Fast Dynamic Voronoi Treemaps

Avneesh Sud  
eXtreme Computing Group  
Microsoft Research  
avneesh.sud@microsoft.com

Danyel Fisher  
Microsoft Research  
danyelf@microsoft.com

Huai-Ping Lee  
Dept. of Computer Science  
University of North Carolina at 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) 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

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.
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 (near 1:1) 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 [4], [5].

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, users can lose track of them [12], [13].

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/narrowing 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.

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 [5], [6] 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. To our knowledge, Circular Partition treemaps have not been constrained to allow animation over 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. The additive weighted Voronoi diagram can be computed in optimal time in continuous 2D domain [15], and efficiently in discrete domains using a GPU [16]. Lloyd [17] presents a practical approach for computing cent (non-weighted) Voronoi diagrams using iterative updates. Mioc et al. present a spatio-temporal Voronoi data structure and formal grammar for dynamic updates of Voronoi maps [18] and application to additive weighted Voronoi diagrams [19].

For Voronoi treemap computation, the area of each weighted Voronoi region must also be computed. 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

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Avg Aspect Ratio</th>
<th>Stable updates</th>
<th>Stable zoom</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slice&amp;Dice [7]</td>
<td>100+</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>Squarified [10]</td>
<td>1.3</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>Ordered [1]</td>
<td>2.8</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Strip [1]</td>
<td>2.5</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Spiral [4]</td>
<td>2.5</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Voronoi</td>
<td>1.3*</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

* Based on the aspect ratio of the bounding boxes of the Voronoi regions.
computation [20]. Recently, Balzer et al. [21] have presented Capacity-Constrained Voronoi Diagrams (CCVDs) for computing power (multiplicative-weighted) CVDs, and performance improvements are presented in [22]. 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 [23]. Performance optimizations for 2D error-bounded discrete non-weighted Voronoi diagrams on the GPU are presented in [24]–[26]. A GPU-based implementation of CVDs is described in [27].

In the last year, a variety of projects have begun to examine the uses of GPU computation for information visualization. McDonnell and Elmqvist [28] suggest using GPU techniques to rapidly render nodes for graphs and charts, and provide a graphics pipeline; similarly, Bailey [29] 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 \( X \subset \mathbb{R}^2 \) represent a compact domain, and \( BB(X) \) denote an axis-aligned bounding box of \( X \). Let \( \mathcal{P} = \{p_1, \ldots, p_n\} \) 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 \textit{sites}. Given a distance function \( d(p, q) \) between two points \( p, q \in \mathbb{R}^2 \), the \textit{Voronoi region} of a site \( p_i \) is the set of points closer to \( p_i \) than to any other site:

\[
\mathcal{V}(p_i, \mathcal{P}) = \{q \in \mathbb{R}^2 \mid d(q, p_i) \leq d(q, p_j) \forall p_j \in \mathcal{P}, j \neq i\}.
\]

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

\[
VD(\mathcal{P}, \mathcal{S}) = \bigcup_{p_i \in \mathcal{P}} \mathcal{V}(p_i, \mathcal{P}) \cap \mathcal{S}
\]

**Weighted Voronoi Diagrams:** Typically, the distance function \( d(p, q) \) is the Euclidean distance function. Using other distance functions for computing Voronoi regions results in generalized Voronoi diagrams [30]. Given a set of weights \( \mathcal{W} \), a unique weight \( w_i \in \mathcal{W} \) is assigned to each site \( p_i \). Using weighted distance functions, a \textit{Weighted Voronoi Diagram} \( VD(\mathcal{P}, \mathcal{S}, \mathcal{W}) \) is generated. An \textit{additive weighted (AW) Voronoi Diagram} \( VD_{aw}(\mathcal{P}, \mathcal{S}, \mathcal{W}) \) is generated by adding a weight to the distance function,

\[
d_{aw}(q, p_i, w_i) = \|q - p_i\| - w_i
\]

This is similar to generating a regular Voronoi diagram where the \( i^{th} \) 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 \( \mathcal{P} \) 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,

\[
p_i = c_i = \frac{\int_{\mathcal{V}(p_i, \mathcal{P})} xd\sigma}{\int_{\mathcal{V}(p_i, \mathcal{P})} d\sigma}
\]

where \( d\sigma \) is the area differential. The CVD minimizes the inertia momentum (and increases the compactness) of each Voronoi region [20], 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:** For purpose of visualization, an approximate Voronoi diagram accurate up to pixel precision is sufficient. Hence, a discrete Voronoi diagram is computed by sampling the domain \( \mathcal{S} \) at a finite set of points \( \mathcal{S} \), and computing the membership of each sample in \( \mathcal{S} \) to a Voronoi region. This computation may be done with random samples [31] or with uniform samples using graphics hardware [32].

