, minz
9:       end if
10:      SyncThreads {All threads sync here to maintain data consistency}
11:   end for
12:   rayDone ← 1 if ray intersects
13:   allDone ← 1
14:   if ( !rayDone ) then
15:      allDone ← 0 {All threads in parallel}
16:   end if
17:   SyncThreads
18:   terminate if allDone
19: end for
20: Perform lighting/shading using the nearest intersection
```

It is tricky to perform step 1 on parallel architectures as multiple elements may increment the same variable. Step 2 can be performed as described in the parallel prefix sum operation by Blelloch et al. [1], which is implemented on the GPU by Sengupta et al. [12]. In step 3 we need to find the index for each element by incrementing the count for current bin which is similar to step 1.

An easy way to perform parallel split is to use the existing atomic operation hardware on GPUs. Atomic operation like add, subtract, increment, etc. , can be performed on the global memory on CUDA. This method suffers from two main drawbacks, first, the global memory access takes 300 to 400 clock cycles and second, the performance of atomic operation suffers in the presence of collisions in memory writes. Thus, for a skewed data or badly arranged data if multiple threads tend to write or increment same memory location the cost is too high, as the global memory is slow to access.

A recent paper by He. et.al. [6] computes the bin counts without atomic operations on the GPU. Each processor/thread handles a disjoint portion of the input and builds the complete histogram for its part in the shared memory. These partial histograms are written to globally accessible memory to separate locations. A parallel prefix sum [12] over this data gives the required counts which is used to send each data to the correct location. Due to their use of separate histograms for each processor, no memory conflicts occur. This procedure can, however be used only when the number of categories is small due to the limited shared memory available on the GPU. Given, 16KB of shared memory is available, the method supports bins only up to 64 using 32 threads per block. Our implementation of the above approach uses a multi pass approach in order to split into more than 64 bins.

Algorithm 2 SHARED_MEMORY_SPLIT :: Split function as implemented on GPU using CUDA. We use a approach similar to that proposed by Shams et al. [13] for performing atomic operations on shared memory

```
1: Compute Histogram per Block
2: Store it bin-wise in global memory, #Bins × #Blocks
3: Scan the histogram array :: Scan elements give index of each bin for each block
4: Load part of scan array corresponding to block into shared memory
5: Read x and category(x)
6: Read the scan histogram value for the bin and increment it atomically
7: Write x to value read from the shared memory
```

We combine elements of the atomic-based algorithm and He et al. [6] algorithm using a shared memory atomic operations. Shared memory is accessible to all threads of the block and is fast, making it an ideal candidate to maintain the common information among threads. We maintain a common histogram in the shared memory for all threads of the block. We perform the atomic operations on the shared memory using a method similar to that proposed by Shams et al. [13] to avoid conflicts. Due to the efficient use of available shared memory, we can perform split for bins up to 2048.

The limited shared memory will restrict the number of bins for a split operation. For large number of bins, we can split the elements in multiple steps by inducing a hierarchy in the bins. All M elements are split to n_1 bins in the first step. In the next step, all elements in each bin gets split into n_2 sub-bins. The data has now been split into $n_1 n_2$ bins. This can be repeated until the number of bins is N. We call this approach *multi level split*. With multi level split, we can perform split with bins as high as 2048×2048, although for best performance on GPU hardware a 3 or higher levels may give a better performance.

We use 32 threads per block and vary the number of blocks depending on size of input list. We divide the input array into chunks corresponding to blocks. For maximum efficiency we use a suitable number of blocks which decides the number of elements handled by each block. We make sure only one thread updates the count of a bin by using

Algorithm 3 RAYCASTING :: Complete algorithm for per frame building data structure and ray casting, deformable triangulated models.

1: Compute tile IDs (x,y) for each triangle along with minimum z-projection coordinate
2: Perform reduction on z-projection coordinate to find out minimum and maximum z-projection value for current frame
3: First level split (Algorithm 2) is performed by looking up x-tile IDs for each triangle
4: Segmented second level split is performed using the y-tile IDs on the output of above step
5: Split in the z-direction is performed by computing a z-tile ID using zMin and zMax for well fitting z-slabs
6: Histogram of triangles falling into tiles and slabs is outputted along with the scan of the histogram from the above step
7: Ray Casting is performed as per Algorithm 1 using the above output

# Z-Slabs	DS Time (msec)			RC Time	Total
	X	Y	Z	(msec)	(msec)
1	6.1	3.5	-	52	62
4	6.3	4.2	4.1	30	44
8	6.0	3	3.1	28	40
16	6.1	3.1	3	24	36

Table 2: Data structure building time and Ray Casting time for varying number of Z-Slabs. For a single slab split is not performed in the z direction. Level 1 split is performed on tiles in X direction, second and third are then performed on Y and Z direction respectively.

