EUROGRAPHICS 2003 / P. Brunet and D. Fellner
(Guest Editors)

Volume 22 (2003), Number 3

Solving Geometric Optimization Problems using Graphics Hardware

Markus Denny

Max-Planck-Institut für Informatik
Saarbrücken
Germany
mdenny@mpi-sb.mpg.de

Abstract

We show how to use graphics hardware for tackling optimization problems arising in the field of computational geometry. We exemplarily discuss three problems, where combinatorial algorithms are inefficient or hard to implement. Given a set S of n point in the plane, the first two problems are to determine the smallest homothetic scaling of a star shaped polygon P enclosing S and to find the largest empty homothetic scaling of P completely contained inside an arbitrary polygonal region. Pixel-exact solutions for both problems are computed in real-time. The third problem is a facility location problem and more difficult to solve. Given the Voronoi diagram $VoD(S)$ of the n points, we try to position another point p in the plane, such that the resulting Voronoi region of p has maximal area. As far as we know there exists no traditional solution for this problem for which we present pixel-exact solutions.

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computer Graphics]: Geometric algorithms, languages, and systems I.3.3 [Computer Graphics]: Display algorithms

1. Introduction

The Voronoi diagram, for short VoD, is a well known and very versatile structure in computational geometry. It is used as the basis for numerous algorithms.

Exploiting a geometric relationship between the VoD and the lower envelope of the arrangement of cones Hoff et al. [17] presented a fast algorithm to render the VoD relying on graphics hardware. This will be shortly reviewed in section two. As proven by Hoff et al. only a few triangles are required to compute a *pixel-exact* distance mesh representing the VoD. In our approach, we also demand a *pixel-exact* assignment of the colors, i.e. Voronoi regions. Then, the number of required triangles increases sizably considering the worst case, which is proven is section two.

As an alternative, we present an algorithm that makes use of depth textures and thus circumvents the above mentioned approximation problem. Additionally, we present a method to speed up the computation such that the running time remains almost constant, independent of the number of processed point sites. In the final part of the second section, we discuss some variations and applications of our approach.

In the third section, we present a pixel-based solution to a facilities location problem, which still seems to remain a too tough nut to crack for pure computational geometry. Given a set S of n points in the plane and a boundary area B, find the position for a new site $s \in B$ such that its Voronoi region has maximal area.

In section four we turn our focus to minmax facilities problems. Given a set S of n points in the plane and a star-shaped polygon P determine the smallest enclosing homothet Q.

The third problem, we pay attention to, is the extremal polygon containment problem. Our goal is to place the biggest homothet Q inside an arbitrarily polygonal region such that no point of S is covered by Q.

A good introduction in geometric computations via graphics hardware can be found in [20].

All computations are executed on an Intel-Pentium^{TM}

800 MHz in combination with a GForce^{TM} 3 graphics adapter.

2. Lower envelope and Voronoi diagrams

2.1. Voronoi diagram

A Voronoi diagram of a set of sites is defined to be a partition of the plane into regions. Each region corresponds to one of the sites and is determined by the property that all points within a region are closer to the corresponding site than to any other site, with respect to some fixed distance function.

Nowadays, VoDs play an important role in many fields besides computational geometry (cf. [12]). A proof of the versatility and importance of Voronoi diagrams may also be found in the large number of surveys and books dealing with Voronoi diagrams (e.g. [2], [3], [22]).

The relationship between the VoD of n sites and the lower envelope of the arrangement of a set of n cones, first observed by Edelsbrunner and Seidel (cf. [10]), emerges as particularly interesting for us (cf. figure 1).

Let $S = \{s_1, \ldots, s_n\} \subset \mathbb{R}^3$ denote the set of point sites in three dimensional Euclidean space. Each of the sites lies in the xy-plane at $z = 0$. Let C denote a right circular cone completely contained in the half-space $z \geq 0$ with the apex of C positioned at $(0,0,0)$, i.e. $C = \{(x,y,z) \in \mathbb{R}^3 | z = \sqrt{x^2+y^2}\}$. If we position such a cone on top of every point site, then the orthogonal projection of the lower envelope of the arrangement of the cones is exactly the partitioning of the plane into Voronoi regions.

