Exact and Adaptive Signed Distance Fields Computation for Rigid and Deformable Models on GPUs

Fuchang Liu, Young J. Kim, *Member, IEEE*

Abstract—Most techniques for real-time construction of a signed distance field, whether on a CPU or GPU, involve approximate distances. We use a GPU to build an exact adaptive distance field, constructed from an octree by using the Morton code. We use rectangle-swept spheres to construct a bounding volume hierarchy (BVH) around a triangulated model. To speed up BVH construction, we can use a multi-BVH structure to improve the workload balance between GPU processors. An upper bound on distance to the model provided by the octree itself allows us to reduce the number of BVHs involved in determining the distances from the centers of octree nodes at successively lower levels, prior to an exact distance query involving the remaining BVHs. Distance fields can be constructed 35-64 times as fast as a serial CPU implementation of a similar algorithm, allowing us to simulate a piece of fabric interacting with the Stanford Bunny at 20 frames per second.

Index Terms—Distance fields, GPU, Octree, Bounding volume hierarchies, Physics simulation.

1 Introduction

A distance field is a scalar field that represents the shortest distance between a point in space and a model. A scalar field is usually approximated by a regular or adaptive grid. Applications of distance fields include collision detection [1], physics simulation [2], [3], [4], motion planning [5], mesh generation [6] and geometric modeling [7], [8].

Brute-force computation of exact distance fields takes a long time, and many methods have been proposed to construct distance fields more efficiently. However, most algorithms are not designed for GPUs, and can not cope with adaptive grids or deformable models. We present an efficient GPU-based method of constructing a global signed distance field on an adaptive grid for any model represented by a mesh of triangles. Previous work on computing distance fields on a GPU has mostly been based on approximate distance computations using uniform grids, which require a lot of memory to achieve high resolution.

We use adaptive grids in the form of octrees, like many CPU-based algorithms for creating distance fields. However, switching to GPUs from CPUs is quite challenging due to the following reasons:

- Dynamic memory allocation and pointer creation are used extensively in constructing octree on a CPU implementation, which may not be effectively implemented on a GPU.
- Hierarchical query structures such as bounding volume hierarchy (BVH) are often employed to accelerate the massive number of distance queries to construct distance fields. However, implementing these structures on GPUs is non-trivial since an intelligent load balancing mechanism is required to effectively utilize thousands of GPU threads. Furthermore, these structures should be able to handle dynamic models, for instance, under deformation on GPUs.
- Efficient and parallel traversal on the above hierarchical structure is similarly non-trivial due to the same load balancing issue.

Main Results: In our paper, we address the aforementioned challenges by using a novel, multi-BVH structure and efficient, parallel distance culling techniques. More specifically, we construct a set of BVH structures (i.e. multi-BVH) by exploiting the space coherence of the input triangles, encoded by Morton code. Then, we employ a culling strategy to quickly determine which BVHs can be discarded since they do not contribute to the final distance fields.

The computational pipeline of our algorithm is shown in Fig. 1. We begin by sorting the triangles in the model using Morton code of their centroids and then build an octree that indexes the triangles by utilizing the space coherence of Morton code. Distance queries are facilitated by a BVH of rectangular swept spheres (RSSs) [9]. We also show how to extend this approach to deformable models, using a multi-BVH structure. We cluster the triangles based on their Morton codes and build a BVH for each group. Determining the optimal axis of an RSS takes

• *Fuchang Liu and Young J. Kim are with the Department of Computer Engineering, Ewha Womans University, Seoul, South Korea. Young J. Kim is the corresponding author.*
E-mail: {liufu, kimy}@ewha.ac.kr

a significant amount of time and therefore we adopt an axis approximation method. The shortest distance between the center of every node in octree and the model is refined by traversing a BVH from its root to its leaves. A tight upper bound can be determined by a multi-resolution grid. We can spawn tens of thousands of distance query threads concurrently on a GPU. In order to reduce the number of BVHs to be queried by each thread, we apply an upper bound to cull them.

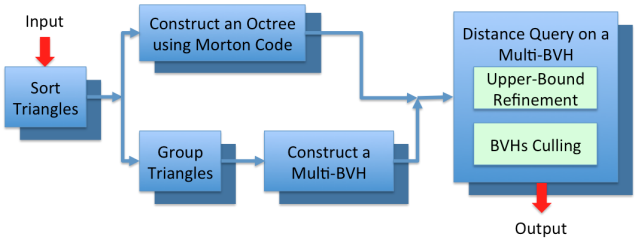


Fig. 1. Pipeline of adaptive distance fields.

The novel aspects of our work can be summarized as follows:

- To the best of our knowledge, our work is the first interactive-rate GPU-based algorithm for adaptive distance fields.
- In order to reduce the computational cost of BVH construction which is dominant in computing distance fields, we employ a multi-BVH structure which improves parallelism efficiency significantly when generating nodes near the top of the BVH.
- Based on the multi-BVH structure, we develop a new BVHs culling technique based on dimension reduction.
- Axis approximation is adopted for effectively constructing small nodes during multi-BVH construction.
- An octree is built in a top-down way on GPUs to represent adaptive distance fields using Morton code.

Organization: The rest of this paper is organized as follows. In Section 2, we will review some previous work. In Section 3, we describe our octree structure. Section 4 describes our method of building multi-BVHs on a GPU. In Section 5 we describe refinement of the upper bound in distance queries and BVH culling to construct multi-BVHs. Finally, in Section 6 we present our results and draw conclusions.

2 Previous Work

Algorithms for constructing a distance field from a triangular mesh can be classified into three categories:

2.1 Methods based on Voronoi Diagrams

In these methods, a distance field is computed from the distances between points lying in a Voronoi cell and its Voronoi site [5], [10]. Hoff et al.[5] create generalized Voronoi diagrams using a graph-based technique which constructs graphs of the sites' distance fields. However, each primitive of the input surface produces a large number of triangles to render, and so the method is inefficient for large meshes. Mauch's [10] characteristics/scan-conversion (CSC) algorithm computes the signed distance field for triangle meshes up to a given maximum distance d. The CSC algorithm does not compute exact Voronoi cells, but instead uses computationally more tractable polyhedra which contain the Voronoi cells. However, for triangular meshes and high grid resolutions, scan conversion and distance computation are much more expensive than construction of the Voronoi diagram. And CSC only computes the distance field in a narrow band around the surface. Sigg et al. [11] introduced the hardware-assisted prism algorithm which is a faster version of CSC. They use OpenGL's ARB fragment program for the nonlinear interpolation of distance values within a single polyhedron slice. Erleben et al. [12] eliminated artifacts due to leaking in the prism scan. Sud et al. [13] use the properties of a Voronoi diagram to cull primitives that do not contribute to the distance field. All these methods are based on a uniform grid.

2.2 Methods based on Distance Transforms

A distance transform (DT) creates distance field close to the surface of a model from which distances elsewhere may be determined [14]. DT methods can be classified into three categories by the method of distance computation that is used: Chamfer DTs, vector DTs and the fast marching method (FMM). Chamfer DTs [15], [16], [17], [18] use a distance template which is centered over each voxel.The distance from the central voxel is set to the minimum of all of its neighbor's distances. The accuracy of chamfer DT gets worse as the distance from the model surface increases. Vector DTs [19], [20], [21] address this problem by determining a vector to the closest point on the surface of the model; those vectors are then propagated outwards using a vector template. FMM [22], [23] is a well-known numerical scheme which can solve the Eikonal equation which determines the distance from a voxel to the surface of the model by applying a first or second order estimator to the distances from its neighbors to the model. None of these techniques are exact except in some special situations. Recently, Cao et al. [24] have proposed exact Euclidean DT which can be applied to a binary image in 2D and higher dimensions at interactive rates on current GPUs, but this method is not directly applicable to meshes, and the memory overhead of distance field computation increases as the cube of the resolution.