thread's id within the warp/block.

Table 1 depicts performance of three different approaches for Split operation. Performance of the global atomic approach is affected by number of clashes, making it data dependent. The approach performs well with high number of bins. Hist/Thread approach is our implementation of He et al. [6], which limits its number of bins to 64 due to limited shared memory. We perform multiple iterations over the data for bins higher than 64. Multi Level Shared uses one histogram per CUDA block. For higher number of bins 2-Level and 3-Level split is used and allows a wide range of number of bins.

4. Ray casting with Multi Level Split

Ray Casting requires small tiles in the image space, the order of 8×8. Thus, a large number of tiles and a moderate number of slabs will work best for real-time ray casting of heavy models. For a 1024×1024 window/image size we use 128×128 tiles in the image space along with 16 slabs in the z-direction, giving $128 \times 128 \times 16$ bins. We hierarchically organize the bins and perform a 3 level split to build the required data structure.

Dragon Preview				
Rotation Angle	0	30	60	90
RTI (M/frame) 16-Slabs	43.7	44.3	33.2	23.9
RTI (M/frame) 0-Slabs	77.2	77.8	77.3	78.8

Table 3: Number of Ray-Triangle Intersections (RTI) performed per frame for Dragon Model ($\sim 1M$ triangles) after multi-sorting triangles to tiles. With increase in depth complexity of the model, z-slabs tend to deliver better performance.

We perform first level of split by dividing the image space into a 128 tiles in the x direction and sorting the triangle to these 128 bins. We then perform segmented split over these 128 bins by dividing each of the partitions to 128 y oriented tiles. For second and third level segmented splits, elements of each partition take part. We then perform a third level split considering the distance of triangles from the camera and binning the triangles into 16 different bins.

We consider highly triangulated models of the order of $1M$ for ray casting on a image size of $1M$ pixels $(1K \times 1K)$. A 2D tiled data structure in the image space is built, which maps rays to CUDA threads and each tile to a CUDA block. To reduce the ray-triangle intersection for each ray, we divide the depth for each tile into z-slabs. High percentage of triangles are small in size and map to not more than $3-5$ tiles. A few triangles do fall into a lot of tiles. We experimented on various models and a triangle belongs to 1.2 tiles on an average. Table 3 shows number of ray-triangle intersections for various views of the Dragon model. When we do not use slabs we perform ray-triangle intersection for all rays of the tile against all triangles belonging to the tile, which remains independent of the view. With slabs, we stop at the closest intersection in a slab. Thus, slabs perform better for views with depth complexity of triangles. RTI/frame drops as Dragon model rotates from a side view (less depth complexity) to a frontal view (more depth complexity). Table 2 shows that the ray tracing time decreases with increasing number of slabs while the DS building time remains unaffected.

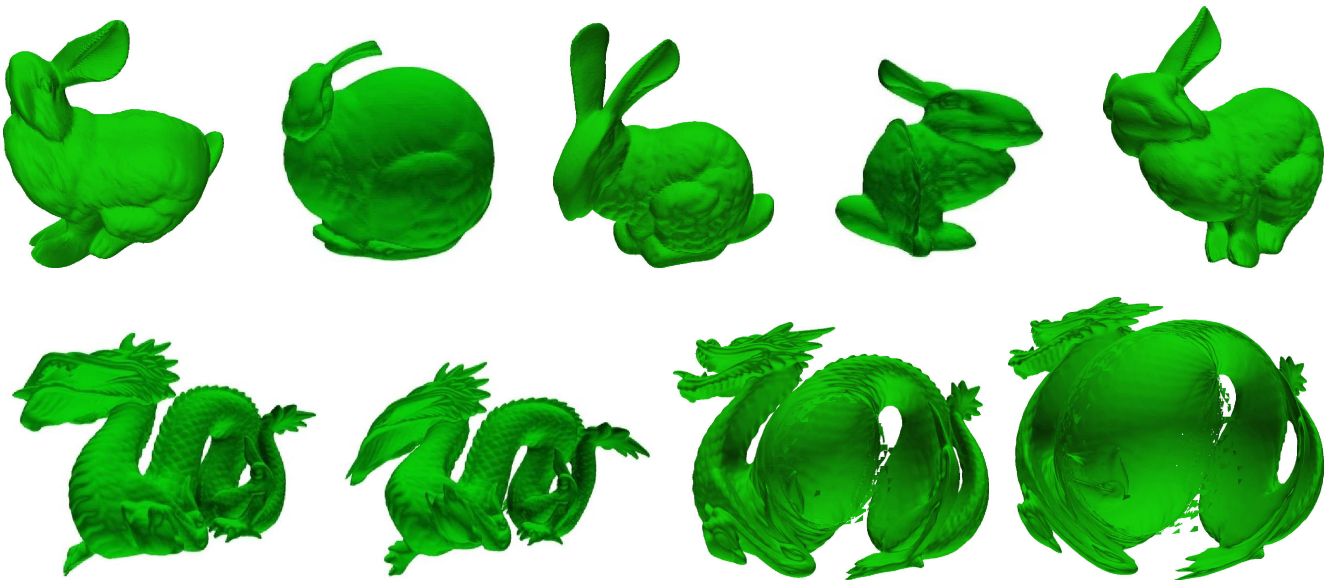

Figure 4: Top : Deformed Bunny Model. Bottom : Dragon Model going under deformation

Models→						
# Triangles	1.09M	870K	70K	346K	97K	641K
Tile Sorting	3.5	2.79	0.3	1.13	0.37	2.05
DS Building	15	13	2	4.6	2.1	11
Ray Casting	35	25	8	17	12	27
Frame Rate	20	26	100	47	70	26

Table 4: Data Structure building and Ray Casting time for various triangulated models on an Nvidia 8800 GTX.

