BADF: BVH-Centric Adaptive Distance Field Computation for Deformable Objects on GPUs

Xiaorui Chen 1, Min Tang 1, cheng li 1, Dinesh Manocha 2, RuoFeng Tong 1
1 Zhejiang University 2 University of Maryland

Abstract

We present a novel method (BADF) for accelerating the construction of ADFs (adaptive distance fields) of rigid and deformable models on GPUs. Our approach is based on constructing a bounding volume hierarchy (BVH) and we use that hierarchy to generate an octree-based ADF. We exploit coherence between successive frames and sort the grid points of the octree to accelerate the computation. Our approach is applicable to rigid and deformable models. Our GPU-based algorithm is about $20-50$X faster than CPU-based algorithms. Our BADF algorithm can construct the distance fields for deformable models with 60K triangles at interactive rates on an NVIDIA GTX GeForce 1060. Moreover, we observe 3X speedup over prior GPU-based ADF algorithms.

1. Introduction

Distance fields are scalar fields that represent the minimum distance from any point in space to a shape or objects in the scene. They are typically computed for a finite number of points in 3D based on uniform or adaptive sampling. Adaptive distance fields (ADFs) are sampled according to local detail and stored in a spatial hierarchy for efficient processing. They are used in many applications including motion planning, proximity queries, surface reconstruction, physics-based simulation, etc. [13]. Compared to uniform distance fields (UDFs), ADFs offer many advantages in terms of quick query processing and lower storage overhead.

Distance fields are typically computed by repeatedly performing distance queries between given points and the boundary of an object. Each query can be accelerated using hierarchic data structures, e.g., spatial hashing or bounding volume hierarchies (BVHs). For rigid models, distance fields can be computed once during a pre-process stage. However, for deformable models, the distance fields need to be updated or recomputed on-the-fly, which can be challenging for real-time applications.

Many techniques have been proposed for faster distance field computation of rigid and deformable models that can exploit multiple cores on CPUs and GPUs. In particular, GPU parallelism can be used to perform the distance queries for multiple points in parallel [14, 18]. While UDF computation techniques use simple, regular grid based representation, ADF methods typically use an octree structure to store the distance fields. The computation of multiple data structures in terms of bounding volume hierarchies and octrees slows down the algorithms. In practice, distance field computation can be expensive and may not be fast enough for interactive applications or for the manipulation of deformable models.

Main Result: We present a novel BVH-based algorithm (BADF), to accelerate the distance field computation on GPUs. Our approach is designed to achieve higher performance on commodity processors for interactive applications. We present new techniques to build integrated data structures and exploit coherence between successive frames for faster computation. The major components of our approach include:

- **BVH-centric streamlined data structure:** We use a fast algorithm to construct the BVH and later use that BVH to compute an octree-based ADF. We first sort the model triangles based on the Morton code [21] and then compute a BVH tree using the spatial information of the Morton code. We use the node relationship within a BVH tree and the location information of the Morton code to generate the octree for ADF computation. The BVH tree bounding box is refitted to reduce the range of distances. Finally, the distance queries are performed according to the octree representation.

- **Accelerate distance queries with spatial-temporal coherence:** Our algorithm records the nearest triangle on the boundary for each query point after the distance query for the current frame. During the next frame, the distance to the nearest triangle is used as an upper bound and we perform efficient pruning for BVH-based culling. This reduces the BVH traversal overhead for distance queries.

- **Accurate distance field by sorting query points:** We avoid redundant distance query calculations of the octree's grid points by sorting all the grid points based its Morton code and processing the sorted grid points for distance queries. Compared to prior techniques, our

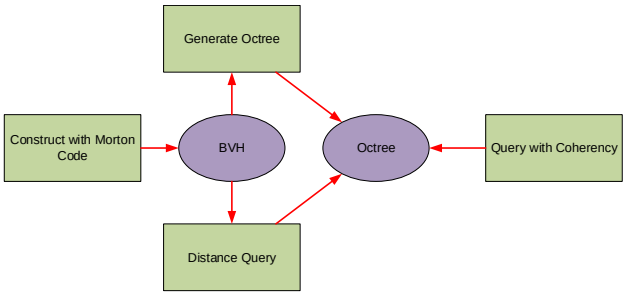

Figure 1. **Algorithm Pipeline:** Our BVH-centric algorithm first construct a BVH with Morton codes. We use this BVH for parallel distance queries and construct an octree-based ADF. We utilize coherence to accelerate the distance queries. The ADF can be computed at almost interactive rates for deformable objects.

approach can be used to compute ADF at a finer resolution with a similar runtime performance.

An overview of our approach is given in Figure 1. We implement BADF and highlight its performance on rigid and deformable models with up to 1M triangles (See Table 1). We highlight our results using a grid resolution of 128^3 on an NVIDIA GTX GeForce 1060. For models of up to 69k triangles, our approach can compute ADF at 46.6 ms per frame (i.e. 20fps). Compared to a recent GPU-based algorithm [18], we obtain up to 3X speedup.

2. Background

We give a brief overview of disance field computation algorithms and describe our notation.