2.3 Methods based on a Voxel/Triangle Overlap List

Methods in this category reduce the cost of calculation by constructing a list of intersecting triangle for each voxel. Chang et al. [25] voxelize a triangular mesh and build a list of triangles for each boundary on the model which they use to build a local signed distance field. Chang et al.'s method is CPU-based and their distance field is uniform and only exists within a narrow band around the surface of the model. K. Yin et al. [26] also use an intersecting triangle list for computing distance fields. But they compute adaptive distance fields on a GPU by generating the adaptive sampling points using an octree, which is built in a bottom-up way. The maximum resolution of their distance field is limited to 512^3. And they must approximate the distance field for those points far from the surface.

More recently, [27] proposed a CUDA-based method for signed distance field calculation using adaptive grids on a GPU. However, this method performs only distance query on a GPU while adaptive grid generation is still executed on a CPU; thus their method may incur a communication overhead between CPUs and GPUs. All the methods we have reviewed here in Section 2 are approximate, local, or executed fully or partially on a CPU. To the best of our knowledge, ours is the only algorithm that can compute exact adaptive signed distance fields for a large triangle mesh at interactive rates using GPUs.

3 Adaptive Grid Generation

Adaptive distance fields (ADFs) are usually represented hierarchically, using structures such as quadtrees in 2D and octrees in 3D. The space occupied by the input model is subdivided recursively using a subdivision rule. Each quadtree or octree node stores the distance from its center to the surface of the model, as well as pointers to parent and child nodes. In this section, we will discuss the generation of an octree structure for ADFs on a GPU.

3.1 Octree Generation

Distance fields are relatively easy to implement on GPUs using regular grids, but these require a lot of storage. It is more efficient to use ADFs [28] which allocate storage more effectively, favoring the regions which contain fine detail. But problems of memory allocation and pointer creation make it difficult to use octrees on a GPU. Recently, octree representations, such as the hashed octree [29], and the linear octree [30], based on a hash table have been proposed in which pointers to children of nodes are replaced by lookup operation. However, these hashing strategies are limited to static cases.

There are usually two ways to construct octrees, either bottom-up [31] or top-down [32] approaches.

In [32] (i.e. top-down fashion), the main problem is lack of parallelism in the initial splits that makes most processors idle and triangle swapping during construction of that can cause global memory read and write overhead. For bottom-up fashion, it costs too much memory in the initialization. Thus, the finest grid resolution of octree is limited by GPU memory. We use a top-down approach to subdivide grids adaptively according to the criterion: the number of triangles associated with each grid, as illustrated in Fig. 2. In Fig. 2, compared to uniform grids generation, we start from a coarse level and only subdivide grids which contain more triangles than a threshold. The challenging issue is still how to maximally exploit GPU's parallelism for adaptive grids.

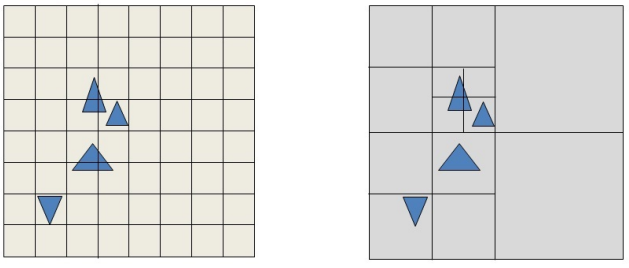


Fig. 2. Different grids subdivision comparison: Take 4 arbitrary triangles for example, 64 grids are generated in uniform grids subdivision on the left. Only 13 grids are generated in adaptive grids subdivision on the right.

3.2 Octree Construction on the GPU using Morton Code

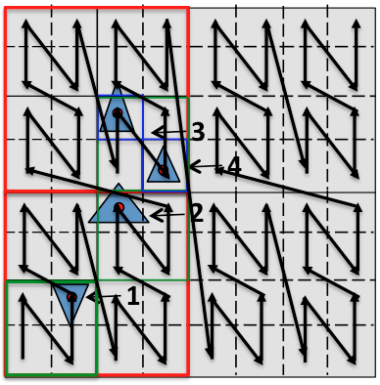
Morton code [33] has been used for building BVHs [31]. Note that a recursive decomposition of 3D space based on Morton code is equivalent to an octree decomposition. Additionally, we have discovered that sorting triangles on their Morton code can suggest which nodes of the octree should be decomposed further. Thus we can use Morton code to parallelize adaptive grids generation and store grids into octree data structures on a GPU.

Our octree construction algorithm has some similarities with Zhou et al.'s data-parallel octree method [34] which also uses Morton code. Zhou et al. build an octree in a bottom-up way, because they also need to build a vertex array for all the octree nodes, which is subsequentially used for surface reconstruction. However, our algorithm uses top-down, starting from a coarse grid rather than the root; we can directly build nodes at depth of D, which corresponds to resolution $(2^D)^3$. Thus, we can arbitrarily define the finest resolution according to user-specified requirement and easily build octree from coarsest depth to finest depth without any intermediate nodes construction.

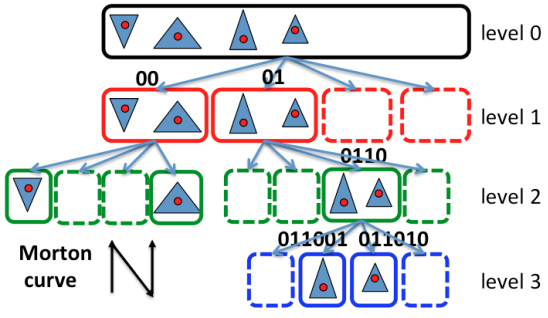
Morton code makes it easy to determine the number of triangles in each node across levels; since all the

triangles contained by the same octree node have the same Morton code, by identifying the triangles having the same Morton code, we can count the number of triangles that each node contains. We use a simple subdivision criterion: if the number of triangles is larger than a threshold, subdivide the node, otherwise, it is a leaf. We store the octree nodes in a node list. A node t contains the following information:
1) The $3m$-bit Morton code, where m is the depth of the node.
2) The coordinates of the center of the node (i.e. the sampling point).
3) Pointers to the parent and first child of the node.

Fig. 3 shows how we use Morton code to decide which node to divide further. At level 1, two nodes contain more than one triangle. If the division criterion as one triangle, then we must divide these two nodes containing eight nodes at level 2. But one node still contains more than one triangle. We continue dividing this node, and obtain four nodes at level 3. Since all the nodes now contain one triangle or none, division is terminated. Fig. 3 (a) shows that the order of the triangles which is (1, 2, 3, 4) has not changed across resolutions.



(a) triangles ordered by Morton code



(b) triangles in the quad tree

Fig. 3. The spatial coherence by sorting triangles with Morton code and quad tree generation.

We summarize our octree construction algorithm as Algorithm 1. Note that the octree subdivision recursively continues until all the octree nodes contain less than one triangle or the octree reaches the maximum depth. During parallel octree subdivision, the number of all children nodes at the same depth can be computed using atomic operations available on modern GPUs, for instance, such as AtomicInc in CUDA; refer to line 22 in Algorithm 1.

