Fast Construction of Inter-Object Spacing Representations

Nathan Morrical, John Edwards

Department of Informatics and Computer Science, Idaho State University, 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, discretizing finely only where needed, Hierarchical kernel envokation isn't necessarily bad. We might move to that to remove some of the Q linear complexity overhead. 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, derive the time complexity, demonstrate experimental results, and discuss extension to 3D. Our results show significant improvement over the current state of the art.

1. Introduction

Reworded this paragraph, since the last sentence contained duplicate information provided in the first sentence. Constructing quadtrees on objects is an important task with applications in collision detection, distance fields, generalized Voronoi Diagram (GVD) construction [1], 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. In other cases, quadtrees are purposed for fast retrieval as is often the case in hierarchical subdivisions of point data. 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.

reworded the first sentence a bit. This object separation is of some use in 2D (e.g. path planning), but becomes a very important problem in 3D. Hierarchically subdividing non-point data in a principled parallel way is surprisingly complex, and this paper lays the groundwork for our continuing work in 3D.

Modeling inter-object spacing is computationally straightforward when that 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. added carriage return here

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 course 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.

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 to point sites [10, 2, 11]. Kim and Liu's work [12] is similar, computing the quadtree on the barycenters of triangles, giving an approximation of an object-resolving quadtree. Most quadtree approaches that support surfaces [13, 14, 15, 16] are designed for efficient rendering, and actual construction of the quadtree is implemented on the CPU. Two works [17, 18] implement Adaptive Distance Fields in parallel on quadtrees but building the quadtree itself is done sequentially. Yin et al. [19] 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

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, called the "vertex 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 (see section 3.4.4).

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 starts by sorting the Morton codes of the given vertices. Our implementation uses an efficient parallel radix sorter described by Ha et al. [20]. Once the vertices are sorted, a binary radix tree, and then an initial quadtree can be constructed in parallel. 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. Prune the octree

During the initial quadtree construction, we can indirectly prune the initial quadtree to simplify conflict detection and reduce our memory footprint. Assume we have a numeric vertex labeling such that each vertex is labeled to match the object it belongs to. The initial binary radix tree (BRT) provided by Karras serves as a bounding volume hierarchy, and is used to generate the initial quadtree by seperating vertices regardless of their label. Since our objective is to resolve between objects of different labels, we can proactively prune the initial BRT, and subsequently the initial quadtree (see figure 1) such that a leaf node can contain multiple vertices as long as they all have the same label.

I tried to be a bit more verbose here and in the paragraph above. Let me know if anything in particular needs more clarification. To prune the initial BRT efficiently, we label each BRT node using the following criterion: if the current node is a leaf node which seperates two vertices with identical labels, label the current node to match the label of the vertices being seperated.

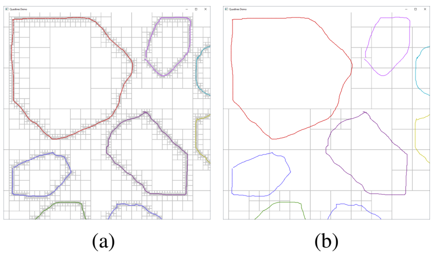

(a) (b)

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.

If the current node is a leaf node that seperates two vertices having mismatching colors, label the current node as -2, i.e. ”required”. Lastly, if the current node is an internal node, i.e. the current node is an ancestor of another BRT node, mark the current node as -1, i.e. ”unknown”. This initial step can be done immediately after Karras’ BRT construction without the need to invoke an additional kernel.

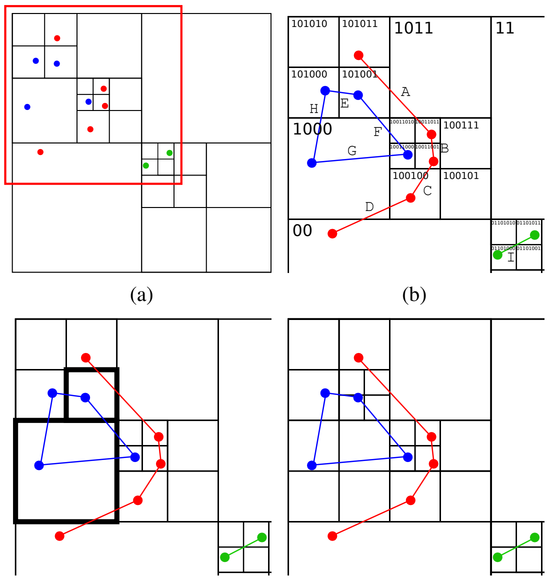
We then propagate the BRT labels up the tree in parallel, marking ”unknown” nodes as ”required” when the labels of the current node’s two children nodes mismatch. Finally, we generate quadtree nodes from only the required internal binary radix tree nodes.

