Parallel quadtree construction on collections of objects

Nathan Morrical, John Edwards*

Idaho State University, Pocatello, ID, USA

Abstract

We present a parallel quadtree algorithm that resolves between geometric objects, modeling space between objects rather than the objects themselves. Our quadtree has the property that no cell intersects more than one labeled object. A popular technique for discretizing space is to impose a uniform grid – an approach that is easily parallelizable but often fails because object separation isn't known a priori or because the number of cells required to resolve closely spaced objects exceeds available memory. Previous parallel algorithms that are spatially adaptive, i.e., discretizing finely only where needed, either separate points only or make no guarantees of object separation. Our 2D algorithm is the first to construct an object-resolving discretization that is hierarchical (saving memory) yet with a fully parallel approach (saving time). We describe our algorithm, demonstrate experimental results, and discuss extension to 3D. Our results show significant improvement over the current state of the art.

1. Introduction

Constructing quadtrees on objects is an important task with applications in collision detection, distance fields, robot navigation, shape modeling, object description, and other applications. Quadtrees built on objects most often model the objects themselves, providing a space-efficient representation of arbitrarily complex objects. However, our work centers on using quadtrees to separate, or resolve, collections of closely spaced objects, i.e., to construct a discretization such that no cell intersects more than one object. Such quadtrees can be thought of as modeling the space between objects.

Modeling inter-object spacing is computationally straightforward when the 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 objects are close together relative to the size of the domain. An approach using a uniform grid would have excessive memory requirements in order to resolve between objects because the uniformly sized grid cell must be small enough to fit between objects at every location in the domain. Thus, an adaptive approach must be used for datasets of closely spaced objects.

To our knowledge, only one algorithm [1] computes an adaptive data structure that fully resolves between objects without using unreasonable amounts of memory, but it does so in serial, with expected performance liabilities. A naive approach to parallelizing quadtree computation would be to assign all available compute units according to a coarse grid, then run the serial algorithm on each compute unit. While simple, there is potential for serious load imbalancing if the close object spacings are not uniformly distributed.

This paper extends the work done by Edwards et al. [1] by computing the quadtree in parallel with an algorithm that is adaptive and independent of object distribution. Our algorithm, which is targeted for the GPU, performs an order of magnitude faster than the previous work and will be an important base for later distance transform and generalized Voronoi diagram computation.

Our algorithm has three main components:

1. Construct a quadtree on object vertices using the Karras algorithm [2]
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 object vertices, object facets, or quadtree cells.

*Corresponding author

Email addresses: bitinat2@isu.edu (Nathan Morrical), edwajohn@isu.edu (John Edwards)

Modeling object separation is of some use in 2D (e.g. path planning), but it is a very important problem in many 3D applications. Hierarchically subdividing space between faceted objects in a principled parallel way is complex, and this paper lays the groundwork for our continuing efforts in 3D.

2. Related work

Serial In an early work, Lavender et al. [3] define and compute octrees over a set of solid models. Two seminal works build octrees on objects in order to compute the Adaptive Distance Field (ADF) on octree vertices. Strain [4] fully resolves the quadtree everywhere on the object surface, and Frisken et al. [5] 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. Boada et al. [6, 7] 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 [8, 9] but are computationally expensive and are restricted to convex sites.

Parallel Many recent works on fast quadtree construction using the GPU are limited either to point sites [10, 2, 11] or to sites that don't overlap octree cells [12]. Most quadtree approaches that support surfaces are designed for efficient rendering and not inter-object resolution. Most of these approaches construct the quadtree on the CPU [13, 14, 15, 16], although Choi et al. [17] succeed in constructing k-D trees in parallel. Two works [18, 19] implement Adaptive Distance Fields in parallel on quadtrees but building the quadtree itself is done sequentially. Yin et al. [20] compute the octree entirely on the GPU using a bottom-up approach by initially subdividing into a complete octree, resulting in memory usage that is no better than using a uniform grid. Crassin and Green [21] build the octree top-down by performing subdivisions at each level. The most similar work to what we do here is Kim and Liu's method [22], which computes the quadtree on the barycenters of triangles, giving an approximation of our quadtree, but without fully resolving between objects. We are unaware of any GPU quadtree construction methods that are fully adaptive and resolve between objects.