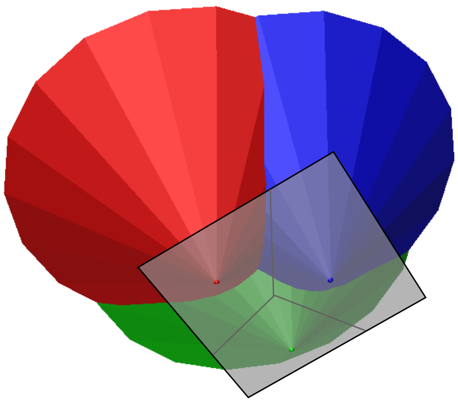

Figure 1:

Construction of a Voronoi diagram employing cones

This relationship is exploited by Hoff et al. [17]. They approximate each cone as a triangle fan. To extract the lower envelope of the arrangement of the cones, the depth buffer is enabled such that only the lowest fragments for each pixel are allowed to pass.

2.2. The error due to the approximation

In order to get a pixel-exact approximation of the VoD Hoff et al. [17] approximate the base of the right circular cone with a regular polygon such that the polygonal chain is inside an annulus of the maximal radius d (the diameter of the image) and $d - \varepsilon$ with $\varepsilon = 1$. Assuming a image size of 1024×1024 pixels, Hoff et al. [17] conclude that $T = 85$ triangles are sufficient for a *pixel-exact* computation of the Voronoi diagram. T is determined by

$$T \geq \left\lceil \frac{\pi}{\arccos \frac{d-\varepsilon}{d}} \right\rceil . \tag{1}$$

This error approximation holds for the *pixel-exact* computation of the depth buffer values. Thus, it suffices the requirements of any further application relying on the pixel-exact distance computation. It does not guarantee a correct assignment of the color of the Voronoi regions. It might happen that many pixels are assigned the wrong color although for any of these pixels hold that the distance between it and its corresponding nearest site is computed fairly accurate.

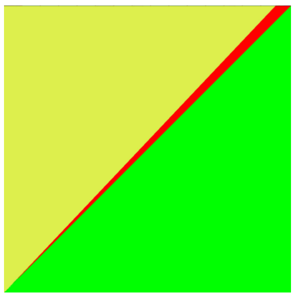

Figure 2:

Construction of a Voronoi diagram employing cones

The worst case is represented by two sites at minimal possible distance located in one of the corners. In our example the two points are positioned at $(1,0)$ and $(0,1)$, i.e. lower left corner. The resulting maximum error occurs at the upper right corner (cf. figure 2). For an image size of 1024×1024 pixels the deviation of the computed bisector from the correct one in the upper right corner is more than 50 pixels.

The number of required triangles per cone can be bounded from above as follows. As mentioned before, the worst case is determined by two sites s and t at nearest possible distance a. The maximum deviation appears at the point of the bisector which is further away from both, thus at the upper right corner (cf. figure 3). To derive an upper bound, we assume that the approximation of the cone for the site s is always correct and the approximation of the cone for t is always as bad as possible. At the maximum height r the base of the cone C_s for s is a circle with radius r and the base of the cone C_t of t is a circle with radius $r - \varepsilon$. Hence, the real approximation is in between the annulus of r and $r - \varepsilon$.