2.1. UDF and ADF

A signed distance field is a volumetric representation of 3D objects. In a 3D domain $\mathcal{B} \in \mathbb{R}^3$, the signed distance field can be represented as a signed distance function, Φ:$\mathbb{R}^3 \rightarrow \mathbb{R}$, defined by the smallest Euclidean distance among all the points on the boundary $\partial\mathcal{B}$ to a given inquiry point $\boldsymbol{\xi}$, i.e.:

$$\Phi(\boldsymbol{\xi}) = s(\boldsymbol{\xi}) \inf_{\boldsymbol{\xi}^* \in \partial\mathcal{B}} \|\boldsymbol{\xi}^* - \boldsymbol{\xi}\|, \quad s(\boldsymbol{\xi}) = \begin{cases} -1, & \boldsymbol{\xi} \in \mathcal{B}, \\ 1, & \text{otherwise.} \end{cases} \tag{1}$$

Signed distance fields can be further classified as UDFs or ADFs based on whether they use uniform or adaptive sampling of a 3D space, respectively [8]. ADF algorithms subdivide the space adaptively into an octree and only store the distance to the object scene at the octree's nodes. Compared to the uniform sampling of UDFs, ADF algorithms are superior in terms of computation time and memory overhead.

ADFs have been used for various applications, including ray tracing [11], collision detection [19], surface reconstruction [4], motion planning [9], visualization [16], and geometric modeling [7].

2.2. ADF construction

The construction of ADFs is faster than the construction of UDFs. However, current ADF computation algorithms are unable to compute distance fields at interactive rates for complex deformable models. The main challenges are computing the octree and performing distance queries for each grid point of the octree.

Different techniques have been proposed to accelerate the ADF computation by exploiting the parallelism of the GPUs. Bastos et al. [1] present a method to compute ADFs on a GPU and store the octree nodes using a hash-based structure. Liu et al. [18] describe a method to compute distance fields on GPUs. They use BVH as an acceleration structure for distance queries and construct it in a top-down manner. All these methods work well on rigid models and do not offer interactive performance for deformable models.

2.3. Voronoi Diagrams and Distance Fields

Distance fields are closely related to the generalized Voronoi diagram (GVD) computation. A GVD divides the 3D space into generalized Voronoi cells based on the primitives closest to each point in the space. Hoff et al. [9] propose a fast method to compute the approximate GVD using interpolation-based polygon rasterization hardware. With the continuous development of GPU architectures and general-purpose programmability, many faster techniques have been proposed for GVD computation [5, 6, 10, 22, 26]. The GVD can be regarded as a subset of the locus of distance field critical points. We can calculate the GVD based on the distance field or we can use an expansion algorithm to calculate the distance field in 3D space based on the GVD.

2.4. Octree Generation

There is considerable work on octree computation, including bottom-up and top-down strategies.

Bottom-up Algorithms: Zhou et al. [28], Karras et al. [15], and Jeroen et al. [2] present techniques to compute octrees. Many prior methods are based on bottom-up approaches and Morton codes. Zhou et al. [28] use the resulting octree for point cloud reconstruction, so it needs to preserve the vertices, edges, and face during octree computation. For a given level, Jeroen et al. [2] mask each particle and group the results with a parallel compaction algorithm. The masking and grouping procedures are repeated for every single level until all the particles are assigned to leaf nodes or the maximal depth of the tree is reached. Karras et al. [15] construct the BVH in parallel. That algorithm detects all the edges and generates an octree based on the information of the edges. On the basis of Karras, Morrical et al. [20] further figure out the relationship between multiple objects. For those conflict cells containing multiple objects, Morri-

cal further differentiates by algorithm. Our approach improves on this method for octree computation.

Top-down Algorithms: Zhou et al. [29] use a top-down construction in another study about kd-tree construction. However, using such parallel constructs on the GPU causes most of the computation cores to be idle, especially at the root of the tree. Liu et al. [18] use a top-down approach to reduce the idle time of a processor core and instead compute multiple BVHs. In contrast to these methods, our approach is better suited to exploiting the multiple cores on the GPU.

2.5. Distance Queries

The time complexity of calculating the distance field algorithm is $O(m*n)$, where m is the number of query points, and n is the time taken for each query point. Many techniques use BVHs to reduce the query time. Other methods tend to reduce the number of query points. The ADF formulation proposed by Frisken et al. [8] reduces a larger number of sampling points by using an adaptive grid. Many other techniques have been proposed to accelerate the distance field computation.

- **Exact Distance Field Around an Object:** Sramek and Kaufmann [24] and Jones et al. [12] present a method to compute distance field shells that only calculates the distance field near the surface of the object. This method can reduce the sampling points by half, but it can not be applied everywhere in the 3D space.
- **Level Sets for Propagating Accurate Distances:** Breen [3] and Kimmel [17] propose level sets for propagating accurate distances throughout the volume. Breen et al. [3] first calculate the closest points for the narrow band and zero set, and then uses the fast marching method to compute the closest point.
- **Different Types of Query Points:** Yin et al. [27] divide the sampling points into three types: the points having triangles at the lowest level, the points on the boundary of the lowest level nodes but without triangles, and the points at the center of the non-lowest-level nodes siblings. This method can reduce a large number of sampling points, but computes an approximate distance for some of the query points.

