

Statistical Data Depth and the Graphics Hardware

Shankar Krishnan, Nabil H. Mustafa, and Suresh Venkatasubramanian

ABSTRACT. The notion of ‘depth’ has been used in statistics as a way to identify the center of the bivariate distribution given by the point set P in \mathbb{R}^2 . We present a general framework for computing such statistical estimators, that makes extensive use of modern graphics architectures. As a result, we derive improved algorithms for a number of depth measures such *location depth*, *simplicial depth*, *Oja depth*, *colored depth*, and *dynamic location depth*. Our algorithms perform significantly better than currently known implementations, outperforming them by at least one order of magnitude and having a strictly better asymptotic growth rate.

1. Introduction

Shape estimators are important in spatial data mining, where they are used to gauge important characteristics of the data. One notion of shape estimation of a set of points is their ‘depth’, which has been used in statistics as a way to identify the center of a bivariate distribution [8]. For example, the *location depth* of a point with respect to a fixed set of input points has often been used in statistics as a way to identify the center of a bivariate distribution [8]. The associated notion of a *depth contour* (the set of all points having depth $\geq k$), has been employed in this context to isolate the “core” of a distribution [20]; the Tukey median is the contour of points at maximum location depth. As explained in [20], one can define a shape called the “bagplot” that effectively separates outliers in a distribution from the core. The bagplot “visualizes the location, spread, correlation, skewness, and tails of the data” [20].

An illustration of this idea is shown in Figure 1. The figure on the left is a set of points representing a bivariate distribution, where we can see a weak correlation between the parameters on the x and y coordinate axis. On the right is the set of depth contours for this data set; notice the inner lighter region, which clearly indicates the correlation. The set of depth contours of a point set are used in various applications such as hypothesis testing, robust statistics, and even in various problems in cell biology (especially for moving point sets).

1991 *Mathematics Subject Classification.* Primary 6207, 65D18; Secondary 6209.

Work has been supported by NSF under grants CCR-00-86013 EIA-98-70724, EIA-99-72879, EIA-01-31905, and CCR-02-04118.

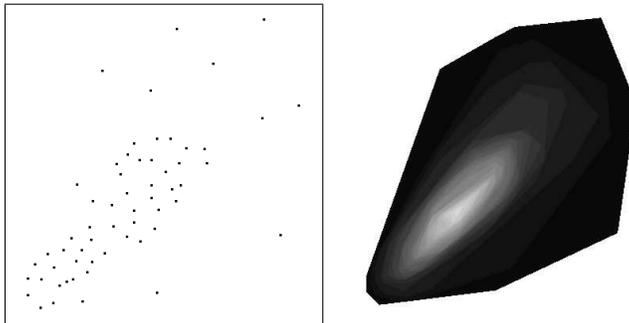


FIGURE 1. An illustration of depth contours

1.1. Organization. We first survey various definitions of statistical depth, and the related computational questions in Section 2. Section 3 gives a basic overview of the graphics rendering system. Section 4 presents the basic tools and techniques underlying all the algorithms. Improved algorithms for the following statistical depth measures are presented: location depth (Section 5), dynamic location depth (Section 6), Oja depth (Section 7), simplicial depth (Section 8), and colored depth (Section 9). Finally, experimental results are presented in Section 10.

2. Defining Data Depth

Over the years, a large number of depth measures have been proposed, and algorithms presented for their fast computation. In general, the greater the depth of a point, the more densely it is surrounded by other points of P . A simple example of a depth measure in \mathbb{R} is the median — given a set of points on the real line, the median has the greatest number of points on either side (at least $\lfloor n/2 \rfloor$).

Once the depth for each point in the plane has been defined, there are actually two quantities which can be computed over this depth measure — the point in the plane with the maximum depth, often called the *median* (analogous to the median case for \mathbb{R}) of P under that depth measure, and the region of all points with depth greater than some integer k . Now, we survey various depth measures, and the related algorithmic results.

2.1. Location depth. Given a set of n points P , the *location depth* of a point u is the minimum number of points contained in any half-plane passing through u . See Figure 2. This notion of depth is not restricted to points in \mathbb{R}^2 — the location depth of a point can be defined in any dimension. Note that in \mathbb{R} , a number's location depth can be computed solely by its rank.

The point in \mathbb{R}^d with the highest location depth is called the *Tukey median*, made popular by John Tukey [22]. It can be shown using Helly's Theorem that in \mathbb{R}^d , there exists a point with location depth at least $\lceil \frac{n}{d+1} \rceil$ [14]. Obviously the maximum depth can be at most $\lceil \frac{n}{2} \rceil$. A related notion to median is the notion of *center-points*, which are points of location depth $\lceil \frac{n}{d+1} \rceil$ in d -dimensional space. The problem of computing *center-points* has been examined earlier; Cole, Sharir and Yap [5] presented an algorithm that computes center-points in the plane in

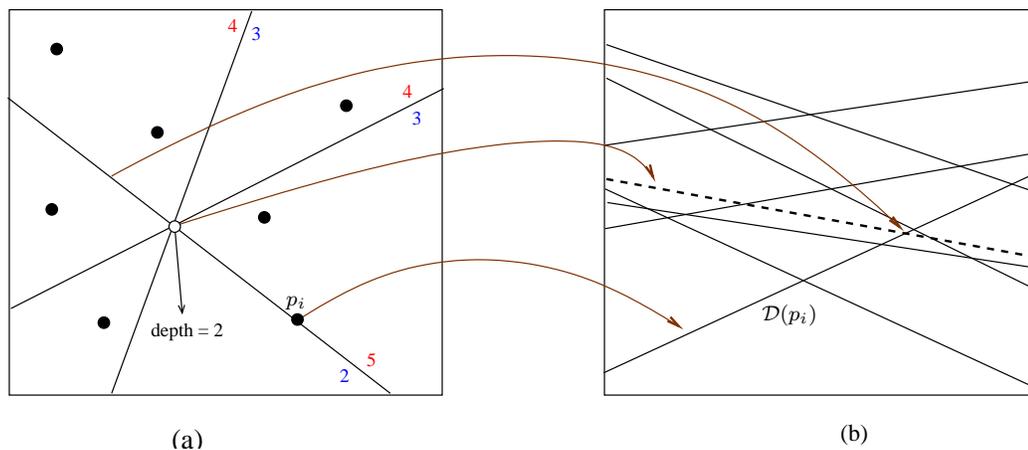


FIGURE 2. A set of points in the plane, and its set of dual lines.

time $O(n \log^5 n)$, and Naor and Sharir extended this to three dimensions in time $O(n^2 \log^6 n)$. Matoušek [14] showed that the Tukey median in \mathbb{R}^2 can be computed in time $O(n \log^5 n)$ by first showing that any point with depth greater than k can be found in $O(n \log^4 n)$ time and binary search on k gives the required time bound. Jadhav and Mukhopadhyay [10] gave a $O(n)$ algorithm to compute the center-point. Recently, Langerman and Steiger [12] presented an algorithm that computes the Tukey median in time $O(n \log^3 n)$, and Chan gave an optimal randomized algorithm that runs in time $O(n \log n)$ [3].

The second algorithmic question relates to computing the set of all depth contours in the plane. First, observe the following relation between depth of a point, and its dual line. Given a point $u \in \mathbb{R}^2$, let $l_u = \mathcal{D}(u)$ be the dual line of u . Let $L = \cup_i \mathcal{D}(p_i)$ be the set of dual lines to all the points in P . Then any line passing through u , say a line l , in the primal plane corresponds to a point $\mathcal{D}(l)$ in the dual plane such that $\mathcal{D}(l)$ lies on the line l_u . Furthermore, due to the order preserving properties of the dual, the number of points of P lying above the line l in the primal plane corresponds to the number of lines in L vertically below the point $\mathcal{D}(l)$ in the dual plane, i.e. the level of the point $\mathcal{D}(l)$. Therefore, to compute the depth of point u in the primal plane, we could look at the dual line l_u , and find the point on l_u with the minimum level (Figure 2). Depth contours also have a natural characterization in terms of the arrangement in the dual plane induced by P — the depth contour of depth k is related to the convex hull of the k and $(n - k)$ levels of the dual arrangement. The algorithm of Miller *et al.* [15] exploits this property by using a topological sweep in the dual to compute the set of all depth contours — they proposed an optimal algorithm that computes all the depth contours for a set of points in time $O(n^2)$, and allows the depth of a point to be queried in time $O(\log n)$. They demonstrate that compared to HALFMED [19] (the previously best known implementation for computing the Tukey median), their algorithm performs far better in practice. For example, on a data set of 1000 points, their algorithm runs in 22.4 seconds, as compared to 9002 seconds for HALFMED.

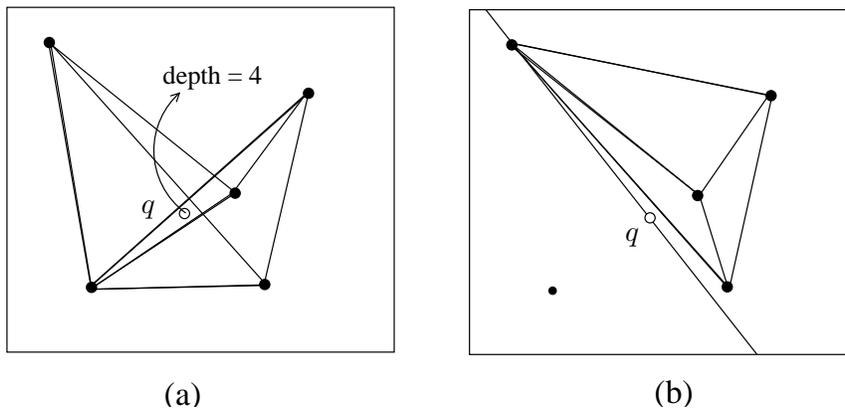


FIGURE 3. (a) The simplicial depth of a single point, (b) Computing simplicial depth by subtracting triangles not containing q .

