Submission to GMP 2017

Parallel Quadtree Construction on Collections of Objects

Submission number: 142

Abstract

We present a parallel quadtree algorithm that resolves between geometric objects. The quadtree has the property that no quadtree cell intersects more than one labeled object. Previous parallel algorithms either spawn kernels hierarchically, separate points only, or make no hard guarantees of object separation. We derive the time complexity and demonstrate excellent results in practice.

1 Introduction

sec:intro

Constructing quadtrees on objects in an important task with applications to collision detection, distance fields, robot navigation, object description, and other applications. Quadtrees built on objects most often model the objects themselves, providing a space-efficient representation of arbitrarily complex objects. Our work however, centers on using quadtrees to separate, or resolve, collections of closely spaced objects. Using a quadtree, we can model the space between objects, the first step in constructing distance fields, detecting collisions, and computing the generalized Voronoi diagram. Modeling inter-object spacing is computationally straightforward when inter-object spacing is large compared to the world bounding box. Approaches typically involve a uniform grid of the space, which leads to efficient computation that often uses graphics processors.

Difficulties arise when spacing between objects is small relative to the size of the domain. An approach using a uniform grid would have excessive memory requirements in order to resolve between objects. What does "resolve" mean? In these cases, the uniformly sized grid cell must be small enough to fit between objects at every location in the domain. To our knowledge, only one algorithm [7] computes an adaptive data structure that fully resolves between objects without using unreasonable amounts of memory.

We present an algorithm that builds a quadtree on arbitrarily spaced objects in parallel. This work extends the work done by Edwards et al [7] by computing the quadtree in parallel with an algorithm targeted for the GPU. Our algorithm performs an order of magnitude faster than the previous work and will be an excellent base for later distance transform and generalized Voronoi diagram computation.

Our algorithm has three main components:

1. Construct an quadtree on object vertices using Karras' algorithm [9]
2. Detect quadtree cells that intersect more than one object, which we call "conflict cells" (contribution)
3. Subdivide conflict cells to resolve objects (contribution)

Each step is done in parallel either on vertices, quadtree cells or object facets.

2 Related work

In an early work, Lavender et al. [12] define and compute quadtrees over a set of solid models. Boada et al. [4, 5] use an adaptive approach to GVD computation, but their algorithm is restricted to GVDs with connected regions and is inefficient for polyhedral objects with many facets. Two other works are adaptive [16, 17] but are computationally expensive and are restricted to convex sites.

Two seminal works build octrees on objects in order to compute the Adaptive Distance Field (ADF) on octree vertices. Strain [15] fully resolves the quadtree everywhere on the object surface, and Frisken et al. [8] resolve the quadtree fully only in areas of small local feature size. Both approaches are designed to retain features of a single object rather than resolving between multiple objects, as is required for GVD computation. Many recent works on fast quadtree construction using the GPU are limited to point sites [3, 9, 19]. Kim and Liu's work [10] is similar, computing the quadtree on the barycenters of triangles, giving an approximation of an object-resolving quadtree. Most quadtree approaches that support surfaces [1, 6, 11, 13] are designed for efficient rendering, and actual construction of the quadtree is implemented on the CPU.

Two works [2, 14] implement Adaptive Distance Fields in parallel on quadtrees but building the quadtree itself is done sequentially. Yin et al. [18] compute the octree entirely on the GPU using a bottom-up approach by initially subdividing into a complete quadtree, resulting in memory usage that is no better than using a uniform grid. We have found no GPU quadtree construction method that is fully adaptive and can resolve between objects.

3 Algorithm

sec:algorithm

We refer to quadtree leaf cells that intersect two or more objects as "conflict cells." A necessary and sufficient condition for a quadtree to resolve objects is to have no conflict cells. Our approach to computing such a quadtree is to first build an initial quadtree using a set S of point samples. We initialize S to be the object vertices. We then detect conflict cells in parallel, followed by augmenting S with sample points such that a subsequent quadtree built on S resolves conflict cells. If S changed, then we iterate.