Algorithm 1 Octree construction
Input: T: list of triangles, l: starting depth, m: maximum depth, α: subdivision threshold
Output: O: list of octree nodes

1: **for** $i \leftarrow 0$ to $2^{3l} - 1$ **in parallel do**
2: $O_i.ID \leftarrow i$
3: $O_i.ParentID \leftarrow NULL$
4: $O_t.ChildID \leftarrow NULL$
5: $O_i.Center \leftarrow ComputeCenter(l)$
6: $O_i.N \leftarrow 0$
7: **end for**
8: **for** input triangle T_i **in parallel do**
9: compute $3m$-bit Morton code C_i^m
10: **end for**
11: sort triangles by $C_i^m (0 \leq i \leq n)$ **in parallel**
12: $j \leftarrow l$, $num \leftarrow 0$
13: **while** $j < m$ **do**
14: **for** C_i^m **in parallel do**
15: $C_i^l \leftarrow$ first $3j$-bit Morton code C_i^m
16: **if** C_i^l is different from C_{i-1}^l **then**
17: $O_i.N \leftarrow i$
18: **end if**
19: **end for**
20: **for each** node O_t **in parallel do**
21: **if** $O_t.N - O_{t-1}.N > \alpha$ **then**
22: $AtomicOperation(num \leftarrow num + 1)$
23: **end if**
24: **end for**
25: **for each** node O_t **in parallel do**
26: **if** $O_t.N - O_{t-1}.N > \alpha$ **then**
27: **for** $i \leftarrow 0$ to 7 **do**
28: $O_i^t.ID \leftarrow O_t.ID + num + i$
29: $O_i^t.ParentID \leftarrow O_t.ID$
30: $O_i^t.Center \leftarrow ComputerCenter(j)$
31: **end for**
32: $O_t.ChildID \leftarrow O_0^t.ID$
33: **end if**
34: **end for**
35: $j \leftarrow j + 1$
36: **end while**

Fig. 4 shows points at the centers of the nodes generated by adaptive division, the nodes of an octree from resolution 8^3 to 1024^3 using Morton Code on GPU. The red points indicate nodes which contain more triangles than a user-specified threshold. We can see that adaptive grids subdivision can generate more sampling points close to surface which contains more details.

4 BVH CONSTRUCTION

In this section, we describe how to build a bounding volume hierarchy (BVH) structure on a GPU, which will be used to accelerate distance queries for octree nodes. First we will describe the construction of a single BVH in two steps, 1) tree construction and 2) fitting rectangular swept spheres (RSSs) [9]. Then,

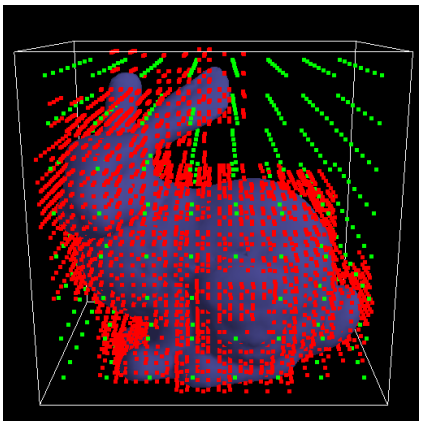


Fig. 4. Adaptive sampling using Morton code for the Stanford bunny model.

we will describe our novel multi-BVHs construction algorithm.

4.1 Tree Construction

In the last few years, many researchers have investigated the use of tight-fitting bounding volumes (BVs) for proximity queries. These include using orientated bounding boxes (OBBs) [35], [36], spherical shells [37] and k-discrete oriented polytopes (k-DOPs) [38], [39]. These tight fitting BVs give better performance than spheres or axis-aligned bounding boxes (AABBs). Rectangular swept sphere (RSS) is the volume filled by a sphere whose center is swept across the surface of a rectangle in 3D [9]. It may also be described as a rectangle offset uniformly in all directions. An RSS fits many models more tightly than other BVs, and its distance from a point can be easily found by calculating the distance from that point to the orienting rectangle and then subtracting the radius of the sphere. We build our BVH top-down using RSSs. At each node, we compute the optimal axis of the orienting rectangle of the RSS, and then determine the sphere radius.

Building a BVH tree on a GPU poses two challenges: parallelization and fitting BVs efficiently in parallel. In our tree construction, we split the underlying primitives and build the tree level by level from top to down. We further exploit the parallelism of GPU by constructing the tree in BFS (breadth-first) by spawning a thread for each node at each level of the tree; we make good use of modern GPUs' threads resource ($10^3 \sim 10^4$ threads available on modern GPUs). Then, each of these threads performs the following tasks concurrently. We compute an OBB (oriented bounding box) for each node, and then split the node across the longest axis of the OBB. Each triangle is allocated to one child node by comparing its centroid with the spatial median of the centroids of triangles along this axis. This computation is also performed in parallel. Fig. 5 shows our splitting scheme.

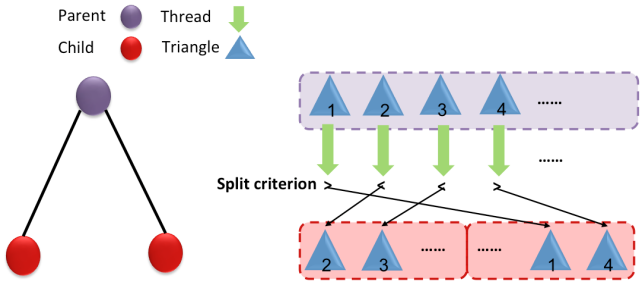


Fig. 5. Splitting nodes: triangles are allocated to child nodes by parallel swapping.

4.2 RSSs Fitting

As we build the tree, we also fit the RSSs to enclose the underlying triangle primitives. To fit an RSS, we first compute an OBB from the underlying triangles, and then choose the smallest of the three dimensions of the OBB as the normal direction of the oriented rectangle. The other dimensions of the OBB fix the orientation of the rectangle and the rectangle dimensions are uniformly grown until the RSS encloses all the triangles. We accelerate computing the variance matrix, center and extent of the OBB on GPUs by employing stream compaction [40] to compute the average and the minimum and maximum coordinates of triangles vertices.

We also use the idea of large and small nodes [32] to increase the effective use of parallelism. A large node is associated with more primitives than a user-specified threshold; the remaining nodes are small. When a RSS tree is being constructed, high-level nodes are associated with a large number of primitives (i.e. large node), and a group of GPU threads are spawned to handle the computational complexity. Meanwhile, a small number of threads are assigned to the nodes at low level (i.e. small nodes). As a result, at least a single thread is mapped to each node. However, for nodes near the root node, there are a relatively small number of nodes (i.e. small amount of workload), which makes the GPU parallelism under-utilized. Take the root for example; only a small portion of GPU threads will be needed, and the rest of GPU threads would go idle. We address this under-utilization problem near the root node by utilizing a multi-BVH structure in the next section. We summarize the splitting and fitting steps as Algorithm 2.

