Hardware-Assisted Computation of Depth Contours

Shankar Krishnan
AT&T Labs – Research
krishnas@research.att.com

Nabil H. Mustafa[*]
Dept. of Computer Science,
Duke University
nabil@cs.duke.edu

Suresh Venkatasubramanian
AT&T Labs – Research
suresh@research.att.com

Abstract

Given a set of points P in the plane, the *location depth* of a point u is the minimum number of points of P lying in a closed halfplane defined by a line through u. The set of all points in the plane having location depth at least k is called the *depth contour* of depth k. In this paper, we present an algorithm that makes extensive use of modern graphics architectures to compute the approximate depth contours of a set of points. The output of our algorithm presents the contours as a coloring of each point with its depth value, as opposed to computing the geometric description of the contour boundary. Our algorithm performs 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

The *location depth* of a point with respect to a fixed set of input points has been used in statistics as a way to identify the center of a bivariate distribution [10]. 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 [22]; the Tukey median is the contour of points at maximum location depth. As explained in [22], 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" [22]. 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.

From a geometric perspective, the location depth of a point is interesting in terms of its relation to k-levels of an arrangement. The problem of computing *centerpoints*
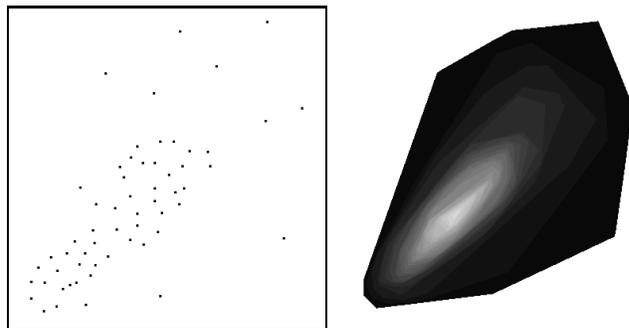

Figure 1: An illustration of depth contours

(which are essentially points of location depth $\lceil \frac{n}{d+1} \rceil$ in a d-dimensional space) has been examined earlier; Cole, Sharir and Yap [2] presented an algorithm that computes center-points in the plane in time $O(n \log^5 n)$, and Naor and Sharir extended this to three dimensions in time $O(n^2 \log^6 n)$. Matoušek [15] proposed an algorithm that for any k, computes the set of points at depth k in time $O(n \log^4 n)$; he also computes the Tukey median in time $O(n \log^5 n)$.

The location depth and depth contour problems have applications in hypothesis testing, robust statistics, and even in various problems in cell biology (especially for moving point sets) [3]. Hence, it is important to develop fast implementations for use in these domains. In the statistics community, programs like ISODEPTH [23], HALFMED [21], and BAGPLOT [22] have been used for this purpose.

More recently, Miller *et al* [17] 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 $O(\log^2 n)$. They demonstrate that compared to HALFMED (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.

1.1 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. In the graphics community, Rossignac et. al. [20] and Goldfeather et. al. [4] use the graphics hardware to perform real-time constructive solid geometry. Greene et. al. [7] have used the "Z query" feature of some specialized graphics hardware to implement their hierarchical Z-buffering algorithm. Hoff *et al* [11] use the hardware Z-buffering capabilities to compute Voronoi diagrams of dynamic primitives in real-time, which is then used for accelerated interactive motion-planning [12]. Krishnan *et al* [13] describe a visibility ordering algorithm for direct volume rendering of unstructured grids using a combination of the Z-buffer and the stencil buffer.

The paper by Hoff *et al* is an example of using the graphics hardware to compute a fundamental geometric object: the Voronoi diagram. In recent work, Mustafa *et al* [18] use similar techniques to simplify large geographic maps in real-time; once again, the use of the graphics hardware enables them to achieve real-time interactivity for a fairly complex problem.

1.2   Algorithm Analysis

Complexity. The use of graphics hardware presents an interesting problem for algorithm design. Since the graphics engine operates like a finite state machine, with each geometric object passing through a rendering pipeline, certain operations can be performed extremely efficiently, and others (like reading values of hardware buffers) are much more expensive. To understand the behavior of algorithms designed on this platform, and to get an intuitive notion of what is "cheap", an abstract cost model of the graphics engine would be extremely useful. We present a preliminary version of such a model in this paper. Similar approaches have been taken by Peercy *et al* [19] and Lindholm *et al* [14].