3. 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 in two stages. We first build an initial quadtree, called the "vertex quadtree," using a set S of point samples. We initialize S to be the object vertices. The second stage is to 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 (see section 3.4.4) which is necessary only if a conflict cell has multiple intersecting objects. The number of iterations is minimized by starting from an initial vertex quadtree. This two-stage approach enables us to resolve between objects fully in parallel regardless of object spacing, i.e., we do not iterate through levels of the quadtree, subdividing as we go.

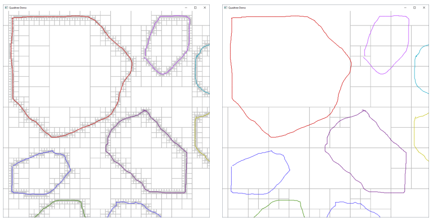
Each step of our algorithm, with the exception of resolving conflict cells, is independent of dimension and can be used for 3D octree applications. But since point sampling for conflict cell resolution is 2D we will use the term quadtree throught the algorithm description for consistency. Our algorithm assumes the objects are faceted where the facets are simplices.

3.1. Build initial quadtree

Our first step is to build a quadtree on the given set of vertices. We use the Karras algorithm [2] which begins by placing the given vertices on a Z-Order curve by computing each vertex's cooresponding Morton code in parallel. Next, Karras sorts the converted points by using a parallel radix sorter, which has a linear execution time. Our implementation uses the efficient four-way parallel radix sorter described by Ha et al. [23]. Once the Morton codes are sorted, the Z-Order curve can be exploited to construct a binary radix tree in a parallel bottom up manner by identifying longest common Morton code prefixes between neighboring points. This resultant binary radix tree can be analyzed in parallel to identify the size and structure of the required vertex quadtree. The strength of this approach 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. Pruning the quadtree

During Karras' initial binary radix tree (BRT) construction, we can prune the BRT to simplify the resultant quadtree. This in turn simplifies the work complexity of conflict cell detection and reduces our overall memory



(a) (b)

Figure 1: (a) The initial quadtree built on the object vertices, in which no quadtree cell contains more than one vertex, can be far more complex than needed to resolve between objects. (b) After pruning the quadtree. Quadtree cells can contain multiple vertices as long as they all have the same label.

footprint. Assume we have a numeric vertex labeling such that each vertex is labeled to match the object it belongs to. The original BRT provided by Karras' algorithm is used to generate a quadtree which separates vertices regardless of their label. Since our objective is to resolve between objects of different labels, we can proactively prune Karras' initial BRT, and subsequently the initial quadtree (see figure 1) by allowing the generated quadtree leaves to contain multiple vertices as long as those vertices have the same label.

To prune the initial BRT efficiently, we label each BRT node C using the following criterion: if C is a leaf node that separates two vertices with identical labels, label C to match the label of the vertices being separated. If C is a leaf node that separates two vertices having mismatched colors, label C as "required". Lastly, if C is an internal node, i.e., it has children, mark it as "unknown". This initial step can be done immediately after the Karras BRT construction without the need to invoke an additional kernel.

We then propagate the BRT labels up the tree in parallel, marking "unknown" nodes as "required" when the labels of the current node's two child nodes don't match. Labels are applied using an atomic compare and swap, and threads terminate if the current ancestor's label was previously "unknown". Finally, we generate quadtree nodes from only the required internal binary radix tree nodes.

3.3. *Identifying conflict cells*

After the pruning in 3.2, the Karras quadtree separates all differently labeled vertices in the dataset. Our goal is to separate differently labeled facets. We first need to identify what quadtree cells require further subdivision. We call these cells "conflict cells" (see figure 2c). To efficiently identify conflict cells, we take advantage of the space filling Morton curve and the existing quadtree hierarchy to reduce the combinatoric complexity of intersection detection between facets and quadtree cells. We use the following approach.

3.3.1. *Initializing conflict cell detection*

Before we detect conflict cells we create a mapping from each quadtree cell c to all facets bounded by c, a technique similar in spirit to the fragment emission and sorting done by Pantaleoni [24]. We first find the "bounding cell" c_f for a facet f, where the bounding cell is the smallest quadtree cell that completely contains f. For each facet f in parallel we determine lcp_f, which is the longest common prefix of the Morton codes of the vertices of f. Then, to find the bounding cell c_f, we iterate in parallel over each facet f and use lcp_f to direct a search through the quadtree. Let F be the number of facets. We allocate two parallel arrays of size F, `BCells` and `FacetMap`. As each c_f is found, the index of c_f is stored in the `BCells` array at index equal to the thread id. At the same time, we store the index of each facet in the `FacetMap`. Initially, FacetMap[i] = i (see figure 3a).