We perform three level split on triangles of the model to build the required data structure. Multi split is performed as each triangle can map to more than one tile (Figure 2). Each *id* comprises of three numbers, x-tile, y-tile and z-bin. Each level of split uses these *ids* to sort triangles to corresponding bins. We use 128×128 tiles in the image space, and up to 16 slabs in the z direction for building an optimal data structure. Table 4 shows break-up timings for various triangulated models. Tile sorting and DS building time is proportional to number of triangles. Ray casting time depends on the number of triangles as well as the depth complexity of the model.

5. Conclusion and Future Work

We present a method to ray cast large deformable models at real time rates by building a suitable sorted data structure in each frame. The approach we take can work with other parallel hardwares, viz. FPGAs, Cell Processors etc. To maximize the performance of ray casting on GPU, we propose an image-space data structure which avoids traversing of data structure for each ray and arranges geometry for a set of rays. For construction of required data structure we develop a fast implementation of multi-split and propose multi level split which supports large number of bins.

We are working on tracing the secondary rays for shadowing, reflection, refraction etc. We have the triangles listed into a 3D grid of cells as part of ray casting for each frame. The secondary rays which spawn at the intersection points can traverse this grid to find second level intersection. We can use the 3-D data structure built each frame for shadow and secondary rays. Even though the data structure changes each frame, it inherently remains a uniform grid. Future GPUs with more cores and better memory performance will be able to ray trace large models interactively.

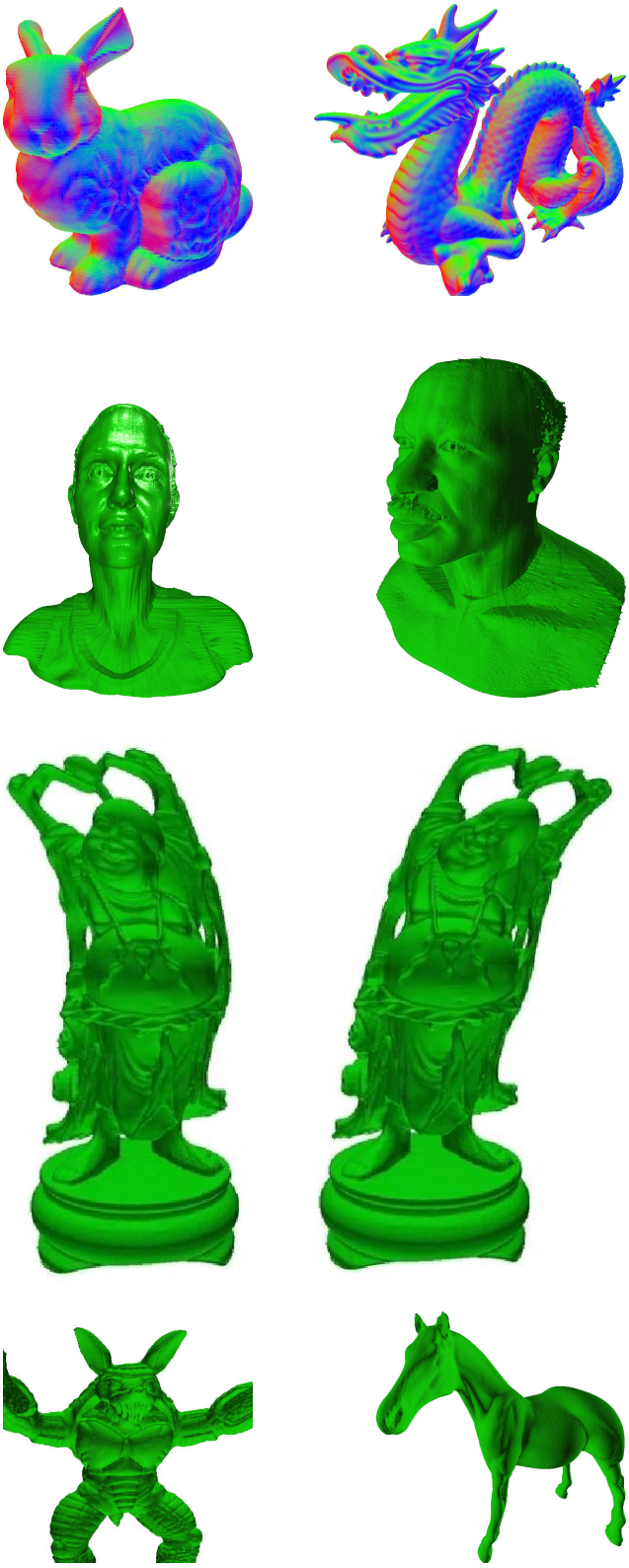

Figure 5: Various triangulated models (70K - 1.2M triangles) ray cast using our method.

