Eurographics Conference on Visualization (EuroVis) 2012
S. Bruckner, S. Miksch, and H. Pfister
(Guest Editors)

Volume 31 (2012), Number 3

Computing Voronoi Treemaps

Faster, Simpler, and Resolution-independent

Arlind Nocaj and Ulrik Brandes

Department of Computer & Information Science, University of Konstanz

Abstract

Voronoi treemaps represent hierarchies as nested polygons. We here show that, contrary to the apparent popular belief, utilization of an algorithm for weighted Voronoi diagrams is not only feasible, but also more efficient than previous low-resolution approximations, even when the latter are implemented on graphics hardware. More precisely, we propose an instantiation of Lloyd's method for centroidal Voronoi diagrams with Aurenhammer's algorithm for power diagrams that yields an algorithm running in $\mathcal{O}(n \log n)$ rather than $\Omega(n^2)$ time per iteration, with n the number of sites. We describe its implementation and present evidence that it is faster also in practice.

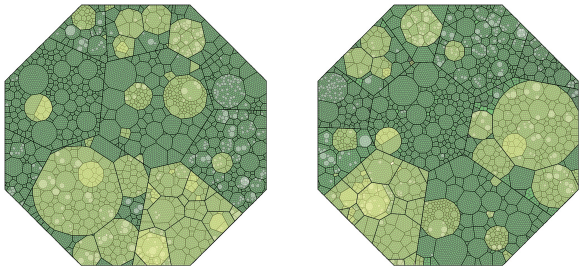
Categories and Subject Descriptors (according to ACM CCS): H.5.2 [Information Interfaces and Presentation]: User Interfaces; I.3.6 [Computer Graphics]: Methodology and Techniques; I.3.8 [Computer Graphics]: Applications

1. Introduction

Much data is either inherently hierarchical, or purposefully made hierarchical for comprehension, abstraction, or interaction. The hierarchical inclusion relations can be represented in a rooted tree, where singleton sets of base elements form the leaves, and each inner node represents the union of the sets represented by its children. Treemaps have been proposed as a space-filling representation of such inclusion-hierarchy trees [Shn92]. Each node of the hierarchy tree is depicted by a rectangle, and rectangles are subdivided recursively into smaller rectangles depicting the children of the corresponding tree node. In most applications, base elements have associated weights, and the area of a rectangle is required to be proportional to the total weight of the corresponding subset. There are several other degrees of freedom in this representation, in particular with respect to layout and rendering, and many of them have been utilized in various applications [Shn09].

A special variant are Voronoi treemaps [BD05], in which more general polygons are used instead of rectangles. These polygons are defined as the regions of centroidal Voronoi diagrams, and the resulting visualizations have good aspect ratio and an appealing, organic look to them. Moreover, they are more robust to changes in time-varying hierarchical data.

Despite the extensive praise they are receiving, Voronoi treemaps are not as frequently used in visualization applications [Got11, HTS09, BFHS09]. Adoption appears to be slowed by their seemingly cumbersome implementation and high computational demands. Common algorithms are based on resolution-dependent approximate computations



(a) Fig. 11 of [BD05] (impl. in C, 5:48 min)

(b) our method (impl. in Java, 0:35 min)

Figure 1: *A hierarchical software structure (Fujaba Tool Suite, 16K nodes). Since no ordering is imposed, comparison should be according to relative area sizes only. Timings on same machine (8 Core Intel Xeon E5345 CPU@2.4 GHz). Our implementation requires 12.5 seconds on more up to date hardware (4 Core Intel Core i7-2600K CPU@3.40GHz).*

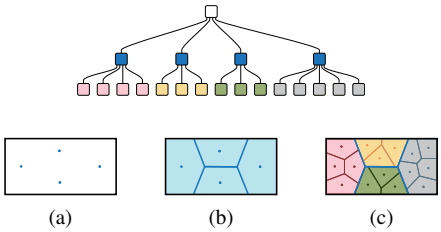


Figure 2: *A hierarchical partition with uniform leaf values (top) and recursive construction of a Voronoi treemap (a–c). Dots correspond to the sites which generate the cells.*

and state of the art appears to be the approach of [SFL10] which achieves run-time improvements over other variants by utilizing fast parallel processing in graphics hardware. We here demonstrate that a carefully revised, yet relatively simple implementation using a conventional algorithm from computational geometry, Aurenhammer's method for power diagrams [Aur87], actually outperforms even the parallelized approximation while also removing the resolution limit.

The remainder is organized as follows. We first recall the definition of Voronoi diagrams and Voronoi treemaps. In Section 3, we then review previous approaches for their computation. The approach proposed here is presented in Section 4 and evaluated in Section 5. We conclude with a brief discussion.

2. Background

2.1. Voronoi Diagrams

Given a set S of n distinct points in the plane, called *sites*, the corresponding Voronoi diagram divides the plane into regions, one for each site. Each region, called *(Voronoi) cell*, consists of exactly those points that have the same closest site. Since display space is usually bounded, we consider bounded Voronoi diagrams that divide up some convex area (e.g., a rectangle) rather than the entire plane.

Formally, we are given a bounded, convex area $\Omega \subset \mathbb{R}^2$ and a set of sites $S = \{s_1, \ldots, s_n\}$. For each point $s_i \in S$ its associated cell $\mathcal{V}(s_i)$ is defined as

$$\mathcal{V}(s_i) = \{p \in \Omega : \|p - s_i\| < \|p - s\| \text{ for each } s \in S - s_i\} \quad (1)$$

where $\|p_1 - p_2\| = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ is the Euclidean distance of points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$. Each cell $\mathcal{V}_s = \mathcal{V}(s)$ is bordered by a polygon $\overline{\mathcal{V}_s}$ of points that have equal distance to at least two sites, or belong to the boundary of Ω. Furthermore the area of a cell (region) is denoted by $A(V_s)$.