2.2. Oja depth. Another measure of depth was proposed by Oja [16], as follows. Given a set of points P in the plane, the Oja depth of a point $u \in \mathbb{R}^2$ is the sum of the area of the triangles formed by u and all pairs of points in P , i.e.,

$$ODepth(u) = \sum_{i,j} \text{Area}(u, p_i, p_j)$$

The same measure can also be extended to d -dimensions by considering d -simplices formed by u and every other configuration of d points of P .

The point (not necessarily unique) with minimum Oja depth is called the Oja median. It is known that the points with minimum Oja median form a convex set [16]. Furthermore, it is known that Oja median occurs as an intersection of lines formed by pairs of points of P . Rousseeuw and Ruts [17] present an $O(n^5 \log n)$ time algorithm for computing the Oja median of a planar point set, which was further reduced to the current best running time of $O(n^3 \log n)$ [1].

2.3. Simplicial depth. Oja depth for a point measured the sum of the areas of $\binom{n}{2}$ triangles in the plane. We can also define a measure which simply counts the number of triangles containing the required point. This leads to *simplicial* depth of a point set, defined by Liu [13]. Let T be the set of all triangles formed by vertices of P — each triangle requires three vertices, and therefore T has $\binom{n}{3}$ triangles. Given a point $u \in \mathbb{R}^2$, the simplicial depth of u , denoted $SDepth(u)$ is the number of triangles of T that contain u . See Figure 3.

The simplicial median is the point with the highest simplicial depth. Liu [13] showed that the simplicial median is invariant to affine transformations. It was shown by Boros and Füredi [2] that there exists a point q such that the simplicial depth of q is $\Theta(n^3)$. The simplicial median can be computed as follows. Construct the set of all $\binom{n}{3}$ triangles. All the points in the region formed by overlay of these $\binom{n}{3}$ triangles have the same simplicial depth, and therefore the simplicial median can be computed in time $O(n^7)$ by constructing each face of the overlay of these triangles, and query for a point there. The running time of the algorithm for computing the simplicial median was reduced to $O(n^4 \log n)$ in [1].

A related question is whether the simplicial depth of a single point can be computed faster. Gil *et al.* [7] present an algorithm for computing the simplicial depth of a query point q in time $O(n \log n)$ as follows.

The idea is to count the set of triangles that do *not* contain q , and then subtract it from $\binom{n}{3}$ to get the set of all triangles that contain q . Given a point $q \in \mathbb{R}^2$, imagine a horizontal line l through q . W.l.o.g assume $q \cup P$ does not contain three collinear points. Also let $l(p)$ be the line passing through p and q . Sort the points in P radially anti-clockwise around q to get the sorted list $\langle p_{i_1}, \dots, p_{i_n} \rangle$. Rotate l anti-clockwise until it hits the first point $p_{i_1} \in P$. It is easy to see that all the triangles formed by $p_i, p_j \in P \cap l(p_{i_1})^+$ and p_{i_1} do not contain q (similarly, all the triangles formed by $p_i, p_j \in P \cap l(p_{i_1})^-$ and p_{i_1} do not contain q). The number of such triangles, say $T(p_{i_1})$, is

$$T(p_{i_1}) = \binom{|P \cap l(p_{i_1})^+|}{2} + \binom{|P \cap l(p_{i_1})^-|}{2}$$

Now continue rotating l until it hits the next point p_{i_2} and similarly the number of triangles not containing q with p_{i_2} as one vertex of such triangles is $T(p_{i_2})$. Thus, it follows that

$$(1) \quad SDepth(q) = \binom{n}{3} - 1/2 \cdot \sum_{j=1}^n T(p_{i_j})$$

since during the anti-clockwise rotation, each triangle not containing q is actually counted twice. Actually, we can define $T(p_{i_j})$ to be just the set of triangles formed by points in the positive half-space, and therefore we count each triangle not containing q exactly once.

2.4. Other depth measures. We briefly survey some of the other known depth measures.

Hyperplane depth is defined for an arrangement of lines [18] (or hyperplanes in higher dimensions). Given an arrangement of lines, the hyperplane depth of a point $q \in \mathbb{R}^2$ is the minimum number of lines a ray emanating from q must cross. The hyperplane median is the point of maximum hyperplane depth. Langerman and Steiger [11] present a $O(n \log n)$ algorithm for computing the hyperplane median in the plane, improving upon the $O(n^3)$ algorithm of Rousseeuw and Hubert [18].

L_1 *depth* can be defined for a point q as simply the sum of distances of q from all the other points of P . Then the L_1 median is the point that minimizes the sum of distances to all the other points in P (it is also known as the Fermat-Weber point). It is easy to see that the L_1 median is invariant to rotations and translations of P . In \mathbb{R} , the L_1 median corresponds to the usual median of the set of numbers. It can be shown that there exists a point $p_i \in P$ such that the sum of distances to all the other points from p_i is at most twice the value of the L_1 median. Naively, by computing this sum of distances to all the other points for each point p_i of P , an approximation to the L_1 median can be computed in time $O(n^2)$ in any \mathbb{R}^d . Indyk [9] gave an elegant algorithm that computes a $(1 + \epsilon)$ -approximation of the median in time $O(n/\epsilon^5)$.

Convex hull peeling methods [21] are yet another way to imagine the skew and spread of planar point sets. Given P , compute the convex hull of P , remove the vertices on the convex hull, and repeat the convex hull “peeling” procedure. Intuitively this results in progressive peeling away of deeper and deeper points. The

last convex hull to be peeled away is in some sense the most deep in the point set. Chazelle [4] showed that this convex hull peeling can be performed in optimal time $O(n \log n)$.

3. The Graphics Rendering System

The graphics rendering system is a finite state machine, consisting of the pipeline described as follows. In the first stage of the pipeline, the user gives the geometric primitives to be “drawn” to the evaluator. The basic primitive is a vertex, which consists of x , y , and z coordinates. Other primitives are lines, triangles, polygons and so forth. The second stage performs various operations such as lighting, clipping, projection and viewport mapping on the input primitives. The third stage rasterizes the primitives, i.e. produces fragments from the geometric primitives which the graphics system will be able to draw. For our purposes, the fragments are just pixels. The fourth stage is per-fragment operations, such as blending and z-buffering. Finally, the fragments that pass the per-fragment operations of the previous stage are drawn, or *rendered*, in the frame buffers. See Figure 4 for the pipeline stages.

The main functionality of the graphics hardware is provided by the per-vertex and per-fragment stages described above. In the design of any graphics pipeline, there are a minimum set of operations required for the per-vertex and per-fragment operations. For example, the depth-test is an example of a necessary per-fragment operation. There have been two approaches toward designing graphics architectures that provide the user with the required functionalities.

Traditionally, the graphics pipeline has been a *fixed function* pipeline. A number of per-vertex and per-fragment operations were “hardwired” into the graphics cards. The parameters required for these operations could be set by the user through a corresponding fixed-function API. With the expressiveness of these operations quickly increasing, the number of these APIs would quickly get out of hand. More importantly, the APIs provide, at best, a high-level control over various per-vertex and per-fragment operations.

The advent of *programmable* pipelines have sought to correct these shortcomings. These programmable pipeline architectures replace the fixed-function vertex and fragment stages with programmable stages. These programmable vertex and fragment engines execute user-defined programs and allow flexible control over operations such as shading and texturing. This user-written vertex program (initially in an assembly-like language, and more recently in languages resembling C) runs on each vertex, and passes the results onto the third stage, rasterization. The fragments produced by the rasterization stage then serve as input to the fragment stage, where again a user-written program (called a pixel shader) processes the incoming fragments to perform, among other things, tests like the depth test. The resulting set of fragments are then sent to be rendered.

We will use only the functionalities provided by the fixed function pipeline (all the operations provided by the fixed function pipeline can be achieved via appropriate vertex and fragment programming).

3.1. Frame Buffers. The frame buffer is a collection of several hardware buffers. Each buffer is a uniform two dimensional grid, composed of cells called *pixels*. Let the number of pixels in each row and column be W and H respectively.

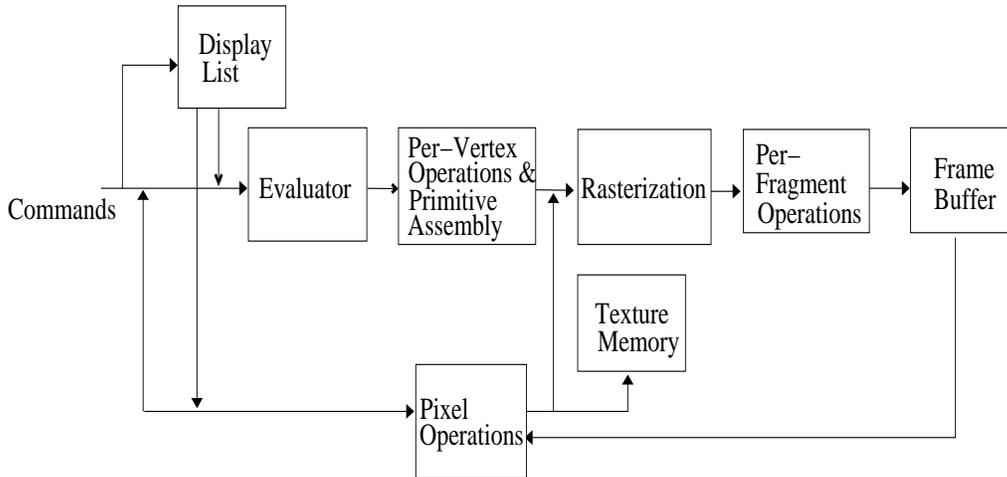


FIGURE 4. The graphics pipeline.

Then, each buffer is essentially a $W \times H$ uniform grid. The buffers we make use of are:

Color Buffer. The color buffer contains the pixels that are seen on the display screen. It contains **RGB** color information¹. Before a fragment is converted and rendered to a particular pixel in the color buffer, it has to undergo various per-fragment operations. If it passes all the operations, it is rendered with the appropriate color.

