Fast Dynamic Voronoi Treemaps

Video at: http://research.microsoft.com/~danyelf/isvd2010_submission_36.mov

Avneesh Sud
Microsoft Research XCG
avneesh.sud@microsoft.com

Danyel Fisher
Microsoft Research
danyelf@microsoft.com

Huai-Ping Lee
UNC-Chapel Hill
lhp@cs.unc.edu

***Abstract*—The Voronoi Treemap is a space-filling treemap technique that relaxes the constraints of rectangular nodes. Its organic shapes maintain a one-to-one aspect ratio, are flexible with their placement, allowing stable zooming and dynamic data values. In this paper, we present algorithms for efficient computation and dynamic update of Voronoi Treemaps. Our GPGPU-based technique allows for rapid computation of centroidal Voronoi Diagrams, providing almost two orders of magnitude speedup over previous work. In addition, we present a hierarchical algorithm for stable updates. Finally, we demonstrate the application of Voronoi treemaps to real-world dynamic datasets, including interactive navigation.**

***Keywords*-Voronoi Diagram; Treemap; GPU; Dynamic;**

I. Introduction

Treemaps are a popular tool for visualizing large amounts of data. They are one of the most successful sophisticated visualizations of weighted hierarchical data available : each node is represented by a 2D shape with an area proportional to its relative weight. Familiar examples include treemaps that visualize disk management systems [1], online discussion groups [2] and stock prices [3]. Increasingly, however, online data has gained a temporal component: we are interested in viewing streaming data, and tracking how it changes. Treemaps are well-posed to handle these new sources of data if they can be adapted to portray changing values.

Most space-filling treemap algorithms today depend on greedy algorithms that fill the space with rectangular shapes: easy to compute and quick to render, rectangles are a convenient shape for visualizations. Yet greedy algorithms, and rectangles, force a choice between stability in the face of dynamic data and a good aspect ratio. While several treemap algorithms have attempted to accommodate dynamic data [4], none have managed to maintain stability as well as a balanced aspect ratio in the face of changes to data.

Voronoi treemaps, presented originally by Balzer *et al.* [5], [6], are a promising alternative approach. By relaxing the constraint of rectangular shapes, they use an optimization algorithm to produce compact Voronoi shapes, which may be dynamically modified in a smooth manner. Voronoi treemaps are, however, computationally-expensive to produce: the original work uses an random-sampling algorithm to compute weighted Centroidal Voronoi Diagrams (CVDs)

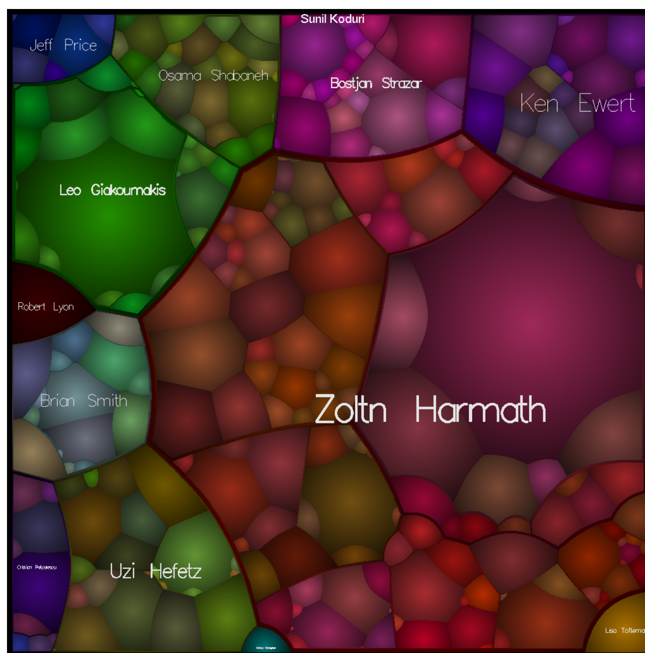

Figure 1: A Voronoi treemap representing an organization hierarchy of 180,000 employees. Every node represents a manager, and is sized proportionate to the number of reports. In this image, we have rendered only the top four levels.

to compute and render Voronoi treemaps. Past literature [5] has suggested that it may be possible to dynamically modify treemaps as data changes, but does not confirm it.

In this paper, we present two main contributions. Firstly, we present a GPU-based iterative algorithm for fast computation of additive weighted-CVDs, exploiting the coherence across consecutive iterations. Using the GPU-based algorithm we can render and animate a Voronoi treemap at interactive rates. Secondly, we present a hierarchical algorithm for computing stable updates of Voronoi treemaps with dynamic data.

In the first section, we discuss related literature, including other approaches to dynamic treemaps. We then discuss the optimizations we have made to computation, rendering, and animation techniques. They use a combination of data interpolation and visual interpolation in order to smoothly

animate treemap nodes. We finish with performance results showing the effects of our optimization.

II. Related Work

In this section, we first review past approaches to treemaps and their tradeoffs with regards to dynamic data. We then discuss prior work on computation of Centroidal Voronoi Diagrams and GPU-based acceleration.

A. Treemap Tradeoffs

The desired properties of the shapes in a dynamic treemap are (a) partition the space without holes (b) low-aspect ratios, (c) stable zoom and (d) stable updates with dynamic data. Optimally, a treemap algorithm minimizes the vacant (uncovered) space, maintains a small aspect ratio, and allows stable updates and zooming without distortion [5], [4].

The classic "slice and dice" algorithm [7] partitions a space in a series of parallel strips. These slices that the algorithm produces can have a very lopsided aspect ratio. This aspect ratio is a critical factor with treemaps: it is very difficult to estimate the relative size of highly imbalanced shapes [8]. As a result, a treemap algorithm that does not guarantee good aspect ratios can be very hard to interpret. Further, zooming in on regions with high aspect ratios requires distortion [9], letter-boxing, or re-generating the image. Other visualization techniques, such as the squarified layout [10] and the ordered treemap [1], produce rectangles with a superior aspect ratio, making it easier to read nodes. However, in order to fill the screen space, these algorithms re-generate the layouts on zooming making them unstable for dynamic navigation.

Appropriately animating data can help users understand how values change by facilitating object constancy [11], and Heer and Roberston [12] have demonstrated that smooth animations can help users track changes in data. While animation is valuable if smooth, it can also be disruptive or confusing, for e.g. if objects all pass through the center, for instance, users can lose track of them [13], [12].

However treemaps are very difficult to animate in a smooth manner as their components change - with trade-offs between low-aspect ratios and stability. Simple resizing of nodes by widening/narroing along 1 axis preserves stability, but results in large distortions and high aspect ratios. Recomputing the layouts at each time step maintains low-aspect ratios, however causes many nodes to rearrange, adversely affecting stability. Other treemap algorithms attempt to reduce the impact of changing data: strip treemaps [1] and spiral treemaps [4] impose ordering constraints on the nodes. While the algorithms continue to minimize aspect ratio, they maintain the fixed order; this prevents nodes from moving too far. When aspect ratio becomes extreme, the boxes jump to other parts of the space. A treemap rearranging can cause precisely this problem. Table I compares the tradeoffs of popular treemap algorithms.