An (ordinary) Voronoi diagram is thus defined as the collection of cells,

$$\mathcal{V}(S) = \{\mathcal{V}(s_1), \ldots, \mathcal{V}(s_n)\} \ .$$

It can be computed in $\mathcal{O}(n \log n)$ time using any of a variety of algorithms [GO04].

2.2. Area Requirements

The area of a cell depends on the relative position of its associated and neighboring sites. Since the cells of a Voronoi diagram are going to be used to depict numerical data elements, we require a mechanism to control their area better.

To have the possibility of influencing the size of the cell area, weighted Voronoi diagrams can be used. Let $W(S) = \{w_1, w_2, \ldots, w_n\}$ be a set of positive real *weights* $w_i = w(s_i) \in \mathbb{R}_{>0}$ associated with sites $S = \{s_1, \ldots, s_n\}$.

Two generalizations of Voronoi diagrams that take weights into account are common, both defined by substituting a weight-dependent distance for the Euclidean distance in (1). In the *additively weighted* (AW) Voronoi diagram [For87], $\|p - s_i\|$ is replaced by $\|p - s_i\| - w_i$. Note that the boundaries between pairs of sites thus become hyperbolic curves. In the *power weighted* (PW) Voronoi diagram [Aur87], or *power diagram* for short, $\|p - s_i\|^2 - w_i$ is used instead and boundaries remain polygonal.

In either variant, increasing a single weight increases the area of the associated cell. While the relation between weights and areas is monotonic, it is in general non-linear. Note also that cells are empty when the defining site is "closer" (w.r.t. to the modified distances) to another site than to itself. This does not happen in ordinary Voronoi diagrams.

We here restrict our attention to power diagrams, but our results are likely to extend to additively weighted Voronoi diagrams because both can be computed in $\mathcal{O}(n \log n)$ time using algorithms of similar practical performance [Aur87, For87].

2.3. Centroidal Voronoi Diagrams

In addition to accuracy in the representation of magnitudes, we would like to ensure good readability. An important readability aspect, among other things supporting the visual comparison of areas, is the aspect ratio of cells, i.e. the ratio of the lengths of sides of the smallest enclosing rectangle. Aspect ratio can be changed by moving sites, and it has been shown that, for ordinary Voronoi diagrams, aspect ratio close to one is achieved when sites are evenly distributed and located in the centroid (the center of mass) of their cell [LWL*09]. The latter property defines *centroidal* Voronoi diagrams (CVDs).

Note that the subdivison of a rectangle by parallel lines as in the slice-and-dice technique [Shn92] also yields, in fact, centroidal Voronoi diagrams (the sites would have to be placed at the center of each rectangle), but the resulting subrectangles are often long and thin and thus have very poor aspect ratio.

2.4. Voronoi Treemaps

A Voronoi treemap is the recursive subdivision of a region into the cells of a centroidal Voronoi diagram defined as follows.

Let $O = \{o_1, \ldots, o_n\}$ be a set of objects, each with an associated positive real value $v_i \in \mathbb{R}_{>0}$, $i = 1, \ldots, n$. We define $v : 2^O \to \mathbb{R}_{>0}$ as the additive extension to subsets of $P \subseteq O$,

$$v(P) = \sum_{i\,:\,o_i \in P} v_i \ .$$

A hierarchical partition of O is a rooted tree $T = (\mathcal{P}, I; r)$ with nodes $\mathcal{P}$ representing subsets of O and edges I representing set inclusion. The root $r \in \mathcal{P}$ represents O and the leaves represent exactly the singleton sets $\{o_i\}$, $i = 1, \ldots, n$. We will identify nodes in the tree with the subsets they represent. Each inner node represents the set formed by the union of the sets of its children.

A hierarchical partition is represented by a Voronoi treemap, if the bounding region represents the entire set of objects, this region is subdivided by a centroidal Voronoi diagram with one cell per child of the root and these cells are subdivided recursively such that the leaves are represented by cells with an area (approximately) proportional to their value. The children of $p \in \mathcal{P}$ are given by $children(p) = \{x \in \mathcal{P} : \ x \subset p\}$. We denote by $A(\mathcal{V}_p) \cdot \frac{v(c)}{v(p)}$ the target area of $c \in children(p)$. See Figure 2 for an example.

3. Previous Approaches

Because of the recursive nature of Voronoi treemaps, it is sufficient to consider a single level of the computation: given a bounded region representing an inner node of a hierarchical partition, subdivide this region by a Voronoi diagram such that each cell corresponds to exactly one child.

To address aspect ratio and area requirements, weighted centroidal Voronoi diagrams are used instead of ordinary ones. Following Balzer and Deussen [BD05], the core scheme for their computation is Lloyd's iterative method for centroidal Voronoi diagrams augmented by weight adaptation to control cell areas. We therefore outline Lloyd's method first, and then present the various approaches building on it.

3.1. Lloyd's Method

Originally, Lloyd's method [Llo57] is designed to determine a centroidal Voronoi diagram inside a bounded region $\Omega \subset \mathbb{R}^2$ such that the density of sites approximates a given density $\rho : \Omega \to \mathbb{R}_{>0}$.

Starting from random initial sites S, an ordinary Voronoi diagram is determined. Then, each site $s \in S$ is moved into the centroid

$$\frac{\int_{\mathcal{V}(s)} \rho(p)\, p \, \mathrm{d}p}{\int_{\mathcal{V}(s)} \rho(p) \, \mathrm{d}p},$$

of its cell, and the Voronoi diagram is recomputed. This process is repeated until an approximately centroidal Voronoi diagram is obtained.

In our present scenario, however, there is no variation in density. While this simplifies the calculation of centroids to the average of the corners of each cell, it also causes the limit solution to be a centroidal Voronoi diagram with evenly distributed sites and thus (near) equal-area cells. The methods described in the next section therefore adapt Lloyd's method to accommodate weights.

3.2. Methods for Voronoi Treemaps

We are not aware of any method to determine a centroidal weighted Voronoi diagram directly from a set of area requirements, even if a solution satisfying one of the two properties is given.