Demanding an orthogonal deviation from the correct bisector of no more than 1 pixel, we get an upper bound on the number of triangles per cone of more than 2500. To ease the calculation, we assume s is at the origin of the xy plane at height $z = 0$ and t is positioned at $(a,0,0)$. Then the cone C_s and C_t are given as $C_s = \{(x,y,z) \in \mathbb{R}^3 | z = \sqrt{x^2+y^2}\}$ and $C_t = \{(x,y,z) \in \mathbb{R}^3 | z = \rho\sqrt{(x-a)^2+y^2}, \rho > 1\}$, where $\rho = d/(d-\varepsilon)$ represents the approximation error of C_t. Again, d denotes the maximal radius (i.e. the diameter of the image). To guarantee a pixel exact approximation, we demand that C_s and C_t intersect at position $(a/2-1, d, z)$, thus

$$\rho\sqrt{((\frac{a}{2}-1)-a)^2+d^2} = \sqrt{(\frac{a}{2}-1)^2+d^2}.$$

Resolving this equality and applying equation (1), we get an upper bound on the number of required triangles.

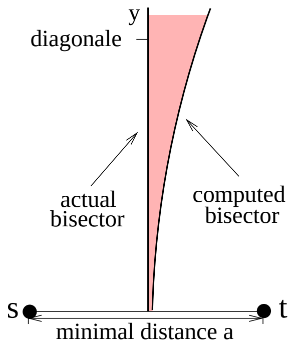

Figure 3:

Worst case approximation error

2.3. Alternative algorithms using depth textures or pixel shaders

As shown above, for a pixel exact computation we have to spend about 30 times more triangles than expected. Apparently, this will have a remarkable impact on the rendering time. Fortunately, this can be circumvented using depth textures.

In a first step, we precompute a *distance* texture D which contains for each entry d_{xy} the corresponding distance to the origin, i.e. $d_{xy} = \sqrt{x^2+y^2}$. For each point site, we render a image wide rectangle textured with D. This guarantees the most exact results maintaining a good rendering time. As the construction of the texture is done only once, it can either be computed by rendering a triangle fan consisting of a rather huge number of triangles, or it can even be precomputed using the CPU's power. This has the additional advantage, that arbitrary distance functions can be realized not relying on a conic representation. For instance, the power diagram (cf. Laguerre-metric [22] can also be realized using this approach.

If the underlying distance function for the computation of the VoD is induced by a Minkowski norm $L_\ell(x,y) = (|x|^\ell + |y|^\ell)^{1/\ell}$ $(\ell \geq 1)$, as for instance the Euclidean norm L_2, then we can make use of its symmetry and reduce the size of the precomputed texture to a quarter.

2.4. Speed up

On closer examination one finds out that in most cases there are ineffective fragments generated. Let p denote the number of pixels of the image under consideration. The rendering of each cone (or depth textured rectangle) generates a fragment for each pixel, thus $p \cdot n$ fragments are created although p fragments are sufficient. Our idea to accelerate the computation, reduces the overhead of needlessly generated fragments. To ease the discussion, we define the width of a Voronoi region with respect to a bounded area to be the maximal distance from its site to any point belonging to that region. Furthermore, the *width of a Voronoi diagram* is the maximum of all widths over all regions.

If we knew the width w of a Voronoi diagram in advance, then we could restrict the height of the cones appropriately. As a consequence the number of required triangles and even more important the number of generated fragments entering the raster pipeline decreases.

Dividing the image using a quad-tree structure gives us a fast approximation for the width of the Voronoi diagram. It divides the image into four equal sized squares. On each level of the quad-tree, each of the four squares is again divided into four squares, resulting in a tree structure with a branching factor of four.

For each level of the initially empty tree and for each site, we mark the node representing the area in which the site resides as being visited.

Our intention is to derive an upper and a lower bound for the width of the Voronoi diagram. In order to derive such bounds, we are interested in the highest level t, for which at least one node is not marked. For the sake of simplicity, we denote the corresponding edge length by e.

Consider a piece of area a node in level t is accountable for. The corresponding edge length e gives a lower bound for the width of the Voronoi diagram. The level one above level t is the lowest level for which all nodes are marked. That is each node *knows* a site which is inside the area corresponding to that node. Hence if we choose the width to be the length of the diagonal (i.e. $e2\sqrt{2}$), we can ensure that the entire image will be covered. That is our upper bound. In case that all leaf nodes are marked, the upper bound reduces to $e\sqrt{2}$ (i.e. the length of the diagonal of the area corresponding to a leaf node).