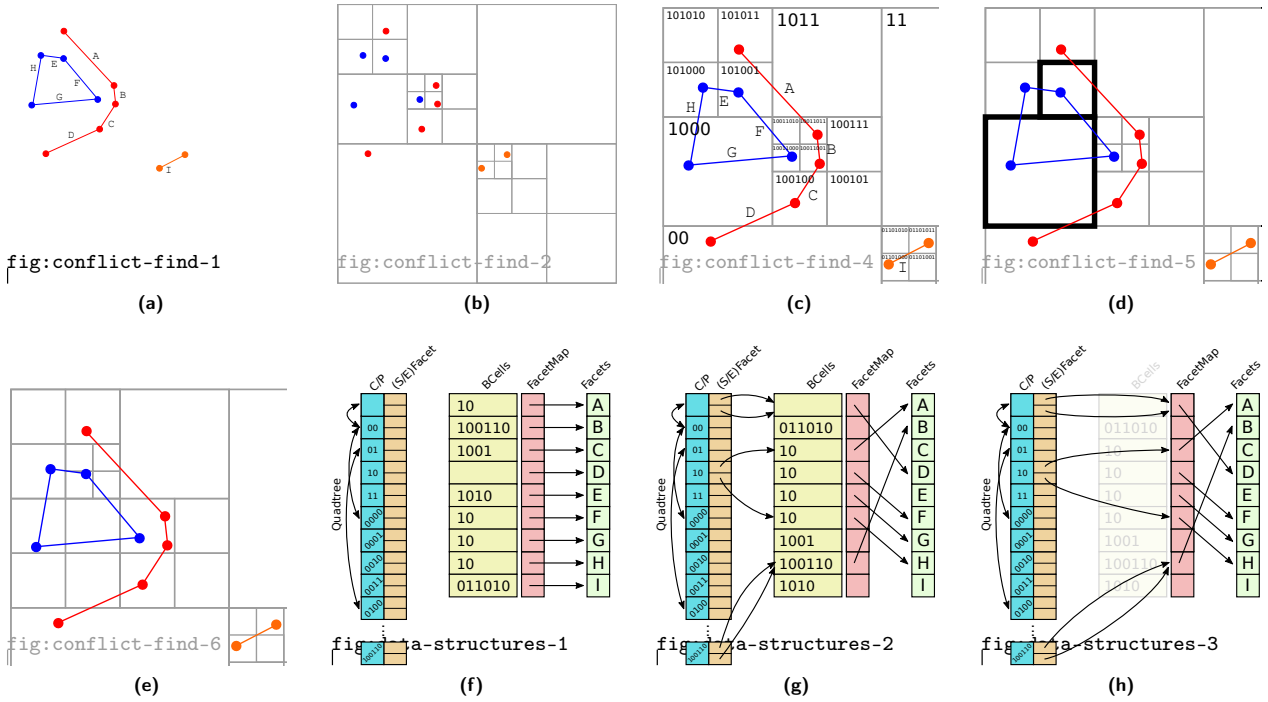

Figure 1: (a) We have three objects, blue, red, and orange with facets labeled A-I. (b) We construct an initial quadtree on the vertices using Karras' algorithm. (c) We then compute the smallest common cell (SCC) for each facet. These pairs are given in figure **??**. (d) Conflict cells, which intersect more than one object, are highlighted. (e) The new quadtree after conflict resolution. (f) The bounding cells (BCells) are stored in an array initially sorted on facet index (letters are used here for clarity). The quadtree array elements are structures which store child and parent pointers ("C/P" in the figure) and start and end facets ("SFacet" and "EFacet"). (g) We sort the BCells array using a parallel radix sort on BCell address for fast indexed access. (h) We then, in parallel on each element of BCells, store the index of the quadtree cell corresponding to the Cell_ID and, if the previous element in the BCells array does not have an identical Cell_ID, store the BCell index in the quadtree Facets field. `fig:conflict-find` `fig:scc-sort`

Most of our algorithm is independent of dimension, and so we will use terms "facet" and "octree" (which lacks a dimension-independent term). The process of finding sample points to resolve conflict cells is in 2D in this paper, so for that section we will use "line" and "quadtree".