References

[1] G. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.

[2] N. A. Carr, J. D. Hall, and J. C. Hart. The ray engine. In *In Proceedings of Graphics hardware*, pages 37–46. Eurographics Association, 2002.

[3] N. A. Carr, J. Hoberock, K. Crane, and J. C. Hart. Fast GPU ray tracing of dynamic meshes using geometry images. In *GI '06: Proceedings of Graphics Interface 2006*, pages 203–209, 2006.

[4] X. Gu, S. J. Gortler, and H. Hoppe. Geometry images. *ACM Trans. Graph.*, 21(3):355–361, 2002.

[5] M. Harris, J. D. Owens, S. Sengupta, Y. Zhang, and A. Davidson. CUDA data parallel primitives library. 2007.

[6] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *SIGMOD '08*, pages 511–524. ACM, 2008.

[7] P. S. Heckbert and P. Hanrahan. Beam tracing polygonal objects. In *Proceedings of the conference on Computer graphics and interactive techniques*, pages 119–127. ACM, 1984.

[8] D. R. Horn, J. Sugerman, M. Houston, and P. Hanrahan. Interactive k-d tree GPU ray tracing. In *Proceedings of I3D 2007*, pages 167–174. ACM, 2007.

[9] A. Knoll, Y. Hijazi, C. Hansen, I. Wald, and H. Hagen. Interactive ray tracing of arbitrary implicits with SIMD interval arithmetic. *Interactive Ray Tracing*, Sept. 2007.

[10] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. In *ACM SIGGRAPH 2005 Courses*, page 268. ACM, 2005.

[11] A. Reshetov, A. Soupikov, and J. Hurley. Multi-level ray tracing algorithm. *ACM Trans. Graph.*, 24(3), 2005.

[12] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for GPU computing. In *Proceedings of Graphics hardware*, pages 97–106, 2007.

[13] R. Shams and R. A. Kennedy. Efficient histogram algorithms for NVIDIA CUDA compatible devices. In *International Conference on Signal Processing and Communication Systems, 2007*.

[14] M. Shevtsov, A. Soupikov, and A. Kapustin. Highly parallel fast KD-tree construction for interactive ray tracing of dynamic scenes. volume 26, pages 395–404. Computer Graphics Forum, 2007.

[15] I. Wald, S. Boulos, and P. Shirley. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph.*, 26(1):6, 2007.

[16] I. Wald, T. Ize, A. Kensler, A. Knoll, and S. G. Parker. Ray tracing animated scenes using coherent grid traversal. *ACM Trans. Graph.*, 25(3):485–493, 2006.

[17] I. Wald, W. R. Mark, J. Günther, S. Boulos, T. Ize, W. Hunt, S. G. Parker, and P. Shirley. State of the art in ray tracing animated scenes. In *STAR Proceedings of Eurographics, 2007*.

[18] L.-Y. Wei, B. Liu, X. Yang, C. Ma, Y.-Q. Xu, and B. Guo. Nonlinear beam tracing on a GPU, MSR-TR-2007-168.

[19] S.-E. Yoon, C. Lauterbach, and D. Manocha. R-LODs: fast lod-based ray tracing of massive models. *Vis. Comput.*, 22(9):772–784, 2006.

[20] K. Zhou, Q. Hou, R. Wang, and B. Guo. Real-time KD-Tree construction on graphics hardware. In *Proceedings of SIGGRAPH Asia*. ACM, 2008.