

Ray Casting Deformable Models on the GPU

Suryakant Patidar
skp@research.iiit.ac.in

P. J. Narayanan
pjn@iiit.ac.in

Center for Visual Information Technology
International Institute of Information Technology Hyderabad

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. Ray casting is an inherently parallel and highly compute intensive operation. 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. 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 [10, 16, 19] and the GPU [1, 9, 20, 18]. A lot of effort has been put into ray casting and ray tracing of static models at real-time rates. 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 mod-



Figure 1: Dragon ray cast (870K triangles)

els 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, GPUs etc. 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.

Earlier, general purpose computing on GPU was performed via the graphics pipeline, which only 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 processing options such as Cell Processor,

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, multi-cores, clusters, CellBE, FPGAs etc. Beam Tracing [6] 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 GPU, ray tracing was performed on CPU or on a cluster of CPUs. A single CPU works sequentially on all 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 [10] 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 a LOD based scheme which integrates simplified LODs of the model into k-d tree, performing efficient ray-triangle intersection. 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. Ray tracing has also been performed on non-triangulated models like implicit surfaces [8] and geometry images [3, 2].

Programmable GPUs performed ray tracing with the limited features of GPU and constrained programming model [1, 9]. Ray tracing was performed as a multi-pass operation due to insufficient capability of the fragment shaders. With growth in programmability of 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 [7]. Recent work by Zhou et. al. [20] builds and ray trace small and medium sized deformable models on the GPU using CUDA. Li-Yi Wie et. al. [18] take the 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 [4, 11]. Data structures have applications in multiple fields and efficient implementation of basic primitives have been addressed with their applications in various fields [5].

2. CUDA Architecture

CUDA is a programming interface to use this parallel architecture for general purpose computing. This interface is a set of library functions which can be coded as an extension of the C language. A compiler generates executable code for the CUDA device. The CPU sees a CUDA device as a multi-core co-processor. The CUDA design does not have memory restrictions of GPGPU. One can access all memory available on the device using CUDA with no restriction on its representation though the access times vary for different types of memory. This enhancement in the memory model allows programmers to better exploit the parallel power of the 8800 GTX processor for general purpose computing.

CUDA Hardware Model :: At the hardware level the 8800 GTX processor is a collection of 16 multiprocessors, with 8 processors each. Each multiprocessor has its own shared memory which is common to all the 8 processors inside it. It also has a set of 32-bit registers, texture, and constant memory caches. At any given cycle, each processor in the multiprocessor executes the same instruction on different data, which makes each a SIMD processor.

CUDA Programming Model :: For the programmer the CUDA model is a collection of threads running in parallel. A warp is a collection of threads that can run simultaneously

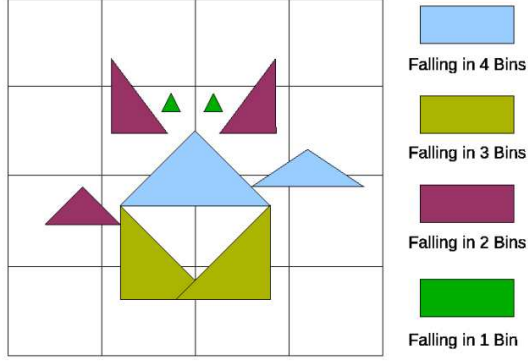


Figure 2: 2D view of the data structure for Ray Casting. Image-space is divided into Tiles.

on a multiprocessor. The warp size is fixed for a specific GPU. The programmer decides the number of threads to be executed. If the number of threads is more than the warp size, they are time-shared internally on the multiprocessor. A collection of threads (called a block) runs on a multiprocessor at a given time. Multiple blocks can be assigned to a single multiprocessor and their execution is time-shared. A single execution on a device generates a number of blocks.

3. 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. 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 based on their position into data structures. 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, 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 k-d tree is reported to be 78 msec and consequent rendering achieved 6fps on the latest GPU. Shevtsov et. al. [14] deliver 7 – 12fps on models consisting of 200K dynamic triangles with shadows and texture.