3.1 Build initial octree

`sec:build-initial-octree`

Our first step is to build an octree on the vertices. This step gives us our first approximation to our final octree. We use Karras' algorithm [9] which sorts the Morton codes of the vertices in parallel, then constructs the binary radix tree in parallel. With the binary radix tree, the quadtree can be constructed with a single parallel call. The strength of this algorithm lies in the fact that overall performance scales linearly with the number of cores, regardless of the distribution of points. That is, even if a large number of vertices are clustered in a small area, requiring deep quadtree subdivision, only a constant number of parallel calls need be made.

3.2 Detect conflict cells

Let the "quadtree address" refer to the unique ID of a quadtree cell C found by concatenating the local addresses of its ancestors from Root to C, where the local address is the Morton code of a 2^{DIM} block of voxels. The address of the root cell is defined as the empty string. Figure 1c shows the address of each leaf cell in a quadtree.

We define a bounding cell (BCell) to be the smallest internal quadtree node which entirely contains a given facet. Given a facet defined by n endpoints $P = \{p_1, p_2, \ldots, p_n\}$, the quadtree address of the BCell is the longest common prefix of the Morton codes of the points in P. Figure 1f gives the addresses of the BCells of the facets in figure 1c.

We begin by constructing an array `BCells` and sibling array `FacetMap` (see figure 1f), which is done in parallel over all facets. Each facet f computes the longest common prefix of its vertices and stores the result in `BCells[`f`]`.

Next we sort the `BCells` and `FacetMap` arrays on the `BCell` values using a parallel radix lexicographical sort (Figure 1g). `BCells` array construction and sorting is done in parallel with the initial Karras octree construction.

Then we use the `BCells` array and octree data structure to find the conflict cells using algorithm 1. We process each leaf cell L in parallel (line 1). We set L's color to -1 (uninitialized). We then investigate each ancestor A of L (line 3). We find the ancestors using the `Parent` field in the octree data structure. Using the `SFacet` and `EFacet` fields, we find, respectively, the first and (inclusive) last of possibly multiple facets bounded by A (line **??**). We use the `FacetMap` array to find all facets bounded by bounding cell A (line 5). Any facet f for which A is the bounding cell could potentially intersect the leaf cell L. We test for intersection between f and L and store the first two facets of differing color (lines 6-15). If at the conclusion of execution L.color is equal to -2 then L is a conflict cell and must be resolved.

Algorithm 1: FIND_CONFLICT_CELLS

Input: Quadtree

```
1  for leaf cell L do in parallel
2      L.color = -1
3      foreach cell A in direct_ancestors(L) do
4          foreach i in {SFacets[A]...EFacets[A]} do
5              f := Facets[FacetMap[i]]
6              if f intersects L then
7                  if L.color == -1 then
8                      L.color = f.color
9                      L.facet[0] = f
10                 end
11                 else if L.color ≠ f.color then
12                     L.color = -2
13                     L.facet[1] = f
14                 end
15             end
16         end
17     end
18 end
```

3.3 Resolve conflict cells

To resolve a conflict cell C, we consider pairs of lines of differing labels that intersect C. Figure 2a shows two lines

$$q(t) = q = q_0 + tv \tag{1}$$
$$r(f) = r = r_0 + fw \tag{2}$$

along with a line

$$p(s) = p = p_0 + su \tag{3}$$

that bisects q and r. Our strategy will be to sample points P on $p(s)$ (figure 2d) such that an quadtree built on $S \cup P$ will completely "separate" q and r, i.e., no descendent cell of c will intersect both q and r. We do this by ensuring that P is sampled such that every box that intersects both q and r also intersects at least two points in P. Because Karras' algorithm guarantees that every leaf cell intersects at most one point, we know that no leaf cell will intersect q and r and thus no leaf cell will be a conflict cell. We will find a series of boxes such that each box's left-most intersection with $p(s)$ is a sample point meeting the above criterion.