The central idea of Balzer and Deussen [BD05] is to apply Lloyd's method to *weighted* Voronoi diagrams to make these centroidal, and to adapt weights during the iteration to meet the area requirement. After each iteration, weights are increased or decreased proportionally to the missing or excess area.

However, because the computation of weighted Voronoi diagrams is found to be "non-trivial" [BDL05], they resort to sample-based approximation. That is, a sufficiently dense number of sample points is tested for their closest site according to the weighted distance function, and sites are subsequently moved to the average of all sample points closest to them. This approach has two major drawbacks with respect to efficiency: it introduces a resolution problem (where higher resolution corresponds to more sample points and thus higher running times) and the cell boundaries are not obtained as part of the process. The sampling needs quadratic running time and thus asymptotically dominates the time spent in each iteration. To actually construct the Voronoi diagram, pairwise boundaries are determined, irrelevant ones discarded, and the remaining ones clipped. This again has at least a quadratic running time and slows down the overall process.

Sud et al. [SFL10] present an improved implementation of the sampling approach using graphics hardware and General Purpose GPU techniques. While actual running times are reduced significantly, the advantage of parallelization on the graphics card could not be used for all parts of the algorithm and the asymptotic running time for a single iteration remained quadratic in the number of sites. Several other authors have worked on computing centroidal Voronoi diagrams completely on GPU [VSCG08, RLW*11], but we are

not aware of one such approach that also allows for weight adaptation.

Instead of computing the treemap level by level, Gotz [Got11] suggests to interleave the iterative computations on all levels. The argument being more flexible reactions to changes in the data, but it seems that the constant change of bounding regions increases the number of iterations rather dramatically. It is interesting to note, though, that in this approach a combinatorial algorithm is used for computing weighted Voronoi diagrams. Unfortunately, its adaptation for Lloyd's method is not described and no running times are reported.

Applications using Voronoi treemaps generally stick to the original computation scheme [HTS09, BFHS09], and authors therefore note that running times are a major limitation.

Let us summarize the main drawbacks of previous approaches as follows:

(speed) at least quadratic running time
(resolution) samples determine accuracy and level of detail
(hardware) practical running times require parallelization

We show in the next section that a careful adaptation of a conventional analytic algorithm with no resolution limit outperforms even the hardware-accelerated approach at very low resolution.

4. Resolution-independent Algorithm

In this section, we introduce our modified Voronoi treemap algorithm. While we stick to the overall scheme of Balzer and Deussen in which Lloyd's method is applied to weighted Voronoi diagrams, our approach differs in three main aspects:

- an analytic algorithm for power diagrams
- a modified update scheme for sites and weights
- a heuristic to reduce the number of iterations

As a byproduct, implementation complexity is reduced as well. The initialization of the sites depends strongly on the application, as an example we use random positions in Ω.

Although we keep Lloyd's method and the weight change step, we need to handle them separately to have non empty regions after each iteration, see Algorithm 1. By doing this separation we could possibly use the $L-BFGS$ optimization method which has been shown to converge faster for the ordinary Voronoi diagram [LWL*09] than Lloyd's method, but its influence on the weighted Voronoi diagram and the area requirements are not known. We thus keep Lloyd's method in this step to be able to compare our implementation with previous work.

4.1. Description

After starting with an initialization we improve the layout in each iteration of the for-loop. In `AdaptPositionsWeights` (line 6) the sites are moved to the centroid but at the same time the weights are decreased if necessary. In `AdaptWeights` (line 8) the weights are changed in such a way that the areas are improved in each iteration. Further the weight of each site $s \in S$ is limited by its nearest neighbor NN_s (line 13 and line 14). After each adaption of the diagram we need to recompute the power diagram (line 7 and line 9). This is necessary since Lloyd's method is a different optimizer than our area error reduction step (`AdaptWeights`). If the area error is below a certain threshold we can cancel the optimization process (line 11).

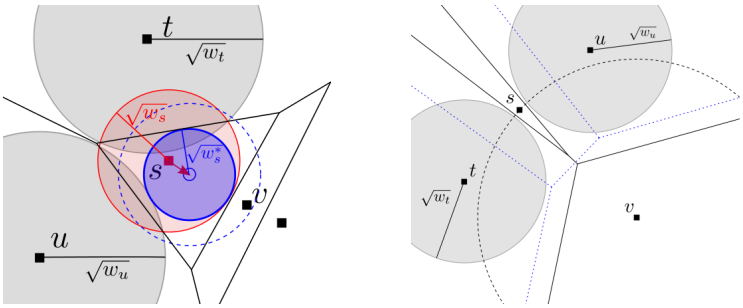
Algorithm 1: Compute Voronoi Treemap (single layer)

Input: $\Omega \subset \mathbb{R}^2$: convex polygon , $p \in \mathcal{P}$: hierarchy node with n children, $i_{\max}$: maximal iteration number, $E_{\text{threshold}}$: error threshold
Output: Voronoi diagram $\mathcal{V}(S)$ with n polygons

1 init. sites $S = children(p)$ with unique positions in Ω
2 init. weights W with small constant ε
3 $error \leftarrow \infty$
4 $\mathcal{V}(S) \leftarrow$ `ComputePowerDiagram` (Ω, S, W)
5 **for** $i \leftarrow 1$ **to** $i_{\max}$ **do**
6 `AdaptPositionsWeights` $(p, \mathcal{V}(S), S, W)$
7 $\mathcal{V}(S) \leftarrow$ `ComputePowerDiagram` (Ω, S, W)
8 `AdaptWeights` $(p, \mathcal{V}(S), S, W)$
9 $\mathcal{V}(S) \leftarrow$ `ComputePowerDiagram` (Ω, S, W)
10 $error \leftarrow \frac{\sum_{s \in S} A(\mathcal{V}_s) - A(\Omega) \cdot \frac{v(s)}{v(p)}}{2 \cdot A_\Omega}$
11 **if** $error < E_{\text{threshold}}$ **then return** $\mathcal{V}(S)$
12 **return** $\mathcal{V}(S)$