Table I: Trade-offs of treemap algorithms. Optimally, algorithms maintain a small aspect ratio and allow stable updates and zooming without distortion

	Avg Aspect Ratio [4]	Stable updates	Stable zoom
Slice&Dice [7]	100+	✓	X
Squarified [10]	1.3	X	✓
Ordered [1]	2.8	X	X
Strip [1]	2.5	X	X
Spiral [4]	2.5	X	X
Voronoi	*1.3**	✓	✓

+ Avg aspect ratio for first dataset, it may increase across time.
* Based on the aspect ratio of the bounding boxes of the Voronoi regions.

By relaxing the requirement to use rectangular regions, treemap algorithms gain freedom to arrange themselves with smaller aspect ratios, and to move around the screen more smoothly. Both Voronoi treemaps [6], [5] and Circular Partition treemaps [14] maintain low aspect ratios, which allows smooth zooming. Voronoi treemaps are generated with an iterative algorithm, which lends itself to enforcing constraints on positions. These constraints, we will show, can be used to animate treemaps as data changes dynamically. In contrast, Circular Partition treemaps do not lend themselves to handling dynamic data.

B. Centroidal Voronoi Diagrams and GPU Computation

The Voronoi diagram is a fundamental geometric data structure. A weighted Centroidal Voronoi Diagram (CVD) provides a structure that is space-filling, provides a good 1:1 aspect ratio for each region, and provides intuitive control of each region. However, while exact algorithms exist for computing weighted centroidal Voronoi diagrams in continuous [15] and discrete [16] 2D domains, general Voronoi diagrams are far more computationally expensive to compute. Lloyd [17] presents a practical approach for computing centroidal (non-weighted) Voronoi diagrams using iterative updates.

In the original work on Voronoi treemaps [5], the authors use a parallel, Monte Carlo sampling based approach for computing weighted CVDs. This approach is very computationally expensive – generating the treemap of 4000 nodes takes over 7 minutes using 8 CPUs. In later work [6] they suggest using the GPU based approach of [16] to accelerate the computation.

CVD computation can also be performed as an optimization task, minimizing the total inertia moment of Voronoi regions. This has been demonstrated for non-weighted CVD computation [18]. Recently, Balzer *et al.* [19] have presented Capacity-Constrained Voronoi Diagrams (CCVDs)

for computing power (multiplicative-weighted) CVDs, and performance improvements are presented in [20]. However, we are not aware of any work on extending above techniques for computation of additive weighted CVD, and application to dynamic datasets.

Recent work has taken advantage of the increasing programmability, and the parallelism, of GPUs to compute discrete approximations of various Voronoi diagrams efficiently. A survey of algorithms for computing ordinary and weighted discrete Voronoi diagrams in 2D using GPUs is presented in [21]. Performance optimizations for 2D error-bounded discrete non-weighted Voronoi digrams on the GPU are presented in [22], [23], [24]. A GPU-based implementation of CVDs is described in [25].

In the last year, a variety of projects have begun to examine the uses of GPU computation for information visualization. McDonnel and Elmqvist [26] suggest using GPU techniques to rapidly render nodes for graphs and charts, and provide a graphics pipeline; similarly, Bailey [27] offers a brief tutorial on using GPU shaders to generate scatterplots and isocontours. In this paper, we use the GPU not merely for rendering large numbers of objects rapidly, but also for accelerating a computation process.

III. Overview

In this section we introduce the notation used in the paper, give a background on Voronoi treemaps, and present an overview of our approach.

A. Notation

Let $\mathcal{X} \subset \mathbb{R}^2$ represent a compact domain, and $BB(\mathcal{X})$ denote an axis-aligned bounding box of $\mathcal{X}$. Let $\mathcal{P} = \{\mathbf{p}_1, \ldots, \mathbf{p}_n\}$ be a set of n distinct points in a compact domain $\mathcal{S} \subset \mathbb{R}^2$ with coordinates $(x_1, y_1), \ldots, (x_n, y_n)$. We call these points *sites*. Given a distance function $d(\mathbf{p}, \mathbf{q})$ between two points $\mathbf{p}, \mathbf{q} \in \mathbb{R}^2$, the *Voronoi region* of a site $\mathbf{p}_i$ is the set of points closer to $\mathbf{p}_i$ than to any other site:

$$\mathcal{V}(\mathbf{p}_i, \mathcal{P}) = \{\mathbf{q} \in \mathbb{R}^2 \mid d(\mathbf{q}, \mathbf{p}_i) \leq d(\mathbf{q}, \mathbf{p}_j) \, \forall \, \mathbf{p}_j \in \mathcal{P}, j \neq i\}$$

The *Voronoi Diagram (VD)* is a partition of the domain $\mathcal{S}$ into (at most) n Voronoi regions:

$$\mathrm{VD}(\mathcal{P}, \mathcal{S}) = \bigcup_{\mathbf{p}_i \in \mathcal{P}} \mathcal{V}(\mathbf{p}_i, \mathcal{P}) \cap \mathcal{S}$$

Weighted Voronoi Diagrams: Typically, the distance function $d(\mathbf{p}, \mathbf{q})$ is the Euclidean distance function. Using other distance functions for computing Voronoi regions results in generalized Voronoi diagrams [28]. Given a set of weights $\mathcal{W}$, a unique weight $w_i \in \mathcal{W}$ is assigned to each site $\mathbf{p}_i$. Using weighted distance functions, a *Weighted Voronoi Diagram* $\mathrm{VD}(\mathcal{P}, \mathcal{S}, \mathcal{W})$ is generated. An *additive weighted (AW) Voronoi Diagram* $\mathrm{VD}_{aw}(\mathcal{P}, \mathcal{S}, \mathcal{W})$ is generated by adding a weight to the distance function,

$$d_{aw}(\mathbf{q}, \mathbf{p}_i, w_i) = \|\mathbf{q} - \mathbf{p}_i\| - w_i$$

This is similar to generating a regular Voronoi diagram where the ith generator is a circle centered at (x_i, y_i) and radius w_i, although the weights may be negative. The boundary of an additive weighted Voronoi region consists of hyperbolic curve segments. For simplicity, we refer to $\mathcal{P}^w$ as the weighted site set: that is, $\mathcal{P}$ and $\mathcal{W}$ combined.

Centroidal Voronoi Diagrams: A centroidal Voronoi diagram (CVD) is a special Voronoi diagram where each site coincides with the center of mass of its corresponding Voronoi region,

$$\mathbf{p}_i = \mathbf{c}_i = \frac{\int_{\mathcal{V}(\mathbf{p}_i)} \mathbf{x} d\sigma}{\int_{\mathcal{V}(\mathbf{p}_i)} d\sigma}$$

where $d\sigma$ is the area differential. The CVD minimizes the inertia momentum (and increases the compactness) of each Voronoi region [18], resulting in regular sized Voronoi regions with a bounding box aspect ratio close to $1:1$. A CVD can be generalized to a weighted CVD using a set of weights and non-Euclidean distance functions.