Robustness. The tremendous improvements in performance that one can achieve by using graphics hardware come at a price. Since the lowest unit processed by the graphics engine is a pixel, features in the input that are at sub-pixel resolution may be lost. Therefore it is important to quantify the error that the output produced by such a method will have, and formulate methods to deal with this problem.

The reader might notice the connection between our work on raster displays and the vast body of work on finite resolution computational geometry [6, 9, 5, 8, 16]. One of the main directions (especially in the context of *geometric rounding* is to modify the output of an exact algorithm (for example for computing line intersections in the plane) so that when snapped to a grid, the topology of the arrangement is preserved (or at least is consistent with the original). More recently, algorithms have been designed to do the output computation and rounding together, for improved efficiency. All of these approaches assume that the algorithm has con-

trol over the rounding process; techniques like shortest-path rounding[16] heavily rely on this. However, we have no control over the rasterization process, and this is what makes techniques from this area difficult to adapt in our domain. Moreover, our methods heavily use operations in the hardware that would be very inefficient to perform in software. We also note that we are not aware of any algorithms for computing depth contours that operate on a finite grid.

1.3   Our Results

In this paper, we present an algorithm that makes extensive use of modern graphics architectures to compute the depth contours of a set of points, and allow querying of location depth. Our implementation is significantly faster than that of Miller *et al*; on a data set of size 1000, our implementation runs in roughly 4 seconds, and our growth rate is linear in n, as compared to the above mentioned implementations. In fact, our implementation takes only 18.6 seconds to compute depth contours for a set of 10,000 points; for this size, [17] does not report timings.

After defining the problem in Section 2, we present an abstract model of the graphics engine in Section 3. In Section 4 we present our hardware-based algorithm. Section 5 defines various notions of error that we then use to evaluate the quality of our results. Section 6 first proposes an asymptotic cost model for the basic hardware operations, based on experiments conducted on various hardware platforms, and then analyses the complexity of our scheme in terms of this model. We present an experimental evaluation of our algorithm in Section 7, and conclude with some interesting open questions in Section 8.

2   Problem Definition

DEFINITION 2.1. (LOCATION DEPTH) *Let* P *be a set of points in* $\mathbb{R}^2$. *The location depth of a point* $u \in \mathbb{R}^2$ *(with respect to* P) *is the minimum number of points of* P *lying in a closed hyperplane defined by a line through* u.

DEFINITION 2.2. (DEPTH CONTOURS) *Let* P *be a set of points in* $\mathbb{R}^2$. *The set of all points in* $\mathbb{R}^2$ *having location depth at least* k *(with respect to* P) *is called the depth contour of depth* k *(or the* k-*hull).*

The depth contour of depth 1 is merely the interior of the convex hull of P. Each contour of depth k is a convex polygon contained inside the contour of depth $k - 1$. For a point set of size n, it is known from Helly's theorem that there always exists a point of depth $n/3$, and the maximum depth can be at most $n/2$.

PROBLEM 2.1. (DEPTH CONTOURS) *Given a point set* P, *compute the set of all depth contours.*

Depth contours have a natural characterization in terms of the arrangement in the dual plane induced by P. This

property was used by Matoušek [15] and Miller *et al* [17] in their algorithms for computing k-hulls.

The mapping from the primal plane **P** to the dual plane **D** is denoted by the operator $\mathcal{D}(\,\cdot\,)$: $\mathcal{D}(p)$ is the line in plane **D** dual to the point p, and similarly $\mathcal{D}(l)$ is the point in plane **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 **P** that is the dual of point q in the dual plane **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 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* [17] exploits this property by using a topological sweep in the dual to compute the set of all depth contours.

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.

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. 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 contain **RGB** color information[1]. 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.

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 $R_1 G_1 B_1$, and a user-specified constant K that can be specified before rendering any primitive (it cannot be varied per fragment though). *SB op* and *DB 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 *ReadBuffer* operation, which reads a specified buffer from the frame buffer into the main memory of the machine.

4 Our Algorithm

Our 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. We then use this information in the second phase to reconstruct the depth contours (without computing convex hulls).