The quad-tree is filled as follows. For each site we traverse the tree bottom-up and mark every node on the path to the root of the tree until we reach the first node already marked.

For each layer, the nodes are stored in an array such that the address of the node can easily be computed applying a bitwise shift operation on the x and y coordinate of the site under consideration.

In the worst case any node of the tree as well as any site is visited at most once. For a image size of 1024×1024 we use a quad-tree of height six. On an Intel-Pentium$^{\text{TM}}$ 800 MHz the processing of 10000 sites requires less than 4 milliseconds which is definitely worth the trouble.

To prove the effect of the speedup, we summarize some test series in figure 4. For all series, the sites are chosen uniformly at random from $\{0,\ldots,1023\}^2$.

The red curve represents the time consumed by the algorithm due to Hoff et al. [17]. Since the height of the cones remains unadapted, the number of generated fragments, and thus the running time of the algorithm, increases linearly with the number of sites.

In contrast to that, the running time remains quasi unaffected of the number of sites, if the height of the cones is adjusted.

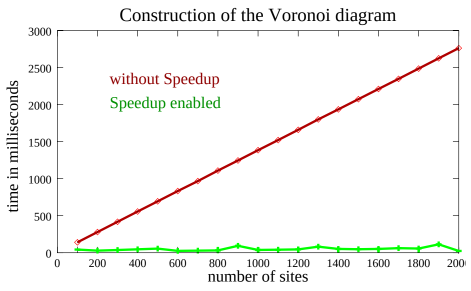

Figure 4:

Rendering time with and without the quad-tree

2.5. Variations

Although we cannot go into detail in this paper, it is worth mentioning that beside VoDs based on Minkowski norms, also VoDs based on an arbitrary distance function can be rendered with the help of depth textures. We can also make use of our approach to compute higher order VoDs using shadow buffering to simulate a second depth buffer. The special case of the furthest point VoD can be rendered by just changing the point of view for the rendered scene. Furthermore, there are straight forward implementations allowing to change the shape of the sites to line or circle segments. Additionally, weighted VoDs can be rendered using just a single depth texture if we have access to the pixel shading functionality. Deploying geometrical properties, these implementations are possible without any impact on the speed and accuracy.

A more detail description of these variation can be found in [8].

3. Settle point computation

3.1. Yet another fast food in town

Assume you are the store manager of a fast food chain. You decide to capture the market in a town you are not yet present. Most naturally, the trading area should be as big as possible. To find the optimal place, you construct a Voronoi diagram of the presently existing stores of your business rivals. The new store is best located at the point, that will have the biggest region, when inserted into the existing diagram as a new site. More precisely, the problem is stated as follows:

Given a set S of n sites within a bounding region B. Find the position for a new site s such that in the Voronoi diagram for the set $S \cup \{s\}$ the Voronoi region R_s of s maximizes the area over all sites $s \in B \setminus S$.

To ease the further description we call this point the *settle* point of a Voronoi diagram.

Previous work

Cheong et al. [6] describe a related problem. Consider the following game, in which a first player chooses an n-point set A inside a square Q. Thereafter, a second player places another n-point set B inside Q. The payoff for the second player is the fraction of the area of Q occupied by the regions of B in the Voronoi diagram of $A \cup B$. Cheong et al [6] give a strategy for the second player that always guarantees him a payoff of at least $\frac{1}{2} + \alpha$ for a constant $\alpha > 0$. Although this work resembles our problem, the point bearing the largest area is not determined. For a very special case, Dehne et al [7] describe a method to maximize a Voronoi region. But up to now, we are not aware of any previous solution delivering the settle point.

Properties

The problem appears to be rather challenging as the position of the settle point can be quite arbitrary. In general, it coincides neither with an edge of the Voronoi diagram nor with an edge of the furthest point Voronoi diagram.