Discrete Voronoi Diagrams: Exact computation of generalized Voronoi diagrams is a hard problem due to its algebraic and combinatorial complexity. Instead, it is common to compute discrete approximations of generalized Voronoi diagrams. A discrete Voronoi diagram is computed by sampling the domain $\mathcal{S}$ at a finite set of points $\widetilde{\mathcal{S}}$, and computing the membership of each sample in $\widetilde{\mathcal{S}}$ to a Voronoi region. This computation may be done with random samples [29] or with uniform samples using graphics hardware [30].

B. Voronoi Treemaps

We briefly summarize our approach for generating Voronoi treemaps, which is built upon the work of Balzer *et al.* [5], [6]. The input to the process is, like for other treemap algorithms, a data structure representing a tree of goal areas $\mathcal{G}$. We refer to this input tree as a 'data tree': Let Ndenote a node in a data tree, and the functions `Par`(N), `Children`(N), `CGoals`(N) denote functions that respectively return the parent node of N, a list of child nodes of N, and a list of the goal areas of the child nodes of N.

The treemap is constructed as a hierarchy of weighted CVDs. Each CVD represents a single node of the treemap and its immediate children; the relative weights of the sites must be balanced so that the areas of the Voronoi regions are proportionate to the goal areas. Balzer *et al.*extend the well-known Lloyds algorithm [17] for computing CVDs. This is an iterative algorithm, in each iteration a weighted Voronoi diagram is computed. The position of each site is updated

to the centroid of its corresponding Voronoi region, and the associated weights are updated using relative errors in area of Voronoi regions. Recursively, each regions is subdivided into sub-regions, using the parent region's boundaries as a container.

In this paper, the tree of these CVDs is referred to as the 'generated tree;' in addition to the information in the 'data tree', let $\texttt{CWtPos}(N)$ be a list of the position and weights of the child nodes of a node Nin the generated tree.

C. Our Approach

Balzer *et al.* [5] investigate both additive and power weighting functions on CVDs; in our implementation, we choose the *additive* weighting function, both for its faster convergence and its appealing organic appearance.

We accelerate the CVD computation by computing a discrete CVD using the GPU, as described in section IV. We extend the GPU-based algorithm of Sud *et al.* [22], to compute additive weighted Voronoi diagrams on the GPU. In addition we exploit the coherence between iterations to further speed up CVD computation. The cost of computation can be greatly reduced by tight bounds on Voronoi regions; we present a predictor-corrector scheme to compute bounds on the change in the Voronoi regions and so reduce the number of distance computations required.

We also provide a revised rule for updating the weights across iterations. Due to lack of convex domain and a closed form approximation of the Jacobian of the cost function, we cannot directly apply a line-search gradient descent algorithm as described in [18]. Instead we present a heuristic to identify the cases where the gradient approximation presented in [5] causes convergence of CVD iterations to fail. Under such cases we bound the step size to provide better convergence of the weighted centroidal Voronoi regions. These improvements to computation of static Voronoi treemaps are presented in section.

In section V, we present our approach for stable update of the Voronoi treemap as the underlying data changes. This results in an updated Voronoi treemap in which the relative placement of nodes is similar to the prior treemap. We then linearly interpolate the positions of the nodes in order to accomplish smooth transitions.

IV. Voronoi Treemap Compute and Render

Our approach for fast computation and rendering of Voronoi treemaps depends on speeding the computation of weighted centroidal Voronoi diagrams, by far the slowest step of Voronoi Treemap computation. We optimize this in following ways: by moving operations in VD computation to the GPU, by utilizing coherence across iterations to restrict computation to as small a region as possible, and by accelerating convergence. In contrast, rendering a generated tree can be performed at interactive rates: it uses only one iteration of GPU-accelerated Voronoi diagram compute per node.

A. GPU-Based Weighted CVD Computation

We use an iterative algorithm for weighted CVD computation, similar to Balzer *et al.* [5]. CVD computation involves computing a weighted discrete Voronoi diagram on the GPU. We scan the framebuffer to compute centroids and tally the relative areas, and update the positions and weights. We then pass these updated positions and weights back to the GPU, repeating until convergence. This algorithm is presented

Algorithm 1: ComputeCVD($\mathcal{S}$, $\mathcal{G}$, ϵ, $\mathcal{P}^w$): This algorithm computes the weighted CVD constrained to a domain $\mathcal{S}$ that matches a set of desired areas $\mathcal{G}$ to within error threshold ϵ, initialized from an original weighted site set

Input: domain $\mathcal{S}$, set of initial site positions $\mathcal{P}^w_{in}$, set of normalized goal areas $\mathcal{G}$, error threshold ϵ
Output: set of final site positions $\mathcal{P}^w_{out}$, s.t. Area($\mathcal{V}(\mathbf{p}_i)$) $\propto g_i$ in VD($\mathcal{P}^w$)

1 Initialize $\mathcal{P}^w \leftarrow \mathcal{P}^w_{in}$
2 Initialize each element in $\mathcal{B}$ as $BB(\mathcal{S})$
3 **repeat**
4 buffer $\leftarrow$ GPUComputeWtVD($\mathcal{P}^w$, $\mathcal{B}$, $\mathcal{S}$)
5 $(\mathcal{A}, \mathcal{C}, \mathcal{B}') \leftarrow$ ScanBuffer(buffer)
6 $\mathcal{T} \leftarrow$ VerifyBounds(buffer)
7 $(stable, \mathcal{P}^{w'}, \mathcal{B}) \leftarrow$ UpdateState($\mathcal{P}^w$, $\mathcal{B}', \mathcal{A}, \mathcal{C}, \mathcal{T}, \epsilon, \mathcal{S}$)
8 $\mathcal{P}^w \leftarrow$ FixOverlap($\mathcal{P}^{w'}$)
9 **until** *stable=true*
10 $\mathcal{P}^w_{out} \leftarrow \mathcal{P}^w$

in Algorithm 1. Function `GPUComputeWtVD` computes an additive weighted discrete Voronoi diagrams on the GPU. Conservative bounding boxes of each Voronoi region are predicted and the distance values to each site are computed for each pixel within the bounding boxes using a pixel shader. Further details are provided in Section IV-A1. Function `ScanBuffer` scans the framebuffer to compute a discrete approximation to the area $\mathcal{A}$ and centroid $\mathcal{C}$ of each Voronoi region. In addition the bounding boxes $\mathcal{B}'$ of each Voronoi region are also computed. In our current implementation this is performed by reading back the framebuffer to the CPU.