---

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

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

Table 1: Operations

4.1 Duals and Bounded Arrangements

Without loss of generality, we assume $p_x, p_y \in [-1, 1]$. Each buffer consists of a $(2W+1) \times (2W+1)$ uniform grid of pixels. Consider the standard duality relation: $p = (p_x, p_y)$ in the primal plane maps to the line $y = -p_x x + p_y$ in the dual plane. Define the (contour) depth of a line $\mathcal{D}(p)$ as the (contour) depth of p. The depth of line $\ell \in \mathcal{D}(P)$ is then $\min_{q \in \ell} \min\{n - level(q), level(q)\}$. The level of points on ℓ 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.

Since our screen size is bounded, it is possible that an intersection point in the dual may lie outside the screen boundary (for example, when two lines are nearly parallel), which would introduce error in computing the depth of a line incident on that point. To remedy this problem, we introduce the idea of *bounded duals*:

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

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

Any two line segments in $(x, y) - \mathcal{BD}(P)$ 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 4.1. (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 $(x, 2y)$-$\mathcal{BD}1(P)$ and $(x, 2y)$-$\mathcal{BD}2(P)$ such that each intersection point $q \in \mathcal{D}(P)$ lies either in $(x, 2y)$-$\mathcal{BD}1(P)$ or $(x, 2y)$-$\mathcal{BD}2(P)$.*

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

The two bounded duals are

$$(1, 2)\text{-}\mathcal{BD}1(p): \quad (p_x, p_y) \rightarrow \quad y = -p_x \cdot x + p_y$$
$$(1, 2)\text{-}\mathcal{BD}2(p): \quad (p_x, p_y) \rightarrow \quad y = -p_y \cdot x + p_x$$

Algorithm 1 `ComputeBoundedArrangement(PtSet P)`
Enable **STENCIL TEST**
Set **STENCIL OPERATION** = INCREMENT
for $\mathcal{BD} = \mathcal{BD}1\ \ \mathcal{BD}2$ **do**
   Initialize $SB = 0$
   for $i = 1 \ldots n$ **do**
      Line Segment $l' = (1, 2)\text{-}\mathcal{BD}(p_i)$.
      DRAW HALF-PLANE (ℓ'^{+}).
Save Stencil Buffer

We leave out the complete calculations in this extended abstract, but it can shown 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 above two bounded duals. $\square$

4.2 Computing Depths using the Stencil Buffer

We now describe how to compute the level of each pixel in the dual plane. The basic idea is as follows. For each point $p \in P$, we compute its bounded dual. Then, for each pixel P_{ij}, we increment the corresponding value in the Stencil Buffer, depending on whether it is above or below the bounded dual segment for point p. Algorithm 1 gives the algorithm with the hardware details, which are explained later.

The notion of "above" and "below" is not well defined for a line segment. We interpret the upper halfspace of a line segment ℓ' (denoted by ℓ'^{+}) as the intersection of the upper halfspace of the line ℓ that supports ℓ' with the screen bounding box (Figure 2).
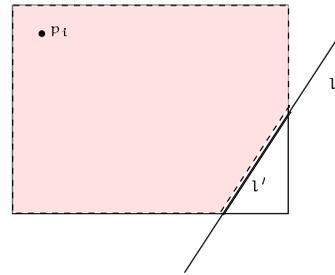

Figure 2: Although l' not visibly below p_i, the bold dashed half-plane representing l'^{+} is sufficient for correctness.

LEMMA 4.2. *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. Two key things need to be explained here - the stencil test and the method of rendering the half-planes.

The stencil buffer allows stencil test to be performed on pixels that are to be rendered by any primitive. Set the stencil test so that if Buffer has the pixel $P_{ij} \in (1,2)$-$\mathcal{BD}(p)^+$, then the stencil test would increment the corresponding pixel in the stencil buffer, i.e. SB_{ij} would become $SB_{ij} + 1$, where $\mathcal{D}(p)$ is rendered line segment. For example, rendering a line segment in the color buffer with stencil test set to INCREMENT increments the stencil buffer values of the pixels contained in the line segment.

An important thing to note is that since we are rendering a bounded dual (a line segment), the positive half-plane that we compute is actually a polygon which shares a boundary with the boundary of the screen (see Figure 2), and therefore if $P_{ij} \in \mathcal{D}(p)^+$, then $P_{ij} \in$ HALF-PLANE$(\mathcal{BD}1(p)^+)$, HALF-PLANE$(\mathcal{BD}2(p)^+)$, which implies that $|T|$ is computed correctly. ⊡

4.3 Computing Depth Contours

As mentioned in Section 2, 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 $\Omega(n^2)$, and hence we will not use this approach here. We present a new grid-based method for computing the contour regions.