Next, we perform a parallel radix sort on the parallel arrays (`BCells` and `FacetMap`) using the bounding cell addresses as the sort key (figure 3b). Finally, since each quadtree cell may bound multiple facets, we compute a range for each quadtree cell by comparing neighboring quadtree indices in the mapping in parallel (the `(F/L)Facet` array in figure 3b). We now have a mapping from a quadtree cell c to facets bounded by c (figure 3c).

3.3.2. *Conflict Cell Detection*

To identify conflicts, we begin by processing each leaf cell L in parallel using Algorithm 1. First, we set L's color to -1, meaning it is unknown whether L is a conflict cell or not. Then, we traverse each direct ancestor A of L using a *Parent* field stored in the quadtree data structure (line 3). For each ancestor traversed, we iterate over the facets bounded by A by using the quadtree cell to facet mapping computed in 3.3.1 (line 4).

For each facet f discovered this way, we test for intersection between f and L. If f intersects L and L's color is -1, we copy f's color to L. Otherwise if f intersects L and L's color does not match f's color, we set L's color to -2, indicating that L is a conflict cell that must be resolved. Note that in Algorithm 1, no atomic operations are required.

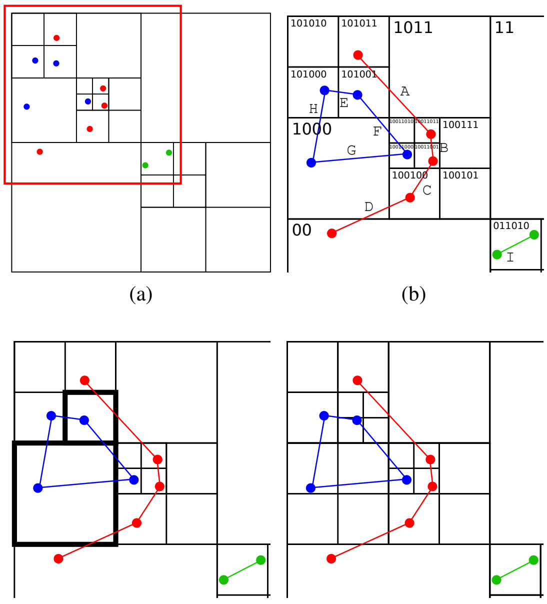


Figure 2: We have three objects, blue, red, and green with facets labeled A-I. (a) Initial pruned vertex quadtree. (b) Zoomed-in to the region outlined by red in (a) and showing the boundary cell (BCell) computation for each facet. (c) Conflict cells, which intersect more than one object, are highlighted. (d) The new quadtree after conflict resolution.

Algorithm 1: FIND_CONFLICT_CELLS

Input: Quadtree

for *leaf cell L* **do in parallel**
    L.color = -1
    foreach *cell A in direct_ancestors(L)* **do**
        foreach *i in* $\{FFacet[A] \dots LFacet[A]\}$ **do**
            f := Facets[FacetMap[i]]
            if *f intersects L* **then**
                if *L.color == -1* **then**
                    L.color = f.color
                    L.facet[0] = f
                end
                else if *L.color* $\neq$ *f.color* **then**
                    L.color = -2
                    L.facet[1] = f
                end
            end
        end
    end
end

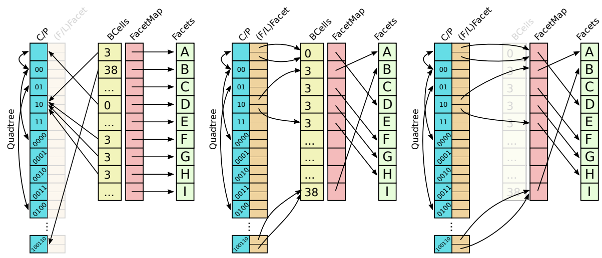


Figure 3: (a) The bounding cells (BCells) are stored in an array initially sorted on facet index (letters are used here for clarity). The quadtree array elements are structures which store child and parent pointers ("C/P" in the figure). (b) We sort the BCells array using a parallel radix sort on BCell address for fast indexed access. We then, in parallel on each element of the BCells array, store the BCells/FacetMap indices of the first and last facets in a given quadtree cell in `FFacet` and `LFacet`, respectively. (c) For a given quadtree cell, we can find all contained facets for use in algorithm 1.