3.3. 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 a 2-bit (3-bit in 3D) Morton code. The address of the root cell is defined as the empty string. Figure 2b 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 contained in the longest common prefix (LCP) of the Morton codes of the points in P. If a given LCP is more specific than any quadtree node, ie the LCP lies within a quadtree leaf, we simply take the quadtree address of the leaf that the LCP lies within. This is often the case with pruned quadtrees where entire facets may lie within a quadtree leaf. This paragraph had some incorrect conclusions, so I tried to clarify a bit more. Figure 3a gives the addresses of the BCells of the facets in figure 2b.

We begin by constructing an array `BCells` and sibling array `FacetMap` (see figure 3a), which is done



(a) (b)
(c) (d)

Figure 2: We have three objects, blue, red, and green with facets labeled A-I. (a) Initial vertex quadtree. This vertex quadtree is unpruned. Can we prune it for consistency? (b) Zoomed-in to the region outlined by red in (a) 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.

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 3b).

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). First, we set L’s color to -1 (uninitialized). We then traverse each ancestor A of L (line 3) by using the `Parent` field in the octree data structure. Using the `FFacet` and `LFacet` fields, we find, respectively, the first and (inclusive) last of possibly multiple facets bounded by A (line 4). The `FacetMap` array is used 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.

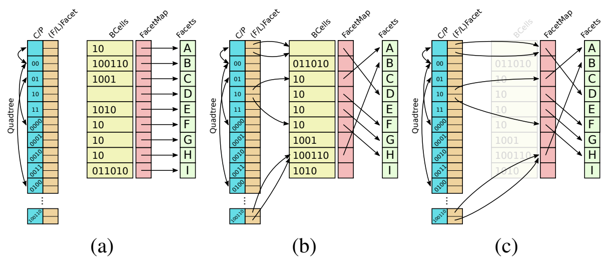


Figure 3: (a) The bounding cells (BCells) are stored in an array initially sorted on facet index (letters are used here for clarity). The quadtree array elements are structures which store child and parent pointers ("C/P" in the figure). (b) We sort the BCells array using a parallel radix sort on BCell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

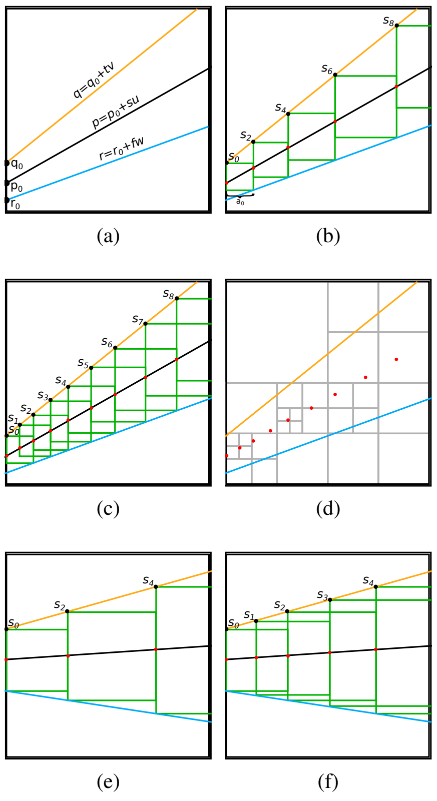


Figure 4: (a) A conflict cell with two lines from different objects. (b) Fitting boxes such that any box intersecting both lines contains at least one sample (red dots). (b) Fitting boxes such that any box intersecting both lines contains at least two samples. This ensures that a quadtree built from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