LEMMA 4.3. *Let* $q \in [-1, 1] \times [-1, 1]$, q *in primal plane* **P**. *Then* $\exists d$ *in the dual plane* **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 4.1, we know that either $(1, 2)$-$\mathcal{BD}1$ or $(1, 2)$-$\mathcal{BD}2$ 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 4.2. ⊡

We now present Algorithm 2 that reconstructs the depth contours from the depth information computed in the dual.

THEOREM 4.1. *Algorithm 2 computes the contour regions in the primal plane correctly.*

Proof. Take any point q in the primal plane **P**. From Lemma 4.3, we know there exists d in the dual plane **D** such that the depth of point q is equal to $\min\{SB_d, n - SB_d\}$

Algorithm 2 ComputeContourRegions(Bounded Arrangement SB)

Initialize SegmentSet = ∅
for $\mathcal{BD} = \mathcal{BD}1$ $\mathcal{BD}2$ **do**
  Clear Color Buffer
  for $i = 1 \ldots n$ **do**
    DRAW($\mathcal{BD}(p_i)$) with Color 1.
  CB = **Get** Color Buffer
  for $i = 1 \ldots W$ **do**
    for $j = 1 \ldots H$ **do**
      if Color $CB_{ij} = 1$ **then**
        $cDepth = \min\{SB_{ij}, n - SB_{ij}\}$
        SegmentSet=SegmentSet∪$(\mathcal{BP}(P_{ij}), cDepth)$
Enable **Depth Test**
Set Depth Test = **LESS**
for (Segment $l = \mathcal{BP}(P_{ij}), cDepth) \in$SegmentSet **do**
  DRAW(l) with z-value = cDepth

---

and that $d \in \mathcal{BD}(p)$ for some $p \in S$, and either $\mathcal{BD}1$ or $\mathcal{BD}2$ (Lemma 4.1). 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. ⊡

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.

5 Error Analysis and Output Accuracy

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.

As we have seen in Section 4 above, our method for computing 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.

5.1 Phase I: 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 5.1. (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 intersections 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.

5.1.1 When intersection points collapse

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 5.1, 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 5.2. (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

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

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 $W \geq \frac{1}{2\min_{t_{ij}^k \in T} v_{ij}^k}$.

5.2 Phase II: Contour Reconstruction

The analysis of Section 4.3 establishes the correctness of Algorithm 2. However, rasterization and pixelization may still introduce errors into the rendering of the contours.

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 5.3. *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 5.2, we are guaranteed that in the dual buffer constructed by Algorithm 1, there exists a pixel that contains p. By Lemma 5.1, we know that the 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 Analyzing the complexity of the algorithm

Our main reason for using a hardware-based approach is to take advantage of the power and speed of modern graphics engines. As the results in Section 7 will show, we are able to achieve a significant improvement in running time as compared to known software-based approaches. However, in order to extend our techniques to other kinds of geometric problems, it is important to understand exactly what operations on the hardware cost us, and which operations are cheaper than others. Such an understanding allows us to model the hardware, providing an abstract framework to analyze algorithms that we may develop, and also provides intuition as to how our algorithms should be designed to work well in practice.

In Section 3, we presented the basic operations that can be performed in the graphics engine. In this section, we will

present a cost model for these operations (based on empirical data from three different platforms), and will analyze the complexity of our algorithm based on this cost model.

6.1 A Cost Model for the Graphics Engine

To model the cost of operations on the engine, we ran tests on 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). We note that the purpose of comparing three different platforms is not to measure relative performance, but to establish that the relative complexity of operations on a *fixed* platform is roughly the same.

Let the cost of rasterizing a single pixel be one unit. One of the basic operations in our algorithm is the rendering a line of length l. In Figure 3(a) we plot the time (T) taken to draw N lines versus N in a display window of dimensions 512×512, where the lines have length uniformly chosen at random between 0 and 512 (and hence have an average length of 256). As we can see in the figure, the time to draw a single line is quite uniform. The time to draw a line of average length 256 is $4.14\,\mu s$ on the Octane, $1.64\,\mu s$ on the Onyx, and $1.72\,\mu s$ on the Linux PC. This time was computed by averaging over all the data points used in Figure 3(a).