Algorithm 2: Update steps

1 `AdaptPositionsWeights` $(p, \mathcal{V}(S), S, W)$
2 **foreach** *site* $s \in S$ **do**
3 $s \leftarrow centroid(\mathcal{V}_s)$
4 $distanceBorder \leftarrow \min_{x \in \overline{\mathcal{V}_s}} \|x - s\|$
5 $w_s \leftarrow (\min(\sqrt{w_s}, distanceBorder))^2$
6 `AdaptWeights` $(p, \mathcal{V}(S), S, W)$
7 $NN \leftarrow$ `Nearestneighbor` (S)
8 **foreach** *site* $s \in S$ **do**
9 $A_{\text{current}} \leftarrow A(\mathcal{V}_s)$ `/* current area */`
10 $A_{\text{target}} \leftarrow A(\Omega) \cdot \frac{v(s)}{v(p)}$ `/* target area */`
11 $f_{\text{adapt}} \leftarrow \frac{A_{\text{target}}}{A_{\text{current}}}$
12 $w_{\text{new}} \leftarrow \sqrt{w_s} \cdot f_{\text{adapt}}$
13 $w_{\max} \leftarrow \|s - NN_s\|$
14 $w_s \leftarrow (\min(w_{\text{new}}, w_{\max}))^2$
15 $w_s \leftarrow \max(w_s, \varepsilon)$



(a) Conflict in `AdaptPositionsWeights`

(b) Conflict in `AdaptWeights`

Figure 3: *Example of two weighted power diagrams and conflicts when updating the sites and their weights. Circles represent the weight and have thus $\sqrt{w_p}$ as radius for a site $p \in S$. (a) Site s (with red circle) is moved to the centroid of its Voronoi cell, which could cause in combination with other site movements the site v to have an empty cell (dotted blue circle contains v). Conflict is solved by reducing the radius of s to $\sqrt{w_s^*}$ (blue circle) if necessary (line 5 of Algorithm 2). (b) Site s is not a neighbor of v in the weighted Voronoi diagram (black lines), but is one in the ordinary Voronoi diagram. Increasing the weight (dotted black circle) of v has thus to be limited by its neighbors in the ordinary Voronoi diagram or by its nearest neighbor (line 14 of Algorithm 2).*

4.2. Complexity

For the complexity analysis we say $n = |S|$ and k is the number of iterations which are made. We first look at the complexity of the update steps (Algorithm 2). Since the Voronoi diagram of n sites has linear complexity, the computation of the centroids for all sites needs $\mathcal{O}(n)$ time. To compute the minimal distance of the centroid to the border of the corresponding cell, one has to determine the smallest distance of the centroid to the set of segments (bisectors) which define the corresponding Voronoi cell. Note that this can be done in constant time for a single segment. For each segment we have to compute the minimal distance to two sites, which needs $\mathcal{O}(n)$ time due to the linear number of segments. We can therefore say that `adaptPositionsWeights` needs $\mathcal{O}(n)$ time and space.

When adapting the weights with `AdaptWeights` we need to compute the nearest neighbor of every site in S, which takes $\mathcal{O}(n \log n)$ by e.g., computing the ordinary Voronoi diagram (power diagram with zero weights). Since all the other steps in the for-loop (line 8) are trivial we need $\mathcal{O}(n \log n)$ time for `AdaptWeights`.

For Algorithm 1 we can now follow that the runtime for a single iteration $\mathcal{O}(n \log n)$ and due to the for-loop the overall runtime is $\mathcal{O}(k \cdot n \log n)$.

4.3. Correctness

In this section we prove the correctness by showing that no Voronoi diagram with empty regions is generated by Algorithm 1.

The following invariant helps us proving the correctness:

$$\forall s,t \in S, s \neq t : \|s-t\| > \max(\sqrt{w_s}, \sqrt{w_t}) \qquad (2)$$

We first show some helpful lemmas.

Lemma 1 Invariant (2) holds for Algorithm 1

Proof By choosing unique coordinates in Ω and $\varepsilon < \min_{p,q \in S}(\|p-q\|)$, Equation 2 holds after the initialization. Since the use of `ComputePowerDiagram()` in Algorithm 1 does not change the sites or weights, it cannot break the invariant.
Since each Voronoi cell of a bounded power diagram is convex, the centroid has to lie in the cell. By moving a site to the centroid of its cell `AdaptPositionsWeights` the invariant could be broken. By decreasing the corresponding weight in line 5 to the distance of the site to its closest bisector the following holds for two neighboring sites $s,t \in S$:

$$\|s-t\| \geq w_s + w_t > \max(w_s, w_t). \qquad (3)$$

This further means that $\sqrt{w_s}$ for a site $s \in S$, can at most be the minimal distance of the site to its Voronoi cell. Since every other site $t \in S$ lies in its own cell $\mathcal{V}_t$ it also lies outside of the cell $\mathcal{V}_s$, thus the distance between s and t is larger than the minimal distance of s to its cell border, $\|s-t\| > \sqrt{w_s}$. Respectively $\|s-t\| > \sqrt{w_t}$ holds, therefore $\|s-t\| > \max(\sqrt{w_s}, \sqrt{w_t})$ is valid after `AdaptPositionsWeights()`.
The limitation of the weight in line 13 further guarantees that the invariant holds after the call of `AdaptWeights()`.
□

Lemma 2 Invariant (2) $\Rightarrow \forall s \in S : \mathcal{V}_s \neq \emptyset$

Proof (by contradiction) Assume there is a site $s \in S$ for which $\mathcal{V}_s = \emptyset$, which means that the point $p = (s_x, s_y) = s$ has a smaller distance to another site $t \in S, t \neq s$ and $w_t > w_s$:

$$\begin{aligned} \|p-t\|^2 - w_t &< \|p-s\|^2 - w_s \\ \|s-t\|^2 &< w_t - w_s \; < w_t = \max(w_s, w_t) \\ \|s-t\| &< \max(\sqrt{w_s}, \sqrt{w_t}) \end{aligned}$$

But this is a contradiction to our assumption that $\forall s,t \in S, s \neq t : \|s-t\| > \max(\sqrt{w_s}.\sqrt{w_t})$. □

Theorem 3 Algorithm 1 does not produce empty Voronoi cells and is thus correct.