3. BADF: Adaptive Distance Field Generation

In this section, we present our novel algorithm for generating adaptive distance fields. The overall pipeline is highlighted in Fig. 1.

3.1. BVH Generation

Our BVH generation algorithm (as shown in Fig. 2) is based on the GPU algorithms proposed by Karras [15]. Their algorithm can construct a BVH fully in parallel. We extend that method so that it can be directly used to generate the octree. After we sort all the leaf nodes that correspond one-to-one with the triangle positions in lexicographic order, the range of leaf nodes covered by each internal node can be represented as a linear range $[i, j]$. We record δ(i, j) of each node, which denotes the length of the longest common prefix between Morton code keys k_i and k_j. For any $m, n \in [i, j]$, $\delta(m, n) \geq \delta(i, j)$. We can calculate the δ of each node by comparing the Morton code keys of its leftmost and rightmost leaf nodes. This δ is used for octree generation.

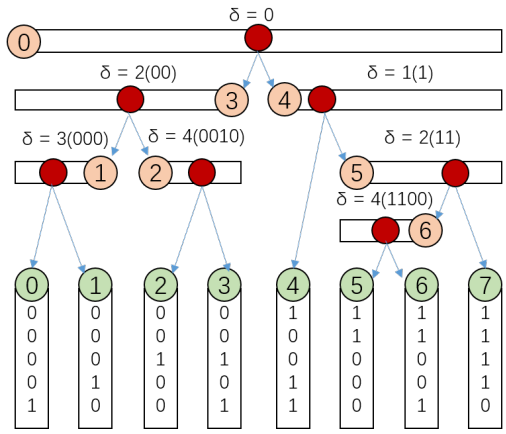

Figure 2. **BVH Generation:** We highlight our parallel BVH computation algorithm. The range of keys covered by each node is indicated by the horizontal bar. The split position, corresponding to the first bit that differs between the keys, is indicated by a red circle. We also compute a δ for each internal node, which is used for octree generation.

3.2. Octree Generation

After constructing the BVH, we use the parent-child node inheritance relationship and spatial information from that relationship to generate the octree. On the original BVH tree, if the δ of the node can be divided by 3, it means that this node is an internal node of the octree, e.g. if $\delta_i = 3$, it implies that the $node_i$ is the child of the root node in the octree. For a parent node with a prefix of length δ_{parent} and a child node with a prefix of length δ_{child}, if $\delta_{child}/3 - \delta_{parent}/3 > 0$, there is an octree node that can be generated between the child and parent nodes in the BVH tree. We first set up a maximum octree resolution controlled by the user as an input parameter. Based on the resolution, we determine the bit length of the Morton code of the octree leaf nodes. We traverse the BVH in a top-down manner, and check if we can insert the octree node on its left or right children. If so, we can record the Morton code prefix for this node. The detailed algorithm is shown in Algorithm 1.

Algorithm 1 Octree Generation: We traverse the BVH in a top-down manner, and check if we can insert the octree node on its left/right children.

```
1: procedure OCTREE GENERATION
2:                                   ▷ Traverse the BVH.
3:     lvnum_i ← number of node in level i
4:     d ← max depth of octree
5:     for N_i in each BVH Node in parallel do
6:         δ_i ← δ of N_i
7:         δ_l ← δ of N_i.LeftChild
8:         δ_r ← δ of N_i.RighttChild
9:         if δ_l/3 − δ_i/3 > 0 then
10:            if δ_l/3 = d then
11:                Add to Octree leaf Node Queue Q
12:        if δ_r/3 − δ_i/3 > 0 then
13:            if δ_r/3 = d then
14:                Add to Octree leaf Node Queue Q
15:                              ▷ Now generate the octree.
16:    for L_i in each Queue Q in parallel do
17:        for k=d:1 do
18:            if k-th ancestor of L_i has not been initialized
   then
19:                initialize k-th ancestor of L_i
```

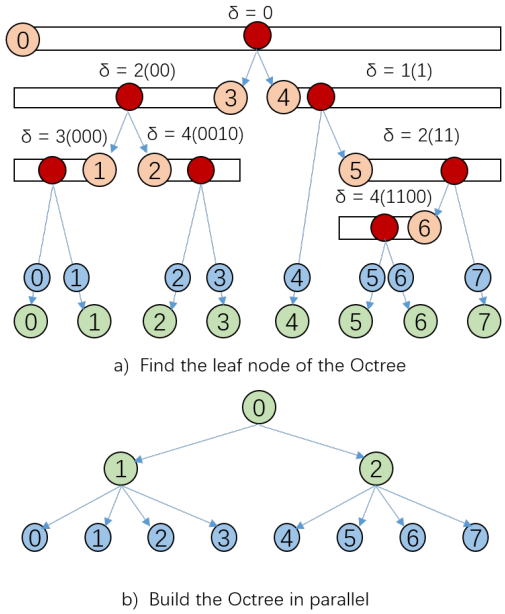

Figure 3. **Octree Generation:** We traverse the BVH in a top-down manner (a), and check if we can insert the octree node on its left/right children (b). Our combination of BVH and octree results in faster computation.