Beam tracing exploits the coherence of pixels belonging

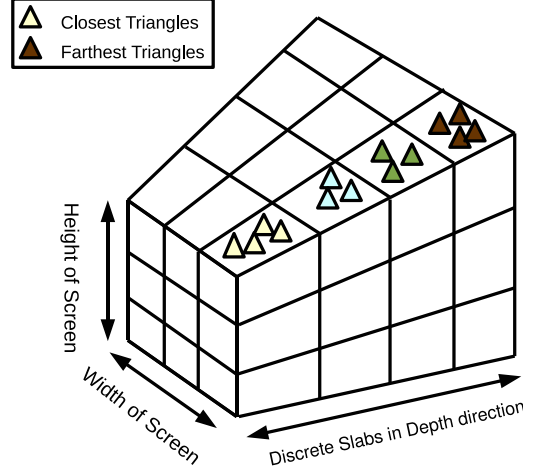


Figure 3: 3D view of the data structure. Tiles in the image-space are divided into frustum shaped slabs in z direction.

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 triangles at real time rates. Zhou et. al. [20] present a real-time kd-tree construction scheme on the GPU which supports ray tracing of deformable models. Small sized models with 80K are ray traced at 6 FPS on the GPU. Shevtsov et. al. [14] also proposed a parallel kd-tree construction on the CPU. They are able to render models with 200K triangles at 7 to 12 FPS. Above approaches work only for small and medium sized models.

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 3-dimensional data structure, with 2-d tiles in the image-space, and depth based slabs in the third dimension. This incorporates the features of beam tracing from the point of view of rendering and restricts the number of triangle intersections per ray to provide a real time ray casting of heavy deformable models.

3.1. Data Structure for Ray Casting

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

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 tri-





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 1: 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.

angles of a tile on z completely is costly. We use a middle approach and divide the z -extent into discrete bins called slabs (3). Each triangle is arranged to a slab based on its nearest z value. Triangles of a slab have no ordering with 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.

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.

3.2 Ray Casting Algorithm

The CUDA algorithm for ray-casting is given in Algorithm 1. The GPU architecture and available resources can place additional constraints on the above process. Under CUDA, 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 current GPU. All triangles of a slab may not 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.