4.3 Multi-BVH Construction

The main problem using BVH construction algorithm on a GPU is the lack of parallelism when generating the nodes near the top of the tree as most of cores are idle. For example, for a GPU equipped with n cores, in order to make full use of this GPU, we need to give it n tasks. For simplicity of discussion, we consider the depth of a complete balanced binary tree with n nodes is $log_2(2n-1)$. Thus, a GPU with n

Algorithm 2 RSS tree construction
Input: T: list of triangles, α: split threshold
Output: R: list of RSS tree nodes

```
1: //Large node construction
2: activelist ← root node
3: while activelist is not empty do
4:    for node i in activelist in parallel do
5:       //Large nodes split
6:       compute RSS of node i
7:       compute split axis r_i and split point P
8:       n_i ← the number of triangles in node i
9:       for triangle T_j of nodes i in parallel do
10:         P_j ← project barycenter of T_j on to r_i
11:         if P_j < P then
12:            add T_j to left child node ch_0^i
13:         else
14:            add T_j to right child node ch_1^i
15:         end if
16:      end for
17:   end for
18:   //Add small nodes into smalllist
19:   add new child node ch^i to nextlist
20:   for node ch^i in nextlist in parallel do
21:      if ch^i is a small node then
22:         add ch^i to smalllist
23:         delete ch^i to nextlist
24:      end if
25:   end for
26:   swap nextlist and activelist
27:   clear nextlist
28: end while
29: //Small node construction
30: for node i in smalllist in parallel do
31:    create local stack S[]
32:    S[] ← node i
33:    while S[] is not empty do
34:       localnode ← S[]
35:       if localnode contains more than α triangles then
36:          do RSS fitting and node splitting in the same
           way as in large node construction
37:          S[] ← created children nodes
38:       else
39:          mark localnode as a leaf node
40:       end if
41:    end while
42: end for
```

cores can not be fully exploited unless the depth of BVH is greater than $log_2(2n-1)$. Other authors [31], [41], [42] have addressed this issue by employing a hierarchical grid decomposition for distributing the workload. But most of these technique use AABBs to exploit spatial coherence along the coordinate axes. It is hard to extend those methods to tighter fitting BVs such as RSSs, which require an optimal axis of arbitrary orientation.

For nodes near the root node, we create a number of work items and fill them into GPU cores to make the GPU threads busy. Specifically, if we decompose the BVH into many subtrees and fill each GPU core with a root of subtree, and the performance of root-level BVH construction would increase. To do this we partition the model into spatially coherent groups of triangles. For each group, we build an independent BVH. In Section 3.2, we described ordering triangles by their Morton codes, which positions them coherently along a space-filling Morton curve. We can also use a Morton code to group triangles into a lattice with a user-specified resolution, such as levels 0 to 3 in Fig. 3. Fig. 6 shows that we group the triangles along the Morton curve. Then we can build a BVH for each group in parallel. We can create as many groups as the number of cores in a GPU. Each individual BVH is built using Algorithm 2.

In constructing a multi-BVH structure, we found out that more than half of the processing time is spent in computing the axis for splitting small nodes and subsequent RSS fitting. We therefore approximate the choice of axes for small nodes, as illustrated in Fig. 7. Instead of computing the axis of an RSS for a small node, we reuse the axis from the root of its subtree. We summarize the multi-BVH construction process as Algorithm 3.

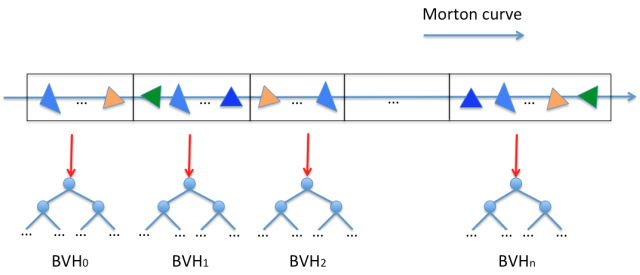


Fig. 6. Grouping triangles along a Morton curve to build a multi-BVH. Each group of triangles is associated with a node at a user-specified resolution.

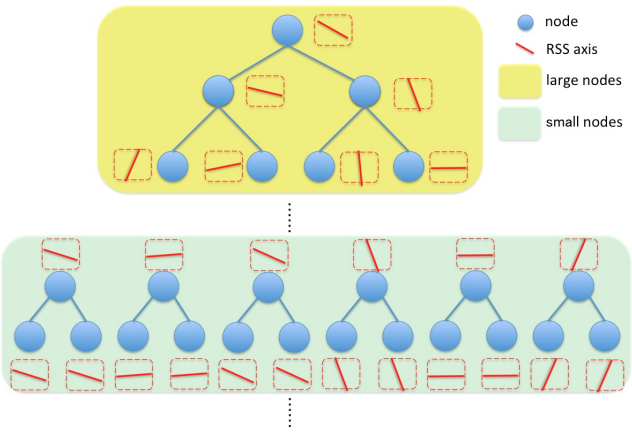


Fig. 7. Optimizing the construction of small nodes by approximating the RSS axis. We compute an RSS axis for every large node and split it along the longest axis. For small nodes, we approximate the RSS axis using the root of the subtree.

5 Parallel Distance Queries

A lot of the work on BVH-based distance computation on CPUs has focused on the choice of a good upper

Algorithm 3 Multi-BVH construction
Input: T: list of triangles which have been sorted in Alg. 1 and l: initial resolution, α: split threshold
Output: R: list of RSS tree nodes

```
for input triangle T_i in parallel do
    compute 3l-bit Morton code C_i^l
end for
for C_i^l in parallel do
    if C_i^l is different from C_{i-1}^l then
        add i to list N
    end if
end for
divide n triangles into m groups, m = |N|
activelist ← m root nodes of groups
//Large nodes construction
while activelist is not empty do
    for node i in activelist in parallel do
        do RSS fitting and node splitting in the same way as in large node construction in Alg. 2
    end for
    swap nextlist and activelist
    clear nextlist
end while
//Small nodes construction
for node i in smalllist in parallel do
    create local stack S[]
    S[] ← node i
    looptimes ← 0
    while S[] is not empty do
        localnode ← S[]
        if looptimes < 1 then
            if localnode contains more than α triangles then
                fit RSS and node splitting in the same way as in large node construction
                S[] ← created children nodes
                save split axis r_i
            else
                mark localnode as leaf node
            end if
        end if
        if looptimes ≥ 1 then
            if localnode contains more than α triangles then
                fit RSS by projecting triangles along the splitting axis r_i
                S[] ← created children nodes
            else
                mark localnode as leaf node
            end if
        end if
        looptimes ← looptimes + 1
    end while
end for
```

bound on distance [9]. Having got that, it is simple to discard any BVH subtree that has no chance of reducing that bound. An upper bound on distance will be updated, if this is possible, when the traversal of the BVH tree reaches its leaves. We will now discuss how to compute a good upper bound on distance when constructing a distance field in the form of an octree and how to cull a BVH using an upper bound.

5.1 Upper Bound Refinement

Using an RSS tree, we can compute the distance between a point at the center of an octree node and the model by traversing the RSS tree, while continually updating the upper bound. The upper bound u on the distance between a point and a model is updated as follows:

1) Add the root node t to a list l, and compute the distance between p and any triangle enclosed in the node t, and use this distance to initialize u.
2) Remove a node $\tilde{t}$ from l, and compute the distance d_N between p and the left child of node $\tilde{t}$. If d_N is smaller than u, add the left child of node $\tilde{t}$ to l. If $\tilde{t}$ is leaf, u is updated to d_N.
3) Repeat 2) for the right child of $\tilde{t}$.
4) Repeat 2) and 3) until l becomes empty.
5) The actual distance between the P and the model is equal to the upper bound u.

We perform this computation for the center of every node in the octree in parallel. Note that we only update the upper bound when we visit leaf nodes.

A tight upper bound can reduce the BVH traversal time that takes significantly. We tighten this upper bound by utilizing the space coherence in an octree. Because each sampling point is represented by an octree node. We can easily find the sampling point corresponding to the parent of a node. At coarse resolution, we use the distance between the center of the node and an arbitrary triangle from the model as an upper bound. For the center p_b of a node at fine resolution, we first find the center p_a of the parent node, and the triangle t_a that is closest to p_a, and then we use the distance between p_b and t_a as an upper bound, as illustrated in Fig. 8; here, d_B is less than d_{AB}, but d_{AB} is tighter than an upper bound computed using an arbitrary triangle such as T_R. Experiments show that this approach can produce a very tight upper bound, leading to effective culling during traversal. We have found it around 15 times faster than using an arbitrary triangle to initialize the upper bound, as shown in Section 6.2.