3.3. BVH Refitting

To facilitate faster distance queries, we need to refit the constructed BVH. This is necessary because the BVH is obtained by a Morton code and the bounding volume of each node corresponds to the sub-space, which may not be the tightest bounding volume of all its triangles. In the process of refitting a BVH, we use a bottom-up approach and recalculate the bounding boxes of each node in parallel in the GPU. Specifically, we first refit all the leaf nodes in parallel, then iteratively refit all the internal nodes with children nodes that have already been refitted. In practice, this parallel algorithm is quite efficient on GPU, and BVH refitting takes only approximately $0.3 - 0.8\%$ of the overall ADF construction time.

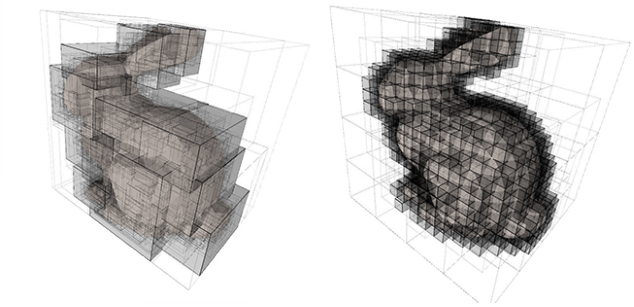

Figure 4. **BVH Refitting on GPU:** Before refitting, the bounding boxes are generated by space occupied by octree nodes. We obtain tight fitting bounding boxes surrounding triangles after refitting.

3.4. Query Reduction

After we compute an octree, we perform a distance query on each octree grid point to construct the final distance field. However, these grid points are repeated for adjacent octree nodes. For each octree node, we need to perform 8 queries for its 8 corners. Most of these queries are redundant since these corners are often shared by many octree nodes. Therefore, we assign each corner a Morton code, and use the code to ensure that there is only one queries for each corner. In practice, we can reduce the number of queries by up to $2 - 3$x.

3.5. Distance Queries

We reuse the BVH to speed up distance queries on the octree corners, which addresses a major efficiency bottleneck in our algorithm. The detailed algorithm is highlighted in Algorithm 2. As shown in Algorithm 2, we perform all the queries in parallel on the GPU (line 25 -45). Each GPU thread works on a query point C_i. In each thread, we traverse the BVH in a top-down manner, and store all the active BVH nodes into a stack S. We process all the BVH nodes in S until it becomes empty (lines 31-43). For each BVH node, if its left or right child is a leaf node, we compute the distance between the triangle in the child and C_i and update the minimum distance $minDist$. After all the BVH nodes in S are processed, we compute the $minDist$ and its corresponding triangle $minItem$ (line 45) for C_i.

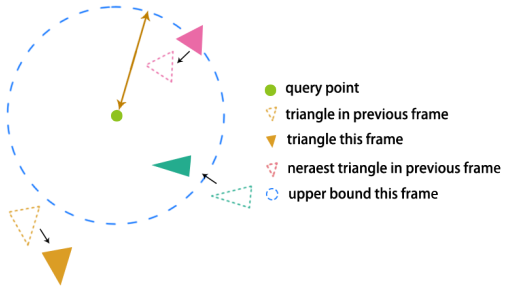

Figure 5. **Utilizing Spatial-Temporal Coherence for Distance Query:** We record the triangle with the minimum distance (the purple one) to the query point during the last frame. The distance to that triangle is used as an upper bound during the next query and reduces the overhead of tree traversal.

We also utilize spatial-temporal coherence to accelerate distance queries for deformable objects. Specifically, we record the triangle(s) with the minimum distance (the purple one in Fig. 5) to the query point in the last frame and use its distance at the current frame as the initial minimum distance for the new query. This storing of distances from the last frame and using them to compute upper bounds accelerates the computation of ADF by 1.5X for deformable models.

3.6. Sign Calculation

We use the angle weighted pseudonormal algorithm [23] to calculate the normal. This algorithm calculates the pseudo-normal by assigning the normal deflection by calculating the angular weights corresponding to the faces adjacent to the vertices. The sign of the query point is judged by the product of the pseudo-normal direction of the query point and the nearest point.

4. Results and Comparison

In this section, we describe our implementation and compare the performance with prior methods.

4.1. Implementation

We have implemented our ADF construction algorithm on an NVIDIA GeForce GTX 1060 (with 1280 cores at 1.5 GHz and 6GB memory). Our implementation uses CUDA toolkit 9.1 and Visual Studio 2013 as the underlying development environment. We use a standard PC (Windows 7 Ultimate 64 bits/Intel I7 CPU@4G Hz/8G RAM) to evaluate performance. We perform single-precision floating-point arithmetic for all the computations on the GPU. We also integrate our algorithm into a GPU-based cloth simulation system, I-Cloth [25], where the ADF computation is used for collision handling. . We have evaluated our BADF algorithm for different resolution distance fields for multiple rigid and deformable models.

Algorithm 2 Distance Query: We use the BVH to speed up distance queries on the octree corners.