Proof This follows directly from Lemma 1 and Lemma 2. □

4.4. Computation of the Power Diagram

It will turn out that the power diagram is easy to compute by using any convex hull algorithm in 3D.

Aurenhammer [Aur87] describes how the general $d-$dimensional power diagram can be computed by using a transformation to the $d+1$ dimensional space. In the same breath he further generalizes his method for higher-order Voronoi diagrams. This sweeping blow results in a very general description and makes it hard to use his algorithm without intensively dealing with computational geometry.

We now reduce the complex algorithm to the important steps for the two dimensional case. It turns out that for the two dimensional case one has mainly to combine several easy transformations with a convex hull computation in 3D. The main steps are:

- transform the weighted sites in 2D to half-planes in 3D
- compute the lower envelope of the half-plane intersections (convex hull)
- map the dual solution back by reversing the applied transformation

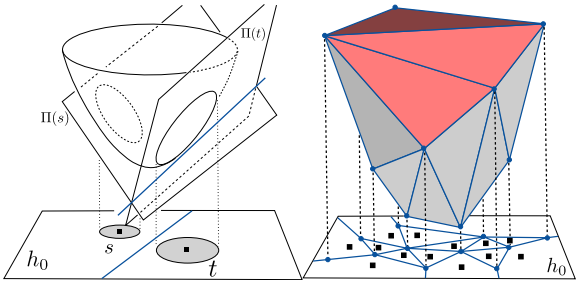
4.4.1. Transformation from 2D to 3D

This transformation uses the property that the power function in 2D can be described as a plane in three-dimensional space. The plane h_0 in $\mathbb{R}^3$ is the plane which is spanned by the x and y-axis, $z = 0$. Let $s = (x_s, y_s) \in S$ and w_s the corresponding weight. Each site lies on h_0, see Figure 4a.

The transform Π maps a sphere with coordinates of s and radius $r = \sqrt{w_s}$ into the three-dimensional plane

$$\Pi(s) : z = 2\begin{pmatrix} x & y \end{pmatrix} \cdot \begin{pmatrix} x_s \\ y_s \end{pmatrix} - \left({x_s}^2 + {y_s}^2 \right) + r^2 \qquad (4)$$

The most important property of Π is that the vertical projection of $\Pi(s) \cap \Pi(t)$ is the bisector of the sites s and t in 2D, see Figure 4 for an illustration, or see section 4.1 in [Aur87] for the proof. The corners and edges of the lower envelope correspond to the vertices and the bisectors of the Voronoi diagram. Since the projection from 3D to 2D can easily be done by just ignoring the z-coordinates, we only have to look at the computation of the lower convex hull.



(a) Transform Π (redrawn from [Aur87])

(b) Lower convex hull mapped from 3D to 2D

Figure 4: *Computation of a power diagram: (a) sites s,t are transformed to the planes $\Pi(s)$ and $\Pi(t)$ in 3D. The bisector between s and t is a vertical mapping of the intersection of $\Pi(s)$ and $\Pi(t)$ back to the two-dimensional plane h_0. (b) The intersection of the projected half-planes creates a lower convex hull in 3D (gray facets) which is a dual solution to the power diagram.*

4.4.2. Lower Convex Hull

A plane in $\mathbb{R}^3$ can be defined by $h : z = ax + bx + c$. The plane-point duality is an interesting relation between planes and points. Each plane h can be represented as a point (or polar) in a dual space by using a *polarity* function Δ of a, b, c:

$$\Delta(h) = \left(\frac{a}{2}, \frac{b}{2}, -c \right). \qquad (5)$$

Δ further maps each point p into a plane $\Delta(p) = \bigcup_{h \supseteq p} \Delta(h)$. An important characteristic of Δ is that it maintains the relative positions of points and planes. If a plane h contains a point p in the normal space, then the plane $\Delta(p)$ contains the point $\Delta(h)$ in the dual plane, see [Cox03] for an extensive description of polarity and duality.

By using the plane duality we can just map the half-planes, which we want to intersect, to points in the dual space. The convex hull of this set of points in dual space is then a dual solution to the wanted lower envelope of half-plane intersections. The dual solution just has to be transformed back to the normal space by using Δ. A face of the convex hull in dual space corresponds to a point in the normal space.

For the algorithm we need to concatenate Δ and Π for $s \in S$, which results in:

$$\Delta(\Pi(s)) = (x_s, y_s, x_s^2 + y_s^2 - \underbrace{w_s}_{r^2}). \qquad (6)$$

As a result of the convex hull in dual space, we have triangulated faces. Each plane defined by a face needs to be transformed back using Δ. For a face f defined by three points $P_1 = (x_1, y_1, z_1), P_2 = (x_2, y_2, z_3), P_3 = (x_3, y_3, z_3) \in \mathbb{R}^3$ the plane $h_f : z = ax + bx + c$ is defined with $a = -\frac{\alpha}{\gamma}$, $b = -\frac{\beta}{\gamma}$, $c = \frac{\delta}{\gamma}$, where α, β, γ and δ are

$$\begin{aligned}
\alpha &= y_1(z_2 - z_3) + y_2(z_3 - z_1) + y_3(z_1 - z_2) \\
\beta &= z_1(x_2 - x_3) + z_2(x_3 - x_1) + z_3(x_1 - x_2) \\
\gamma &= x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2) \\
\delta &= x_1(y_2 z_3 - y_3 z_2) + x_2(y_3 z_1 - y_1 z_3) + x_3(y_1 z_2 - y_2 z_1).
\end{aligned} \tag{7}$$

Eqn. 7 is used to determine the plane from a triangle, and transform it back from dual space.