Function `VerifyBounds` performs the correction on the Voronoi region bounds. The bounds predicted in previous iteration are verified by scanning the framebuffer, marking invalid bounds to be subsequently updated, as presented in section IV-A2. Function `UpdateState` updates the position,

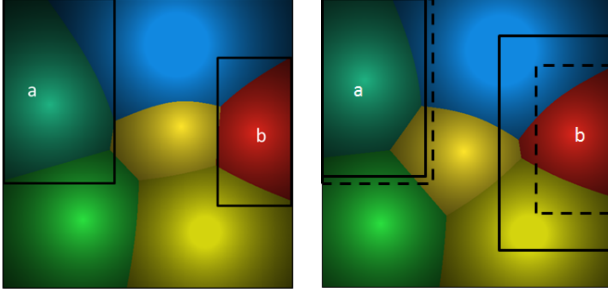

Figure 2: Optimizing CVD computation bounding boxes based on previous iterations. (left) The bounding boxes of nodes a and b. (right) Based on relative error, region (b) was smaller than desired by 20%, so we expand its bounding box by a constant multiple of 20%. Distance computations to b are restricted to pixels inside its bounding box.

weights and estimated Voronoi region bounds for each site. The position is updated similar to the Lloyd's algorithm [17]. We extend Algorithm 2 in Balzer *et al.* [5] to compute the new weights, but add new heuristics to provide stables convergence in certain cases when there is large disparity of goal areas. Finally, the function also predicts new Voronoi region bounds (Figure 2), and is discussed in detail in Section IV-A3. Function `FixOverlap` corrects for overlap between the circles corresponding to the Voronoi sites and is described in lines $4-15$ of Algorithm 4 in [5].

1) Weighted Voronoi Diagram Computation on GPUs: We build upon the work of Sud *et al.* [22] for computing weighted Voronoi diagrams using the GPU. Instead of range-based culling, we compute bounds on the Voronoi regions using coherence across consecutive iterations as shown in section IV-A2. Therefore for each site, exactly 1 tile bounding the Voronoi region is rasterized. The distance vectors to the vertices are computed and passed as texture coordinates. A pixel shader computes the distance value at each pixel, and uses the depth test for storing the minimum distance. The domain $\mathcal{S}$ is scaled to be a subset of $[0, \frac{\sqrt{2}}{2}] \times [0, \frac{\sqrt{2}}{2}]$. The color buffer is used to store the id of the closest site.

Our approach for computing AW-Voronoi Diagram using the GPU follows similarly. A quad covering the AW-Voronoi region is rasterized, and the distance vectors to the vertices are passed as texture coordinates. The weight is passed as a uniform parameter. The weighted distance values are computed by a pixel shader and the minimum is computed via the depth buffer. However, current GPUs have a 24-bit fixed precision depth buffer, and the depth values are clamped to the range $[0, 1]$. The weight term in AW distance function is unbounded and can even be negative. This can cause the distance function to be outside $[0, 1]$ range. This results in errors during GPU computation of AW-Voronoi diagrams. To address this issue, we introduce an affine transform on the distance function. Let w_{min} and w_{max} be the minimum and maximum weights among all sites, and D_{max} be the maximum Euclidean distance among all pairs of sites ($D_{max} \leq 1$ from the domain definition). We now define the affine transforms on distance function between a point $\mathbf{q}$ and site $\mathbf{p}_i$, and weight

$$\begin{aligned} d'_{aw}(\mathbf{q}, \mathbf{p}_i) &= \frac{\|\mathbf{q} - \mathbf{p}_i\|}{D_{max} + w_{max} - w_{min}} + w'_i \\ w'_i &= \frac{w_{max} - w_i}{D_{max} + w_{max} - w_{min}} \end{aligned}$$

It can be shown that $0 \leq w'_i \leq 1$ and $0 \leq d'_{aw}(\mathbf{q}, \mathbf{p}_i) \leq 1$. This affine transform is semantically equivalent to translating the distance cones along Z axis. Therefore the projection of the distance cones to XY plane, and the discrete Voronoi diagram does not change, even though the relative weight ratios are not preserved. Since the range of distance d'_{aw} is $[0, 1]$ the AW Voronoi diagram can now be computed using the depth buffer on the GPU.

2) Voronoi Region Bounds: As the CVD converges, the change in the bounds of each Voronoi region across consecutive iterations diminishes (see figure 2). We exploit this coherence to compute an approximate bounding box for the next iteration using the current bounding box and the error in area of each Voronoi region (lines 12-19 of Algorithm 4). This predicted bound may be invalid if it is too small to contain the Voronoi region that should be computed. We present a simple test for validity of each predicted bound using the continuity of the distance field.

Suppose two adjacent pixels $\mathbf{q}_1$ and $\mathbf{q}_2$ are on the boundary of ith and j^{th} Voronoi regions, namely $\mathbf{q}_1 \in \mathcal{V}(\mathbf{p}_i, \mathcal{P})$ and $\mathbf{q}_2 \in \mathcal{V}(\mathbf{p}_j, \mathcal{P})$, and the size of a pixel is $\|\mathbf{q}_2 - \mathbf{q}_1\| = \delta$. Then both regions are valid if

$$|d_{aw}(\mathbf{q}_1, \mathbf{p}_i, w_i) - d_{aw}(\mathbf{q}_2, \mathbf{p}_j, w_j)| < \delta,$$

Conversely, the bounding box for the ith site is invalid if $d_{aw}(\mathbf{q}_2, \mathbf{p}_j, w_j) - d_{aw}(\mathbf{q}_1, \mathbf{p}_i, w_i) > \delta$, and the bounding box for the jth site is invalid if $d_{aw}(\mathbf{q}_1, \mathbf{p}_i, w_i) - d_{aw}(\mathbf{q}_2, \mathbf{p}_j, w_j) > \delta$. If the predicted bound is invalid, then the bounding box reset to the entire domain, and an additional CVD iteration is performed.

3) CVD Convergence: We use an iterative update rule similar to modified Lloyd's algorithm. The update rule for the weights presented in [5] approximates the gradient of the cost function by the error in areas of each site. This involves two drastic simplifications - the cost function is linear over the entire domain, and the Jacobian is an identity matrix (i.e. change in weight of site $\mathbf{p}_j$ does not affect the area of another site $\mathbf{p}_i$). Due to these assumptions, the prior approach fails to converge stably when there is large disparity between the maximum and minimum goal areas. This results in overcompensating for large errors multiple

times and oscillations when the computed weight changes sign. and is shown in Figure 3. Such scenarios were common in our real-world datasets.

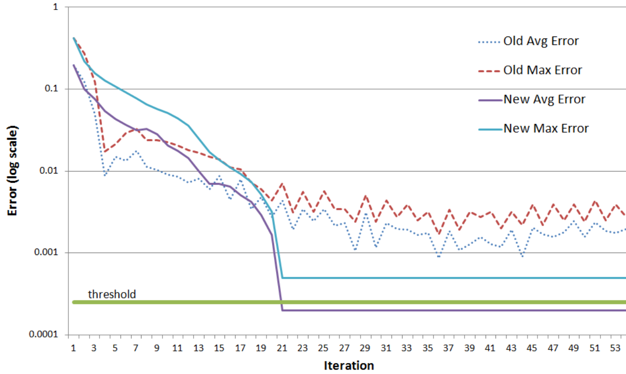

Figure 3: This graph compares the convergence of the AW CVD using the approach presented in [5] (old) and our `UpdateState` rules (new) for 5 sites with random goal weights in the range $[1, 100]$. Iterations terminate when average error $<$ threshold $\epsilon = 0.00025$ (thick line). Our approach converges quickly, whereas the prior approach keeps oscillating.