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 currentBatch from global memory
5:     SyncThreads
6:     if ( !doneRay ) then
7:       Perform ray-triangle intersections with all the triangles
8:       Keep track of closest triangle, minz
9:     end if
10:    SyncThreads {All threads sync here to maintain data consistency}
11:  end for
12:  doneRay  $\leftarrow$  1 if ray intersects
13:  allDone  $\leftarrow$  1
14:  if ( !doneRay ) then
15:    allDone  $\leftarrow$  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

```

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* 1. The 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.

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

#Bins	CPU			Global Atomic			He [5]	Multi Level Shared								
	1M	4M	16M	1M	4M	16M	16M	1M			4M			16M		
								1-L	2-L	3-L	1-L	2-L	3-L	1-L	2-L	3-L
32	21.0	76	357	238	950	3815	x	2.5	-	-	11	-	-	90	-	-
64	21.1	79	365	170	673	2684	125	2.73	-	-	25	-	-	109	-	-
128	21.8	95	371	85	340	1370	x	2.78	-	-	25	-	-	148	-	-
256	22.0	93	378	44	177	710	x	2.87	-	-	26	-	-	168	-	-
512	22.6	93	391	33	132	520	x	2.9	6.5	-	25	39	-	169	186	-
1024	21.6	88	378	23	101	393	x	3.38	6.4	10	25	39	44	172	183	215
2048	23.1	93	397	21	80	319	x	4.3	5.5	8.3	26	37	41	158	225	205
4096	24.1	98	412	17	68	266	x	x	5.4	7.5	x	44	37	x	249	184
8192	25.6	101	428	15	58	273	x	x	5.7	6.8	x	40	40	x	270	198
16K	30.6	106	444	14	57	279	x	x	5.6	6.3	x	39	37	x	288	209
32K	41.7	127	533	15	59	286	x	x	5.5	6.4	x	37	35	x	241	217
64K	39.6	241	1069	15.5	64	305	x	x	5.7	6.4	x	37	36	x	234	225
128K	55.2	381	1527	15.3	62	303	x	x	6.0	6.2	x	32	37	x	216	222
256K	94.3	550	2291	15.7	67	303	x	x	9.0	6.2	x	38	38	x	221	221
512K	170	791	3246	15.5	64	301	x	x	12	6.1	x	39	38	x	222	225

Table 2: Comparison of the three Split operation for range of #bins and #elements. He et. al proposed a split operation to perform fast Relational Joins on the GPU. Due to over-use of shared memory, maximum number of bins are limited to 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/provided.

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 3-d 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.

4. 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. 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 function 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.

It is tricky to perform step 1 on parallel architectures as multiple elements may increment the same variable. Step 2 can be performed using an efficient prefix sum operation as described in work by Sengupta et. al. [11]. 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 making it simple to perform writes to single value by multiple threads on GPU. 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 [5] 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. These partial histograms are written to globally accessible memory to separate locations. A parallel prefix sum [11] over this data gives the required