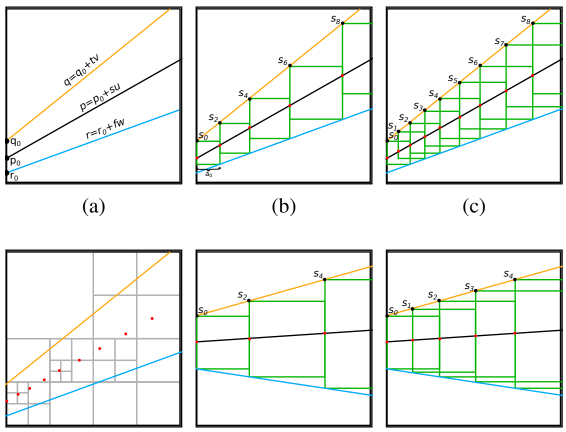


Figure 4: (a) A conflict cell with two lines from different objects. (b)-(c) Fitting boxes such that any box intersecting both lines contains at least one sample (red dots). (b) Fitting boxes such that any box intersecting both lines contains at least two samples. This ensures that a quadtree built from 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. (e)-(f) The adjacent case.

3.4. Resolve conflict cells

We present a conflict cell resolution algorithm for pairs of lines in 2D. For a conflict cell C, our approach is to find sample points inside the cell such that no leaf cells in a quadtree constructed over the sample points intersect both lines. In this section we derive equation (28) which computes the number of samples required to resolve the cell. We also derive equation (22) which computes the samples themselves. The power of our approach lies in the fact that both expressions are closed-form and neither one is iterative, so we can evaluate the first in parallel over leaf cells and the second in parallel over all samples that we need to compute.

To resolve a conflict cell C, we consider pairs of lines of differing labels that intersect C. Figure 4a 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 4d) such that a quadtree built on $S \cup P$ will completely "separate" q and r, i.e., no descendent leaf 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. In the following discussion, p^x and p^y refer to the x and y coordinates of point p, respectively.

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. The smallest box sampled at point $p(s)$ has edge length $a(s)$ as shown in figure 4b. We break the problem of finding $a(s)$ into two cases:

1. The *opposite* case (figure 4b) is where $w^y > 0$, so each box intersects q and r at its top-left and bottom-right corners, respectively.
2. In the *adjacent* case (figure 4e), $w^y < 0$, so the line intersections are adjacent at the top-left and bottom-left corners of the box.

3.4.1. 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,

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

where

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

and

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

3.4.2. 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

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

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

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

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

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

where

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

and

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

3.4.3. 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 4b and 4e). 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

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

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

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

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

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

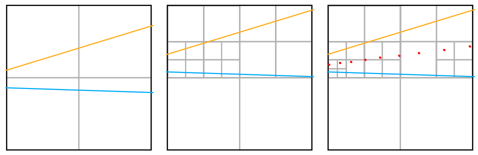


Figure 5: Best and worst cases given two lines. The same number of conflict resolution samples are generated regardless of where the lines are located. (a) Base case: two lines can be resolved by a single quadtree subdivision. (b) Worst case: the same two lines translated slightly in y now require five subdivisions to be resolved. (c) The number of cells generated from the shown resolution samples is within a constant factor of the worst case.

3.4.4. Iteration

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 quadtree from points (section 3.1). We finish when the only conflicts identified are at the maximum depth.

3.5. Optimality

Define an optimal final quadtree to be one in which only conflict nodes have children, and let an optimal final quadtree's size be n total nodes. Our iterative sampling algorithm results in a quadtree that has a size within a constant factor of n in the worst case (see figure 5). We omit the proof as well as an average case analysis because optimality can be achieved by simply removing unnecessary nodes in one final parallel pruning step.