We consider only cases where the slope of p is in the range $0 \leq m \leq 1$. All other instances can be transformed to this case using rotation and reflection. We begin by fitting the smallest box centered on a point p that intersects both q and r. We break the problem into two cases: the *opposite* case (see Figure 2b) is where $w^y > 0$, so each box intersects q and r at its top-left and bottom-right corners, respectively. The *adjacent* case (see Figure 2e) is where $w^y < 0$, so the line intersections are adjacent at the top-left and bottom-left corners of the box.

3.3.1 Finding $a(s)$ – *opposite* case

Given a point $p(s)$, we wish to find $a = a(s)$, which will give us the starting x coordinate for the next box. Consider the top-left corner of the box $q(t(s)) = q(t)$ and the bottom-right corner $r(f(s)) = r(f)$.

Because $p^x(s) = q^x(t)$,

$$t = \frac{p^x(s) - q_0^x}{v^x} = \frac{p_x^x - q_0^x + su^x}{v^x} \tag{4}$$

Because our boxes are square,

$$r(f) = r_0 + fw = q_0 + tv + a\begin{bmatrix}1\\-1\end{bmatrix} \tag{5}$$

From (5),

$$f = \frac{1}{w^y}(q_0^y + tv^y - a - r_0^y) \tag{6}$$
$$a = r_0^x + fw^x - q_0^x - tv^x \tag{7}$$