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. [12] for performing atomic operations on shared memory

- 1: Compute Histogram per Block
- 2: Store it bin-wise in global memory, $\#Bins \times \#Blocks$
- 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

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 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.

We combine elements of the atomic-based algorithm and He et. al. [5] 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 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. [12] to avoid conflicts. Due to our efficient use of available shared memory, we can now 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 subbins. 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 – level or higher may give a better performance.

We use 32 threads per block and varying 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 thread’s id within the warp/block.

Ray Casting requires small tiles in the image space, the order of 8×8 . Thus, large number of tiles and moderate

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

- 1: Compute up to four (x,y) tile IDs 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 to four 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 (msec)	Total (msec)
	X	Y	Z		
1	6.1	3.5	3.8	52	75
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 3: Data structure building time and Ray Casting time for varying number of Z-Slabs. Z-Slabs=1 corresponds to brute force ray-triangle intersection within a Tile, thus 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.

number of slabs will work best for an efficient real-time ray casting of heavy models. For a 1024×1024 window/image size we would need 128×128 tiles in the image space along with another 16 slabs in the z -direction, thus making it a $128 \times 128 \times 16$ number of bins. We hierarchically organize the bins and perform a 3 level split to build the required data structure.

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.

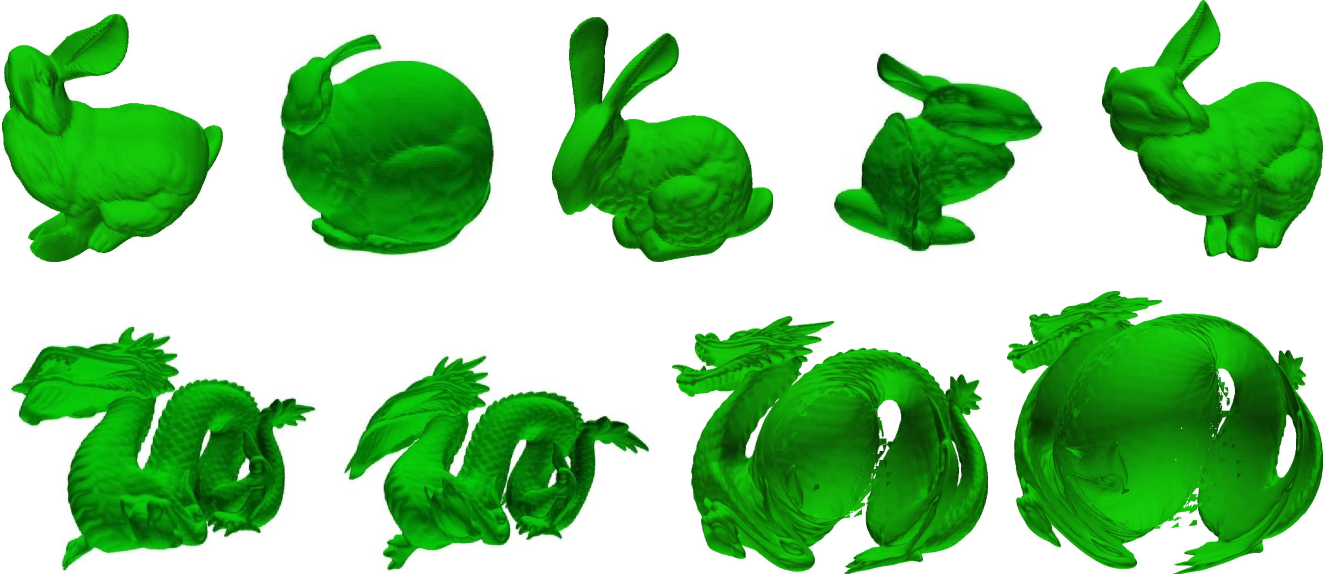


Figure 4: Top : Deformed Bunny Model. Bottom : Dragon Model going under deformation







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.

5. Ray casting with Multi Level Split

We consider highly triangulated models of the order of $1M$ for ray casting on a image size of $1M$ ($1K \times 1K$) pixels. We assume triangles to map to $1 - 20$ pixels on the window and use this assumption to consider a triangle falling into not more than 4 tiles (a tile is 8×8 pixels in the image space). A 2-d tiled data structure in the image space is built which maps rays to CUDA threads and each tile to a CUDA block. Instead of building a data structure in the world space and traversing the rays, we build a grid like structure in the image space to bring the geometry to the rays. Thus, a group of rays undergo ray-triangle intersection with a small set of geometry. To reduce the ray-triangle intersection for each ray, we divide the depth for each tile into z-slabs. A third level of hierarchy is thus built for a real time ray casting of heavy models. Multi level split which incorporates multi-split (a triangle mapping to more than