The cost of drawing a line varies with the length of the line. Roughly, it takes l units of time to draw a line of length l. Figure 3(b) shows the results of plotting the time to draw versus line length for the three platforms. For each data point, we drew $100,000$ lines of the same (pixel) length and measured the time taken to draw this set. The variation in times is somewhat irregular, but one can conclude that the cost of drawing a line is proportional the number of pixels in it. Thus, if the length of a line is l, and the size of a pixel is $1/W$, the cost of rendering the line is proportional to lW.

A similar graph can be plotted for triangles (see Figure 4(b)). Once again, we observe that the cost of drawing a triangle is proportional to its area. The graphics engine triangulates general convex polygons before rendering them, so the relation between rendering time and area holds even in this case.

At various points in our algorithm, we perform stencil tests to fill the buffers in a particular way. Since the stencil test is part of the pipeline, the additional cost of performing such a test should be a small constant. In Figures 3(c),4(c), we plot the time taken to draw random lines with and without stencil tests enabled. For the Onyx, enabling a stencil test has an almost negligible effect on the rendering speed. On the Octane, there is a distinct difference, but the extra cost (as measured by the difference in slopes of the two line) is a small constant.

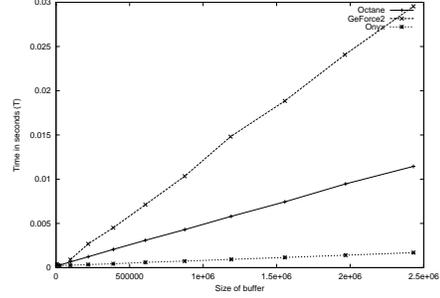Finally, we measure the cost of extracting an "output"

Figure 5: Time to read back a buffer into main memory

(reading a buffer from the chip into main memory). For all three platforms, we measure the time taken to read the color buffer of variable-sized windows into main memory. The results are shown in Figure 5. Again, we see a clear linear behavior; we can model the cost of reading a buffer into main memory as proportional to the size of the buffer. We summarize the costs of various operations - line of length l: $\Theta(l)$; triangle of area Δ: $\Theta(\Delta)$; buffer read (size B): $\Theta(B)$; stencil test: $\Theta(1)$.

6.2 Complexity of Our Algorithm

We are now in a position to analyze the complexity of the algorithm described in Section 4. Let the point set have size n. 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 $\text{Pix}(l)$. Then the running time of Algorithm 2 is $W \bigcup_{l \in \mathcal{D}(p)} \text{Pix}(l)$. Since $\bigcup_{l \in \mathcal{D}(p)} \text{Pix}(l) \leq W^2$, 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 `read-back`. 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 in Section 4, 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.
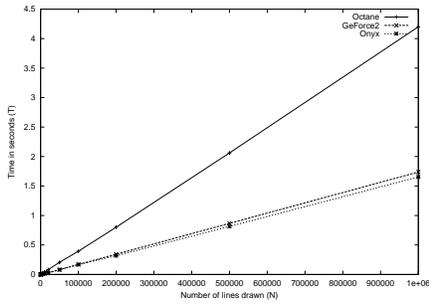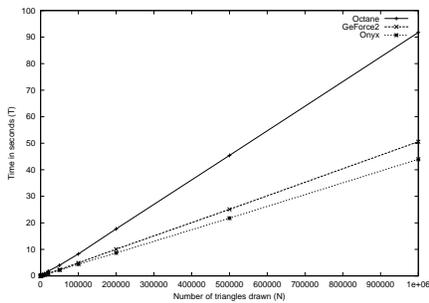
(a) Rendering time vs number of lines drawn
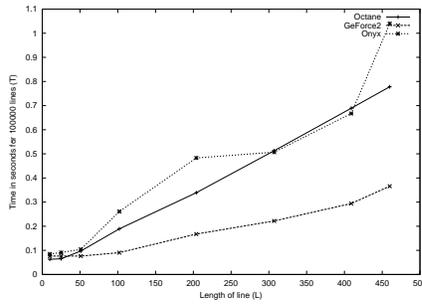
(b) Time taken (in seconds for 100000 lines) vs line length in pixels
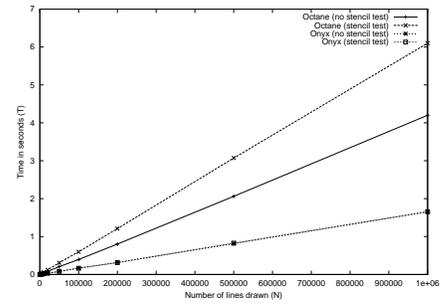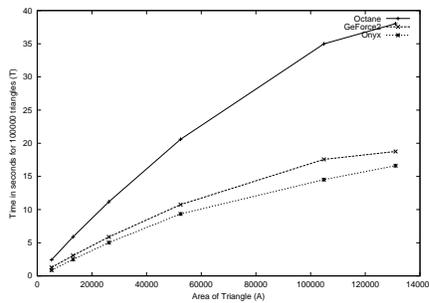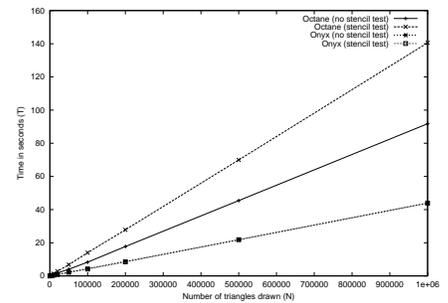
(c) The effect of enabling stencil tests