Substituting equations (4) and (6) into equation (7) and solving for a,

$$a(s) = \hat{\alpha}_o s + \hat{\beta}_o \tag{8}$$

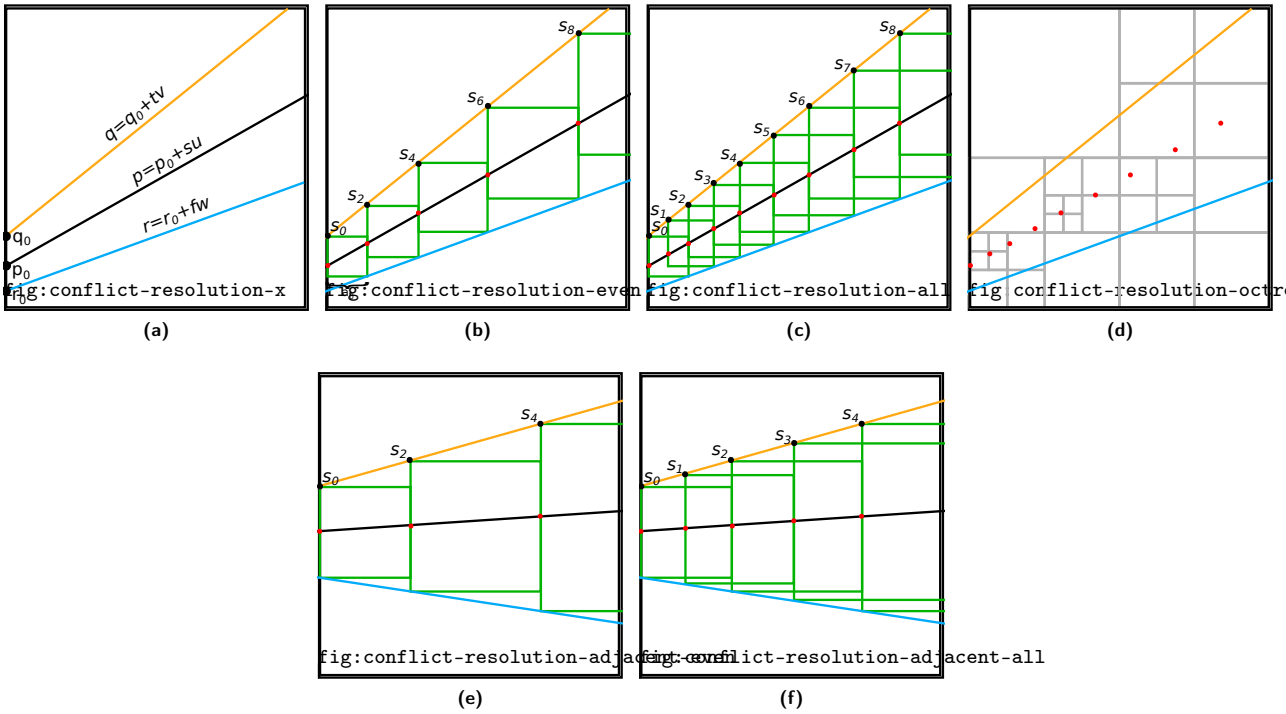

Figure 2: (a) A conflict cell with two lines from different objects. (b) Fitting boxes such that any box intersecting both lines contains at least one sample (red dots). (b) Fitting boxes such that any box intersecting both lines contains at least two samples. This ensures that an quadtree built from the samples using Karras' algorithm (panel (d)) will have no leaf cells that intersect both lines, ensuring that the new quadtree is locally free of conflict cells. `fig:conflict-resolution`

where

$$\hat{\alpha}_o = \frac{u^x |w \times v|}{v^x (w^x + w^y)} \tag{9}$$

and

$$\hat{\beta}_o = \frac{|w \times v|(p_0^x - q_0^x) + v^x(|r_0 \times w| + |w \times q_0|)}{v^x (w^x + w^y)} \tag{10}$$

3.3.2 Finding $a(s)$ – *adjacent* case

Consider the top-left corner of the box $q(t(s)) = q(t)$ and the bottom-left corner $r(f(s)) = r(f)$. $r(f)$ is now defined as

$$r(f) = r_0 + fw = q_0 + tv + a \begin{bmatrix} 0 \\ -1 \end{bmatrix} \tag{11}$$ `eqn:ra`

Equations (4) and (6) remain the same while (7) becomes

$$0 = r_0^x + fw^x - q_0^x - tv^x \tag{12}$$ `eqn:a1a`

Substituting equations (4) and (6) into equation (12) and solving for a,

$$a(s) = \hat{\alpha}_a s + \hat{\beta}_a \tag{13}$$ `eqn:aa`

where

$$\hat{\alpha}_a = \frac{u^x}{v^x w^x} \tag{14}$$

and

$$\hat{\beta}_a = \frac{w^x(p_0^x - q_0^x) + |w \times q_0| + |r_0 \times w|}{w^x} \tag{15}$$

3.3.3 Sampling

In both the *opposite* and the *adjacent* cases, $a(s)$ is of the form $a(s) = \hat{\alpha}s + \hat{\beta}$. We now use $a(s)$ to construct a sequence of values $S = \{s_0, s_1, s_2, \ldots, s_n\}$ that meet our sampling criterion. We first construct the even samples (see Figures 2b and 2e). Given a starting point $p(s_0)$,

$$p^x(s_{i+2}) = p^x(s_i) + a(s_i) \tag{16}$$

Substituting in equations (3) and (8)/(13),

$$p_0^x + s_{i+2}u^x = p_0^x + s_i + \hat{\alpha}s_i + \hat{\beta} \tag{17}$$

Solving for s_{i+2} gives the recurrence relation

$$s_{i+2} = \alpha s_i + \beta \tag{18}$$ eqn:recurrence

where

$$\alpha = 1 + \frac{\hat{\alpha}}{u^x} \tag{19}$$

and

$$\beta = \frac{\hat{\beta}}{u^x} \tag{20}$$

Constructing the odd samples is identical, except that we start at

$$s_1 = \left(1 + \frac{\hat{\alpha}}{2u^x}\right) s_0 + \frac{\hat{\beta}}{2} \tag{21}$$

which is the point in the center of the first box in the x-dimension.