one tile) and segmented-splits is described in Figure 5.

For the purpose of ray casting we consider triangles to be sorted to tiles and slabs. Given, each triangle can fall into multiple tiles, we perform a first level of multi-split where each triangle is considered as many times as the number of tiles its falling into. Each triangle is appended to the output list multiple times corresponding to different tiles it falls in. The output size for Level 1 is greater than original number of triangles in a model. Level 1 outputs a X-Tile sorted list of triangles which are further considered for segmented-splits which is single-split as the triangles are now considered only once. At first level of split (L1), number of CUDA blocks are configurable. We consider 64 blocks which equally divide the input number of elements in order to perform the split.

First level of split outputs a X-tile sorted list of triangles with the starting point for each X-tile partition and the number of triangles belonging to the partition. Second

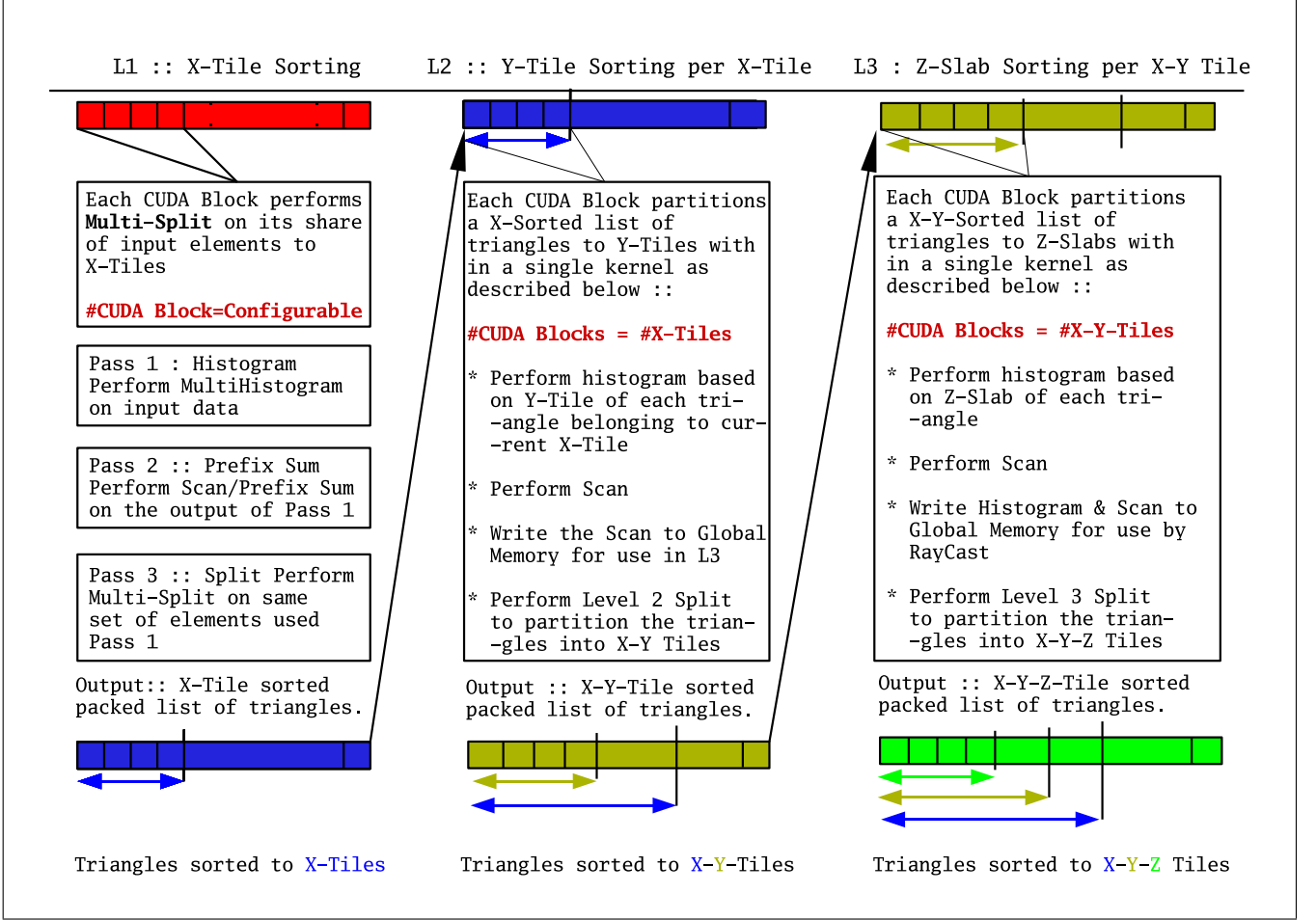


Figure 5: Multi Level Split as performed on triangles against X-Y Tiles and Z-Slabs for Ray casting. Level 1 (L1) splits the data and outputs a X-Tile sorted list of triangles, similarly Level 2 (L2) performs a segmented-split on output of L1 to output a X-Y Tile sorted list of triangles. L3 performs segmented split on the above list to obtain the final packed list of triangles sorted by Z-Slabs in each X-Y Tile.

level of split (L2) uses the output from L1 and performs a segmented-split on each of the partitions. We now have a hierarchy where each CUDA block performs split on a X-tile partition hence, we have as many CUDA block as there are number of X-tiles. Each block loads the partition of elements corresponding to its block/X-tile and builds the histogram based on triangle's Y-tile. Within the kernel each block performs a scan and uses it to perform the split, thus each block rearranges the X-tile partition by partitioning them into Y-tiles within the X-tile.

Second level of split outputs a X-Y-tile sorted list of triangles along with the histogram and scan of each partition. Third level (L3) divides each of these partitions (# X-Y-Tiles) based on Z-slab of each triangle. Z-slabs are decided based on distance of each triangle from the camera center. Each triangle is projected on the screen using the ModelView and Projection matrix for the current frame.