▷ Perform Distance Query between a Triangle and a Point.
procedure DISTQUERYTP(Triangle(a, b, c), Point p)
    ▷ Calculate the distance from p to three corners.
    $dist_0 = distance(a, p)$
    $dist_1 = distance(b, p)$
    $dist_2 = distance(c, p)$
    ▷ Calculate the distance from p to three edges.
    $dist_3 = distance(ab, p)$
    $dist_4 = distance(bc, p)$
    $dist_5 = distance(ca, p)$
    ▷ Calculate the distance from the point to the triangle plane.
    $dist_6 = distance(\Delta abc, p)$
    ▷ Get the maximum distance
    return $min(dist_i)$ for $i \in [0, 6]$

▷ Perform Distance Query between a Bounding Box and a Point.
procedure DISTQUERYBP(Box(b_i, $i \in [0, 7]$), Point p)
    ▷ Calculate the distance from the point to the eight vertices.
    $dist_i = distance(b_i, p)$ for $i \in [0, 7]$
    ▷ Get the maximum distance.
    return $min(dist_i)$ for $i \in [0, 7]$

▷ Perform Distance Query for the Octree Corners.
procedure DISTANCE QUERY(Octree Corners C_i, BVH B)
    for query point C_i **in parallel do**
        create local stack S
        ▷ Use coherency for deformable objects
        $minItem = T_i'$
        $minDist = DistQueryTP(minItem, C_i)$
        $S.push(B.root())$
        while do$S.emty() == false$
            $curNode = S.pop()$
            if $curNode.rightChild().isleaf()$ **then**
                $dist = DistqueryTP(curNode.rightItem(), C_i)$
                if $dist < minDist$ **then**
                    $minDist \leftarrow dist$
                    $minItem \leftarrow curNode.rightItem()$
            else
                $dist \leftarrow DistQueryBT(curNode.rightBox(), C_i)$
                if $dist < minDist$ **then**
                    $S.push(curNode.rightChild())$
                ▷ Same to do with the left child.
            Processing $curNode.leftChild()$
            ▷ Get the maximum distance.
        Record $minDist$ and $minItem$ for C_i.

Table 1. **ADF Construction Time of Rigid Models on an NVIDIA GeForce GTX 1060**

Benchmarks	Triangles	Construction Time
Andy	34K	65.9ms
Bunny	70K	46.6ms
Armadillo	213K	53.6ms
Dragon	871K	88.9ms
Buddha	1.1M	101.3ms

4.2. Benchmarks

The construction time of our algorithm for different rigid models is shown in Table 1. As shown in the table, our algorithm is capable of constructing ADF for models with several hundreds of thousands triangles within tens of milliseconds on a commodity GPU.

We also use two benchmarks with deforming objects for evaluation, as shown in Fig. 7. We replace our distance field algorithm with collision detection and collision response in the I-Cloth cloth simulation system [25]. We use ADF to compute the distance between the cloth and an object and use that to add penalty forces on the cloth nodes to compute the separation forces, which are a function of the distance.

The first deformable benchmark used for evaluation corresponds to Andy: a boy performing Kungfu. It is a human body model with 33k triangles and the jacket has 22k triangles. In this scenario, the overall cloth simulation can run at approximately $7-10$ FPS. Another benchmark used is the Sphere-cloth: A ball moving forward and backward is interacting with a cloth hanging with two corners. This benchmark has a piece of rectangle cloth with 16k triangles and a sphere with 1k triangles. Our cloth simulation (with ADF computation) can run at approximately $10-13$ FPS. As shown in Fig. 8, our algorithm runs pretty fast for all the frames except the 1st frame. This highlights the benefit of our algorithm in terms of using frame-to-frame coherence.

4.3. Comparison

Prior GPU-based ADF construction algorithm: We compared our results with [18]. Since the source code implementation of that algorithm is not available, we estimated its performance on GTX 1060 based on the reported performance data and the number of GPU cores. We observe that BADF is up to 3X faster and provides an average speedup of 1.9X, as shown in Fig. 9.

Our algorithm shows good performance improvement because we use a more parallel GPU-based algorithm to construct the octrees and BVH trees. Liu and Kim [18] construct an octree and a BVH using a top-down construction method, so the time complexity of the algorithm is $O(nlogn)$. In contrast, we use the parallel construction method, so the time complexity of our ADF construction algorithm is $O(n)$. Therefore, as the number of triangles increase, we observe better performance improvement on these benchmarks using our algorithm.

Rigid Models vs. Deformable Objects: Our algorithm is better suited to constructing dynamic or time-varying distance fields (e.g., for deforming objects). Our approach can exploit the frame-to-frame coherency between the frames. The use of coherence can provide 1.5X speedup, on average.

Different ADF Resolutions: We also evaluate the performance of our ADF construction algorithm by varying specified resolutions. As show in Fig. 10, as the resolution increases, the ADF construction time slows down accordingly.

5. Conclusion and Limitations

We present a GPU-based ADF construction algorithm for rigid or deformable objects. We have also integrated it into an interactive cloth simulation system to compute proximity constraints and collision response. The main data structure of our algorithm is a tight BVH that is constructed in parallel on GPU. This BVH is used both for octree generation and perform efficient distance queries on the octree nodes. We also utilize frame-to-frame coherence to accelerate the queries. Our algorithm can compute ADF for complex deformable models at interactive rates on commodity GPUs and offers up to 3X speedup over prior methods.