Stencil Buffer. This buffer is used for per-fragment operations on a fragment before it is finally rendered in the color buffer. As the name suggests, it is used to restrict drawing to certain portions of the screen. We explain this in the next section.

Depth Buffer. The depth buffer stores the depth value for each pixel. Depth is usually measured in terms of distance to the eye, so pixels with larger depth-buffer value are (usually) overwritten by pixels with smaller values.

The pixel at the i -th row and j -th column of the screen is denoted as P_{ij} , $1 \leq i \leq W, 1 \leq j \leq H$. The corresponding value of the pixel P_{ij} in the color buffer is denoted as CB_{ij} ; similarly, the corresponding values in the stencil and depth buffers are denoted as SB_{ij} and DB_{ij} respectively.

3.2. Operations. Geometric primitives are first converted into fragments (through the process of rasterization), which then pass through some per-fragment operations, after which they are finally rendered in the color buffer. Using these per-fragment operations, we formulate a set of basic hardware-based operations. One major restriction which makes the design of algorithms using the graphics pipeline challenging is that each operation is per-fragment, i.e. each operation is local to a pixel. This makes gaining global information inherently difficult, and inefficient, and the main task is therefore to solve a problem globally using local operations.

¹It also contains information relating to transparency of pixels, also called the α -value.

Color buffer operations	Stencil buffer operations	Depth buffer operations
$CB_{ij} \leftarrow \odot(CB_{ij}, R_1G_1B_1)$	$SB_{ij} \leftarrow K$	$DB_{ij} \leftarrow \odot(DB_{ij}, z)$
$CB_{ij} \leftarrow CB_{ij} \oplus R_1G_1B_1$	$SB_{ij} \leftarrow SB_{ij} \oplus 1$	
$\odot = \{\min, \max\}, \oplus = \{+, -\}$	$SB_{ij} \leftarrow !SB_{ij}$	
If-conditionals:		
If $(SB_{ij} \{<, >, \leq, \geq, =, \neq\} K)$ then $CB_{ij} \leftarrow R_1G_1B_1, SB_{ij} \leftarrow SB \text{ op}$ else $SB_{ij} \leftarrow SB \text{ op}$		
If $(DB_{ij} \{<, >, \leq, \geq, =, \neq\} z)$ then $CB_{ij} \leftarrow R_1G_1B_1, DB_{ij} \leftarrow DB \text{ op}$ else $DB_{ij} \leftarrow DB \text{ op}$		

TABLE 1. Operations.

We now give some examples of the per-fragment operations. Each operation takes as input a fragment and some data from the frame buffer, performs an operation and then updates the frame buffer appropriately.

More specifically, we are given an input fragment (pixel) P_{ij} with depth value z and color $R_1G_1B_1$, and a user-specified constant K that can be specified before rendering any primitive (it cannot be varied per fragment though). $SB \text{ op}$ and $DB \text{ op}$ are any stencil buffer or depth buffer operations. Table 1 lists a subset of the per-fragment operations.

I/O Operations. The user interacts with the pipeline by drawing primitives and retrieving the rendered results. The main input operation is the *Draw* operation, which is used to draw a given primitive. The main output operation is the *Read-Buffer* operation, which reads a specified buffer from the frame buffer into the main memory of the machine.

3.3. Hardware-Assisted Algorithms. Our work is based on the paradigm of *hardware-assisted algorithms*. The growing power and sophistication of graphics cards, (and their easy accessibility) has made it possible to implement fairly complex operations directly on the graphics chip. Today, there are a vast number of applications where the GPU has been used primarily as a fast co-processor to accelerate a computation. A full overview of the work on hardware-assisted algorithms is beyond the scope of this paper; the reader is referred to the site <http://www.GPGPU.org> for a fairly complete list of all work in this area.

Although the GPU has been used in a variety of settings, probably its most effective use has been in the area of physical simulation and visualization, where its rendering capabilities and computing power can be harnessed at the same time. Numerous papers have studied simulation of physical processes on the GPU, and there has been some work on data analysis and visualization as well.

4. Basic techniques

In this section we describe the basic techniques used by the various data depth algorithms. However, they are also of independent interest.

Let P denote a set of n points in the plane. Without loss of generality, we assume $p \in [-1, 1] \times [-1, 1]$ for all $p \in P$. The line passing through two points p and q is denoted by $l(p, q)$. The positive (negative) halfplane of a line l is denoted by l^+ (l^-). Define $f(X) = |X \cap P|$ to be the number of points of P contained in the geometric object X .

Consider the standard duality relation: $p = (p_x, p_y)$ in the primal plane (denoted as \mathbf{P}) maps to the line $y = -p_x \cdot x + p_y$ in the dual plane (denoted as \mathbf{D}).

The mapping from the primal plane \mathbf{P} to the dual plane \mathbf{D} is denoted by the operator $\mathcal{D}(\cdot)$: $\mathcal{D}(p)$ is the line in plane \mathbf{D} dual to the point p , and similarly $\mathcal{D}(l)$ is the point in plane \mathbf{D} dual to the line l (we denote the inverse operator by \mathcal{P} , i.e. $\mathcal{P}(q)$ is the line in the primal plane \mathbf{P} that is the dual of point q in the dual plane \mathbf{D}). Each point $p \in P$ maps to a line $\ell = \mathcal{D}(p)$ in the dual plane. Also define $\mathcal{D}(P) = \cup_{p \in P} \mathcal{D}(p)$. The set of lines $\mathcal{D}(P)$ define the dual arrangement. In this dual arrangement, we denote the *level* of a point x to be (in the standard way) the number of lines of $\mathcal{D}(P)$ that lie strictly below or passing through x .

The duality relation satisfies these two properties (see [6] for more details):

- **Incidence.** For a line l passing through points p and q , its dual point $\mathcal{D}(l)$ is the intersection of the lines $\mathcal{D}(p)$ and $\mathcal{D}(q)$.
- **Order-preservation.** If the point p lies below line l , the dual line $\mathcal{D}(p)$ lies below the dual point $\mathcal{D}(l)$.

Our main objective in this section is: Preprocess P such that for any line $l' = l(u, v)$, where $u, v \in [-1, 1]^2$ are points in the primal plane \mathbf{P} , $f(l(u, v)^+)$ can be computed quickly. Due to the order-preservation property of the duality relation $\mathcal{D}(\cdot)$ above, the number of points of P in the positive halfplane of l' is exactly the number of dual lines below the dual point of l' , and it follows that $f(l'^+) = \text{level}(\mathcal{D}(l'))$. Therefore our strategy would be to compute the dual arrangement $\mathcal{D}(P)$ to answer the level queries.

However, note that each buffer consists of a $(2W + 1) \times (2W + 1)$ uniform grid of pixels. Thus to guarantee bounded error, all our computations are restricted in the space $[-1, 1] \times [-1, 1]$. Therefore it is possible that the point $\mathcal{D}(l')$, formed by the intersection of the lines $\mathcal{D}(u)$ and $\mathcal{D}(v)$, may lie outside the screen boundary (for example, when the two lines $\mathcal{D}(u), \mathcal{D}(v)$ are nearly parallel). Thus we would be unable to compute $f(l'^+)$ using the arrangement computation. To remedy this problem, we introduce the notion of *bounded duals* in the next section, which are then used in the hardware-based computation of the arrangement described in Section 4.2.

4.1. Bounded arrangements.

DEFINITION 4.1 (Bounded Dual). Let p be a point in \mathbf{P} , and let $\ell = \mathcal{D}(p)$. The bounded dual of the point p is a line segment defined as $(x, y)\text{-}\mathcal{BD}(p) = \{(d_x, d_y) \in \ell \mid |d_x| \leq x \text{ and } |d_y| \leq y\}$.

Let $(x, y)\text{-}\mathcal{BD}(P) = \bigcup_{p \in P} (x, y)\text{-}\mathcal{BD}(p)$. Define $\mathcal{BP}(p)$ similarly as the reverse mapping of a point in the dual plane to a line segment in the primal plane.

Any two line segments in $(x, y)\text{-}\mathcal{BD}(P)$ must intersect within the range $[-x, x] \times [-y, y]$, or not intersect at all. The next lemma shows that we can capture all the intersections with a small range by a union of two bounded duals.

LEMMA 4.2 (Bounded Union Lemma). *Given any point set P , where for all $p \in P, p \subseteq [-x, x] \times [-y, y]$, there are two bounded duals $(x, 2y)\text{-}\mathcal{BD}1(P)$ and $(x, 2y)\text{-}\mathcal{BD}2(P)$ such that each intersection point $q \in \mathcal{D}(P)$ lies either in $(x, 2y)\text{-}\mathcal{BD}1(P)$ or $(x, 2y)\text{-}\mathcal{BD}2(P)$.*

PROOF. We need and prove the lemma for $x = 1, y = 1$; the general case can be proved similarly.

The two bounded duals are

$$\begin{aligned} (1, 2)\text{-}\mathcal{BD}1(p) : (p_x, p_y) \in \mathbf{P} &\rightarrow y = -p_x \cdot x + p_y \\ (1, 2)\text{-}\mathcal{BD}2(p) : (p_x, p_y) \in \mathbf{P} &\rightarrow y = -p_y \cdot x + p_x \end{aligned}$$

Also, the reverse dual equations, mapping points in dual to lines in the primal, are:

$$\begin{aligned} (1, 2)\text{-}\mathcal{BP}1(p) : (p_x, p_y) \in \mathbf{D} &\rightarrow y = p_x \cdot x + p_y \\ (1, 2)\text{-}\mathcal{BP}2(p) : (p_x, p_y) \in \mathbf{D} &\rightarrow p_x \cdot y = x - p_y \end{aligned}$$

It can be verified that these reverse duals indeed preserve incidence and order. To show that these set of bounded duals capture all the intersections, we show that the x and y coordinates of each intersection point lie within the intervals $[-1, 1] \times [-2, 2]$ for at least one of the two bounded duals. For convenience, we drop the prefix $(1, 2)$ - in the rest of the proof.