Figure 5 illustrates such a case, based on a Voronoi diagram of set of eight points. To ease perception, the eight sites are colored in green. Further, the slim lines represent the edges of the Voronoi diagram, and the fat lines represent those of the furthest Voronoi diagram. The settle point is marked by a red circle. For any other point, it holds that its brightness represents its potential area.

Pixel based adaptation

The easiest method to accomplish our task is to insert all possible points, one at a time, in the existing Voronoi diagram, compute the size of the area of the just constructed region, and delete the point again. Obviously, this is by far too inefficient and time consuming. After a point is inserted, the

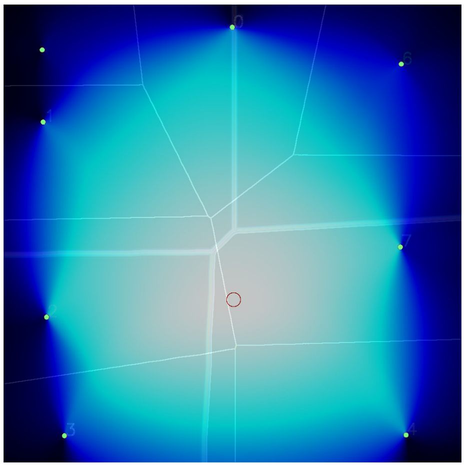

Figure 5:

Voronoi, and furthest point Voronoi diagram with superposed settle point computation

entire image must be read back into main memory to count the number of newly colored pixels.

Instead, we do the following. Let p be the pixel under consideration. We investigate for which potential new sites the pixel p would contribute to their regions (cf. figure 6). For that reason, we draw a circle around p. The radius of the circle equals the distance between p and its nearest site s_p. If a pixel q_{in} inside the circle is chosen to be the new site, then p will belong to its region, since the distance between p and q_{in} is less than the distance between p and s_p. Similarly, p will never belong to the region of a pixel q_{out}, outside the circle.

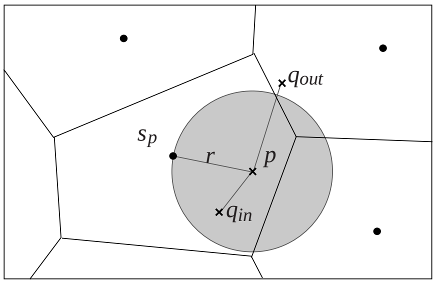

Figure 6:

Contribution area of p

Given a set of point sites $S = \{s_1, \dots, s_n\}$, we render the VoD(S) as a first step. After examining the depth buffer each pixel *knows* about the distance to its nearest site. Based on this knowledge, we do the following for each pixel p: We draw a circle around p of radius r, where r equals the distance between p and its nearest site. p contributes 1 to the potential area of any pixel inside the circle (e.g. q_{in}). We count these events abusing the color and stencil buffers. (The stencil buffer is part of the pixels individual memory used for per-fragment testing. It is commonly applied to realize masking regions.) Whenever a circle overlaps a pixel its stencil buffer is incremented by 1. The fragment is allowed to pass the stencil test, not until an overflow in the stencil buffer occurs. This has the effect, that one of the four color buffers value increases by 1, provided that blending is appropriately enabled. This is why we initialize the color buffer with zero and the stencil buffer with 1. We select the blending factors to be all one. Furthermore, we use the *test of equality with zero* as stencil compare function and reference value. All stencil actions are set to *increment with wrap around* the stencil buffer by 1. In addition to that, the circles are at first colored with red equals 1 (i.e. 0x01) and green, blue and alpha equal zero.

As mentioned before, whenever a circle overlaps a pixel, its stencil buffer increments. Every 256 times the stencil buffer wraps around, and due to the blending settings, the red color buffer increments by 1 (only the first wrap around happens after 255 hits since the stencil buffer is initialized to 1). Accordingly, we can count $256 \times 256 - 1$ hits in the red buffer. After that, we rotate the color values of the circles to sum up the hits successively in the remaining three color buffers. In doing so, we can count $4 \times 256 \times 256 - 1 - 3 \times 256 = 261375$ events before we have to read back the image and reset the color and stencil buffers to continue counting. Finally, we add up all intermediate images together with the last image in the main memory to get the final image. Any pixel with the greatest value is the requested candidate. With new pixel shading hardware, there are 32 bit per channel color buffers, such that $4 * 2^{32} * 256$ events can be counted without a readback.