In our algorithm, we present a new heuristic for the site weights to provide a more stable convergence of CVD iterations. The detailed algorithm is presented in Algorithm 4 (Appendix). The position of a site is updated using the centroid of the Voronoi regions (line 5). The weight is updated using the error values, while preventing it from getting too close to 0 (lines 6 - 10). In addition, under drastic changes in the weight (whenever the weight changes sign), the updated weight is bounded to a small value (lines 10 - 12). Finally, the bounding box is adjusted as described above.

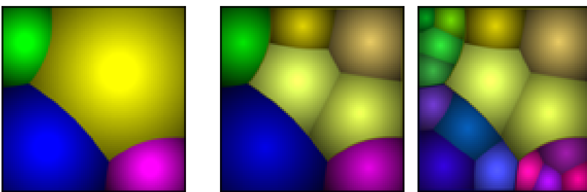

Figure 4: Hierarchical computation and rendering of 2-level Voronoi Treemap: (left) the first level AW VD. (center) VD of yellow child is rendered inside its Voronoi region. (right) the completed render of all children

B. Treemap Computation

The complete Voronoi treemap is computed by recursively traversing the tree top-down and computing the weighted CVD, highlighted in Algorithm 2. For a given node N_i

Algorithm 2: ComputeTreemap(N, $\mathcal{S}$, ϵ): This algorithm recursively computes the positions and radii of nodes in the Voronoi Treemap corresponding to subtree of N, given a bounding domain $\mathcal{S}$, and error threshold ϵ

Input: DataNode N_i, domain $\mathcal{S}$, error threshold ϵ
Output: Generated tree rooted at N_o with final site positions and radii

1 $\mathcal{S}_N \leftarrow$ `GPUComputeMask` $(N_i) \cap \mathcal{S}$
2 Initialize $\mathcal{G} \leftarrow$ `CGoals`(N_i)
3 Initialize $\mathcal{P}^w$ to random points in $\mathcal{S}_N$ and unit weight
4 $\mathcal{P}^w \leftarrow$ `ComputeCVD` $(\mathcal{S}_N, \mathcal{G}, \mathcal{P}^w, \epsilon)$
5 `CWtPos`$(N_o) \leftarrow \mathcal{P}^w$
6 **foreach** *Child* C_i *of* N_i; *Child* C_o *of* N_o **do**
7 $\quad C_o \leftarrow$ `ComputeTreemap` $(C_i, \mathcal{S}, \epsilon)$
8 **end**

at depth d in the tree, the positions of the child nodes `CPos`(N_i) are treated as the set of sites $\mathcal{P}$, and the domain of computation $\mathcal{S}_{N_i}$ is the Voronoi region of node N_i in the AW CVD of this parent. Note that each AW-CVD is computed independently at full grid resolution, therefore the mask $\mathcal{S}_{N_i}$ must be computed for each node at full resolution. We can recursively compute the mask for the tree at any level by noting that the mask at one level is the set intersection of its parents. The function `GPUComputeMask` implements this: it computes a high-resolution mask corresponding to $\mathcal{S}_{N_i}$. The function performs recursive set intersections on 2D domains corresponding to Voronoi regions using the stencil buffer. The stencil values are incremented at each level. The output of `GPUComputeMask` is the stencil buffer where all pixels in $\mathcal{S}_{N_i}$ have a stencil value=d. The function performs d calls to `GPUComputeWtVD`, but only draws sites that have bounding boxes that overlap the bounding box of N_i. No buffer readbacks need to be performed. With this observation, we note that a Generated Treemap can be represented merely as a tree of $\mathcal{P}^w$ (and their bounding boxes), and so can be serialized easily, allowing for offline and remote computation.

C. Interactive Rendering

In this section, we present our approach for rendering the Generated Treemaps at high-resolutions at interactive rates using the GPU. Using a compact tree representation of $\mathcal{P}^w$, an image corresponding to the Voronoi Treemap is computed. The rendering algorithm traverses the Generated Tree, and fills the space by rendering the weighted CVD of each node into its Voronoi region. The rendering algorithm is similar to `ComputeTreemap` in Algorithm 2, with the following changes. The site positions and weights are initialized from the computed node positions and weights in the Generated Tree. Therefore only one iteration of `ComputeWtWD` is performed, instead of `ComputeCVD`. Since

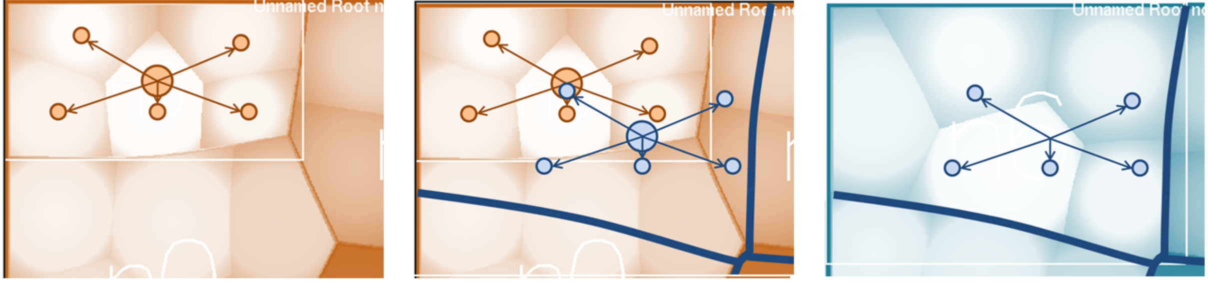

Figure 5: Re-seeding children for dynamic updates. The node's centroid and five children are initially located higher up (left). When the centroid is re-computed (center, dark line), the children are re-seeded (center). The finished state (right).

all the Voronoi diagrams are combined into a single image, the Voronoi diagram of each child is rendered inside the Voronoi of the parent, as shown in Figure 4. Therefore the function `GPUComputeMask` computes the domain of computation $\mathcal{S}_{N_i}$ by marking the pixels belonging to Voronoi region of N_i. This computation can be performed in constant time. Finally, the recursive traversal of Generated Tree can be pruned at nodes for which the projected CVD is less than 1 pixel.

V. Smooth Updates for Dynamic Data

The Voronoi treemap algorithm is amenable to dynamic data updates. In particular, the goals $\mathcal{G}$ can be dynamically altered. We extend algorithm 2 to enable data updates. While sites did not have initial positions in the previous algorithm, the treemap is now initialized with the previous layout; we wish to modify that layout by the smallest degree that will accommodate the new areas. For each CVD, we initialize the sites with the previous centroids and weights, but with the new goal areas. Once we have computed the new CVD, however, the children's centroids may no longer be located within the parents' bounds; thus, we then re-center the children based on the new computed location (Figure 5). The output of algorithm 3 is the set of new centers for the treemap's nodes. Figure 6 shows the animation in action.