Projection coordinate range from 0.0 to 1.0. Triangles tend fall into a small range due to non-linear depth distribution. We therefore compute a zMin and zMax which correspond to the z projection coordinate of closest and farthest triangle respectively. We divide the constant number of Z-slabs between zMin and zMax, getting a well fitted and well distributed partition of triangles. With # X-Y-tiles number of CUDA blocks, we partition each of the set into these discrete Z-slabs to output a 3-d data structure along with the counts and starting point for each partition to be used by ray casting process. We use an image size of 1024×1024 which is divided into 128×128 tiles each of size 8×8 . Each tile is partitioned into 16 slabs. We perform a $128 \times 128 \times 16$ multi level split to build the desired data structure.

Algorithm 4 Shadow Ray Cast

- 1: For each shadow ray, enumerate a list of voxels between the point and light source from the 3D grid data structure using a modified line drawing algorithm in 3-D
 - 2: List of voxels for each rays of a tile are merged removing the duplicates to list the potential voxels in front to back order between the tile and the light source
 - 3: Perform ray triangle intersection for shadow rays on the above list. (similar to Algorithm 1)
 - 4: A ray stops at an intersection marking the pixel as shadowed
-

6. Possible Extensions

We can trace secondary rays which are responsible for shadowing, reflection, refraction and other effects. 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 in order to find second level intersection. Figure 6 shows the grid based data structure for a frame and the secondary rays starting at the intersection point and directed towards the lights for the purpose of shadowing. 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.

Once the grid-based data structure is built, it can be used to render shadow rays and secondary rays etc. We generate the shadow rays from the point of intersections and enumerate tiles in each's path using a modified line-drawing algorithm. Each cell or slab is processed in the path of the ray till intersection or free space. Given a grid data structure in camera view, we transform it to a space where the grid sits on the origin and grows in positive x, y and z direction for ease of computation. In order to perform shadowing, we transform the lights and rays to the above mentioned world space and traverse the grid cells to find an intersection. The color at a pixel is accumulated from the first pass of ray casting and multiple second level intersections. Image space effects like ambient occlusion have been recently performed on the GPU [13]. Our system can incorporate fast image space operations along with shadowing and secondary rays. Algorithm 4 gives the pseudo code for shadowing by a point light source. Multiple lights can be handled similarly. For area lights, multiple shadow rays can be generated for each intersection.

7. Conclusion