Our approach has some limitations. First, the quality and resolution of ADF depends on a user specified resolution and shape of the models. Secondly, the benefit of frame-to-frame coherence are observed after the first few frames. Moreover, when we use the ADF for cloth simulation, the penalty force based method can not guarantee all the penetrations can be overcome in current time step.

There are many avenues for future research. In addition to overcoming the limitations, we would like to extend our algorithm form using a single GPU to multiple GPUs and use that for interactive distance field computation on deformable models with millions of triangles.

References

[1] T. Bastos and W. Celes. GPU-accelerated adaptively sampled distance fields. In *2008 IEEE International Conference on Shape Modeling and Applications*, pages 171–178. IEEE, 2008. 2

[2] J. Bédorf, E. Gaburov, and S. P. Zwart. A sparse octree gravitational n-body code that runs entirely on the GPU processor. *Journal of Computational Physics*, 231(7):2825–2839, 2012. 2

[3] D. E. Breen, S. Mauch, and R. T. Whitaker. 3d scan conversion of csg models into distance volumes. In

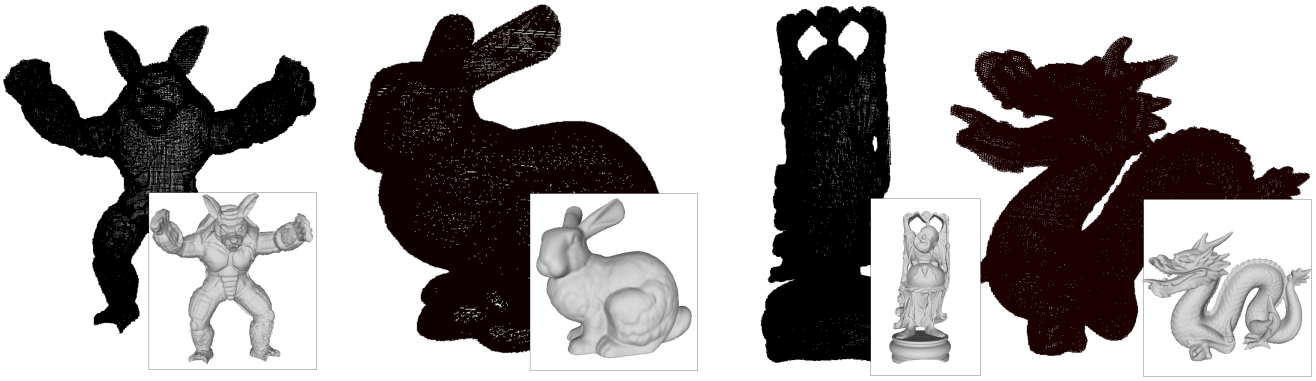

Figure 6. **ADF Construction Results of Rigid Models for the Armadillo, Bunny, Buddha and Dragon**