In case of the availability of pixel shading functionality, we ease the process of counting. We use a depth texture assigning each pixel a 4 byte memory cell, which can be used for counting.

A variation for successful stores

Let us recall the example of the store manager from the beginning of the section. Assume you are already involved in the city, thus some of the stores are already your stores. Then we can easily restrict the search for the new settle point excluding your current trading area. The pixels belonging to your area are just unregarded, i.e. no new site can profit from them.

3.2. Speed up

The idea to speed up the algorithm is based on the observation that the computation made for some sets of pixels repeats exactly for other sets. To gain a better insight, we trace the computation for a horizontal row of pixels $p_1, \dots, p_\delta$ all belonging to the site s_p. Pixel p_i is at distance i away from s_p.

A circle with radius i is drawn around each p_i with the effect that the *area counters* of pixels inside the circle are changed. Assume we construct a stamp T combining all the affected pixels inside any of the drawn circle. T is a rectangle sufficiently large to enclose any of the circles.

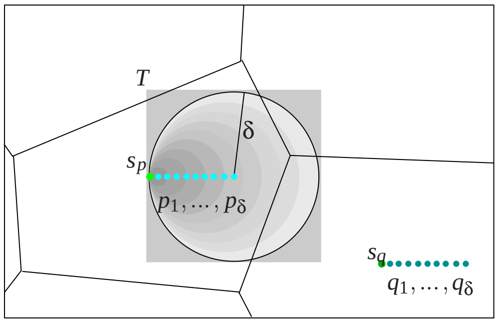

Figure 7:

Contribution area of p

This stamp can now be used in the course of the computation. For any row of pixels $q_1,\dots,q_\delta$ belonging to a site s_q, we omit the rendering of the single circles. Instead, we use T to stamp the area around the pixels appropriately, all at once.

This could be exploited in the following way. We precompute the circles for various configurations of pixels. Instead of restricting ourselves just to pixel rows, we make sample triangular groups (located around a virtual site) and compute the effect of these pixel triangles on the surrounding area.

Armed with a large set of precomputed patches, we can now reduce the number of pixels we have to process in the ordinary way. After the Voronoi diagram of the initial n sites is computed, we do the following for each site.

Find the biggest pixel triangle out of our precomputed samples, which fits inside the sites region, such that the virtual site's position maps onto the real sites position. Thereafter, any pixel inside the triangle is marked as processed (see figure 8). For the rest of the pixels we proceed as before. We draw circles of appropriate radius around these. At the end of the ordinary algorithm we just add the corresponding patches to the image in order to get the complete result.

In our implementation, the number of pixels which remain to process is reduced by 60 percent on average.

Exploiting this idea yields another additionally accelerating effect, we can benefit from. Since fewer pixels have to be processed, buffer overflows occur less frequently. Thus we can reduce the number of times, the buffers have to be read back in order to prevent a buffer overflow.

Running time anomaly

Interestingly, there is an apparent anomaly concerning the running time. The greater the input set of sites, the quicker

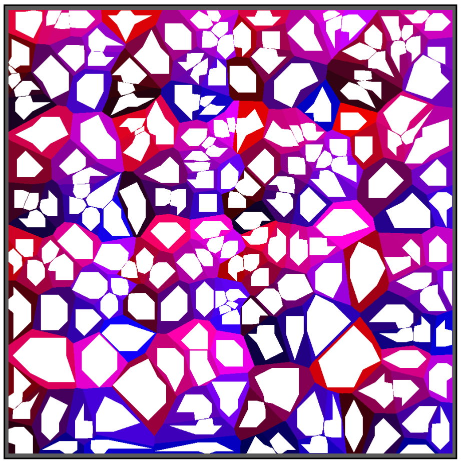

Figure 8:

Voronoi diagram with blank patches of precomputed areas

the settle point can be computed. This is because the area and width of the Voronoi regions decrease as more sites share the same bounded area. Circles from points far away from their corresponding site cover a lot more fragments than the points nearer to the site, thus fewer fragments need to be updated with a larger input set. The rendering time varies between half a minute for set of about 1000 sites and up to five minutes for set of three sites.