Note that we have recorded the closest triangle for the center of each node in the octree into a list during the distance query. Therefore, a tight upper bound of the center of a node can be initialized by retrieving the closest triangle to the center of its parent. We map a thread to perform the tight upper bound initialization and distance query for a single sample point. In each thread, we use a stack to perform BVH traversal. Each node has pointers to its two children. We track the traversal by saving the nodes which have been traversed. We push the parent node on to the stack and visit its left child sub-tree. Then we pop the parent node and visit the right child. Finally, we determine the sign of the distance field using the angle-weighted pseudo-normal [43].

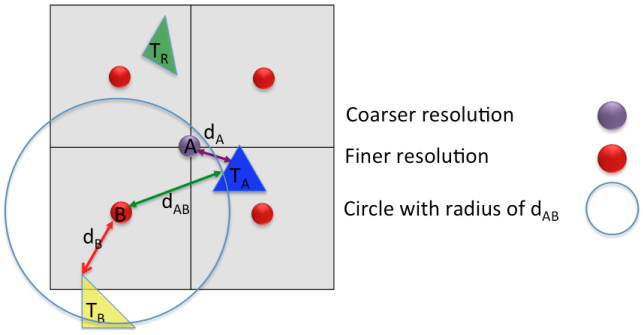


Fig. 8. Initializing an upper bound in a multi-resolution grid. T_A and T_B are the triangles closest to points A and B. T_R is any other triangle. d_A, d_B are the distances between A and T_A, B and T_B. d_{AB} is the distance between B and T_A.

5.2 Distance Query Using Multi-BVHs

To perform a distance query using multi-BVHs, we find all the candidate BVHs which are near to the query point. Then, we compute the distance between the BVHs and the sample point in parallel, as illustrated in Fig. 9.

Algorithm 4 Distance query on multi-BVHs
Input: T: list of RSS tree, P: list of sampling points, U: list of initial upper bound
Output: D: list of distances between sampling points and the model

```
1: //Space culling
2: R ← RSS of the root of the multi-BVH
3: for R_i in parallel do
4:    I_i^x ← project R_i on to the x-axis
5: end for
6: for P_i in parallel do
7:    j ← 0
8:    while j < |R| do
9:       if [P_i^x − U_i^x, P_i^x + U_i^x] ∩ I_i^x then
10:         if U_i ≤ dist(P_i, R_j) then
11:            add index of R_j in T to list l_i
12:            add index of P_i to list f_i
13:         end if
14:      end if
15:      j ← j + 1
16:   end while
17: end for
18: //Distance query
19: L = [l_0, l_1, ··· , l_m], F = [f_0, f_1, ··· , f_m], m = |P|
20: for L_i in parallel do
21:    do distance query between P_{F_i} and BVH whose root is L_i
22:    add the distance to list dist_i
23: end for
24: Dist = [dist_0, dist_1, ··· , dist_m]
25: for dist_i in parallel do
26:    D_i ← compute minimum of dist_i
27: end for
```

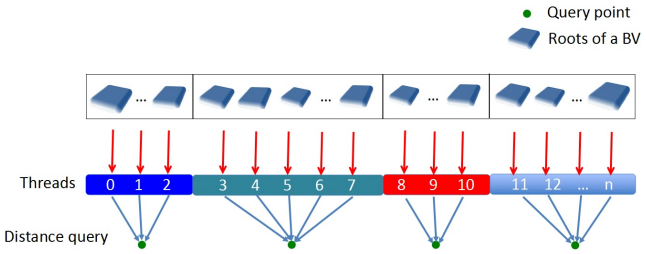


Fig. 9. Processing a distance query with a multi-BVH. We find the BVH related to each query point, and spawn as many threads as the number of BVHs. We perform the distance query for each thread and then choose the minimum distance for the query point.

5.3 Culling BVHs Using an Upper Bound

We can increase the efficiency of a distance query by reducing the number of candidate multi-BVHs that we have to consider. As illustrated in Fig. 10, if we have n BVHs, for every point we need to iterate these n BVHs for distance query. Actually, we can skip BVHs which are far away. The distance is only determined by m close BVHs, and $m \ll n$. We can use the upper bound, set as described in Section 5.1, as a culling radius for each query point. However, checking the overlap between culling sphere and the root nodes of a multi-BVH involves the expense of computing the distance between a point and a BV (i.e. an RSS). We reduce this cost by using a two-step overlap check along a single axis:

1) Project all the BVs on to some axis, such as the x-axis, and check whether the culling sphere overlaps the BVs along that axis. This only requires a logical OR operation.
2) Only if it overlaps, then check the distance between the center of the culling sphere and the BVs.

This space culling substantially reduces the number of candidate BVHs for each query point. We summarize space-based BVH culling and distance query using a multi-BVH as Algorithm 4.

6 Results and Discussion

6.1 Implementation

We have implemented our algorithm using Visual Studio C++2008 and NVIDIA CUDA 4.2 on a PC equipped with an Intel quad-core 2.66GHz CPU, a 4.0Gb main memory, and an NVIDIA Geforce GTX580 with 1.5Gb memory, running under Windows 7. We tested our algorithm on six different datasets including the Stanford Bunny, Armadillo, Hammer, Hand, Dragon and Buddha whose combinatorial complexities range from 70K to 1M triangles, as shown in Table 2. We set the subdivision threshold α for octree construction to one which can guarantee at least one point is sampled if an octree node contains more than

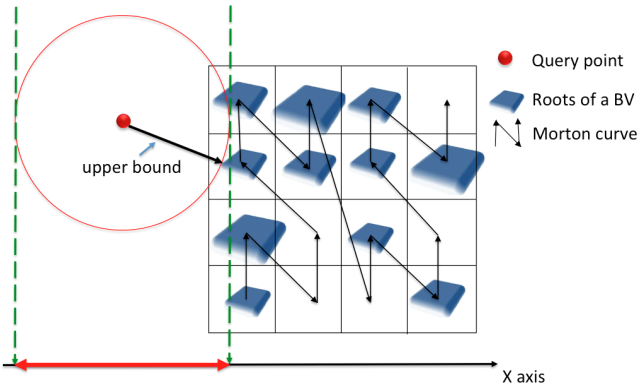


Fig. 10. Culling a multi-BVH using an upper bound. For each query point, we only keep the BVHs which have BVs at their roots that overlap with a sphere centered on the query point and having the upper bound as its radius.

one triangle. We also set the starting depth to three and maximum depth to ten. During BVH construction, small nodes are associated with less than 32 triangles. We choose 32 that corresponds to the CUDA warp size. If we choose the number less than 32, it makes some threads idle for large node construction; otherwise, it degrades the parallelism for small node construction. The number of groups that we partition the test models into for multi-BVH construction is computed by counting the number of triangles with different Morton code in Algorithm 3.

6.2 Experimental Results

Rigid Models: We applied our GPU-based parallel algorithm to six geometric models as illustrated in Figure 13, and generated adaptive distance fields with resolutions from 8^3 to 1024^3. The implementation can handle more than 1 million triangles and 4 million octree nodes. The average number of BVs for each resolution is given in Table 1. We define the average number of BVs by using the total number of BVs to divide the number of sampling points. We can see that the number of BVHs quickly falls as the resolution increases. This property improves the performance of distance query, since the number of sampling points in high resolution is much more than the number of sampling points in low resolution. We give the computation time of the distance fields, together with the number of triangles and points in the octree for six models in Table 2.