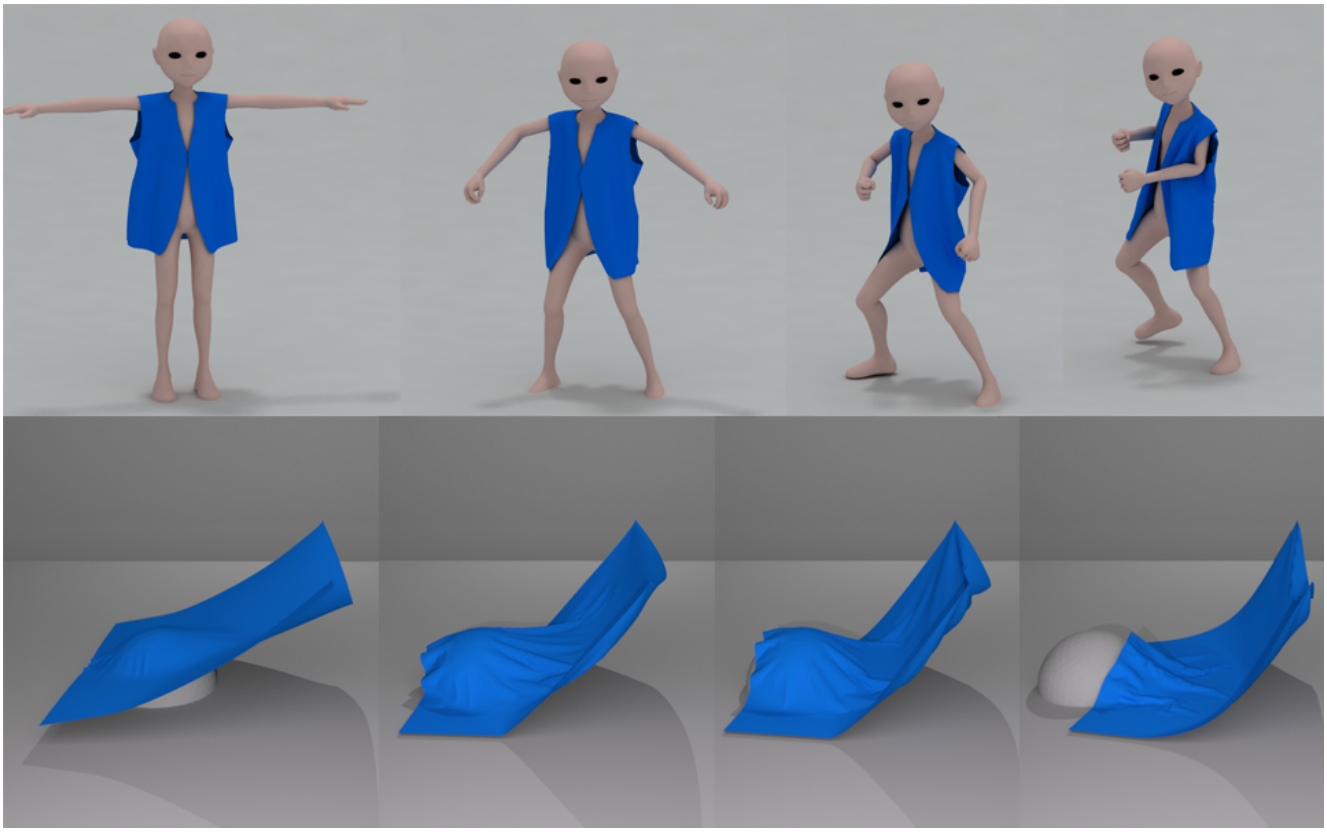

Figure 7. **Benchmarks for Deformable Models:** We use two complex benchmarks, Andy (up) and Sphere (down), and our ADF algorithm can compute distance fields at $15 - 22$FPS.

IEEE Symposium on Volume Visualization (Cat. No. 989EX300), pages 7–14. IEEE, 1998. 3

[4] F. Calakli and G. Taubin. Ssd: Smooth signed distance surface reconstruction. In *Computer Graphics Forum*, volume 30(7), pages 1993–2002. Wiley Online Library, 2011. 2

[5] T.-T. Cao, K. Tang, A. Mohamed, and T.-S. Tan. Parallel banding algorithm to compute exact distance transform with the GPU. In *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, pages 83–90. ACM, 2010. 2

[6] I. Fischer and C. Gotsman. Fast approximation of high-order voronoi diagrams and distance transforms on the GPU. *Journal of Graphics Tools*, 11(4):39–60, 2006. 2

[7] S. F. Frisken and R. N. Perry. Designing with distance fields. In *ACM SIGGRAPH 2006 Courses*, pages 60–66. ACM, 2006. 2

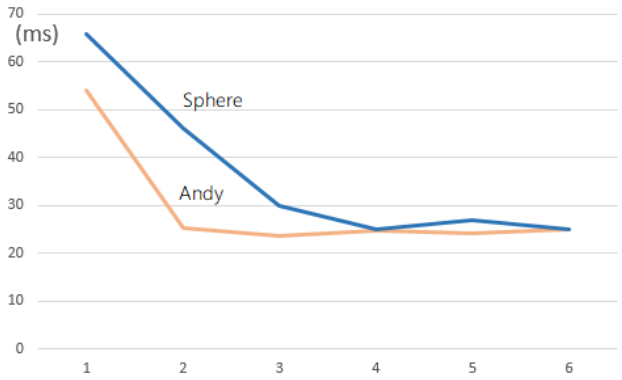

Figure 8. **ADF Computation of Deformable Models:** For the two benchmarks, Andy and Sphere, we observe significant speedup on ADF construction by using the temporal-spatial coherency between animation frames. We observe improved performance after the first few frames due to coherence.

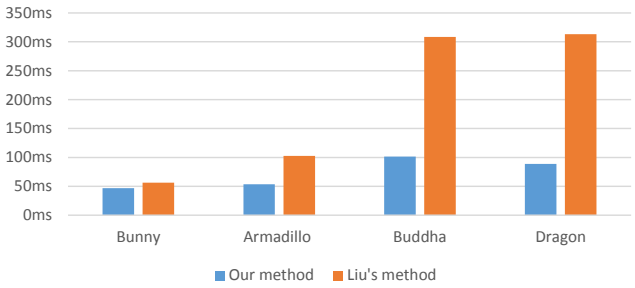

Figure 9. **Performance Comparison with [18]:** We observe up to 3X speedups among all the benchmarks. These speedups are obtained due to better parallelism, reuse of BVH, frame-to-frame coherence, and culling of redundant queries.