Let $p = (p_x, p_y), q = (q_x, q_y) \in \mathbf{P}$ with $p_x, p_y, q_x, q_y \in [-1, 1]$. The corresponding dual lines are

$$\begin{aligned} \mathcal{BD}1(p) : y = -p_x \cdot x + p_y, \quad \mathcal{BD}2(p) : y = -p_y \cdot x + p_x \\ \mathcal{BD}1(q) : y = -q_x \cdot x + q_y, \quad \mathcal{BD}2(q) : y = -q_y \cdot x + q_x \end{aligned}$$

and the intersections are

$$\begin{aligned} \mathcal{BD}1(p) \cap \mathcal{BD}1(q) : x_1 = \frac{q_y - p_y}{q_x - p_x}, y_1 = \frac{p_y q_x - p_x q_y}{q_x - p_x} \\ \mathcal{BD}2(p) \cap \mathcal{BD}2(q) : x_2 = \frac{q_x - p_x}{q_y - p_y}, y_2 = \frac{p_x q_y - p_y q_x}{q_y - p_y} \end{aligned}$$

First assume that $|x_1| \geq |x_2|$ (the other case is symmetric). Then

$$\begin{aligned} \frac{|q_y - p_y|}{|q_x - p_x|} &\geq \frac{|q_x - p_x|}{|q_y - p_y|} \\ |q_y - p_y| &\geq |q_x - p_x| \\ \frac{|q_x - p_x|}{|q_y - p_y|} &= |x_2| \leq 1, \text{ and hence } x_2 \in [-1, 1]. \end{aligned}$$

For $|y_2|$, we have

$$\begin{aligned} |y_2| &= \frac{|p_x q_y - p_y q_x|}{|q_y - p_y|} \leq \frac{|p_x - q_x| + |q_y - p_y|}{|q_y - p_y|} \\ &\leq \frac{|p_x - q_x|}{|q_y - p_y|} + 1 \leq 2 \end{aligned}$$

since $|q_y - p_y| \geq |p_x - q_x|$, and $|p_x q_y - p_y q_x| \leq |p_x - q_x| + |q_y - p_y|$ for all $|p_x|, |q_x|, |p_y|, |q_y| \leq 1$. Therefore we get all intersections with these two duals in the window $[-1, 1] \times [-2, 2]$. The above algebra is tight. To see this, consider the case where $p_x = q_y = 1$, and $p_y = q_x = c$, where $|c| \leq 1$. Note that the condition $|q_y - p_y| \geq |p_x - q_x|$ is still satisfied. Then $|y_2| = \frac{|1-c^2|}{|1-c|} = |1+c|$, and thus arbitrarily close to 2 for c arbitrarily close to 1. The case when $|x_1| \leq |x_2|$ is similar. \square

Algorithm 1 ComputeArrangement (PtSet P)

Input: Point Set $P = \{p_1, \dots, p_n\}$

Output: Arrangement $\{SB1, SB2\}$

Enable **STENCIL TEST**

Set **STENCIL OPERATION** = INCREMENT

for $\mathcal{BD} = \mathcal{BD1} \ \mathcal{BD2}$ **do**

 Initialize $SB = 0$

for $i = 1 \dots n$ **do**

 Line Segment $l' = (1, 2)\text{-}\mathcal{BD}(p_i)$.

DRAW HALF-PLANE (l'^+) .

 Save Stencil Buffer

RETURN $\{SB1, SB2\}$

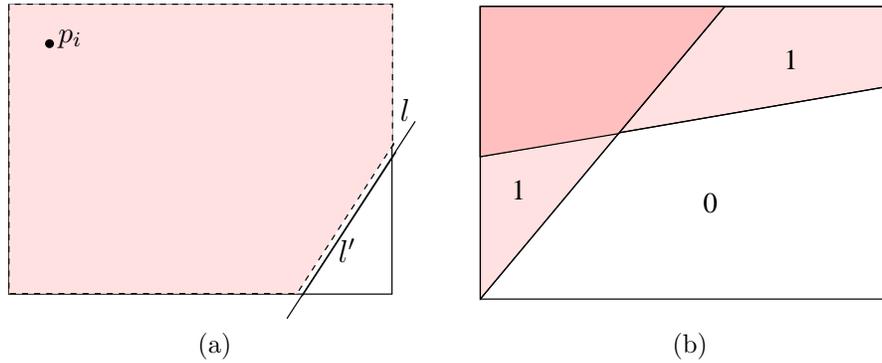


FIGURE 5. (a) Rendering HALF-PLANE(l'^+) is sufficient for correctness, (b) two half-planes with the corresponding regions of different levels.

4.2. Computing arrangements. Recall that our objective was to process P to answer halfspace range counting queries $f(l^+)$ for any line l via the level query $level(\mathcal{D}(l))$ in the arrangement $\mathcal{D}(P)$. We will represent the arrangement $\mathcal{D}(P)$ as the pair $\{(1, 2)\text{-}\mathcal{BD1}(P), (1, 2)\text{-}\mathcal{BD2}(P)\}$, and process the two duals to answer level queries for points in the range $[-1, 1] \times [-2, 2]$. It follows from Lemma 4.2 that this is sufficient to answer the counting query for any line l in \mathbf{P} .

For each of the two duals, we compute the level of each pixel: for each point $p \in P$, compute its bounded dual, and for each pixel P_{ij} , increment the corresponding value in the Stencil Buffer, depending on whether it is above or below the bounded dual segment for p . Algorithm 1 gives the details.

The notion of “above” and “below” is not well defined for a line segment. We interpret the upper halfspace of a line segment l' as the intersection of the upper halfspace of the line l that supports l' with the screen bounding box (see Figure 5 (a)).

LEMMA 4.3. *After Algorithm 1 is done, each pixel P_{ij} 's value SB_{ij} contains $|T|$, where $T = \{p \in P : P_{ij} \in \mathcal{D}(p)^+\}$.*

PROOF. First note that if $P_{ij} \in \mathcal{D}(p)^+$, then $P_{ij} \in \text{HALF-PLANE}(\mathcal{BD1}(p)^+)$, and $P_{ij} \in \text{HALF-PLANE}(\mathcal{BD2}(p)^+)$. The stencil buffer executes the stencil test on the pixels that are to be rendered on by any primitive. We set the stencil test so that for all the pixels in the buffer, the stencil operation would increment the corresponding value in the stencil buffer, i.e. $SB_{ij} = SB_{ij} + 1$, if $P_{ij} \in (1, 2)\text{-}\mathcal{BD}(p)^+$, where $\text{HALF-PLANE}(\mathcal{D}(p)^+)$ is the rendered primitive (see Figure 5(b)). Thus finally each SB_{ij} contains $|T|$. \square

5. Location depth

Given a set P of n points, the location depth of a point p , denoted $ldepth(p)$, is defined as the minimum number of points of P lying in a closed halfplane defined by a line through p . Define the location depth of a line l , $ldepth(l)$, as the minimum number of points lying in the two halfplanes defined by the line, i.e.

$$ldepth(l) = \min\{f(l^+), f(l^-)\}$$

Recall that due to the order-preservation property of the duality relation $\mathcal{D}(\cdot)$ defined in Section 4, the number of points of P in the positive halfplane of l is exactly the number of dual lines below the dual point of l , and it follows that $f(l^+) = level(\mathcal{D}(l))$ and $f(l^-) = n - level(\mathcal{D}(l))$. Consequently the location depth of a point p is equal to

$$\min_{q \in \mathcal{D}(p)} \min\{n - level(q), level(q)\}$$

The level of points on $\ell = \mathcal{D}(p)$ change only when ℓ intersects some other line $\ell' \in \mathcal{D}(P)$. Therefore, to compute the depth of a line, we need only examine its depth at each intersection point in the dual.

As mentioned in Section 4, the depth contours correspond to convex hulls of k - and $(n - k)$ -levels in the dual plane. We observe here that it is possible to compute convex hulls in a hardware-assisted fashion. However, this approach takes $\Omega(n)$ time for each depth contour, leading to an overall complexity of $\Omega(n^2)$. We now present a new grid-based method for computing the contour regions with $O(nW^2)$ running time.

This algorithm proceeds in two phases. First, we compute the dual arrangement and store the depths of all lines in the dual. However, since our dual (like the primal plane) is pixelized, we are actually able to compute the depth of every pixel in the dual by computing the arrangement as described in Section 4.2. We then use this information in the second phase to reconstruct the depth contours (without computing convex hulls). Algorithm 2 illustrates the algorithm.

LEMMA 5.1. *Let $q \in [-1, 1] \times [-1, 1]$, q in primal plane \mathbf{P} . Then $\exists d$ in the dual plane \mathbf{D} such that $q \in \mathcal{P}(d)$, $d \in \mathcal{D}(p)$ for some $p \in P$, $|d_x| \leq 1$, $|d_y| \leq 2$, and $Depth(q) = \min\{SB_d, n - SB_d\}$.*

PROOF. Let l be the line that realizes the contour depth of q . Then l must pass through some other point $p \in P$ (we can always rotate the minimizing line until it contains p). In the dual plane, $\mathcal{D}(l)$ is a point, for which we have $\mathcal{D}(l) \in \mathcal{D}(q)$, and $\mathcal{D}(l) \in \mathcal{D}(p)$. From Lemma 4.2, we know that either $(1, 2)\text{-}\mathcal{BD1}$ or $(1, 2)\text{-}\mathcal{BD2}$ must contain $d = \mathcal{D}(l) = \mathcal{D}(q) \cap \mathcal{D}(p)$, and therefore $-1 \leq d_x \leq 1$, $-2 \leq d_y \leq 2$. The depth follows from Lemma 4.3. \square