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]...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 ≠ f.color then
                    L.color = -2
                    L.facet[1] = f
                end
            end
        end
    end
end
```

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 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. 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. We break the problem into two cases:

1. The *opposite* case (see 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 (see 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 $S = \{s_0, s_1, s_2, \dots, s_n\}$ 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

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

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 octree from points (section **??**). We finish when the only conflicts identified are at the maximum depth.

3.5. *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.

Time complexity

1. Build octree using Karras' algorithm [2] - $O(D)$.
2. Prune the Binary Radix Tree from Karras [2] - $O(D)$ Does this sound right?
3. 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 - $O(\log M)$. Shouldn't the big o here be O(n), where n is the number of levels in the quadtree? 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.
4. 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) $S \leftarrow S \cup S'$ - $O(1)$.

5. 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 $O(M + D)$ 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.

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 octree depth at each intersection, causing an octree of size $O(DM^2)$.

4. Results and conclusions

Our implementation[1] of the algorithm supports polygons and polylines which needn't be manifold or connected. All tests were run on a Razer Blade Stealth with an Intel i7 6500u 3.10 ghz dual core processor, 8GB of memory, and an Nvidia GTX 1070 graphics card. Figure **??** shows results on four datasets: a simple toy dataset showing conflict cell detection and resolution (5a-5b); a more complex maze dataset (**??**), and a complex dataset with many objects at very different scales (8a-8f). 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.

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

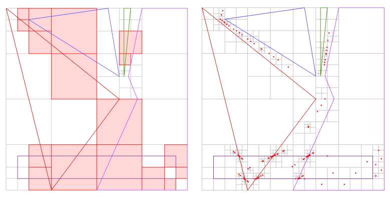


Figure 5: (a) A toy dataset showing conflict cells after building the quadtree from object vertices. (b) The toy dataset showing how samples are collected.

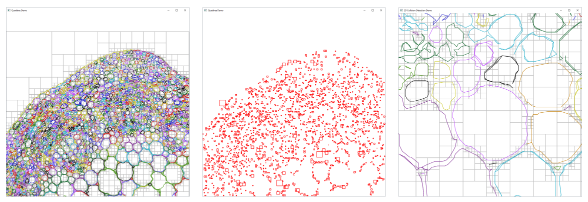


Figure 6: A large set of uniquely labeled polygons constructed from connected component analysis on a photograph of vascular cambium, a type of plant tissue. (a) Initial vertex quadtree after pruning. (b) All conflict cells of the initial quadtree. (c) After conflict cell resolution. No quadtree cell intersects more than one object.

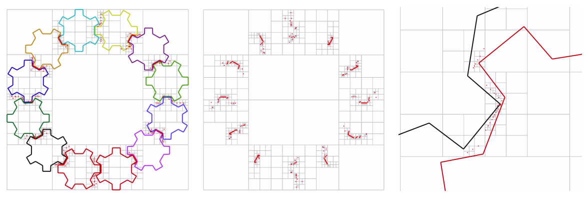


Figure 7: (a) A dataset of gears with close tolerance. The resolved quadtree with sampled points is shown. (b) Showing just the quadtree and sample points. (c) A zoomed-in image showing the close object spacing compared to the large domain.

[1]Source code will be made available at our website.

dataset	objects	object facets	quadtree depth		time (millisec)		quad cells ($\times 10^3$)	
			Ours	Prev	Ours	Prev	Ours	Prev
Fig. 5a	5	24	10	9	54	3	177	1168
Fig. 8a	470	4943	24	24	128	465	38	157
Fig. **??**	2	27,998	9	8	148	429	43	66
Fig. **??** x2	2	113,084	10	9	414	1778	125	262

Table 1: Nate, please send me an updated table in whatever format (I can throw it into Latex). We should have timings on simple, maze, vascular and gears comparing GVD to PGVD with and without pruning. Not sure if we'll end up including the non-pruning results. We should also include the # objects, # facets and # quadtree cells, as are included in this table. Table of quadtree computation statistics and timings on datasets that are unmanageable using other methods. 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; *quad cells* - number of quadtree cells. Dataset "**??** x2" is a maze dataset increased in size by a factor of two in each dimension from **??**.

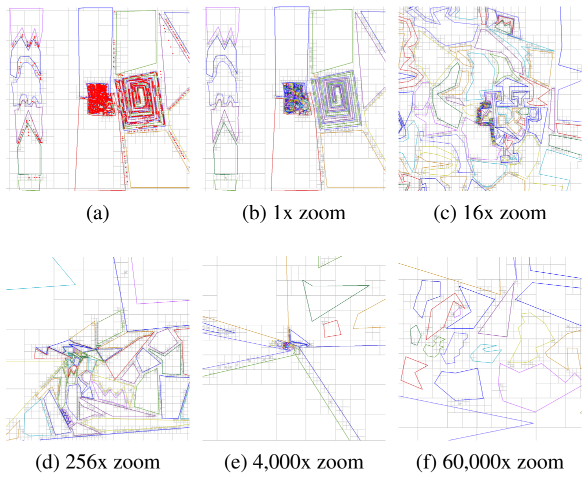

(a) (b) 1x zoom (c) 16x zoom
(d) 256x zoom (e) 4,000x zoom (f) 60,000x zoom

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.

We are in the process of integrating our algorithm with animated systems, generating quadtrees in real-time for collision detection, distance transforms, and generalized Voronoi diagram computation. Our implementation continues to be refined and optimized, and we expect to shortly have a version with an order of magnitude improvement over the state of the art. Importantly, we are also working on an extension to 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.

[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¡ i¿ n¡/i¿-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] Kim YJ, Liu F. Exact and adaptive signed distance fieldscomputation for rigid and deformablemodels on gpus. IEEE Transactions on Visualization and Computer Graphics 2014;20(5):714–25.

[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] 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.

[18] 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.

[19] 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.

[20] 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.