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 \( N \) denote a node in a data tree, and the functions \( \text{Par}(N) \), \( \text{Children}(N), \text{Goals}(N) \) 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 region 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 CWtPos(N) be a list of the position and weights of the child nodes of a node N in 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. [24], 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. 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 IV.

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 COMPUTATION AND RENDERING

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 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 A and centroid C of each Voronoi region. In addition the bounding boxes 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, weights and estimated Voronoi region bounds for each site. The position is updated similar to the Lloyd’s algorithm [17].
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.

We extend Algorithm 2 in Balzer et al. [5] to compute the new weights, but add new heuristics to provide stable convergence in certain cases when there is large disparity of goal areas. Finally, the function also predicts new Voronoi region bounds (Fig. 2). 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. [24] 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 $S$ is scaled to be a subset of $[0, \sqrt{2}/4] \times [0, \sqrt{2}/4]$. 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 distance 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 $w_{\text{min}}$ and $w_{\text{max}}$ be the minimum and maximum weights among all sites, and $D_{\text{max}}$ be the maximum Euclidean distance among all pairs of sites ($D_{\text{max}} \leq 0.5$ from the domain definition). We now define the affine transforms on distance function between a point $q$ and site $p_i$, and weight $w_i$,

$$d_{aw}'(q, p_i) = ||q - p_i|| + w_i'$$

$$w_i' = \frac{w_{\text{max}} - w_i}{2(w_{\text{max}} - w_{\text{min}})}$$

It can be shown that $0 \leq w_i' \leq 0.5$ and $0 \leq d_{aw}'(q, p_i) \leq 1$. This 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 10-19 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 $q_1$ and $q_2$ are on the boundary of $i^{th}$ and $j^{th}$ Voronoi regions, namely $q_1 \in \mathcal{V}(p_i; P)$ and $q_2 \in \mathcal{V}(p_j; P)$, and the size of a pixel is $\|q_2 - q_1\| = \delta$. Then both regions are valid if

$$|d_{aw}(q_1, p_i, w_i) - d_{aw}(q_2, p_j, w_j)| < \delta.$$ 

Conversely, the bounding box for the $i^{th}$ site is invalid if $d_{aw}(q_2, p_j, w_j) - d_{aw}(q_1, p_i, w_i) > \delta$, and the bounding box for the $j^{th}$ site is invalid if $d_{aw}(q_1, p_i, w_i) - d_{aw}(q_2, p_j, w_j) > \delta$. If the predicted bound is invalid, then the bounding box reset to the entire domain, and an additional CVD iteration is performed.

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 $p_j$ does not affect the area of another site $p_i$). Due to these assumptions, the prior approach fails to converge stably when there is large disparity between the maximum and minimum goal areas. Such scenarios were common in our real-world datasets.

In our algorithm, we present a new heuristic for the site weights to provide a more stable convergence of CVD iterations. Although this approach does not provide theoretical
bounds on convergence, in our experiments it converged with small error for most datasets. The detailed algorithm is presented in Algorithm 4 (Appendix).