We solve the recurrence relation (18) using the characteristic polynomial to yield

$$s_i = k_1 + k_2 \alpha^i \tag{22}$$ eqn:sample

where the k variables are split into those for even values of i and those for odd values of i, and are given as

$$k_1^{even} = \frac{\beta}{1-\alpha} \tag{23}$$

$$k_1^{odd} = \frac{\beta}{1-\alpha} \tag{24}$$

$$k_2^{even} = \frac{\alpha s_0 + \beta - s_0}{\alpha - 1} \tag{25}$$

$$k_2^{odd} = \frac{\alpha s_1 + \beta - s_1}{\alpha - 1} \tag{26}$$

The last step to formulating P for parallel computation is to determine how many samples we will need. Let $p(s_{exit})$ be the point at which the line p exits the cell.

$$k_1 + k_2 \alpha^i < s_{exit} \tag{27}$$

results in

$$i < \log_\alpha \frac{s_{exit} - k_1}{k_2} \tag{28}$$ eqn:num-samples

In short, we first use equation (28) compute the number of samples required to resolve the cell. The second step is to compute the samples themselves, which is done in parallel over all new samples to be computed, using equation (22).

3.3.4 Iterate

Because conflict cell resolution only considers two facets at a time, we may have to iterate multiple times if more than two facets intersect a given cell. If new sample points were found then we add them to the current set S of sample points and return to building the octree from points (section 3.1). We finish when no new sample points are found.

3.4 Time complexity analysis

Let $M = |F|$ and $N = |V|$, where F are the object facets and V are the object vertices. Let D be the depth of the octree. In this analysis we assume sufficient parallel units to maximize parallelization.

1. Build octree using Karras' algorithm [9] - $O(D)$.
2. Detect conflict cells
   (a) Build BCells array - $O(D)$. Building of the array runs in parallel for each facet f. The facet looks at each vertex (we assume simplices with a constant number of dimensions), computes Morton codes and finds the longest common prefix among vertices. This requires looking at each bit, of which there are $O(D)$.
   (b) Sort BCells array - $O(\log M)$. The array has M elements, and we use a parallel radix sort with log complexity.
   (c) Index BCells with octree data structure - $O(D)$. This runs in parallel on leaf cell IDs and each kernel requires a search of the octree for a given cell ID, taking at most D steps.
   (d) Find facets that intersect each leaf cell - Worst case $O(M + D)$, average case $O(D)$. In unusual datasets, a single leaf cell will be intersected by $O(M)$ facets. On average, however, leaf cells intersect a small number of facets, and thus this step is dominated by the depth D of the octree due to visiting each ancestor of the leaf cell.
3. Resolve conflict cells
   (a) Compute new sample points - $O(1)$. The first step computes, in parallel over conflict cells, the number of samples required to resolve the cell using equation (28). The second step is to compute the samples themselves, which is done in parallel over all new samples to be computed, using equation (22).
   (b) $S \leftarrow S \cup S'$ - $O(1)$.

dataset	objects	object Δs ($\times 10^3$)	quadtree depth	quadtree cells ($\times 10^3$)	quadtree memory (Mb)	GVD (sec)	GVD Δs ($\times 10^3$)
Fig. ??	3	7	8	54	3	0.9	83
Fig. ??	4	15	12	146	9	3.9	232
Fig. ??	470	5	24	158	8	2.0	151
Fig. ??	448	4015	8	2716	151	195	8100
Fig. ??	35	1500	8	496	70	19	2700

Table 1: Table of quadtree/GVD computation statistics and timings on datasets that are unmanageable using other methods. Columns are: *objects* - the number of objects in the dataset; *object* Δs - the number of line segments (2D) or triangles (3D) of all objects in the dataset; *quadtree depth* - required quadtree depth in order to resolve objects; *quadtree cells* - total number of leaf quadtree cells; *quadtree memory* - amount of memory used by the quadtree; *GVD (sec)* - seconds to perform all steps of GVD computation; *GVD* Δs - number of line segments (2D) or triangles (3D) in the GVD. `tab:timings`

4. Iterate - $O(Q)$ iterations. In the worst case, all facets intersect a single cell, requiring potentially $Q = O(M^2)$ iterations. In our testing, we have never seen Q exceed fill in with some value here.