Note that the algorithm itself does not have a notion of removing or adding nodes to the tree, and so does not allow the tree to be restructured. We are able to work around this by deleting nodes by shrinking them to nothing; we can add nodes by growing them from zero size. To do this, we seed a node with a minimal desired size at the time step before the node must appear; it grows through the time step.

A. Linear Interpolation to Smooth Transitions

The technique above is amenable to interpolation: we can interpolate goal weights to generate intermediate keyframe trees in between stages in our data. However, it does not make sense to interpolate for every frame of an animation:

Algorithm 3: ComputeTreemapUpdate(N, $\mathcal{S}$, ϵ): This algorithm extends Algorithm 2 given a new tree of goal areas

Input: Previous generated tree Nrooted at N_i, domain $\mathcal{S}$, error threshold ϵ

Output: Generated tree rooted at N_o with final site positions and radii

1. $\mathcal{S}_{N_o} \leftarrow$ `GPUComputeMask`(`Par`(N_o)) $\cap \mathcal{S}$
2. Initialize $\mathcal{G} \leftarrow$ `CGoals`(N_i)
3. Initialize $\mathcal{P}^w \leftarrow$ `CWtPos`(N_i)
4. `CWtPos`$(N'_o) \leftarrow$ `ComputeCVD`$(\mathcal{S}_{N_o}, \mathcal{P}^w, \epsilon)$
5. offset $\leftarrow$ `pos`(`CWtPos`(N'_o)) $-$ `pos`(`CWtPos`(N_o))
6. **foreach** *Child* C_i *of* N_i, *Child* C_o *of* N_o **do**
7.   `pos`(`CWtPos`(C_i)) $\leftarrow$ `pos`(`CWtPos`(C_i)) $+$ offset
8. $C_o \leftarrow$ `ComputeTreemapUpdate`$(C_i, \mathcal{S}, \epsilon)$
9. **end**

when `ComputeCVD` converges on a local minimum, the individual frames may oscillate.

It is possible to use a visual interpolation in the VD space: we can linearly interpolate the *centroids* and *weights* of corresponding trees. This causes Voronoi regions to slide through and past each other: each node moves smoothly on the interpolated frames, while keyframes maintain accuracy at fixed intervals. However, in some circumstances, a CVD may converge at visibly different local minima: nodes might be interchanged or moved. The linear interpolation between these may generate intermediate frames with undesirable artifacts, such as nodes sliding through each other.

It is necessary to balance these two forms of interpolation: data-based interpolation is expensive and potentially jerky, but generates keyframes that are accurate centroidal Voronoi diagrams; visual interpolation is smooth, but can lead to artifacts if the distance between data frames is too great.

VI. Implementation and Results

We have implemented our algorithm on a PC running Windows 7 with a $2.4Ghz$ Intel Core2 CPU, 4GB memory and an NVIDIA GTX260 GPU. We used DirectX9 graphics

Figure 6: Three snapshots of a stable animation. CodeRed stays at the top of the screen, and Producer stays at the bottom right, even as the yellow bottom-left subtree undergoes dramatic growth. See the accompanying video for the full animation sequence.

API, and HLSL for implementing the shaders. The discrete centroidal Voronoi diagram is computed to an offscreen surface with 32-bit floating point precision. The resolution of the compute buffer was chosen between 32×32 and 256×256, and the error threshold for convergence of CVD computation $\epsilon \approx \frac{1}{M}$, where $M \times M$ is the compute buffer resolution (note AW-CVD for each node is computed at $M \times M$ resolution, independent of depth in tree). In addition to the error threshold, a maximum number of iterations $(100-200)$ was enforced on the CVD loop in the event that the CVD is stuck in a local minimum. As an optimization in `ComputeCVD`, as the CVD begins to converge (after first few iterations), the function `VerifyBounds` is called every k^{th} iteration as the conservative bounds remain accurate. For Voronoi treemap computation and rendering, the set of nodes for CVD compute is maintained in a priority queue sorted by bounding box area. This results in the nodes with largest area being computed first. The computed Voronoi treemaps, were rendered at resolutions between 512×512 and 1024×1024.

A. Results

In this section, we illustrate the output with several visualizations, and analyze the performance of our algorithm. Figure 1 shows the management structure of a large corporation. As the video shows, users can interactively navigate the hierarchy. Figure 6 shows three snapshots from an animation based on real data. Additional results can be found in the accompanying video.

The performance of our CVD computation algorithm is presented in 7. In the top half, we show the total time to compute a CVD, averaged per iteration and separated by steps in Algorithm 1); in the bottom of the figure, we show total time for our optimizations: the `ComputeCVD` and `VerifyBounds` functions. By virtue of our optimizations, growth is sub-linear with the number of sites. The bottlenecks, instead, are in scan and update: scan is constrained by a costly readback, while update is quadratic in the number of sites. Most treemaps we have observed have used the middle grid sizes and between 5 and 50 children per node,

Table II: Times to compute and render various random treemaps. *N/Lvl*= No nodes per level (branching factor), *Lvl* = Num levels, *N* = total num nodes, *Compute* = time to compute treemap (in seconds) at 64×64 and 128×128 resolutions, *Display* = time to render the treemap (in ms) at 512×512 and 1024×1024.

N/Lvl	*Lvl*	*N*	*Compute(s)*		*Display(ms)*	
			64	128	512	1024
2	10	2047	7.1	18.9	75.3	102.4
8	4	4681	10.2	28.5	46.8	78.5
10	3	1111	2.4	6.3	23.3	52.1
10	4	11111	23.7	71.8	75.1	105.3
10	5	111111	257.1	777.2	679.2	689.7

so neither of these is the major constraint.

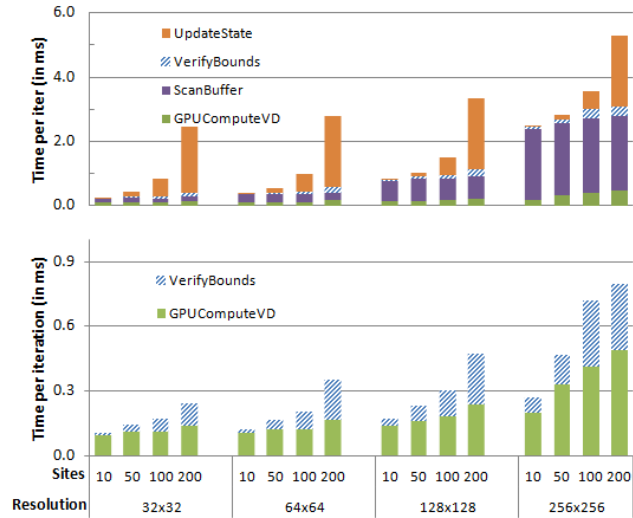

Figure 7: CVD computation timing across grid resolutions and site counts. (top) GPU and CPU computation combined. Scan includes buffer readback to CPU; Update includes `FixOverlap`. (bottom) Benefits of our coherence-based optimizations: growth of GPU operations are sublinear with sites.