4. Implementation

We have implemented[1] our algorithm using OpenCL. Figure 6 shows the stages of the major kernels. Every kernel call is parallel on vertices, facets, quadtree nodes or Morton code bits. Our implementation uses 64-bit Morton codes, which were sufficient for all of our datasets. There is no way of knowing *a priori* what the object spacing is going to be, however, 64-bit codes were sufficient for the most demanding datasets while retaining reasonable radix-sort timings. Choosing a large number of bits for the Morton code results in little or no wasted effort in later refinement, affecting the amount of later pruning only if intra-object vertex

[1]Source code is available at `www2.cose.isu.edu/~edwajohn/research/pquad`.

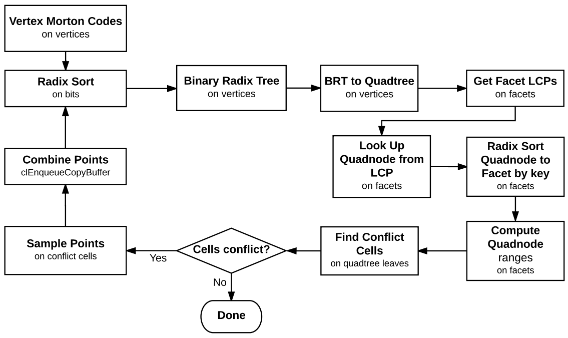


Figure 6: Kernel calls, with some calls omitted for clarity. The name of the kernel is in larger font while the the elements on which the parallelism runs are given in smaller font. The majority of calls are facet- or vertex-parallel.

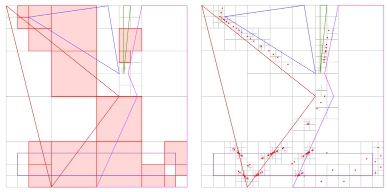
spacing is small compared to spacing between objects. The number of bits also has no effect on the number of conflict cells unless the Morton codes are too small to resolve vertices.

Our implementation of the algorithm supports polygons and polylines which needn't be manifold or connected. Intersecting lines are not handled as a special case, i.e., the quadtree is simply resolved to its maximum depth. Special handling can be implemented per application as needed, e.g., for collision detection applications.

5. Results and conclusions

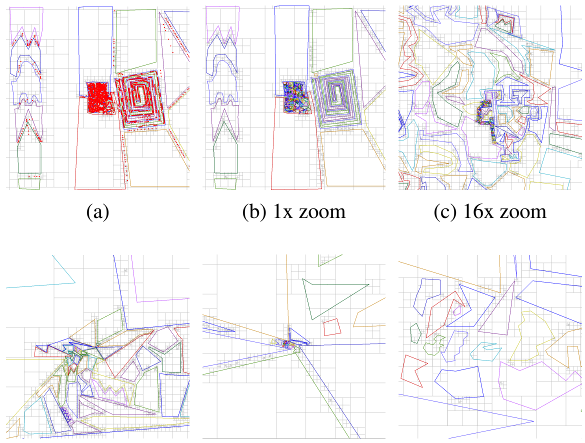
All tests were run on an Intel i7 6500u 3.10 GHz dual core processor, 8 GB of memory and an Nvidia GTX 1070 graphics card. Figure 7 shows results a simple toy dataset showing conflict cell detection and resolution. A very complex dataset with many objects at very different scales is shown in figure 8. It demonstrates that our method can handle datasets far beyond the memory limits of uniform grid approaches while still fully resolving between objects. The gears dataset (figure 9) again shows a large domain-to-object-spacing ratio, as well as non-convexities. The vascular dataset shown in figure 10 demonstrates our method on polylines derived from biological image data, which is often noisy with non-manifoldness and intersections. Table 1 shows timings for our implementation compared to the previous state-of-the-art. Our implementation is significantly faster and also generates fewer quadtree cells. See Appendix A for a runtime complexity analysis.

As can be seen in table 1, there is overhead with our approach: running our algorithm on small datasets yields smaller gains. In fact, our approach actually per-



(a) (b)

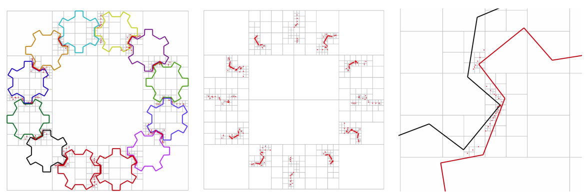
Figure 7: (a) A toy dataset showing conflict cells after building the quadtree from object vertices. (b) The toy dataset showing how samples are collected.