We compared our GPU approach to a CPU implementation using the PQP proximity library[1]. Our GPU implementation is 35 to 64 faster, as shown in Figure 11, and we see that the differential increased with the complexity of geometry. Usually more complexity of geometry has more triangles and generates more sampling points in high resolution. Our multi-BVH construction has a good GPUs utilization for a large number of triangles. Meanwhile, we can cull multi-BVH to a very small number in high resolution, thus our distance query is efficient and can get a good performance compared to CPUs' methods. We also compared the time taken by a distance query with a tight and upper bound derived from an arbitrary triangle. On the Stanford Bunny, the query with a tight upper bound takes 26 ms, with an arbitrary bound takes 372 ms. Note that since our tight upper bound can be directly derived from the sampling points' parent nodes in octree. No extra computation time is needed.

We also compared the performance of a single BVH with a multi-BVH. The results are shown in Figure 12. On a small model, such as the Bunny, a multi-BVH is slightly faster than a single BVH. But when there are lots of triangles, the multi-BVH outperforms a single BVH significantly. Our method has a good scalability with the triangles of geometry increased. We sample points densely close to the places where contains fine detail. And Figure 14 shows the times required to construct a single BVH and a multi-BVH in processing the adaptive distance fields shown in Figure 13. In Figure 14 (a), we can see more than a half the computation time is spent on BVH construction when using a single BVH, but this is only 30% of the computation time with a multi-BVH.

Deformable Models: We linked our distance field algorithm to a physics simulation. We inflated the Bunny (69K triangles) and dropped a cloth (4K vertices) around it. In this benchmark, the distance fields are recomputed for every simulation step. We computed responsive forces using penetration depth obtained from the distance field of bunny. We simulated the response of the cloth against the bunny. Figure 15 shows frames from the resulting simulation, which was computed at 20 frames per second.

Impact of Axis Approximation: We tested how the axis approximation technique mentioned in Section 4.3 performs in terms of BVHs construction and distance query on the six datasets. The axis approximation for small nodes speeds up BVHs construction about 40% on average compared to the conventional RSS [9]. While this makes distance query about 22% slower, it yields more than 15% overall performance improvement implying that sacrificing the quality of BVHs a bit shifts the bottleneck from construction stage to distance query stage and amortizes the overall cost.

6.3 Comparisons

We compare our algorithm against other state-of-the-art BVH and octrees techniques running on GPUs qualitatively. Making quantitative comparisons is not easy, because of differences in GPU computing platforms that rapidly changes and application scenarios.

1. http://gamma.cs.unc.edu/SSV/

Thus, we just describe significant differences against several representative previous work.

LBVH [31] builds a binary tree top-down by bucketing primitives based on Morton code. However, they fit BVs by merging the nodes' children's BVs. This method was originally designed for AABBs, and it is rather non-trivial to extend to tighter BVs. This method also has the unwanted side-effect that a parallel implementation can create chains of singleton nodes in the tree, requiring an additional post-process to walk tree and roll up these singleton chains. Indeed, [31] has been extended to an RSS hierarchy [44] by adding an initial step in which an RSS is fitted around all the triangles to determine the optimal orientation of the BV in the LBVH. Then the Morton code is computed from the positions of the transformed triangles. However, this orientation is then fixed for all the nodes in the BVH, whereas we build a tree top-down and compute the optimal axis for every node individually. Zhou et al. [32]'s method makes poor use of the GPU when building the lower levels of their tree, and their BV is an AABB, which has looser fit than RSS. A similar group of authors [34] present a GPU implementation of octrees based on Morton code. Their methods are based on point-sets and the build an octree in bottom-up. We also use Morton code, but we build an octree top-down, without the need for computing intermediate nodes requiring extra memory.

To the best of our knowledge, no other work exists to compute distance fields on GPUs adaptively and exactly.

7 Conclusions

We have proposed a fast method of computing adaptive distance fields on a GPU. Because it takes too long to build a single BVH on a GPU, we introduce the multi-BVH and axis approximation for small nodes. We use an octree data structure which is built using Morton code, and show how to compute a very tight upper bound for distance queries. By using space-based BVH culling, we quickly reduce the number of candidate BVHs for each query point.

Although our multi-BVH technique significantly outperforms a single BVH, we have observed that the efficiency of distance queries is reduced for a multi-BVH, since we need to traverse more nodes, some of which are duplicated over several BVHs. We are considering various ways of addressing this issue, such as caching BVHs on shared memory in the GPU, exploiting the similarity of traversal paths between consecutive query points, and stackless BVH traversal. Also for future work, we would like to apply our fast distance fields technique to various applications such as physically-based animation, geometric modeling and robot motion planning.

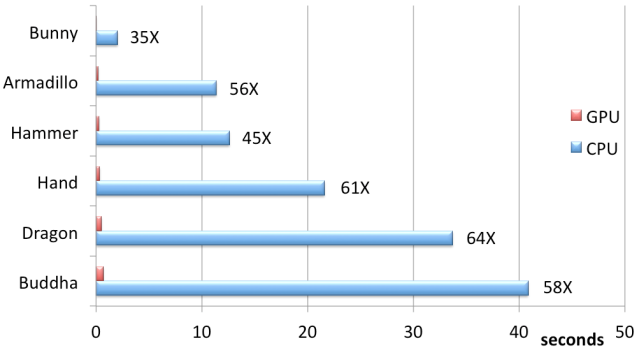


Fig. 11. Distance field computation times on a GPU and a CPU.

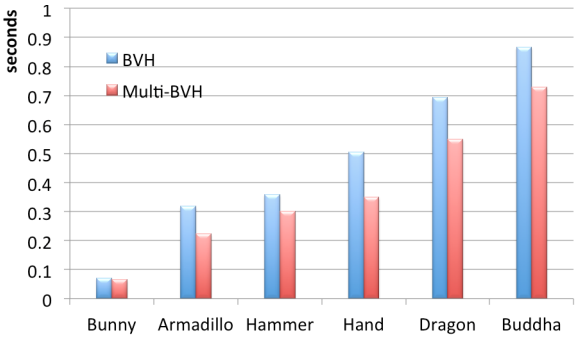


Fig. 12. Distance field computation times using a single-BVH and a multi-BVH. The times include adaptive sampling, BVH construction and distance queries.

Acknowledgments

This research was supported in part by NRF in Korea (No.2012R1A2A2A01046246, No.2012R1A2A2A06047007). We thank Takahiro Harada for useful discussions.

References

[1] T. Morvan, M. Reimers, and E. Samset, "High performance gpu-based proximity queries using distance fields," *Computer Graphics Forum*, vol. 27, no. 8, pp. 2040–2052, 2008.

[2] S. Fisher and M. C. Lin, "Deformed distance fields for simulation of nonpenetrating flexible bodies," in *Proc. Eurographics Workshop on Computer Animation and Simulation*, 2001, pp. 99–111.

[3] E. Guendelman, R. Bridson, and R. Fedkiw, "Nonconvex rigid bodies with stacking," *ACM Transactions on Graphics(Proceedings of SIGGRAPH 2003)*, vol. 22, no. 3, 2003.

[4] R. Bridson, S. Marino, and R. Fedkiw, "Simulation of clothing with folds and wrinkles," in *Proc. ACM SIGGRAPH/ Eurographics Symposium on Computer Animation*, 2003, pp. 28–36.