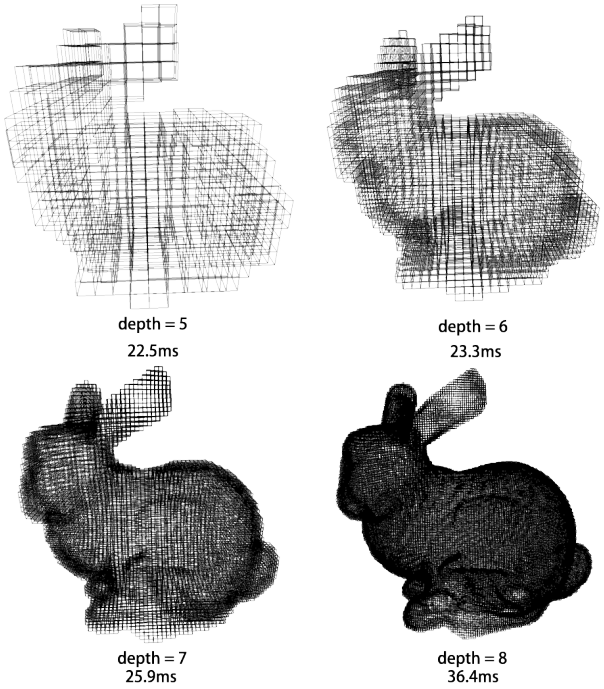

Figure 10. **Performance with Different Resolutions:** As the resolution increasing, the ADF computation time slows down accordingly. BADF can compute the distance fields at interactive rates for higher levels of the octree.

[8] 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. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 249–254. ACM Press/Addison-Wesley Publishing Co., 2000. 2, 3

[9] K. E. Hoff III, J. Keyser, M. Lin, D. Manocha, and T. Culver. Fast computation of generalized voronoi diagrams using graphics hardware. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 277–286. ACM Press/Addison-Wesley Publishing Co., 1999. 2

[10] H.-H. Hsieh and W.-K. Tai. A simple GPU-based approach for 3d voronoi diagram construction and visualization. *Simulation modelling practice and theory*, 13(8):681–692, 2005. 2

[11] O. Jamriška and V. Havran. Interactive ray tracing of distance fields. In *Central European Seminar on Computer Graphics*, volume 2, pages 1–7, 2010. 2

[12] M. W. Jones. The production of volume data from triangular meshes using voxelisation. In *Computer Graphics Forum*, volume 15(5), pages 311–318. Wiley Online Library, 1996. 3

[13] 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*, 12(4):581–599, 2006. 1

[14] M. W. Jones and M. Chen. A new approach to the construction of surfaces from contour data. In *Computer Graphics Forum*, volume 13(3), pages 75–84. Wiley Online Library, 1994. 1

[15] T. Karras. Maximizing parallelism in the construction of BVHs, octrees, and k-d trees. In *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics*, pages 33–37. Eurographics Association, 2012. 2, 3

[16] T. Kerwin, B. Hittle, H.-W. Shen, D. Stredney, and G. Wiet. Anatomical volume visualization with weighted distance fields. In *Eurographics Workshop on Visual Computing for Biomedicine*, volume 2010, page 117. NIH Public Access, 2010. 2

[17] R. Kimmel. Fast marching methods for computing distance maps and shortest paths. 1996. 3

[18] F. Liu and Y. J. Kim. Exact and adaptive signed distance fieldscomputation for rigid and deformablemodels on GPUs. *IEEE transactions on visualization and computer graphics*, 20(5):714–725, 2014. 1, 2, 3, 6, 8

[19] N. Mitchell, M. Aanjaneya, R. Setaluri, and E. Sifakis. Non-manifold level sets: A multivalued implicit surface representation with applications to self-collision processing. *ACM Transactions on Graphics (TOG)*, 34(6):247, 2015. 2

[20] N. Morrical and J. Edwards. Parallel quadtree construction on collections of objects. *Computers & Graphics*, 66:162–168, 2017. 2

[21] G. M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. 1966. 1

[22] G. Rong and T.-S. Tan. Variants of jump flooding algorithm for computing discrete voronoi diagrams. In *4th International Symposium on Voronoi Diagrams in Science and Engineering (ISVD 2007)*, pages 176–181. IEEE, 2007. 2

[23] F. Ségonne, J. Pacheco, and B. Fischl. Geometrically accurate topology-correction of cortical surfaces using nonseparating loops. *IEEE transactions on medical imaging*, 26(4):518–529, 2007. 5

[24] M. Sramek and A. E. Kaufman. Alias-free voxelization of geometric objects. *IEEE transactions on visualization and computer graphics*, 5(3):251–267, 1999. 3

[25] M. Tang, T. Wang, Z. Liu, R. Tong, and D. Manocha. I-Cloth: Incremental collision handling for GPU-based interactive cloth simulation. *ACM Transaction on Graphics (Proceedings of SIGGRAPH Asia)*, 37(6):204:1–10, November 2018. 5, 6

[26] X. Wu, X. Liang, Q. Xu, and Q. Zhao. GPU-based feature-preserving distance field computation. In *2008 International Conference on Cyberworlds*, pages 203–208. IEEE, 2008. 2

[27] K. Yin, Y. Liu, and E. Wu. Fast computing adaptively sampled distance field on GPU. 2011. 3

[28] K. Zhou, M. Gong, X. Huang, and B. Guo. Data-parallel octrees for surface reconstruction. *IEEE Transactions on Visualization and Computer Graphics*, 17(5):669–681, 2011. 2

[29] K. Zhou, Q. Hou, R. Wang, and B. Guo. Real-time kd-tree construction on graphics hardware. In *ACM Transactions on Graphics (TOG)*, volume 27(5), page 126. ACM, 2008. 3