Figure 3: Line measurements

(a) Rendering time vs number of triangles drawn

(b) Time taken (in seconds for 100000 triangles) vs triangle area in pixels

(c) The effect of enabling stencil tests

Figure 4: Triangle Measurements

7 Experiments

We now present experiments in support of our algorithm. We compare the performance of the algorithm on the three different platforms described in Section 6. 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.

7.1 Running Times

Measuring the performance of the algorithm is non-trivial. The OpenGL pipeline consists of two main parts - Transfor-

mation 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.

In this suite of experiments, we generate random data sets (akin to [17]) of different sizes to test our algorithm. Figure 6 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
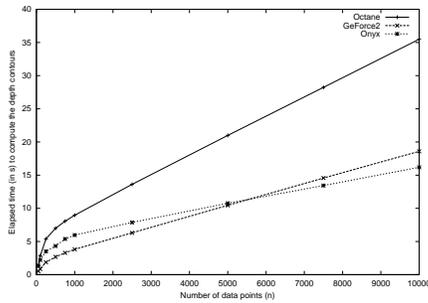
Figure 6: Elapsed running time as a function of n

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* [17], 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.

7.2 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 7. The software interface that we provide allows us to mark rectangular areas of the depth contour map to zoom into, and in Figure 7(c),(d) we show the effect of progressively zooming in closer and closer to the point set.

As noted in Section 5, 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.

8 Conclusions

The improvements achieved by using graphics hardware suggest that the general paradigm of hardware-assisted geometric computation is very useful. We plan to pursue the modeling work started in Section 6 to develop an accurate abstract cost model of the graphics engine. This will hopefully enable us (and others) to achieve a better theoretical understanding of the performance of such algorithms.

The speed of our implementation also makes it possible to think of further applications, the most tantalizing of which is computing the depth contours of *moving points*. In fact, we do not know of any algorithm in the *kinetic framework* [1] for computing contours of moving points; such an algorithm would be interesting for the above reasons.

References

[1] BASCH, J., GUIBAS, L., AND HERSHBERGER, J. Data structures for mobile data. In *Proc. 8th ACM-SIAM Sympos. Discrete Algorithms* (1997), pp. 747–756.

[2] COLE, R., SHARIR, M., AND YAP, C. K. On k-hulls and related problems. *SIAM Journal on Computing 15*, 1 (1987), 61–77.

[3] DUCA, K. Personal communication.

[4] GOLDFEATHER, J., MOLNAR, S., TURK, G., AND FUCHS, H. Near real-time CSG rendering using tree normalization and geometric pruning. *IEEE Computer Graphics and Applications 9*, 3 (1989), 20–28.

[5] GOODRICH, M., GUIBAS, L., HERSHBERGER, J., AND TANENBAUM, P. Snap rounding line segments efficiently in two and three dimensions. In *Proc. 13th ACM Symp. on Computational Geometry* (1997), pp. 284–293.

[6] GREENE, D., AND YAO, F. F. Finite resolution computational geometry. In *Proc. 27th IEEE Symp. Foundations of Computer Science* (1986), pp. 163–172.

[7] GREENE, N., AND KASS, M. Hierarchical Z-buffer visibility. In *Proc. ACM SIGGRAPH* (1993), pp. 231–238.