[5] K. E. Hoff, J. Keyser, M. C. Lin, D. Manocha, and T. Culver, "Fast computation of generalized voronoi diagrams using graphics hardware," *ACM Transactions on Graphics(Proceedings of SIGGRAPH 1999)*, pp. 277–286, 1999.

[6] N. Molino, R. Bridson, J. Teran, and R. Fedkiw, "A crystalline, red green strategy for meshing highly deformable objects with tetrahedra," in *Proc. International Meshing Roundtable 12*, 2003, pp. 103–114.

[7] J. A. Baerentzen, "Manipulation of volumetric solids, with application to sculpting," Ph.D. dissertation, IMM, Technical University of Denmark, DK, 2001.

[8] G. Varadhan and D. Manocha, "Accurate minkowski sum approximation of polyhedra models," *Graphical Models*, vol. 68, no. 4, pp. 343–355, 2006.

Resolution	Bunny	Armadillo	Hammer	Hand	Dragon	Buddha
8^3	48	44	34	27	41	47
16^3	5	5	8	6	7	7
32^3	4	4	5	4	5	5
64^3	3	3	4	3	3	4
128^3	2	3	3	3	3	3
256^3	2	2	2	2	3	3
512^3	2	2	2	2	3	3
1024^3	2	2	2	2	3	3

TABLE 1
Average number of BVs per sampling point for distance fields of different resolution.

Model	Bunny	Armadillo	Hammer	Hand	Dragon	Buddha
Triangles	69,664	345,944	433,152	654,666	871,414	1,087,716
Test Points	317,840	1,478,200	1,558,100	2,753,104	3,575,712	4,251,512
Computation time (ms)	66	224	301	350	550	729

TABLE 2
Computation times of distance fields.

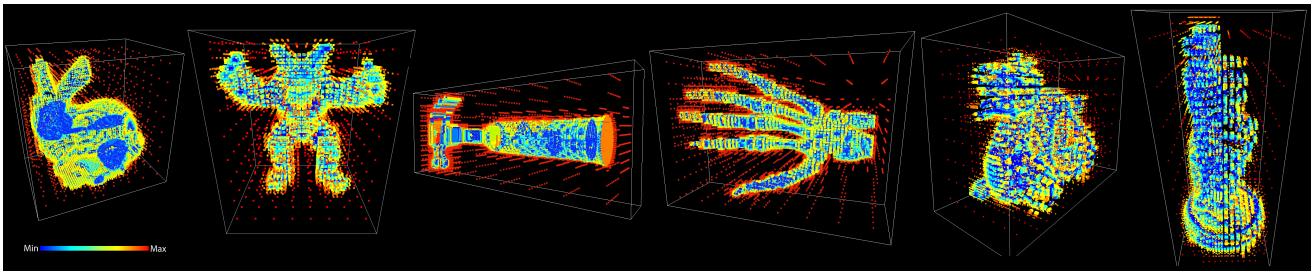
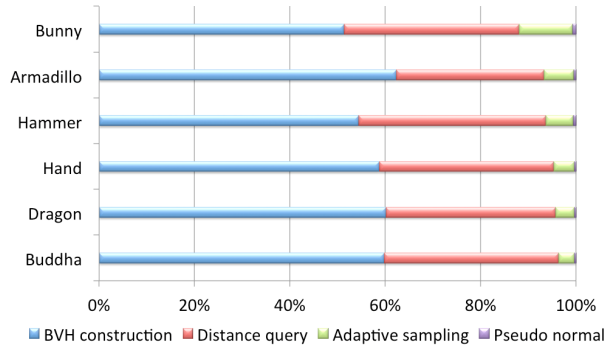
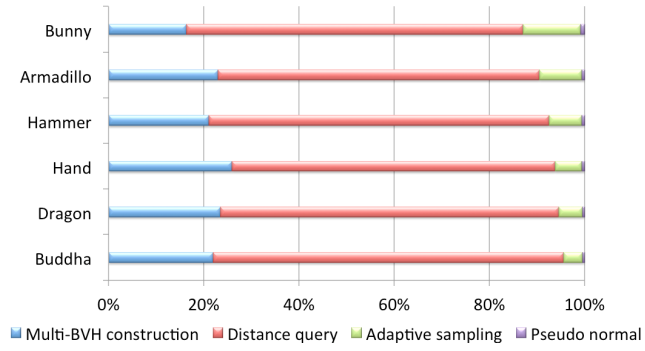


Fig. 13. Distance fields on adaptive grids for the Bunny, Armadillo, Hammer, Hand, Dragon and Buddha models. Colors from blue to red denote increasing distance.

[9] E. Larsen, S. Gottschalk, M. C. Lin, and D. Manocha, "Fast distance queries with rectangular swept sphere volumes," in *Proc. IEEE Conf. on Robotics and Automation*, 2000, pp. 3719–3726.
[10] S. Mauch, "Efficient algorithm for solving static hamilton-jacobi equation," Ph.D. dissertation, California Inst. of Techn., Perdue, CA, 2003.
[11] C. Sigg, R. Peikert, and M. Gross, "Signed distance transform using graphics hardware," in *Proc. 14th IEEE Visualization Conference(VIS'03)*, 2003, pp. 83–90.
[12] K. Erleben and H. Dohlmann, *GPU Gems 3: Signed Distance Fields Using Single-Pass GPU Scan Conversion of Tetrahedra*. Addison-Wesley, 2007.
[13] A. Sud, M. A. Otaduy, and D. Manocha, "Difi: Fast 3d distance field computation using graphics hardware," *Computer Graphics Forum*, vol. 23, no. 3, pp. 557–566, 2004.
[14] M. W. Jones, J. A. Baerentzen, and M. Sramek, "3d distance fields: a survey of techniques and applications," *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 4, pp. 581–599, 2006.
[15] A. Rosenfeld and J. L. Pfaltz, "Sequential operations in digital picture processing," *Journal of the ACM*, vol. 13, no. 4, pp. 471–494, 1966.
[16] F. Rhodes, "Discrete euclidean metrics," *Pattern Recognition Letters*, vol. 13, no. 9, pp. 623–628, 1992.
[17] K. J. Zuiderveld, A. H. J. Koning, and M. A. Viergever, "Acceleration of ray-casting using 3d distance transforms," in *Proc. Visualization in Biomedical Computing*, 1992, pp. 324–335.
[18] G. Borgefors, "On digital distance transforms in three dimensions," *Computer Vision and Image Understanding*, vol. 64, no. 3, pp. 368–376, 1996.
[19] P.-E. Danielsson, "Euclidean distance mapping," *Computer Graphics and Image Processing*, vol. 14, pp. 227–248, 1980.
[20] J. C. Mullikin, "The vector distance transform in two and three dimensions," *CVGIP: Graphical Models and Image Processing*, vol. 54, no. 6, pp. 526–535, 1992.
[21] R. Satherley and M. W. Jones, "Vector-city vector distance transform," *Computer Vision and Image Understanding*, vol. 82, no. 3, pp. 238–254, 2001.
[22] J. A. Sethian, "A fast marching level set method for monotonically advancing fronts," in *Proc. National Academy of Science*, vol. 93, 1996, pp. 1591–1595.
[23] H. K. Zhao, "Fast sweeping methods for eikonal equations," *Mathematics of Computation*, vol. 74, pp. 603–627, 2004.
[24] T. T. Cao, K. Tang, A. Mohamed, and T. S. Tan, "Parallel banding algorithm to compute exact distance transform with the gpu," in *Proc. ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 2010, pp. 83–90.
[25] B. Chang, D. Cha, and I. Ihm, "Computing local signed distance fields for large polygonal models," in *Proc. IEEE-VGTC symposium on Visualization*, 2008, pp. 799–806.
[26] K. Yin, Y. Liu, and E. Wu, "Fast computing adaptively sampled distance field on gpu," in *Proc. 19th Pacific Conf. on Computer Graphics and Applications*, 2011, pp. 25–30.
[27] T. Park, S. Lee, J. Kim, and C. H. Kim, "Cuda-based signed distance field calculation for adaptive grids," in *Proc. IEEE International Conference on Computer and Information Technology*, 2010, pp. 1202–1206.
[28] S. F. Frisken, R. N. Perry, A. P. Rockwood, and T. R. Jones, "Adaptively sampled distance fields: A general representation of shape for computer graphics," *ACM Transactions on Graphics(Proceedings of SIGGRAPH 2000)*, pp. 249–254, 2000.