**Figure 3:** 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

**Algorithm 2:** ComputeTreemap(N, 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 S, and error threshold ϵ

```
1 S_N ← GPUComputeMask(N_i) ∩ S
2 Initialize G ← CGoals(N_i)
3 Initialize P^w to random points in S_N and unit weight
4 P^w ← ComputeCVD(S_N, G, P^w, ϵ)
5 CWtPos(N_o) ← P^w
6 foreach Child C_i of N_i; Child C_o of N_o do
7   C_o ← ComputeTreemap(C_i, S, ϵ)
8 end
```

**B. Treemap Computation**

The complete Voronoi treemap is computed by recursively traversing the tree bottom-up and computing the weighted CVD, highlighted in Algorithm 2. For a given node N_i at depth d in the tree, the positions of the child nodes CPos(N_i) are treated as the set of sites P, and the domain of computation 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 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 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 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 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 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 ComputeWtVD is performed, instead of ComputeCVD. Since 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 Fig. 3. Therefore the function GPUComputeMask computes the domain of computation S_{N_i} by marking the pixels belonging to Voronoi region of N_i. This computation can be performed in constant time using a single render pass to the stencil buffer. 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 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 (Fig. 4). The output of algorithm 3 is the set of new centers for the treemap’s nodes. Fig.5 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
Figure 4: 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).

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

\begin{algorithm}
\caption{ComputeTreemapUpdate($N$, $S$, $\epsilon$)}
\begin{algorithmic}
\State $S_{N_o} \leftarrow \text{GPUComputeMask}(\text{Par}(N_o)) \cap S$
\State Initialize $G \leftarrow \text{CGoals}(N_i)$
\State Initialize $P^w \leftarrow \text{CwPos}(N_i)$
\State $\text{CwPos}(N'_o) \leftarrow \text{ComputeCVD}((S_{N'_o}, P^w, \epsilon)$
\State $\text{offset} \leftarrow \text{pos}(\text{CwPos}(N'_o)) - \text{pos}(\text{CwPos}(N_o))$
\ForAll {Child $C_i$ of $N_i$, Child $C_o$ of $N_o$}
\State $\text{pos}(\text{CwPos}(C_i)) \leftarrow \text{pos}(\text{CwPos}(C_i)) + \text{offset}$
\State $C_o \leftarrow \text{ComputeTreemapUpdate}(C_i, S, \epsilon)$
\EndFor
\end{algorithmic}
\end{algorithm}

Figure 5: 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.

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 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 $32 \times 32$ and $256 \times 256$, 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 $512 \times 512$ and $1024 \times 1024$.

A. Results

In this section, we illustrate the output with several visualizations, and analyze the performance of our algorithm. Fig. 1 shows the management structure of a large corporation. As the video shows, users can interactively navigate the hierarchy. Fig. 5 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 6. 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 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, so neither of these is the major constraint.

For performance analysis of Voro

<table>
<thead>
<tr>
<th>N/Lvl</th>
<th>Lvl</th>
<th>N</th>
<th>Compute(s)</th>
<th>Display(ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>10</td>
<td>2047</td>
<td>7.1</td>
<td>18.9</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>4681</td>
<td>10.2</td>
<td>28.5</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>1111</td>
<td>2.4</td>
<td>6.3</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>11111</td>
<td>24.7</td>
<td>71.8</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
<td>111111</td>
<td>257.1</td>
<td>777.2</td>
</tr>
</tbody>
</table>

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 $64 \times 64$ and $128 \times 128$ resolutions, Display = time to render the treemap (in ms) at $512 \times 512$ and $1024 \times 1024$. 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 $O(N + \sum_{i=1}^{N} \text{area}(B_i))$, where $\text{area}(B_i)$ is num pixels contained in the estimated bounding box of $i$ Voronoi region. Now $\sum_{i=1}^{N} \text{area}(B_i) \leq \sum_{i=1}^{N} \beta \text{area}(V(p_i)) \leq \beta M^2$, where $\beta \geq 1$ 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 $O(N M^2 + N^2)$. Jump flooding variants [26] 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. [24] 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 [21] 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 incorporate the dynamic Voronoi diagram data structure for animation of Voronoi treemaps [18]. 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.

VIII. ACKNOWLEDGEMENTS

We thank Naga Govindaraju, Kori Inkpen Quinn, Mary Czerwinski, John Lambert, and Chinnay Kulkarni for their valuable insights and support. We thank the anonymous reviewers for several useful additions.

REFERENCES


### Appendix

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

**Input:** set of current weighted site positions $P^w$.

- bounding boxes $B$ of Voronoi regions, goal weights $G$, Voronoi region areas $A$, Voronoi region centroids $C$, validity of Voronoi region bounding boxes $T$, error threshold $\epsilon$, domain $S$.

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

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