Algorithm 2 ComputeLocationDepth(PtSet P)

```

    BoundedArrangementBD = ComputeArrangement( $P$ )
    Initialize SegmentSet =  $\emptyset$ 
    for  $\mathcal{BD} = \mathcal{BD1} \ \mathcal{BD2}$  do
        Clear Color Buffer
        for  $i = 1 \dots n$  do
            DRAW(  $\mathcal{BD}(p_i)$  ) with Color 1.
        CB = Get Color Buffer
        for  $i = 1 \dots W$  do
            for  $j = 1 \dots H$  do
                if Color  $CB_{ij} = 1$  then
                     $cDepth = \min\{SB_{ij}, n - SB_{ij}\}$ 
                    SegmentSet = SegmentSet  $\cup$  ( $\mathcal{BP}(P_{ij}), cDepth$ )
            Enable Depth Test
            Set Depth Test = LESS
        for (Segment  $l = \mathcal{BP}(P_{ij}), cDepth$ )  $\in$  SegmentSet do
            DRAW(  $l$  ) with  $z$ -value =  $cDepth$ 
    
```

THEOREM 5.2. *Algorithm 2 computes the contour regions in the primal plane correctly.*

PROOF. Take any point q in the primal plane \mathbf{P} . From Lemma 5.1, we know there exists d in the dual plane \mathbf{D} such that the depth of point q is equal to $\min\{SB_d, n - SB_d\}$ and that $d \in \mathcal{BD}(p)$ for some $p \in S$, and either $\mathcal{BD1}$ or $\mathcal{BD2}$ (Lemma 4.2). In Algorithm 2, we render the dual line segment in the primal plane for every pixel that lies on the bounded dual of any point in P . Therefore, we render the dual line segment for d , $\mathcal{P}(d)$. The depth buffer ensures that the minimum SB_d is finally written, which corresponds to the depth of the point q . \square

Eliminating the stencil buffer. In some commercially available graphics cards, the resolution (number of bits) of the stencil buffer is not always sufficient to perform the depth computation for sufficiently large input. However the “increment” operation that we need from the stencil buffer can be simulated by adding colors in the color buffer (as described in section 3.2), using the *blending* function call that is standard in all graphics engines.

Complexity of our algorithm. Recall that the buffer size is $(2W + 1) \times (2W + 1)$. Consider the first phase, described in Algorithm 1. We draw n polygonal regions (really a clipped halfplane), at a total cost of $O(nW^2)$. Since the stencil buffer has only 8 bits per pixel, we may have depth value overflow with more than 512 points ($2^8 * 2$). Therefore, we have to read the stencil buffer back into main memory and continue, implying that we need $n/512$ readbacks.

In the second phase, described in Algorithm 2, we again render each dual line in the dual plane, at a cost of $O(nW)$. Then we examine each pixel of the dual plane, and for those that are part of a dual line, we render the primal line. Let the set of pixels that are contained in line l be $Pix(l)$. Then the running time of Algorithm 2 is $W \bigcup_{l \in \mathcal{D}(p)} Pix(l)$. Since $\bigcup_{l \in \mathcal{D}(p)} Pix(l) \leq W^2$, we obtain a total cost of $O(W^3)$.

Thus the total complexity of the algorithm is $O(nW + W^3) + nCW^2/512$, where C is a cost of a single **readback**. We do not collapse this with the $O(nW + W^3)$ term to emphasize that C is large compared to the constant hidden by the O -notation.

As mentioned, our implementation eliminates the stencil buffer by using blending in the color buffer. The major effect that this modification has is to reduce the number of **readbacks** in the first stage of the algorithm. Roughly speaking, since the color buffer has 3 8-bit channels (one for each color) we reduce the number of **readbacks** by a factor of 3 to $n/1536$, thus yielding a significant performance gain.

Discretization errors. The output produced by our algorithm is a rasterized set of contours. This digitization necessarily produces errors; in this section we analyze the nature and magnitude of these errors, showing that our algorithm has minimal error with respect to an algorithm that outputs vector objects (lines, points etc) rather than rasterized objects.

Our method for computing the depth contours has two phases. In the first, we construct a map (in the dual plane) that for each dual pixel gives the depth of its corresponding primal line. In the second phase, we use these depth values to reconstruct the contours in the primal plane. There are three main sources of error: (1) Sub-pixel detail in the input, (2) Computing the depth of a pixel in the dual plane, and (3) Rendering the lines (corresponding to each dual pixel) back in the primal.

Computing the depths. Input points may lie within a single pixel. As a result of digitization, all of these points will be collapsed into one pixel. Thus, in the sequel, we will assume that every pixel contains at most one point. We start with a lemma that quantifies the error incurred in snapping points to the grid. Let us assume that the *world coordinates* (that the input is presented in) are in the range $[-1, 1]$, and the pixel grid is of dimension $(2W + 1) \times (2W + 1)$, with pixel coordinates in the **integer** range $[-W, W]$. Let $[x]$ denote the value of x after rounding. A coordinate x in world coordinates is transformed to pixel coordinates by the mapping $x \rightarrow [Wx]$.

LEMMA 5.3 (Perturbation Lemma). *Let ℓ be a line in the primal plane, and let $p = \mathcal{D}(\ell)$ be its dual point. Let p' be a point in the dual plane such that $\|p' - p\|_\infty = \delta$. If ℓ' is the (primal) line such that $p' = \mathcal{D}(\ell')$, then the l_∞ distance between ℓ and ℓ' is at most $2[W\delta]$ in pixel coordinates.*

PROOF. Let the line ℓ be described by the equation $y = mx + c$. The dual point p is thus $(-m, c)$. One such point p' is the point $(-m - \delta, c + \delta)$. This yields $\ell' : y' = (m + \delta)x + c + \delta$. Thus, for a fixed x , the difference $y' - y = \Delta y$ is $\delta(x + 1)$.

Switching to pixel coordinates, the difference in pixels $\Delta Y = [W\Delta y]$ is $[W(\delta(x + 1))]$, which is at most $2[W\delta]$. \square

As noted earlier, the correctness of the depth values computed in the dual depends on whether we correctly identify all intersection points in the dual arrangement. As a result of pixelization, it is possible that two intersection points may collapse, or that in general three intersection points may collapse to one (corresponding to making three almost-collinear primal points collinear). We now present a necessary condition for such an event to happen.

Each pixel in the screen is a square of side $1/W$. Consider three points $p_i = (a_i, b_i), i = 1, 2, 3$. The three dual lines are $l_i : y = -a_i x + b_i$, and the intersection points are: $(l_i, l_j) = (\frac{b_i - b_j}{a_i - a_j}, \frac{a_i b_j - a_j b_i}{a_i - a_j}), (i, j) = \{(1, 2), (2, 3), (1, 3)\}$.

For the three intersection points to lie inside the same pixel, the component-wise absolute difference in the x and y coordinates (in object coordinates) must be less than $\frac{1}{2W}$.

From Lemma 5.3, we can thus conclude that the three intersection points collapse if the ℓ_∞ distance between the corresponding primal lines is less than $2W(1/2W) = 1$ (pixel), thus yielding the following lemma:

LEMMA 5.4 (Intersection Point Collapse). *Given three points p_1, p_2, p_3 that are not collinear, our dual mapping will identify distinct intersection points for each pair of dual lines if the corresponding primal lines are at least 1 unit (in pixel coordinates) apart from each other.*

Using the same argument, we can also derive a lower bound on the dimension of the pixel grid, W . Let the set of distinct triples of points from the original point set be T . Given an element $t_{ij}^k = \langle p_i, p_j, p_k \rangle \in T$, define v_{ij}^k as

$$v_{ij}^k = \min \left(\left| \frac{b_i - b_k}{a_i - a_k} - \frac{b_j - b_k}{a_j - a_k} \right|, \left| \frac{a_i b_k - a_k b_i}{a_i - a_k} - \frac{a_j b_k - a_k b_j}{a_j - a_k} \right| \right)$$

Then the condition that the intersections of the lines $(\mathcal{D}(p_i), \mathcal{D}(p_k))$ and $(\mathcal{D}(p_j), \mathcal{D}(p_k))$ in the dual will not collapse becomes $v_{ij}^k \geq 1/2W$. Extending this condition over all elements in T , we get $W \geq \frac{1}{2 \min_{t_{ij}^k \in T} v_{ij}^k}$.

Contour reconstruction. Let \mathcal{I} be an instance of the depth contour problem, and let \mathcal{A} be an algorithm that computes depth contours exactly (i.e using infinite precision geometry). Let \mathcal{O} denote $\mathcal{A}(\mathcal{I})$, and let S be the display produced by rendering \mathcal{O} . Let S' be the display produced by our algorithm.

LEMMA 5.5. *Let ℓ' be a line in \mathbf{P} drawn in the last step of Algorithm 2. Then ℓ is at most 1 pixel away from the corresponding line ℓ in \mathcal{O} .*

PROOF. Let $p = \mathcal{D}(\ell)$ be the dual point corresponding to ℓ . From Lemma 5.4, we are guaranteed that in the dual buffer constructed by Algorithm 1, there exists a pixel that contains p . By Lemma 5.3, we know that the line ℓ' drawn in the primal is at most $2W(1/2W) = 1$ unit away from ℓ . \square

Clearly, the result of rendering ℓ on the display cannot make its distance to ℓ' worse by more than a pixel width (by our rasterization assumption). Hence, we conclude that our algorithm preserves the visual fidelity of the ideal output \mathcal{O} .

6. Dynamic Location Depth

In the previous section, we presented a linear time algorithm that computes the location depth of a set of static points in \mathbb{R}^2 . Although the dependency of the algorithm on n is linear, the algorithm performs buffer readbacks, which are computationally costly. We now present another algorithm, with a completely different approach, that avoids buffer readbacks altogether (though the dependency on n becomes quadratic). This allows this algorithm to maintain, in real time, the depth contours of a small set of moving points.