For performance analysis of Voronoi treemap computation and rendering, we used randomly generated trees with branching factor of $2-10$ children per nodes per level and a random weight in the range $[1, 20]$. Our timings are presented in Table II.

B. Analysis and Discussion

The main bottleneck in Voronoi Treemap computation is the AW CVD compute for each node. For a grid size $M \times M$, with N sites, cost of 1 call to `GPUComputeWtVD` is $O(N + \sum_{i=1}^{N} \texttt{area}(B_i))$, where $\texttt{area}(\texttt{B}_\texttt{i})$ is num pixels contained in the estimated bounding box of i Voronoi region. Now $\sum_{i=1}^{N} \texttt{area}(B_i) \leq \sum_{i=1}^{N} \beta_i \texttt{area}(\mathcal{V}(\mathbf{p}_i)) \leq \beta M^2$, where $\beta_i \geq 1$ is a measure of tightness of the computed

Voronoi region bounds and $\beta = \max_N(\beta_i)$. As the CVD computation converges, `error` $\rightarrow \epsilon$, then $\beta \rightarrow 1$. Therefore cost of `GPUComputeWtVD` = $O(\beta M^2 + N), 1 \leq \beta \leq N$, where $\beta \approx N$ for first iteration and $\beta \rightarrow 1$ after first few iterations. Cost of `ScanBuffer` and `VerifyBounds` = $O(M^2)$, of `UpdateState` = $O(N)$, and `FixOverlap` = $O(N^2)$. Therefore cost per CVD iteration in Algorithm 1 = $O(\beta M^2 + N^2)$. In Algorithm 2, cost of `GPUComputeMask` for node $i = O(d_i M^2 + d_i N)$, where d_i is depth of node i ($\beta \approx 1$ for all parent nodes).

We can now perform a comparative analysis with some prior work on weighted CVD computation. Using approach of Hoff *et al.* [16] would make cost per iteration $O(NM^2 + N^2)$. Jump flooding variants [24] have cost per iteration $O(M^2 \log M + N^2)$, in addition there are no error bounds on the computed AW-CVD. The approach of Sud *et al.* [22] has better asymptotic cost for a single iteration, however has large constants due to bottleneck of reading back visibility query results from GPU. We can further improve performance of our algorithm by using our coherence based bounds in conjunction with other GPU-based weighted VD computation algorithms. CCVTs [19] have per iteration cost $O(N^2 + NM^2 \log \frac{M}{N})$, although convergence is better than our approach.

For comparison of full Voronoi Treemap computation, Balzer *et al.* [5] reported that a 4K node treemap with 10 levels takes approximately 57 CPU-minutes (7:13 min using eight 2.4Ghz CPUs) to compute. Although exact comparisons are impossible without access to identical datasets, our work represents approximately two orders of magnitude improvement for similar treemaps (see Table II).

C. Limitations

While our algorithm allows interactive rendering, navigation and animations of Voronoi treemaps, the tree cannot yet be computed at rendered rates. CVD computation is greatly affected by slow readbacks from GPU. Thus, animations must be computed in advance (or with a delay). Even during the animation, however, users can interactively zoom into the treemap and explore regions that may not be immediately visible. In addition, choice of compute resolution and error threshold is based on heuristics, which affects convergence and accuracy of the output.

Because the computation process does not guarantee convergence, in some extreme cases the algorithm terminates at a local minimum. During animation, the treemap may oscillate between local minima, causing nodes to move around on screen. The visual interpolation method helps reduce the effects of this movement.

VII. Conclusions and Future Work

The Voronoi treemap is a promising visualization technique that can allow for dynamic updates. In this paper, we have reviewed the trade-offs between dynamic data and aspect ratio. We have discussed our modifications to Balzer *et al.*'s original work, which include improved convergence, optimizations for GPUs, and smooth animations for dynamic data. We have presented results that show that we are able to navigate and render the visualization interactively.

In future work, we hope to continue to improve all aspects of the application. New paradigms for putting more computation on the GPU, using DirectX11 or CUDA will speed up the system. They will allow us to port much of Algorithm 1 to the faster GPU; this will allow us to move more of the computation to the GPU and avoid expensive GPU/CPU communication overhead. We have begun to experiment with implementing the algorithm over an optimization-based framework that will improve convergence, and adding an aspect of path planning to the animation interpolation to remove artifacts. Last, we would like to test the efficacy of animated Voronoi Treemaps against other visualization techniques through user tests.

As streaming data becomes more prevalent from a variety of sources, ranging from social media to online government, visualization tools that can accommodate large-scale hierarchical data with dynamic and stable updates will be increasingly valuable. The animated Voronoi treemap will prove to be a useful tool.

References

[1] B. B. Bederson, B. Shneiderman, and M. Wattenberg, "Ordered and quantum treemaps: Making effective use of 2d space to display hierarchies," *ACM Trans. Graph.*, vol. 21, no. 4, pp. 833–854, 2002.

[2] M. A. Smith and A. T. Fiore, "Visualization components for persistent conversations," in *CHI*, 2001, pp. 136–143.

[3] M. Wattenberg, "Visualizing the stock market," in *CHI '99: CHI '99 extended abstracts on Human factors in computing systems*. New York, NY, USA: ACM, 1999, pp. 188–189.

[4] Y. Tu and H.-W. Shen, "Visualizing changes of hierarchical data using treemaps," *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 6, pp. 1286–1293, 2007.

[5] M. Balzer and O. Deussen, "Voronoi treemaps," in *IEEE InfoVis*. Los Alamitos, CA, USA: IEEE Computer Society, 2005.

[6] M. Balzer, O. Deussen, and C. Lewerentz, "Voronoi treemaps for the visualization of software metrics," in *ACM SoftVis '05*. New York, NY, USA: ACM, 2005, pp. 165–172.

[7] B. Shneiderman, "Tree visualization with tree-maps: 2-d space-filling approach," *ACM Trans. Graph.*, vol. 11, no. 1, pp. 92–99, 1992.

[8] V. Di Maio, "Threshold effect in visual perception of geometrical figures," *Perceptual and Motor Skills*, vol. 87, pp. 340–342, 1998.

[9] R. Blanch and E. Lecolinet, "Browsing zoomable treemaps: Structure-aware multi-scale navigation techniques," *IEEE*

Transactions on Visualization and Computer Graphics, vol. 13, no. 6, pp. 1248–1253, November 2007.

[10] M. Bruls, K. Huizing, and J. van Wijk, “Squarified treemaps,” in *Proc. of Joint Eurographics and IEEE TCVG Symp. on Visualization (TCVG 2000)*. IEEE Press, 2000, pp. 33–42. [Online]. Available: citeseer.ist.psu.edu/bruls99squarified.html

[11] G. G. Robertson, S. K. Card, and J. D. Mackinlay, “Information visualization using 3d interactive animation,” *Commun. ACM*, vol. 36, no. 4, pp. 57–71, 1993.

[12] J. Heer and G. G. Robertson, “Animated transitions in statistical data graphics,” *IEEE Trans. Vis. Comput. Graph.*, vol. 13, no. 6, pp. 1240–1247, 2007.