(a) (b) 1x zoom (c) 16x zoom

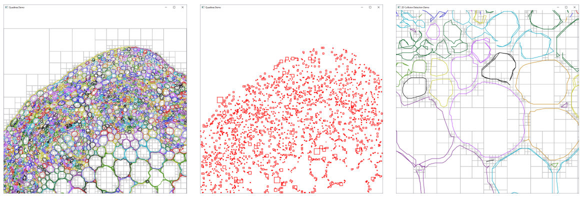
(d) 256x zoom (e) 4,000x zoom (f) 60,000x zoom

Figure 8: (a) A complex dataset with 470 objects at vastly different scales in object size and spacing. (b)-(f) Complex dataset at different zoom levels up to 60K magnification. This shows the importance of an adaptive method such as a quadtree. A uniform grid would require 2^{48} cells to resolve between objects. The quadtree shown here has 22,429 cells.



(a) (b) (c)

Figure 9: (a) A dataset of gears with close tolerance. The resolved quadtree with sampled points is shown. (b) Showing just the quadtree and sample points. (c) A zoomed-in image showing the close object spacing compared to the large domain.



(a) (b) (c)

Figure 10: A large set of uniquely labeled polygons constructed from connected component analysis on a photograph of vascular cambium, a type of plant tissue. (a) Initial vertex quadtree after pruning. (b) All conflict cells of the initial quadtree. (c) After conflict cell resolution. No quadtree cell intersects more than one object. Our method works even though objects in this dataset are often non-manifold and have self-intersections.

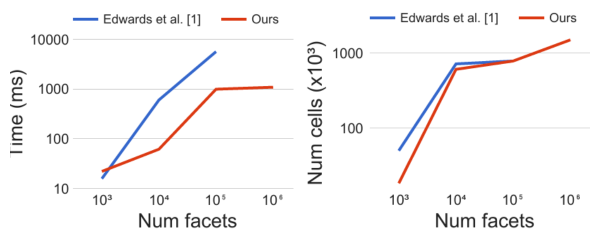
dataset	objects	object facets	quadtree depth	time (millisec)	
				Ours	Prev
Fig. 7a	5	24	9	5	3
Fig. 9	12	288	10	8	24
Fig. 8a	470	4943	24	40	277
Fig. 10	2162	39,338	12	36	376

Table 1: Table of quadtree computation statistics and timings. *Ours* is the approach described in this paper and *Prev* is the approach by Edwards et al. [1]. Columns are: *objects* - the number of objects in the dataset; *object facets* - the number of line segments (2D) of all objects in the dataset; *quadtree depth* - required quadtree depth in order to resolve objects; *time (ms)* - milliseconds to build the quadtree

forms worse on the toy dataset. The power of our algorithm becomes obvious on large, complex datasets, where our performance time gains are significant.

Figure 11 shows the results of a scaling study, where we increased the number of objects and facets by orders of magnitude. Our algorithm consistently shows timings an order of magnitude faster than the state of the art. The approach of Edwards et al. failed on datasets with 10^6 facets or more.

As noted in the introduction, our continuing work is in fast construction of octrees modeling inter-object space in 3D. Every step in our method has a straightforward extension to 3D with the exception of point sampling for conflict resolution (see section 3.4), which is where we are focusing our efforts.



(a) (b)

Figure 11: Scaling tests. The dataset used is a square domain with increasing numbers of non-intersecting lines. (a) Our algorithm consistently performs an order of magnitude faster than the algorithm of Edwards et al. [1] as the number of facets increases. (b) The quadtree that we build is roughly equal in size to that of the previous work.

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

[2] Karras T. 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. Eurographics Association; 2012, p. 33–7.

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

[4] Strain J. Fast tree-based redistancing for level set computations. Journal of Computational Physics 1999;152(2):664–86.

[5] Frisken SF, Perry RN, Rockwood AP, Jones TR. 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. ACM Press/Addison-Wesley Publishing Co.; 2000, p. 249–54.

[6] Boada I, Coll N, Sellares J. The voronoi-quadtree: construction and visualization. Eurographics 2002 Short Presentation 2002;:349–55.