Algorithm 3: Compute Power Diagram

Input: Ω: convex polygon, $S : n$ unique sites, W : set of weights
Data: L: double connected edge list (convex hull) from which a sequence of faces $F(s^*) = (f_1, \ldots, f_k)$ can be derived for each $s^* \in S^*$, F: list of the triangulated faces in L
Output: Power diagram $\mathcal{V}(S)$ with n polygons

1 **if** $n < 4$ **then**
2 Compute $\mathcal{V}(S)$ by intersecting the bisectors
3 **return** $\mathcal{V}(S) \cap \Omega$
4 **else**
5 $S^* \leftarrow \bigcup_{s \in S} \Delta(\Pi(s))$
6 $\{F(s^*) : s^* \in S^*\} \leftarrow$ convexHull(S^*)
7 $\mathcal{V}(S) \leftarrow \{\}$
8 $\overline{F} \leftarrow \{f \in F :$ normal $n_f = (x, y, z)$ with $z < 0\}$
9 **for** $s^* \in S^*$ **do**
10 $(f_1, f_2, \ldots, f_k) \leftarrow F(s^*) \setminus \overline{F}$
11 $P \leftarrow (\Delta(f_1), \Delta(f_2), \ldots, \Delta(f_k))$
12 **for** $p = (x, y, z) \in P$ **do** $p \leftarrow (x, y)$
13 $\mathcal{V}(S) \leftarrow \mathcal{V}(S) \cup \{P\}$
14 **return** $\mathcal{V}(S) \cap \Omega$

4.4.3. Description of the Power Diagram Algorithm

In line 1 of Algorithm 3 we need to check whether the minimal number of points for a convex hull in 3D is given, if this is not the case we can determine the cells by intersecting the bisectors (see [OBSC00]). Otherwise (line 4), the sites are transformed to planes in 3D and then to points in the dual space by using the function in Eq. 6. Note that this requires nothing more than two multiplications and one subtraction to directly get the point we need in dual space from a given site.

The convex hull of the set S^*, which is the dual solution to an intersection of half-planes, is then computed in line 6. The result of the convex hull computation is stored in a double connected edge list. If two sites $s_1, s_2 \in S$ are neighbors in $\mathcal{V}(S)$, then an edge exists between s_1^* and s_2^* in the convex hull of S^*. An Edge directed from s_1^* to s_2^* has a pointer to the face it belongs to and to the edge which goes from s_2^* to s_1^*. By walking along these edges one can easily get all the edges belonging to a site $s^* \in S^*$ and the corresponding faces as a sequence in the right order (line 10). This can clearly be done in $\mathcal{O}(n)$ time and space where $n = |S^*|$. Note that a face in dual space represents a point in normal space, when transformed back using Δ in line 11. The points we need to represent a polygon for a cell are determined by dropping the z-coordinate (line 12). Since not all cells of a Power diagram are closed, they need to be closed such that they are a subset of Ω. As a last step the resulting Voronoi diagram needs to be intersected with the bounding polygon Ω (line 14).

4.5. Implementation Details

In the following we analyze the available algorithms for the convex hull of a set of points in 3D with regard to our Voronoi treemap algorithm.

Several optimal $\mathcal{O}(n \log n)$ algorithms exist for the convex hull computation in 3D [PH77], where n is the number of points. Unfortunately the divide and conquer algorithm of Preparata and Hong [PH77, Ede87] is very difficult to implement and has poor practical performance [Day90]. We are not aware of a complete and stable implementation. Although output-sensitive algorithms exist [CMS93, CM95, ES91], which exploit the fact that in many applications few points are on the convex hull, they are not important in our case. Since every site of our Power diagram has a non empty region, the corresponding point in 3D has to lie on the convex hull. We thus propose to use the randomized incremental algorithm of Clarkson and Shor [CS89] which is expected to run in $\mathcal{O}(n \log n)$, where n is the number of points. It is simple to implement and many stable implementations are already available; see [O'R98] for an implementation in C or [dBCvKO08] for a general description.

For the intersection of two convex polygons we use the linear-time algorithm of O'Rourke et al. [OCON82]. The implementation for polygons with integer coordinates is described in [O'R98] and can be adapted with few changes for polygons with real-valued coordinates.

5. Performance

While our approach is asymptotically faster than previous ones, this does not ensure that it is faster on typical problem instances as well. We first show that our method compares favorably with previous ones already during a single iteration, and then present a new heuristic to further reduce running times speeding up convergence.

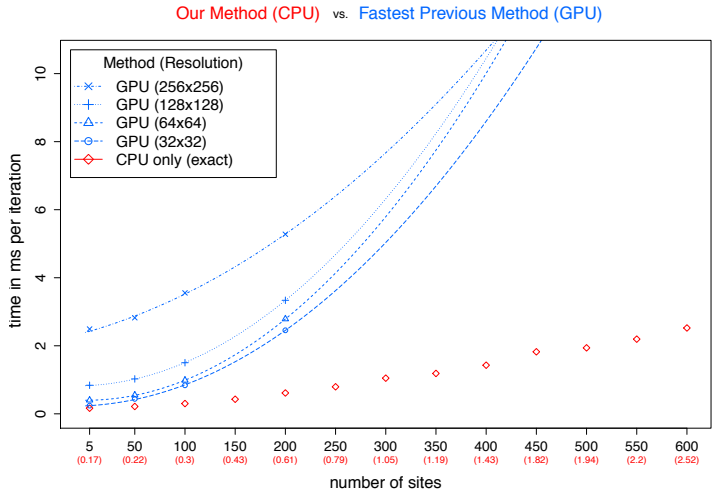


Figure 5: *Timings for a single iteration (sites vs. time in ms). Blue dots are timings reported in [SFL10] for the GPU-accelerated approach at different resolutions, with quadratic curves fitted to extrapolate to larger instances. Red dots are our timings for our CPU-only approach, taken on a comparable machine.*

5.1. Single Iteration Comparison

Since concrete timings are given only in [BD05, SFL10], but to the best of our knowledge, the approach of Sud et al. [SFL10] is the fastest implementation of Voronoi treemaps to date.