[13] K.-P. Yee, D. Fisher, R. Dhamija, and M. A. Hearst, “Animated exploration of dynamic graphs with radial layout,” in *INFOVIS*, 2001, pp. 43–50.

[14] K. Onak and A. Sidiropoulos, “Circular partitions with applications to visualization and embeddings,” in *SCG ’08: Proceedings of the twenty-fourth annual symposium on Computational geometry*. New York, NY, USA: ACM, 2008, pp. 28–37.

[15] F. Aurenhammer, “Power diagrams: Properties, algorithms, and applications,” *SIAM Journal of Computing*, vol. 16, no. 1, pp. 78–96, 1987.

[16] K. E. Hoff, III, T. Culver, J. Keyser, M. Lin, and D. Manocha, “Fast computation of generalized Voronoi diagrams using graphics hardware,” in *Computer Graphics Annual Conference Series (SIGGRAPH ’99)*, 1999, pp. 277–286.

[17] S. P. Lloyd, “Least squares quantization in PCM’S,” *Bell Telephone Labs Memo*, 1957.

[18] Y. Liu, W. Wang, B. Lvy, F. Sun, D. M. Yan, L. Lu, and C. Yang, “On centroidal voronoi tessellation - energy smoothness and fast compu tation,” Hong-Kong University and INRIA - ALICE Project Team, Tech. Rep., 2008, accepted pending revisions.

[19] M. Balzer and D. Heck, “Capacity-constrained Voronoi diagrams in finite spaces,” in *Proceedings of the 5th Annual International Symposium on Voronoi Diagrams in Science and Engineering*, ser. Voronoï’s Impact on Modern Science, K. Sugihara and D.-S. Kim, Eds., no. 4(2), Kiev, Ukraine, September 2008, pp. 44–56.

[20] H. Li, D. Nehab, L.-Y. Wei, P. Sander, and C.-W. Fu, “Fast capacity constrained voronoi tessellation,” Microsoft Research, Tech. Rep. MSR-TR-2009-174, 2009.

[21] F. Nielsen, “An interactive tour of voronoi diagrams on the gpu,” in *ShaderX^6: Advanced Rendering Techniques*. Charles River Media (http://www.charlesriver.com/), 2008.

[22] A. Sud, N. Govindaraju, and D. Manocha, “Interactive computation of discrete generalized voronoi diagrams using range culling,” in *Proc. International Symposium on Voronoi Diagrams in Science and Engineering*, October 2005.

[23] J. Schneider, M. Kraus, and R. Westermann, “Gpu-based real-time discrete euclidean distance transforms with precise error bounds,” in *VISSAPP (1)*, A. Ranchordas and H. Araújo, Eds. INSTICC Press, 2009, pp. 435–442.

[24] G. Rong and T.-S. Tan, “Variants of jump flooding algorithm for computing discrete voronoi diagrams,” in *ISVD ’07: Proceedings of the 4th International Symposium on Voronoi Diagrams in Science and Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 176–181.

[25] C. N. Vasconcelos, A. M. Sá, P. C. P. Carvalho, and M. Gattass, “Lloyd’s algorithm on gpu,” in *ISVC (1)*, ser. Lecture Notes in Computer Science, G. B. et al., Ed., vol. 5358. Springer, 2008, pp. 953–964.

[26] B. McDonnel and N. Elmqvist, “Towards utilizing gpus in information visualization: A model and implementation of image-space operations,” *IEEE Transactions on Visualization and Computer Graphics (Proc. InfoVis 2009)*, vol. 15, pp. 1105–1112, 2009.

[27] M. Bailey, “Using gpu shaders for visualization,” *IEEE Computer Graphics and Applications*, vol. 29, no. 5, pp. 96–100, 2009.

[28] F. Aurenhammer, “Voronoi diagrams: a survey,” Tech. Univ. Graz, Graz, Austria, Tech. Rep., 1988.

[29] L. Ju, Q. Du, and M. Gunzburger, “Probabilistic methods for centroidal voronoi tessellations and their parallel implementations,” *Parallel Computing*, vol. 28, no. 10, pp. 1477–1500, 2002.

[30] K. E. Hoff III, T. Culver, J. Keyser, M. Lin, and D. Manocha, “Fast computation of generalized Voronoi diagrams using graphics hardware,” in *ACM Symposium on Computational Geometry*, 2000, pp. 375–376.

APPENDIX

Algorithm 4: UpdateState($\mathcal{P}^w$, $\mathcal{B}$, $\mathcal{G}$, $\mathcal{A}$, $\mathcal{C}$, $\mathcal{T}$, ϵ, $\mathcal{S}$): This algorithm tests for convergence and updates the state vector during 1 iteration of ComputeCVD. Constant α is a conservative bias.

Input: set of current weighted site positions $\mathcal{P}^w$, bounding boxes $\mathcal{B}$ of Voronoi regions, goal weights $\mathcal{G}$, Voronoi region areas $\mathcal{A}$, Voronoi region centroids $\mathcal{C}$, validity of Voronoi region bounding boxes $\mathcal{T}$, error threshold ϵ, domain $\mathcal{S}$

Output: boolean flag of convergence stable, and updated set of site positions and weights, and bounding boxes $\mathcal{P}^w, \mathcal{B}'$

1 $0 < \delta \ll 1; \alpha > 1$
2 $A_{tot} \leftarrow \sum_{a_i \in \mathcal{A}} a_i$
3 stable $\leftarrow true$
4 **for** $i = 1$ **to** $|\mathcal{P}^w|$ **do**
5 $\quad \mathbf{p}'_i \leftarrow \mathbf{c}_i$
6 $\quad w'_i \leftarrow w_i$
7 $\quad$ error $\leftarrow g_i - a_i/A_{tot}$
8 $\quad$ **if** $|w_i| < \delta$ **then** $w'_i \leftarrow \texttt{sign}(w_i) \cdot \delta$
9 $\quad w'_i \leftarrow w'_i + |w'_i| \cdot \frac{\text{error}}{w_i}$
10 $\quad$ **if** $w'_i \cdot\ w_i < 0$ **then**
11 $\quad\quad$ **if** $|w_i| > \delta$ **then** $w'_i \leftarrow \texttt{sign}(w_i) \cdot \delta$
12 $\quad\quad$ **else** $w'_i \leftarrow \texttt{sign}(w'_i) \cdot \delta$
13 $\quad$ **if** $t_i = true$ **then**
14 $\quad\quad d \leftarrow \frac{\|B_i.\max - B_i.\min\|}{2}$
15 $\quad\quad \max B'_i \leftarrow \max B_i.(\alpha + d \cdot \frac{\text{error}}{g_i})$
16 $\quad\quad \min B'_i \leftarrow \min B_i.(\alpha - d \cdot \frac{\text{error}}{g_i})$
17 $\quad$ **else**
18 $\quad\quad B'_i \leftarrow BB(\mathcal{S})$
19 $\quad\quad$ stable $\leftarrow false$
20 $\quad$ **if** $|\text{error}| > \epsilon$ **then** stable $\leftarrow false$
21 **end**