First use the procedure **ComputeArrangement** to compute the dual arrangement of the point set P . Then enable the depth test, and for every line l passing through a pair of input points, render the positive and negative half-planes of l with their depth values corresponding to $f(l^+) = \text{level}(\mathcal{D}(l))$ and $f(l^-) = n - \text{level}(\mathcal{D}(p))$

Algorithm 3 ComputeLocationDepth(PtSet P)

```

BoundedArrangement { $SB1, SB2$ } = ComputeArrangement( $P$ )
Enable Depth Test
Set Depth Test = LESS
for  $i = 1 \dots n$  do
  for  $j = (i + 1) \dots n$  do
     $l = l(p_i, p_j)$ 
    /* Using { $SB1, SB2$ } to answer the level query  $level(\mathcal{D}(l))$  */
    DRAW ( HALF-PLANE(  $l^+$ ),  $z$ -value =  $f(l^+)$ , color =  $f(l^+)$  )
    DRAW ( HALF-PLANE(  $l^-$ ),  $z$ -value =  $f(l^-)$ , color =  $f(l^-)$  )

```

respectively. The entire algorithm is shown in Figure 3. Note that the depth test ensures that the final color value at a point u is the minimum $f(\cdot)$ value over all the halfplanes that contain u .

THEOREM 6.1. *Algorithm 3 computes the contour regions in the primal plane correctly, and takes $O(n^2W^2)$ time with no readbacks.*

PROOF. We show that for each point in the plane, the location depth computed by the above algorithm is correct. Take a point $u \notin P$ in the primal plane and without loss of generality, let l^+ be the halfplane defined by line l that realizes the location depth of u . We show that (i) there exist two points $p, q \in P$ such that $u \in l(p, q)^+$ and $f(l^+) = f(l(p, q)^+)$, and (ii) for all pairs $p, q \in P$ such that $u \in l(p, q)^+$ (respectively $l(p, q)^-$), $f(l^+) \leq f(l(p, q)^+)$ (respectively $f(l(p, q)^-)$). These together imply the correctness.

- i. Translate the line l vertically downwards until it hits the first point in P , say p . Then rotate it about p clockwise or anti-clockwise (whichever finds the second point first) until it hits the second point q (See Figure 6(a)). Now note that $u \in l(p, q)^+$ and $f(l^+) = f(l(p, q)^+)$. If it was a negative halfplane realizing the depth of u , the line l would be translated vertically upwards.
- ii. Take any two points p and q such that $u \in l(p, q)^+$. Translate the line $l(p, q)$ vertically upwards until it hits the point u (See Figure 6(b)). Let this line be l' . Note that $f(l'^+) \leq f(l(p, q)^+)$ (since l' lies above $l(p, q)$), and $f(l^+) \leq f(l'^+)$ (by definition of l). Thus $f(l^+) \leq f(l(p, q)^+)$. Similarly if $u \in l(p, q)^-$ translate the line $l(p, q)$ downwards and a similar case holds.

There are $O(n^2)$ halfplanes, and each takes $O(W^2)$ time to render. \square

7. Oja depth

Take any point $q \in \mathbb{R}^2$. Let $p_i, p_j \in P$ be two points in the input pointset. The Oja depth of q is simply the sum of the areas of the triangles formed by q and all pairs of points in P . Given p_i and p_j , let l_{ij} be the line passing through p_i and p_j . The area of the triangle formed by q, p_i, p_j is $1/2 \cdot d(p_i, p_j) \cdot d(q, l_{ij})$. Essentially, the area of the triangle can be thought of as the weighted distance of q from l_{ij} , $w_{ij} \cdot d(q, l_{ij})$, where $w_{ij} = \frac{d(p_i, p_j)}{2}$. So the area of triangle formed by

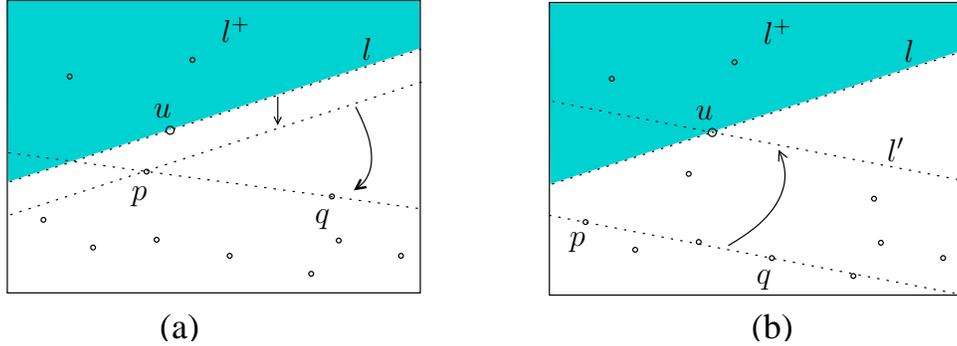


FIGURE 6. (a) Points p and q such that $f(l^+) = f(l(p, q)^+)$, (b) $f(l^+) \leq f(l'^+) \leq f(l(p, q)^+)$.

points r, p_i, p_j is then simply $w_{ij} \cdot d(r, l_{ij})$. And the Oja depth of a point q becomes $\sum_i \sum_{j \neq i} w_{ij} \cdot d(q, l_{ij})$.

Given a point $p_i = (x_i, y_i)$, look at the cone $z = w_{ij} \cdot \sqrt{(x - x_i)^2 + (y - y_i)^2}$, where w_{ij} is some constant. It is easy to verify that the height of the cone at point q is exactly $w_{ij} \cdot d(q, p_i)$. Therefore take the line, l_{ij} , through the two points p_i and p_j , and let w_{ij} be as defined above. Then if we place the cone defined above at every point on this line, we essentially get a wedge defined by the line. It follows that the height of this wedge at point q is $w_{ij} \cdot d(q, l_{ij})$ which is exactly the area of the triangle q, p_i, p_j .

The algorithm is the following. For each one of the $\binom{n}{2}$ lines defined by pairs of points of P , let the weight of each line l_{ij} be $d(p_i, p_j)/2$. Then by using the adding feature provided by the blending function of the color buffer, we can render all the wedges as defined above for each line. After rendering the wedge of each line, the value at each pixel of the color buffer is simply the sum of the distances of the pixel to the lines which is the sum of the areas of all the triangles defined by that pixel and all pairs of points of P , which is the Oja depth. Thus we get the Oja contours in the color buffer.

8. Simplicial depth

Our algorithm for computing simplicial depth contours follows the general paradigm of the location depth contour algorithm of Section 5. We first compute the bounded duals of each point in P , $\mathcal{BD}1(P)$ and $\mathcal{BD}2(P)$ exactly as computed in Section 4.2. In the second phase, we again go over all the pixels in the duals $\mathcal{BD}(P)$, and for each pixel, render the dual line in the primal plane with a specific color. However, instead of maintaining the minimum color at each pixel, as in location depth contours, we subtract the color of the line from the color of all the pixels it contains. The complete algorithm is shown in Algorithm 4.

THEOREM 8.1. *Algorithm 4 computes the simplicial contour regions in the primal plane correctly.*

PROOF. Take any point q in the primal plane \mathbf{P} . Sort the points anti-clockwise around q to get $\langle p_{i_1}, \dots, p_{i_n} \rangle$. Take any line through q and p_{i_j} , $l(p_{i_j})$. Then as explained above, $T(p_{i_j})$ is defined by the quantity $|l(p_{i_j})^+ \cap P|$. Recall that

Algorithm 4 ComputeSimplicialRegions(Bounded Arrangement SB)

```

Initialize SegmentSet =  $\emptyset$ 
for  $\mathcal{BD} = \mathcal{BD}1 \ \mathcal{BD}2$  do
  Clear Color Buffer.
  for  $i = 1 \dots n$  do
    DRAW(  $\mathcal{BD}(p_i)$  ) with Color 1.
  CB = Get Color Buffer
  for  $i = 1 \dots W$  do
    for  $j = 1 \dots H$  do
      if Color  $CB_{ij} = 1$  then
         $cColor = \binom{SB_{ij}}{2}$ 
        SegmentSet = SegmentSet  $\cup$  ( $\mathcal{BP}(P_{ij}), cColor$ )
  Enable Color Blending
  Set Color Blending Function = SUBTRACT
  Set Color Buffer:  $CB_{ij} = \binom{n}{3}$ 
  for (Segment  $l = \mathcal{BP}(P_{ij}), cColor$ )  $\in$  SegmentSet do
    DRAW(  $l$  ) with Color-Value =  $cColor$ 

```

$\mathcal{D}(l(p_{i_j}))$ is actually a pixel, say pixel r , lying on the dual line $\mathcal{D}(p_{i_j})$. Furthermore, $|l(p_{i_j})^+ \cap P|$ is exactly SB_r . Therefore $T(p_{i_j}) = \binom{SB_r}{2}$. Since we render the dual line segment in the primal plane for every pixel that lies on the bounded dual of p_{i_j} , we would render the dual line of pixel r , with color $T(p_{i_j})$. This color gets subtracted from the already present color in the pixel containing q , which was initially set to $\binom{n}{3}$. The above subtraction is done for every point in P (and hence every $T(\cdot)$ is subtracted), and thus finally the color value at pixel containing q is exactly $SDepth(q)$ from Equation 1. The same is true for all the other points in the plane. \square

The time complexity of this algorithm is the same as the time complexity of the location depth contours algorithm.

Remark. One has to be a little careful when performing the above procedure for each of the two duals $\mathcal{BD}1$ and $\mathcal{BD}2$ due to overlapping pixels. It could happen that the dual pixel of the line through q and some p_i is present in both the duals $\mathcal{BD}1$ and $\mathcal{BD}2$. Naively performing the above procedure for each of the two duals then results in subtracting $T(p_i)$ twice from the simplicial depth value of q . Hence, after performing Algorithm 4 for bounded dual $\mathcal{BD}1$, only certain (easily computable) pixels of the other bounded dual $\mathcal{BD}2$ need to be examined.