We build and ray cast heavy deformable models at real time rates. The approach we take can be incorporated with other parallel hardware, viz. FPGAs, Cell Processors etc.

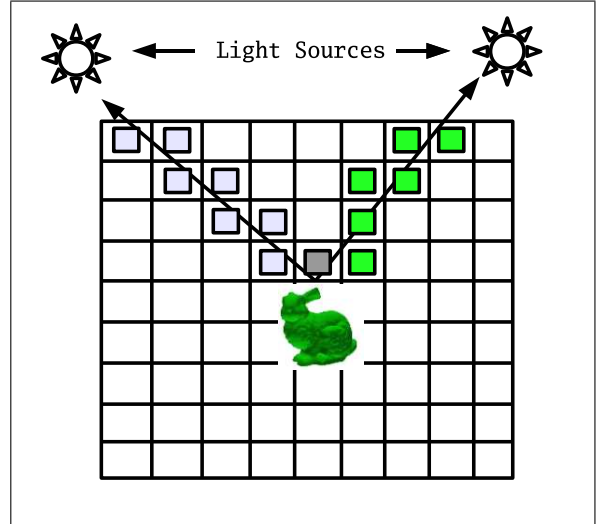
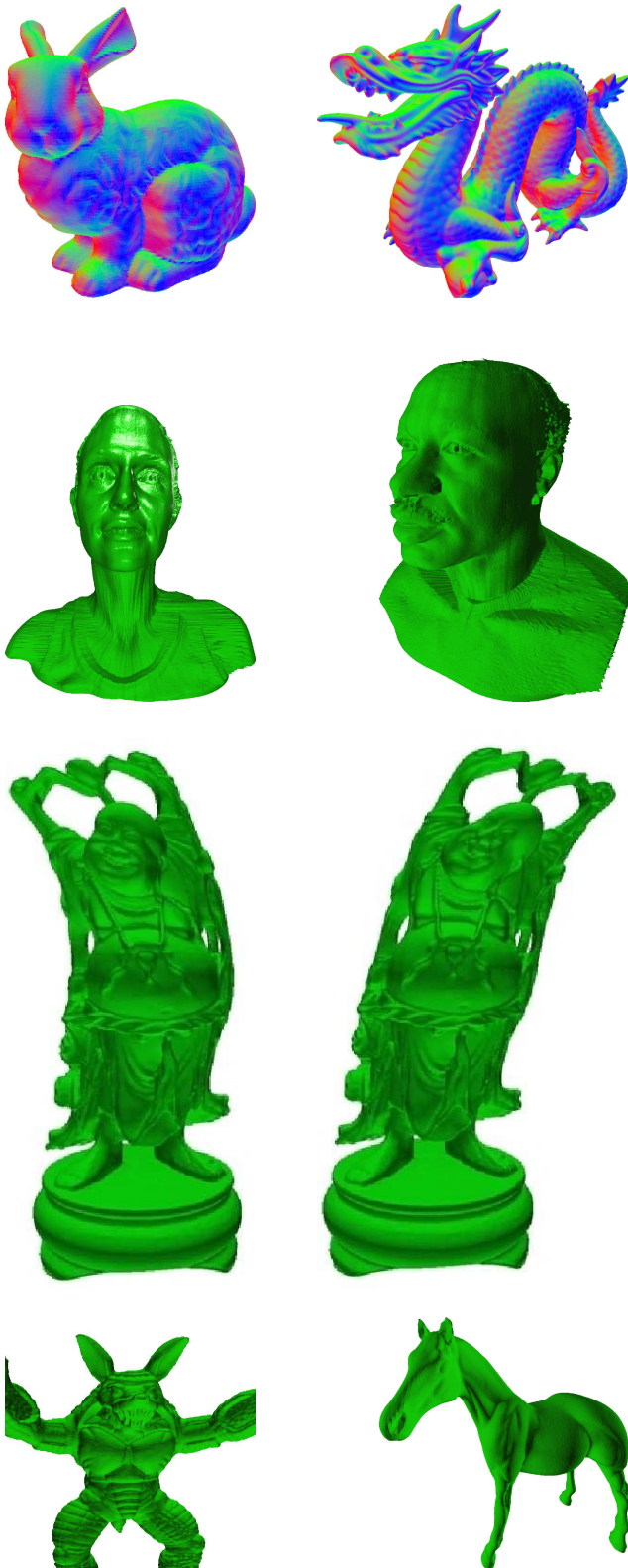


Figure 6: 3-D Grid is projected on to 2-D for explanation. Voxels connecting the intersecting point of the geometry and light can be found by an algorithm similar to line drawing algorithm in 3-D.

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 restrict our models to highly dense triangulated models which limits the number of tiles a triangle can fall in. We deliver a complete ray casting system which can build and ray cast heavy deformable models at real time rates. Our extensions toward ray tracing using the grid data structure aim to ray trace heavy deformable models at interactive rates.

References

- [1] N. A. Carr, J. D. Hall, and J. C. Hart. The ray engine. In *Proceedings of Graphics hardware*, 2002.
- [2] 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*.
- [3] X. Gu, S. J. Gortler, and H. Hoppe. Geometry images. *ACM Trans. Graph.*, 21(3), 2002.
- [4] M. Harris, J. D. Owens, S. Sengupta, Y. Zhang, and A. Davidson. Cuda data parallel primitives library. 2007.
- [5] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*.
- [6] P. S. Heckbert and P. Hanrahan. Beam tracing polygonal objects. In *Proceedings of the conference on Computer graphics and interactive techniques*, 1984.



- [7] D. R. Horn, J. Sugerman, M. Houston, and P. Hanrahan. Interactive k-d tree gpu raytracing. In *In Proceedings of I3D 2007*, 2007.
- [8] A. Knoll, Y. Hijazi, C. Hansen, I. Wald, and H. Hagen. Interactive ray tracing of arbitrary implicits with simd interval arithmetic. *Interactive Ray Tracing, 2007. RT '07. IEEE Symposium on*, 2007.
- [9] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. In *ACM SIGGRAPH 2005 Courses*, 2005.
- [10] A. Reshetov, A. Soupikov, and J. Hurley. Multi-level ray tracing algorithm. *ACM Trans. Graph.*, 24(3), 2005.
- [11] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for gpu computing. In *In Proceedings of Graphics hardware*, 2007.
- [12] R. Shams and R. A. Kennedy. Efficient histogram algorithms for NVIDIA CUDA compatible devices. In *ICSPCS, 2007*.
- [13] P. Shanmugam and O. Arikan. Hardware accelerated ambient occlusion techniques on gpus. In *Proceedings of symposium on Interactive 3D graphics and games*, 2007.
- [14] M. Shevtsov, A. Soupikov, and A. Kapustin. Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes. *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), 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), 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. 2008.
- [19] S.-E. Yoon, C. Lauterbach, and D. Manocha. R-lods: fast lod-based ray tracing of massive models. *Vis. Comput.*, 22(9), 2006.
- [20] K. Zhou, Q. Hou, R. Wang, and B. Guo. Real-time kd-tree construction on graphics hardware, msr-tr-2008-52. 2008.