(a) Single-BVH



(b) Multi-BVH

Fig. 14. Timing breakdown of processes in a single-BVH and a multi-BVH

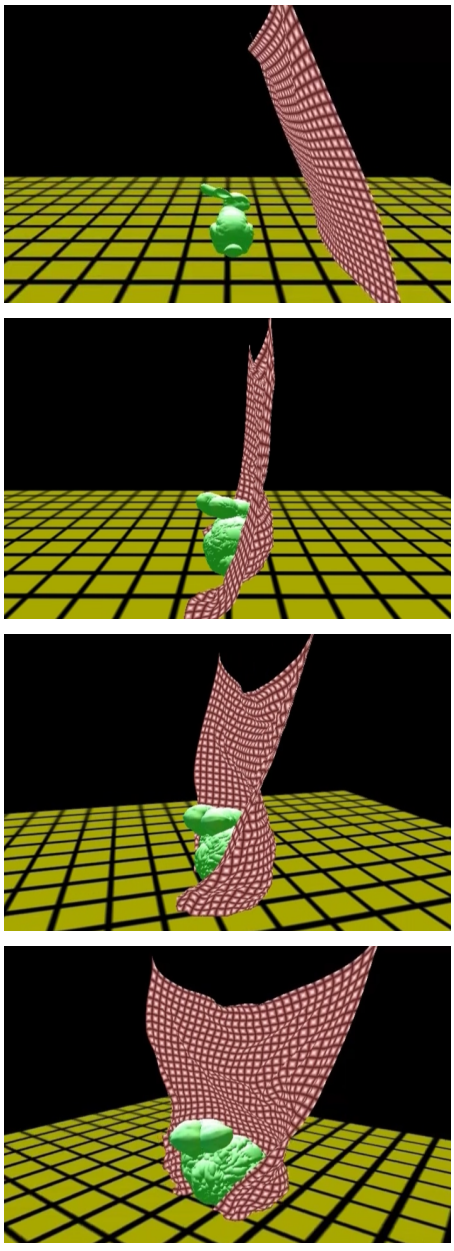
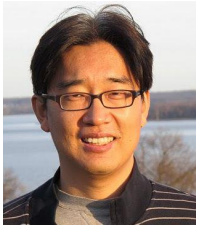


Fig. 15. Simulation of a cloth falling around the Stanford Bunny.

[29] S. Lefebvre and H. Hoppe, "Perfect spatial hashing," *ACM Transactions on Graphics(Proceedings of SIGGRAPH 2006)*, vol. 25, no. 3, 2006.
[30] T. Bastos and W. Celes, "Gpu-accelerated adaptively sampled distance fields," in *Proc. IEEE International Conference on Shape Modeling and Applications*, 2008, pp. 171–178.
[31] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, "Fast bvh construction on gpus," *Computer Graphics Forum*, vol. 28, no. 2, pp. 375–384, 2009.
[32] K. Zhou, Q. M. Hou, R. Wang, and B. N. Guo, "Real-time kd-tree construction on graphics hardware," *ACM Transactions on Graphics(Proceedings of SIGGRAPH Asia 2008)*, vol. 27, no. 5, p. 126, 2008.
[33] G. M. Morton, "A computer oriented geodetic data base; and a new technique in file sequencing," Ottawa, Canada: IBM Ltd, Tech. Rep., 1966.
[34] K. Zhou, M. M. Gong, X. Huang, and B. N. Guo, "Data-parallel octrees for surface reconstruction," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 5, pp. 669–681, 2011.
[35] G. Barequet, B. Chazelle, L. Guibas, J. Mitchell, and A. Tal, "Boxtree: A hierarchical representation of surfaces in 3d," in *Proc. Eurographics*, 1996, pp. 387–396.
[36] S. Gottschalk, M. Lin, and D. Manocha, "Obb-tree: A hierarchical structure for rapid interference detection," in *Proc. ACM Siggraph*, 1996, pp. 171–180.
[37] S. Krishnan, A. Pattekar, M. Lin, and D. Manocha, "Spherical shell: A higher order bounding volume for fast proximity queries," in *Proc. Third International Workshop on Algorithmic Foundations of Robotics*, 1998, pp. 122–136.
[38] M. Held, J. Klosowski, and J. S. B. Mitchell, "Real-time collision detection for motion simulation within complex environments," in *Proc. ACM Siggraph Visual Proceedings*, 1996, p. 151.
[39] J. Klosowski, M. Held, J. S. B. Mitchell, H. Sowizral, and K. Zikan, "Efficient collision detection using bounding volume hierarchies of k-dops," *IEEE Transactions on Visualization and Computer Graphics*, vol. 4, no. 1, pp. 21–37, 1998.
[40] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan primitives for gpu computing," in *Proc. Graphics Hardware'07*, 2007, pp. 97–106.
[41] J. Pantaleoni and D. Luebke, "Hlbvh: Hierarchical lbvh construction for real-time ray tracing of dynamic geometry," in *Proc. ACM SIGGRAPH Symposium on High Performance Graphics(HPG'10)*, 2010, pp. 87–95.
[42] K. Garanzha, J. Pantaleoni, and D. McAllister, "Simpler and faster hlbvh with work queues," in *Proc. ACM SIGGRAPH Symposium on High Performance Graphics(HPG'11)*, 2011, pp. 59–64.
[43] J. A. Baerentzen and H. Aanaes, "Signed distance computation using the angle weighted pseudonormal," *IEEE Transactions on Visualization and Computer Graphics*, vol. 11, no. 3, pp. 243–253, 2005.
[44] C. Lauterbach, Q. Mo, and D. Manocha, "gproximity: Hierarchical gpu-based operations for collision and distance queries," *Computer Graphics Forum*, vol. 29, no. 2, pp. 419–428, 2010.

Fuchang Liu is a postdoctoral research fellow of computer science and engineering at Ewha Womans University. He obtained his PhD degree and BS degrees in Computer Science from Nanjing University of Science and Technology in 2009, and 2004, respectively. His research interests include computer graphics, GPUs rendering, and collision detection.



Young J. Kim is an associate professor of computer science and engineering at Ewha Womans University. He received his PhD in computer science in 2000 from Purdue University. Before joining Ewha, he was a postdoctoral research fellow in the Department of Computer Science at the University of North Carolina at Chapel Hill. His research interests include interactive computer graphics, computer games, robotics, haptics, and geometric modeling. He has published more than 50 papers in leading conferences and journals in these fields. He also received the best paper awards at the ACM Solid Modeling Conference in 2003 and the International CAD Conference in 2008, and the best poster award at the Geometric Modeling and Processing conference in 2006. He was selected as best research faculty of Ewha in 2008, and received the outstanding research cases award from Korean research foundation in 2008.