9. Colored depth

We now present an extended version of the classical location depth contour problem. Let c be a (color) function $c : P \rightarrow \{1, \dots, k\}$. The extended location depth of a line l is the minimum number of distinctly colored points of P lying on either side,

$$EDepth(l) = \min\{ |c(P \cap l^-)|, |c(P \cap l^+)| \}$$

Then $EDepth(q) = \min_{q \in l} EDepth(l)$, for a point $q \in \mathbb{R}^2$. The set of all points in \mathbb{R}^2 having *extended* location depth at least k is called the extended depth contour

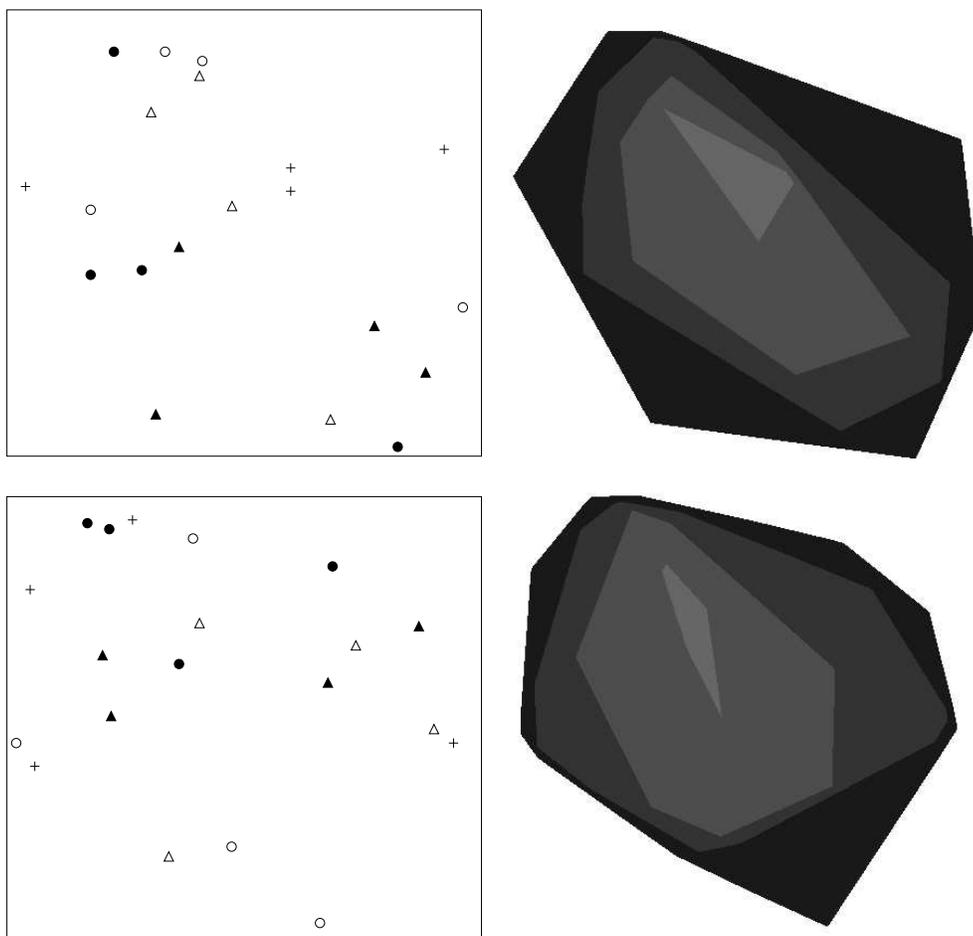


FIGURE 7. Extended depth contours for two different colored point sets.

of depth k . See Figure 7 for an illustration of extended contours computed by our algorithm. Similar to classical depth contours, it is possible to show that the extended depth contours are nested, and each extended contour region is convex.

LEMMA 9.1. *For all possible k , the extended depth contour of depth k is convex.*

PROOF. By definition, the $(k + 1)$ -extended contour is contained inside the k -extended contour. Every point in k -extended contour, but not in $(k + 1)$ -extended contour has a line l passing through it with k distinctly colored points in one halfplane, say l^+ . Then l^+ (and l) do not intersect the $(k + 1)$ -extended contour.

Now assume that $(k + 1)$ -extended contour is not convex. Then there is a point inside the convex hull of $(k + 1)$ -extended contour that is in k -extended contour, but not in $(k + 1)$ -extended contour. But then any line passing through this point must intersect the $(k + 1)$ -extended contour – a contradiction. \square

Algorithm 5 ComputeExtendedArrangement(PtSet P)

```

Sort  $P$  on color  $\mathcal{C}$  to get  $P_1, \dots, P_k$ .
Set  $\bar{p}_{i_j} = (p_{i_j}, \frac{i+j/|P_i|}{k}), p_{i_j} \in P_i$ .
Enable STENCIL TEST, DEPTH TEST
Set STENCIL OP = { DEPTH-PASS; INCREMENT, DEPTH-FAIL: NOP }
Initialize StencilBuffer = 0
for  $P_i, i = k, (k-1), \dots, 1$  do
  for  $p_{i_j} \in P_i, j = 1, 2, \dots, |P_i|$  do
    RENDER  $\bar{p}_{i_j}$ .
Save Stencil Buffer

```

If one can compute the arrangement $\mathcal{D}(P)$ such that the depth of each point q in this dual is equal to the extended depth of the line $\mathcal{P}(q)$ in the primal, that combined with Algorithm 2 of the location depth contour algorithm yields an algorithm to compute the set of extended depth contours.

Given an arrangement \mathcal{L} of colored lines, the level of a point in \mathcal{L} is the number of *distinctly* colored lines strictly below or passing through it. For each point $p_i \in P$, define $c(\mathcal{D}(p_i)) = c(p_i)$. Sort the polygons $\mathcal{D}(p_i)^+$ by color, i.e. $\mathcal{L} = \mathcal{L}_1 \cup \dots \cup \mathcal{L}_k$, where \mathcal{L}_i is the subset of \mathcal{L} with color c_i . Let $\mathcal{L}_i = \{\mathcal{D}(p_{i_1})^+, \dots, \mathcal{D}(p_{i_m})^+\}$. Define $\bar{\mathcal{D}}(p_{i_j}) = (\mathcal{D}(p_{i_j})^+, \frac{i+j/|\mathcal{L}_i|}{k})$ where $\mathcal{D}(p_{i_j})^+ \in \mathcal{L}_i$. Then define $\bar{\mathcal{L}}_i = \{\bar{\mathcal{D}}(p) | \mathcal{D}(p)^+ \in \mathcal{L}_i\}$. This operation *lifts* each polygon in the set \mathcal{L}_i to a location in three dimensions in the z -range $[\frac{i}{k}, \frac{i+1}{k}]$.

Note that each polygon belonging to \mathcal{L}_i has z -value less than any polygon belonging to set $\mathcal{L}_{i'}, i < i'$. Render all the polygons in the following order: render \mathcal{L}_k first, and proceed until we reach \mathcal{L}_1 (in a back-to-front order). Within each set \mathcal{L}_i , polygons are rendered front-to-back, i.e. in the order of increasing z -value. During this process, the graphics state is set so that the stencil buffer (at a pixel) gets incremented only once for each color by using the *min* depth test operation. Algorithm 5 describes this formally.

LEMMA 9.2. *After running the above algorithm, SB_{i_j} contains $|T|$, where $T = \{c(p) | p \in P, P_{i_j} \in \mathcal{D}(p)^+\}$.*

PROOF. Polygons $\mathcal{D}(p)^+ \in \mathcal{L}_i$ are rendered front-to-back, and the z -buffer depth checking enforces the constraint that any pixel belonging to at least one polygon in \mathcal{L}_i has the corresponding stencil value incremented *exactly* once. Given a pixel q , define $\delta_{q,i} = 1$ if $q \in \cup_j \mathcal{D}(p_{i_j})^+$, and 0 otherwise. We render all the polygons of the set \mathcal{L}_i after rendering the polygons of $\mathcal{L}_{i'}, i' > i$. The lifting procedure guarantees that the depth of polygons in \mathcal{L}_i is always less than the depth of any polygon in $\mathcal{L}_{i'}, i' > i$. This implies that the depth test is trivially satisfied between polygons of different color. After rendering all the polygons, the stencil buffer at pixel q contains $\sum_{i=1}^k \delta_{q,i}$ which is the required value. \square

Repeating the same computation for the negative half-planes $\mathcal{D}(P)^-$, it follows from Lemma 9.2 that we can compute the arrangement of colored lines, and after Algorithm 2 then yields the actual set of extended depth contours.

10. Experiments

We now present experiments in support of our location depth algorithm. We compare the performance of the algorithm on the three different platforms: the SGI Octane (R12000 300MHz CPU, 512 MB Memory, EMXI graphics card), the SGI Onyx (R10000 194MHz CPU, 2GB Memory, InfiniteReality graphics card), and the LinuxPC-Nvidia GeForce 3 (AMD Athlon 900 MHz CPU, 768 MB Memory, Nvidia GeForce 2 graphics card). In all the platforms, the code was written in C++ and made extensive use of OpenGL libraries. On the SGI machines, it was compiled using CC and on the Linux machine, with g++.

We present two suites of experiments. In the first set, we evaluate the running time of the algorithm for various data sets of different sizes. In the second set of experiments, we illustrate the zooming capability of our algorithm. As mentioned earlier, the ability to navigate a data set in this fashion is very useful for interactive visualization, and also allows us to explore in more detail regions of the contour map that may have too much detail to be displayed at the initial screen resolution.

10.1. Running times. Measuring the performance of the algorithm is non-trivial. The OpenGL pipeline consists of two main parts - Transformation and Lighting, and the Rasterization part. The performance of the graphics engine in most typical applications is a complicated function determined by a combination of the transformation and rasterization parts. We report two different time measurements for each run; the clock time as reported by `clock()` and the real time elapsed. For the Linux PC, the two measurements are close to each other, indicating that the `clock()` routine is a reliable measure of rendering time, but for the SGI machines, there is a large disparity. Typically, performance measurements on graphics hardware assumes that all the primitives are transferred to the graphics subsystem once and its cost is amortized over several frames. The benchmarks we describe here (the line and triangle drawing) is different because we measure the performance over a single frame. Therefore, the cost of transferring the primitives dominates our performance.