In a single iteration of the adapted method of Lloyd, Sud et al. [SFL10] determine new sites, new weights, and an (approximate) additively weighted Voronoi diagram, whereas we determine new sites and weights, a power diagram, again new weights (including a power diagram), and again a power diagram.

Note that differences in objective, programming language, and hardware requirements do not allow for reasonably controlled experiments. In fact, we do not even have access to the hardware-accelerated code. The testing environments are sufficiently similar, though, to allow for a rough comparison of absolute times. The runtime of our approach only depends on the quantity of the sites and not on their coordinates, anyway we chose random site coordinates for each run.

Sud et al. [SFL10]: PC, Windows 7, Intel Core2 CPU, 2.4 GHz, 4 GB memory, NVIDIA GTX260 GPU

Our setting: MacBook, MacOS 10.6, Intel Core2 CPU, 2.5 GHz, 4 GB memory, without GPU (Java 6)

As can be seen in Figure 5, the hardware-accelerated approach is dominated by our analytical method in terms of growth, but also already on small instances and even with respect to the poorest resolution. Since our implementation is entirely in Java, and no optimizations have been attempted, we think that it is justified to conclude that our method is more efficient.

Note that any approach, including ours, can benefit from parallelization on different cores once the top level of the hierarchy has been dealt with. This was already exploited in [BD05].

5.2. Fewer Iterations

Up to now we have argued that our method is faster in each iteration. We now show that the number of iterations can also be reduced significantly by including assumptions about future updates.

Instead of general extrapolation techniques, we propose to use a heuristic based on an observation made during our preliminary experiments. The number of iterations needed until convergence was particularly high when the variance of required areas was large.

This appears to be due to cells with large area requirements having to push smaller cells in their vicinity away to gain space. A large number of iterations is required to propagate these movements.

Figure 6 shows an example in which the weight of site t needs to be increased to enlarge the cell. If the neighboring cells do not have excess area, they would become smaller in the next iteration, their site would be moved outward to the new centroid, their weight increased and their area enlarged in the next iteration. This process is cut short by adding a radial displacement vector to the sites of cells near t. Note that sites may be subject to displacement triggered by several growing cells near them, and we simply add them up so that they cancel each other out when contradictory.

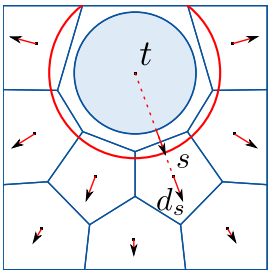


Figure 6: *The large growing cell of a site t needs to push surrounding sites s away. To reduce the number of iterations, we determine displacements d_s depending on the distance from t directly rather than alternating between weights and centroid updates for the same effect.*

To determine which sites should cause movement of others, we use the ratio of current and target area as a criterion. If a cell has a large enough target area, we start generating displacement vectors if the misrepresentation is significant. In our experiments, excluding cells that occupy less then 5–10% of the total area or which already have more then 2/3 of their area worked well, but further experiments are necessary to tune these parameters to a context.

Moreover, we made the magnitude of displacement linearly dependent on the distance from the underrepresented cell to avoid excessive flow and oscillations. See Algorithm 4 for details.

Algorithm 4: Site Update with speedup heuristic

```
1  AdaptPositionsWeights(V(S),S,W)
2      K ← sites causing displacement
3      foreach site s ∈ S do
4          c ← centroid(V_s)
5          c* ← c + d_s
6          if (c* ∩ V_s) = ∅ then
7              p ← cc* ∩ ∂V_s
8              c* ← c + (d_s / ||d_s||) · (1 − ε)||p − c||
9          s ← c*
10         distanceBorder ← min_{p∈∂V_s} ||p − s||
11         w_s ← (min(√w_s, distanceBorder))^2
```

The impact of our heuristic on convergence is illustrated by the following experiment. We created 100 instances by distributing 50 sites randomly in a 2x1 rectangle and assigning area requirements drawn from a power-law distribution. Figure 7 summarizes the convergence behavior of our method with and without the speed-up heuristic. Note that, in the long run, there is no difference in quality of area representation, but the number of iterations needed is reduced by approximately 70%.

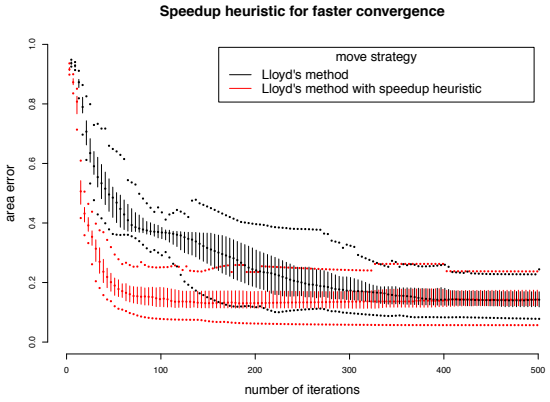


Figure 7: *Boxplots showing area misrepresentation after each iteration of our adaptation of Lloyd's method with (red) and without (black) the speedup heuristic. For 100 sample instances, initial sites have been distributed uniformly in a rectangle with aspect ratio 2, and target areas drawn from a power-law distribution with $f(x) = \frac{1}{x^4}$.*

This heuristic is compatible with other instantiations of Lloyd's method as well.

6. Discussion

We described an asymptotically optimal algorithm for Voronoi treemaps. Our straightforward implementation of this algorithm outperforms tuned and hardware-accelerated implementations of previous approaches. In contrast to previous approaches it is based on a combinatorial algorithm for weighted Voronoi diagrams and therefore resolution-independent; as a consequence, zooming does not require recomputation. Like all previous approaches, our method is an adaptation of Lloyd's method for centroidal Voronoi diagrams, and yields only locally optimal area representation.

The second element of our contribution is a new site update strategy that reduces the number of iterations in Lloyd's method considerably. This technique is not specific to our approach and can be used in other schemes as well, but it would be interesting to explore further the tuning of its parameters.