[8] GUIBAS, L., AND MARIMONT, D. Rounding arrangements dynamically. *Internat. J. Comput. Geom. Appl. 8* (1998), 157–176.

[9] HOBBY, J. Practical segment intersection with finite precision output. Technical Report 93/2-27, Bell Laboratories, 1993.

[10] HODGES, J. A bivariate sign test. *Annals of Mathematical Statistics 26* (1955), 523–527.

[11] HOFF, III, K. E., CULVER, T., KEYSER, J., LIN, M., AND MANOCHA, D. Fast computation of generalized Voronoi diagrams using graphics hardware. In *Proc. ACM SIGGRAPH* (1999), pp. 277–286.

[12] HOFF, III, K. E., CULVER, T., KEYSER, J., LIN, M., AND MANOCHA, D. Interactive motion planning using hardware-accelerated computation of generalized voronoi diagrams. In *Proc. IEEE International Conference on Robotics and Automation* (2000), pp. 2931–2937.

[13] KRISHNAN, S., SILVA, C., AND WEI, B. A hardware-assisted visibility-ordering algorithm with applica tions to volume rendering. In *Data Visualization* (2001), pp. 233–242.

[14] LINDHOLM, E., KILGARD, M. J., AND MORETON, H. A user-programmable vertex engine. In *ACM SIGGRAPH*, p. 149.

[15] MATOUŠEK, J. Computing the center of planar point sets. *DIMACS Series in Discrete Mathematics and Computer Science 6* (1991), 221–230.

	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

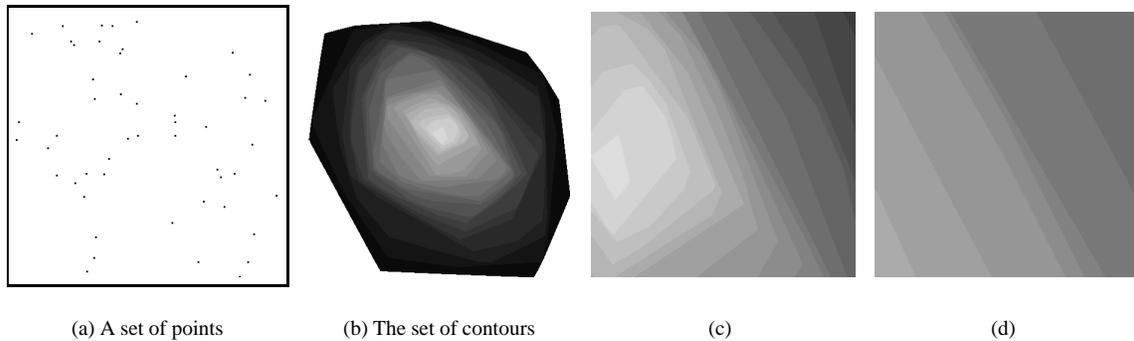Table 2: Running times (in seconds) on the three platforms

(a) A set of points   (b) The set of contours   (c)   (d)

Figure 7:

[16] MILENKOVIC, V. Shortest path geometric rounding. *Algorithmica 27*, 1 (2000), 57–86.

[17] 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.

[18] MUSTAFA, N., KOUTSOFIOS, E., KRISHNAN, S., AND VENKATASUBRAMANIAN, S. Hardware-assisted view-dependent map simplification. In *Proc. 17th ACM Symp. on Computational Geometry* (2001), pp. 50–59.

[19] PEERCY, M. S., OLANO, M., AIREY, J., AND UNGAR, P. J. Interactive multi-pass programmable shading. In *ACM SIGGRAPH*, p. 425.

[20] ROSSIGNAC, J., AND REQUICHA, A. Depth-buffering display techniques for constructive solid geometry. *IEEE Computer Graphics and Applications 6*, 9 (1986), 29–39.

[21] ROUSSEEUW, P. J., AND RUTS, I. Constructing the bivariate tukey median. *Statistica Sinica 8* (1998), 827–839.

[22] ROUSSEEUW, P. J., RUTS, I., AND TUKEY, J. W. The bagplot: A bivariate boxplot. *The American Statistician 53*, 4 (1999), 382–387.

[23] RUTS, I., AND ROUSSEEUW, P. Computing depth contours of bivariate point clouds. *Computational Statistics and Data Analysis 23* (1996), 153–168.