[7] Boada I, Coll N, Madern N, Antoni Sellares J. Approximations of 2D and 3D generalized Voronoi diagrams. International Journal of Computer Mathematics 2008;85(7):1003–22.

[8] Teichmann M, Teller S. Polygonal approximation of Voronoi diagrams of a set of triangles in three dimensions. In: Tech Rep 766, Lab of Comp. Sci., MIT. 1997,.

[9] Vleugels J, Overmars M. Approximating Voronoi diagrams of convex sites in any dimension. International Journal of Computational Geometry & Applications 1998;8(02):201–21.

[10] Bédorf J, Gaburov E, Portegies Zwart S. A sparse octree gravitational N-body code that runs entirely on the GPU processor. Journal of Computational Physics 2012;231(7):2825–39.

[11] Zhou K, Gong M, Huang X, Guo B. Data-parallel octrees for surface reconstruction. Visualization and Computer Graphics, IEEE Transactions on 2011;17(5):669–81.

[12] Li Z, Wang T, Deng Y. Fully parallel kd-tree construction for real-time ray tracing. In: Proceedings of the 18th meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games. ACM; 2014, p. 159–.

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

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

[15] Laine S, Karras T. Efficient sparse voxel octrees. Visualization and Computer Graphics, IEEE Transactions on 2011;17(8):1048–59.

[16] Lefebvre S, Hoppe H. Compressed random-access trees for spatially coherent data. In: Proceedings of the 18th Eurographics conference on Rendering Techniques. Eurographics Association; 2007, p. 339–49.

[17] Choi B, Komuravelli R, Lu V, Sung H, Bocchino RL, Adve SV, et al. Parallel SAH kD tree construction. In: Proceedings of

the Conference on High Performance Graphics. Eurographics Association; 2010, p. 77–86.
[18] Bastos T, Celes W. GPU-accelerated adaptively sampled distance fields. In: Shape Modeling and Applications, 2008. SMI 2008. IEEE International Conference on. IEEE; 2008, p. 171–8.
[19] Park T, Lee SH, Kim JH, Kim CH. CUDA-based signed distance field calculation for adaptive grids. In: Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on. IEEE; 2010, p. 1202–6.
[20] Yin K, Liu Y, Wu E. Fast computing adaptively sampled distance field on GPU. In: Pacific Graphics Short Papers. The Eurographics Association; 2011, p. 25–30.
[21] Crassin C, Green S. Octree-based sparse voxelization using the GPU hardware rasterizer. OpenGL Insights 2012;:303–18.
[22] Kim YJ, Liu F. Exact and adaptive signed distance fields computation for rigid and deformable models on GPUs. IEEE Transactions on Visualization and Computer Graphics 2014;20(5):714–25.
[23] Ha L, Krüger J, Silva CT. Fast four-way parallel radix sorting on GPUs. In: Computer Graphics Forum; vol. 28. Wiley Online Library; 2009, p. 2368–78.
[24] Pantaleoni J. VoxelPipe: a programmable pipeline for 3D voxelization. In: Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics. ACM; 2011, p. 99–106.

Appendix A. 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 quadtree, and D_{max} be the maximum depth of the quadtree. In this analysis we assume sufficient parallel units to maximize parallelization.

Time complexity

1. Build quadtree using Karras' algorithm [2], including pruning - $O(D_{max})$.
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 - $O(D_{max})$. We use a parallel radix sort with linear complexity dependent on the max quadtree depth.
   (c) Index BCells with quadtree data structure - $O(D)$. This runs in parallel on leaf cell IDs and each kernel requires a search of the quadtree 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 quadtree 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) $S \leftarrow S \cup S'$ - $O(1)$.
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, Q has not exceeded 4.

The final complexity of each iteration is $O(M + D_{max})$ worst case and $O(\log M + D_{max})$ average case. In practice we must fix the depth of the quadtree 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.

Space complexity

The primary data structures are shown in figure 3a. The quadtree data structure is size $O(|S|)$ and the remaining arrays are of size M. As $|S| \geq M$, our final space complexity is $O(|S|)$. The number of samples in S depends on the dataset. In 2D, in the worst case, the facets can form an arrangement of maximum number of intersections, which is $M(M-1)/2 = O(M^2)$. If this is the case then we subdivide to the maximum quadtree depth at each intersection, causing a quadtree of size $O(DM^2)$.