The final complexity of each iteration is $O(M + D)$ worst case and $O(\log M + D)$ average case. In practice we must fix the depth of the octree to a constant value in order to use a predetermined integer size for the Morton codes, which brings the average case complexity to $O(\log M)$. Taking iteration into account, the final complexity is $(Q \log M)$ average case.

4 Results and applications

Our implementation[1] of the algorithm supports polygons and triangulated objects, and our wavefront initialization step is implemented on the GPU using OpenCL. All tests were run on a MacBook Pro laptop with a dual-core 2.9 GHz processor, 8 GB memory, and Intel HD 4000 graphics card. Figure **??** shows our implementation of the GVD computation pipeline, and Figure **??** shows the computed GVD on a more challenging dataset. We compare our method with other work and then show examples in three application settings: path planning, proximity queries, and exploded diagrams.

4.1 Comparison to other methods

5 Conclusions

References

[1] J. Baert, A. Lagae, and P. Dutré. Out-of-core construction of sparse voxel octrees. In *Proceedings of the 5th High-Performance Graphics Conference*, pages 27–32. ACM, 2013.

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

[3] J. Bédorf, E. Gaburov, and S. Portegies Zwart. A sparse octree gravitational< i> n</i>-body code that runs entirely on the gpu processor. *Journal of Computational Physics*, 231(7):2825–2839, 2012.

[4] I. Boada, N. Coll, N. Madern, and J. Antoni Sellares. Approximations of 2d and 3d generalized voronoi diagrams. *International Journal of Computer Mathematics*, 85(7):1003–1022, 2008.

[5] I. Boada, N. Coll, and J. Sellares. The voronoi-quadtree: construction and visualization. *Eurographics 2002 Short Presentation*, pages 349–355, 2002.

[6] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann. Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pages 15–22. ACM, 2009.

[7] J. Edwards, E. Daniel, V. Pascucci, and C. Bajaj. Approximating the generalized voronoi diagram of closely spaced objects. *Computer Graphics Forum*, 34(2):299–309, 2015.

[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.

[9] 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.

[10] Y. J. Kim and F. Liu. Exact and adaptive signed distance fieldscomputation for rigid and deformablemodels on gpus. *IEEE Transactions on Visualization and Computer Graphics*, 20(5):714–725, 2014.

[11] S. Laine and T. Karras. Efficient sparse voxel octrees. *Visualization and Computer Graphics, IEEE Transactions on*, 17(8):1048–1059, 2011.

[12] D. Lavender, A. Bowyer, J. Davenport, A. Wallis, and J. Woodwark. Voronoi diagrams of set-theoretic solid models. *Computer Graphics and Applications, IEEE*, 12(5):69–77, 1992.

[13] S. Lefebvre and H. Hoppe. Compressed random-access trees for spatially coherent data. In *Proceedings of the 18th Eurographics conference on Rendering Techniques*, pages 339–349. Eurographics Association, 2007.

[14] T. Park, S.-H. Lee, J.-H. Kim, and C.-H. Kim. Cuda-based signed distance field calculation for adaptive grids. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pages 1202–1206. IEEE, 2010.

[15] J. Strain. Fast tree-based redistancing for level set computations. *Journal of Computational Physics*, 152(2):664–686, 1999.

[1] Source code is available at http://cedmav.org/research/project/33-gvds.html.

[16] M. Teichmann and S. Teller. Polygonal approximation of voronoi diagrams of a set of triangles in three dimensions. In *Tech Rep 766, Lab of Comp. Sci., MIT*, 1997.

[17] J. Vleugels and M. Overmars. Approximating voronoi diagrams of convex sites in any dimension. *International Journal of Computational Geometry & Applications*, 8(02):201–221, 1998.

[18] K. Yin, Y. Liu, and E. Wu. Fast computing adaptively sampled distance field on gpu. In *Pacific Graphics Short Papers*, pages 25–30. The Eurographics Association, 2011.

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