Other issues to address in future work include the extension of the Voronoi treemap to higher dimensions and parallel computation of the 3d convex hull. The most immediate, however, is whether Fortune's algorithm [For87] yields similar improvements for additively weighted Voronoi diagrams.

Acknowledgments. This research was partially supported by DFG via grant GRK/1042. We thank Michael Balzer and Oliver Deussen for providing us with the data to reproduce their example in Figure 1.

References

[Aur87] AURENHAMMER F.: Power diagrams: properties, algorithms, and applications. *SIAM Journal on Computing 16*, 1 (1987), 78–96. 2, 6

[BD05] BALZER M., DEUSSEN O.: Voronoi Treemaps. In *Proceedings of IEEE Symposium of Information Visualization (InfoVis'05)* (2005), IEEE Computer Society Press, p. 7. 1, 3, 8

[BDL05] BALZER M., DEUSSEN O., LEWERENTZ C.: Voronoi Treemaps for the Visualization of Software Metrics. In *Proceedings of the 2nd ACM Symposium on Software metrics* (2005), ACM Press, pp. 165–172. 3

[BFHS09] BERNHARDT J., FUNKE S., HECKER M., SIEBOURG J.: Visualizing Gene Expression Data via Voronoi Treemaps. In *Proceedings of the 2009 Sixth International Symposium on Voronoi Diagrams* (2009), IEEE Computer Society Press, pp. 233–241. 1, 4

[CM95] CHAZELLE B., MATOUŠEK J.: Derandomizing an output-sensitive convex hull algorithm in three dimensions. *Computational Geometry 5* (1995), 27–32. 7

[CMS93] CLARKSON K., MEHLHORN K., SEIDEL R.: Four results on randomized incremental constructions. *Computational Geometry 3* (1993), 185–212. 7

[Cox03] COXETER H. S. M.: *Projective Geometry*. Springer, 2003. 6

[CS89] CLARKSON K., SHOR P.: Applications of random sampling in computational geometry, ii. *Discrete & Computational Geometry 4* (1989), 387–421. 7

[Day90] DAY A. M.: The implementation of an algorithm to find the convex hull of a set of three-dimensional points. *ACM Transactions on Graphics 9* (January 1990), 105–132. 7

[dBCvKO08] DE BERG M., CHEONG O., VAN KREVELD M., OVERMARS M. H.: *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 2008. 7

[Ede87] EDELSBRUNNER H.: *Algorithms in combinatorial geometry*. EACTS Monographs on Theoretical Computer Science. Springer-Verlag, 1987. 7

[ES91] EDELSBRUNNER H., SHI W.: An $o(n\log^2 h)$ time algorithm for the three-dimensional convex hull problem. *SIAM Journal on Computing 20* (March 1991), 259–269. 7

[For87] FORTUNE S.: A sweepline algorithm for Voronoi diagrams. *Algorithmica 2* (1987), 153–174. 2, 9

[GO04] GOODMAN J. E., O'ROURKE J.: *Handbook of Discrete and Computational Geometry*. Discrete Mathematics and its Applications. CRC Press, 2004. 2

[Got11] GOTZ D.: *Dynamic Voronoi Treemaps: A Visualization Technique for Time-Varying Hierarchical Data*. Tech. Rep. RC25132, IBM Research Division, Thomas J. Watson Research Center, 2011. 1, 4

[HTS09] HORN M. S., TOBIASZ M., SHEN C.: Visualizing biodiversity with voronoi treemaps. In *Proceedings of the 2009 Sixth International Symposium on Voronoi Diagrams* (2009), IEEE Computer Society Press, pp. 265–270. 1, 4

[Llo57] LLOYD S. P.: *Least squares quantization in PCM's*. Tech. rep., Bell Laboritories Memo, 1957. 3

[LWL*09] LIU Y., WANG W., LÉVY B., SUN F., YAN D.-M., LU L., YANG C.: On centroidal voronoi tessellation: energy smoothness and fast computation. *ACM Transactions on Graphics 28* (2009), 101:1–101:17. 2, 4

[OBSC00] OKABE A., BOOTS B., SUGIHARA K., CHIU S. N.: *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*, 2nd ed. Wiley, 2000. 7

[OCON82] O'ROURKE J., CHIEN C.-B., OLSON T., NADDOR D.: A new linear algorithm for intersecting convex polygons. *Computer Graphics and Image Processing 19*, 4 (1982), 384–391. 7

[O'R98] O'ROURKE J.: *Computational Geometry in C*. Cambridge University Press, 1998. 7

[PH77] PREPARATA F. P., HONG S. J.: Convex hulls of finite sets of points in two and three dimensions. *Communications of the ACM 20* (February 1977), 87–93. 7

[RLW*11] RONG G., LIU Y., WANG W., YIN X., GU X. D., GUO X.: Gpu-assisted computation of centroidal voronoi tessellation. *IEEE Transactions on Visualization and Computer Graphics 17* (2011), 345–356. 3

[SFL10] SUD A., FISHER D., LEE H.-P.: Fast dynamic voronoi treemaps. In *Proceedings of the 7th International Symposium on Voronoi Diagrams in Science and Engineering* (2010), ISVD '10, IEEE Computer Society, pp. 85–94. 2, 3, 8

[Shn92] SHNEIDERMAN B.: Tree visualization with tree-maps: 2-d space-filling approach. *ACM Transactions on Graphics 11* (1992), 92–99. 1, 2

[Shn09] SHNEIDERMAN B.: Treemaps for space-constrained visualization of hierarchies. webpage, June 2009. http://www.cs.umd.edu/hcil/treemap-history/, visited on November 2011. 1

[VSCG08] VASCONCELOS C., SÁ A., CARVALHO P., GATTASS M.: Lloyd's algorithm on gpu. In *Advances in Visual Computing*, vol. 5358 of *Lecture Notes in Computer Science*. Springer-Verlag, 2008, pp. 953–964. 3