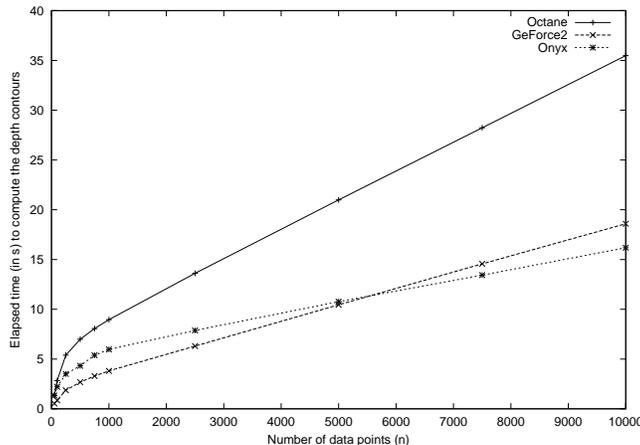


FIGURE 8. Elapsed running time as a function of n .

In this suite of experiments, we generate random data sets (akin to [15]) of different sizes to test our algorithm. Figure 8 plots the running time vs size of input on the three platforms that we experimented on. In all cases, the predicted linear behavior (with respect to n) of the algorithm is clearly demonstrated.

We also present the actual running times in Table 2. We were unable to compare our implementation directly to the implementation of Miller *et al* [15], which appears to be the fastest method for computing depth contours. However, we used a similar methodology to generate our input, and we note that they report a time of 22.4 seconds to compute the entire contour plot for a data set of 1000 points, and the plot that they present indicates the quadratic growth of their algorithm. In comparison, a 1000 point data set is processed by our algorithm runs in 3.81 seconds on a SGI Onyx, and even for 10,000 points, our algorithm takes only 16.18 seconds. It is quite likely that as n increases, our algorithm will exhibit much more than a 10-fold improvement in speed over their method.

	SGI Octane		SGI Onyx		LinuxPC/GeForce 2	
	clock	Elapsed	clock	Elapsed	clock	Elapsed
50	0.93	1.36	1.29	1.34	0.6	0.53
100	1.45	2.84	1.98	2.21	0.94	0.85
250	2.26	5.39	3.2	3.50	1.49	1.87
500	2.78	6.99	3.98	4.32	1.93	2.68
750	3.01	8.06	4.43	5.38	2.21	3.29
1000	3.27	8.96	4.83	5.97	2.49	3.81
2500	3.76	13.61	5.97	7.87	4.00	6.30
5000	4.31	20.99	7.14	10.76	6.34	10.45
7500	4.84	28.23	8.88	13.43	8.7	14.56
10000	5.45	35.48	10.34	16.18	11.09	18.59

TABLE 2. Running times on the three platforms.

10.2. Zooming. Using our approach, we can also zoom into a depth contour rendering to examine areas in close detail. Note that a mere zooming of the frame buffer is not sufficient; especially for highly detailed data sets, the depth contours may need to be re-computed as we zoom in.

Consider the data set and its depth contour set presented in Figure 9. The software interface that we provide allows us to mark rectangular areas of the depth contour map to zoom into, and in Figure 9(c),(d) we show the effect of progressively zooming in closer and closer to the point set.

Zooming has the effect of increasing the resolution in the defined region, thus allowing us to view features that may have been hidden inside a pixel at a lower resolution.

References

- [1] ALOUPIS, G. On computing geometric estimators of location, 2001. M.Sc. Thesis, McGill University.
- [2] BOROS, E., AND FÜREDI, Z. The maximal number of covers by the triangles of a given vertex set on the plane. *Geom. Dedicata* 17 (1984), 69–77.

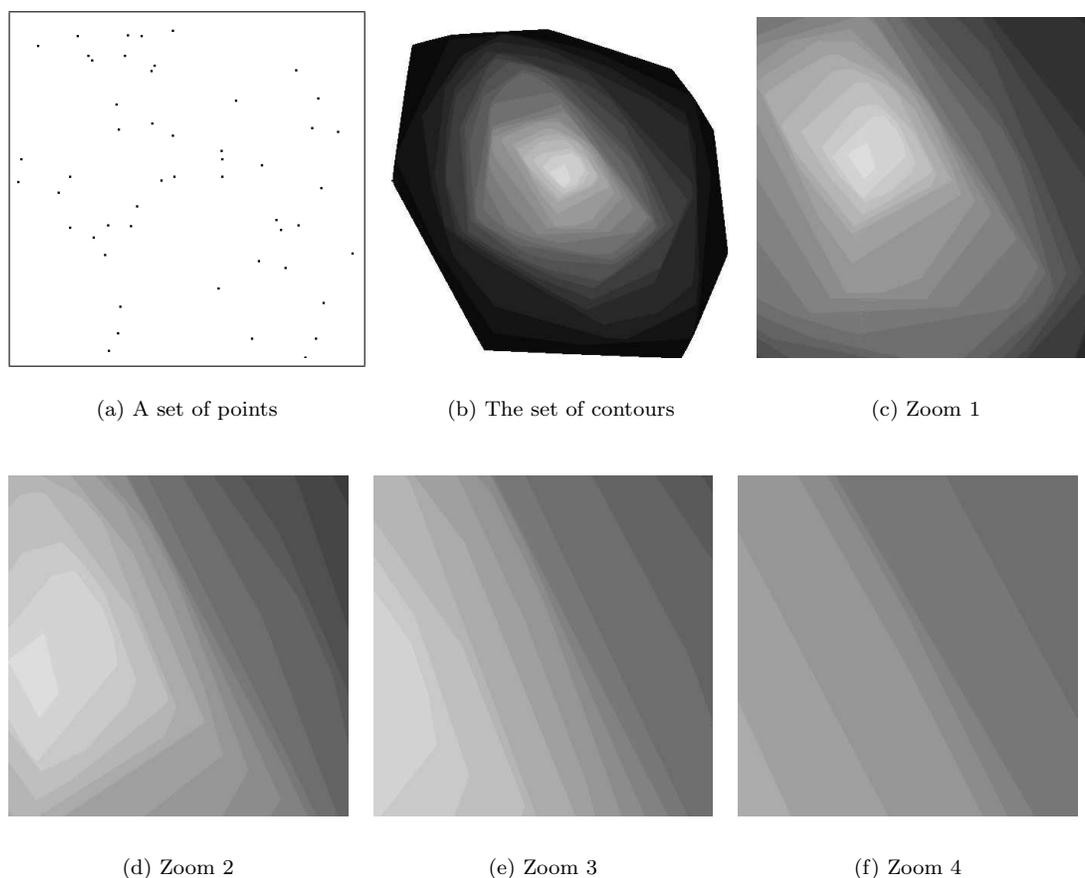


FIGURE 9. Progressive zooms.

- [3] CHAN, T. M. An optimal randomized algorithm for maximum tukey depth. In *SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms* (2004), Society for Industrial and Applied Mathematics, pp. 430–436.
- [4] CHAZELLE, B. On the convex layers of a planar set. *IEEE Transactions on Information Theory* 31 (1985), 509–517.
- [5] COLE, R., SHARIR, M., AND YAP, C. K. On k -hulls and related problems. *SIAM Journal on Computing* 15, 1 (1987), 61–77.
- [6] EDELSBRUNNER, H. *Algorithms in Combinatorial Geometry*, vol. 10 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Heidelberg, West Germany, 1987.
- [7] GIL, J., STEIGER, W., AND WIGDERSON, A. Geometric medians. *Discrete Mathematics* 108 (1992), 37–51.
- [8] HODGES, J. A bivariate sign test. *Annals of Mathematical Statistics* 26 (1955), 523–527.
- [9] INDYK, P. Sublinear time algorithms for metric space problems. In *Proc. 31st ACM Symposium on Theory of Computing* (1999).
- [10] JADHAV, S., AND MUKHOPADHYAY, A. Computing a centerpoint of a finite planar set of points in linear time. *Discrete Computational Geometry* 12 (1994), 291–312.
- [11] LANGERMAN, S., AND STEIGER, W. The complexity of hyperplane depth in the plane. In *Japan conference on discrete and computation geometry* (2000).

- [12] LANGERMAN, S., AND STEIGER, W. Optimization in arrangements. In *Proceedings of the 20th International Symposium on Theoretical Aspects of Computer Science (STACS 2003)* (2003), vol. 2607 of *LNCS*, Springer-Verlag, pp. 50–61.
- [13] LIU, R. A notion of data depth based upon random simplices. *The Annals of Statistics* 18 (1990), 405–414.
- [14] MATOUŠEK, J. Computing the center of planar point sets. *DIMACS Series in Discrete Mathematics and Computer Science* 6 (1991), 221–230.
- [15] MILLER, K., RAMASWAMI, S., ROUSSEEUW, P., SELLARES, T., SOUVAINE, D., STREINU, I., AND STRUYF, A. Fast implementation of depth contours using topological sweep. In *Proc. 12th ACM-SIAM Sympos. Discrete Algorithms* (2001), pp. 690–699.
- [16] OJA, H. Descriptive statistics for multivariate distributions. *Statistics and Probability Letters* 1 (1983), 327–332.
- [17] ROUSSEEUW, P., AND RUTS, I. Bivariate location depth. *Applied Statistics* 45 (1996), 516–526.
- [18] ROUSSEEUW, P. J., AND HUBERT, M. Regression depth. *Journal of the American Statistical Association* 94 (1999), 388–402.
- [19] ROUSSEEUW, P. J., AND RUTS, I. Constructing the bivariate tukey median. *Statistica Sinica* 8 (1998), 827–839.
- [20] ROUSSEEUW, P. J., RUTS, I., AND TUKEY, J. W. The bagplot: A bivariate boxplot. *The American Statistician* 53, 4 (1999), 382–387.
- [21] SHAMOS, M. *Recent results and new directions in algorithms and complexity*. Academic Press, 1976, pp. 251–280.
- [22] TUKEY, J. Mathematics and the picturing of data. In *Proc. of the international congress of mathematicians* (1975), pp. 523–531.

AT&T LABS – RESEARCH
E-mail address: krishnas@research.att.com

DEPT. OF COMPUTER SCIENCE, DUKE UNIVERSITY
E-mail address: nabil@cs.duke.edu

AT&T LABS – RESEARCH
E-mail address: suresh@research.att.com