4. Minmax facilities problem

4.1. Smallest Enclosing Homothet

Imagine the alliance of several communities sharing the wish for a common radio station. To keep down costs, they agree in building just one radio transmitter station capable to cover all communities. If we think of the communities as points in the plane, the goal is to determine the point that minimizes the distance to its farthest community. As we will see, the center of the smallest enclosing circle is the desired place. Furthermore, the radius of the circle is a lower bound on the transmitting power which has to be deployed to cover all the communities.

The minimal enclosing circle, for short MEC- problem as introduced above is a special case of the smallest enclosing homothet problem (see figure 9).

Given a set S of n points in the plane and a simple star-shaped polygon P with $(0,0) \in P$. Determine the smallest homothet H of P such that all points of S are contained inside H.

This problem can also be stated as a minmax facility location problem. Given the set S and the star-shaped polygon $P \ni (0,0)$, we search for a translation vector $t \in \mathbb{R}^2$ mini-

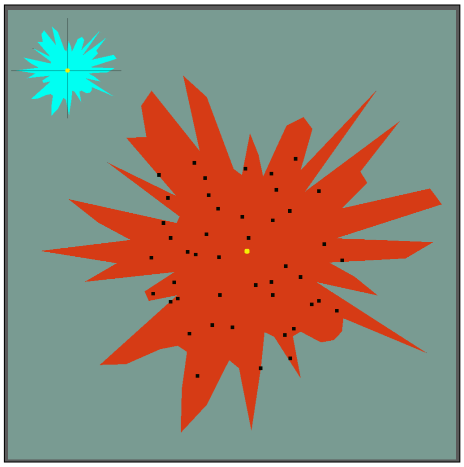

Figure 9:

Polygon P (in light blue) and the smallest enclosing homothetic scaling

mizing the maximal scaling factor $\lambda \in \mathbb{R}^+$ such that each site $s_i \in (\mathfrak{t} + \lambda P)$.

A straight forward application of our more general approach addresses quality assurance systems. Think of the predetermined polygon as a reference value for a machine-made component. Then the scaling factor computed for a set of sample data points scanned from an actual component gives us the deviation factor as a measure for the quality of the production process.

4.1.1. Previous work

For that special case of the afore mentioned MEC–problem (the polygon H takes the shape of a circle) the prune-and-search techniques for linear programming developed by Nimrod Megiddo in 1983 can be adapted to solve this problem in linear time [21]. In 1991, a further enhancement for the computation of the minimum enclosing circle was presented by Welzl ([27]). He came up with a fast randomized method, which could also be used to compute smallest enclosing ellipsoids. Later on, this method was further improved by Gärtner and Schönherr (cf. [16]).

A related problem is considered by Schwarz et al.[24], who presented a linear time algorithm for finding a minimal area parallelogram enclosing a convex polygon in 1994. About twenty years earlier, Freeman and Shapira faced the problem of computing the minimum area rectangle [15]. In 1985 Aggarwal, Chang and Yap [1] provided an $O(n \log n \log k)$ algorithm computing the minimum area k-gon circumscribing a convex n-gon. They exploited a lemma stated by DePano in [9], who on his part proposed solutions for the minimal enclosing equiangular polygon or regular k–gon.

Although these problems are related to the one stated here, there are two major differences. First, we compute the minimal enclosing homothetic scaling, thus rotation of the polygon is not allowed. Second, our constraint on the shape of the polygon is less demanding. It does not have to be convex but just star-shaped.

4.1.2. Pixel-based approach

In a primary step, we develop an algorithm to solve the MEC–problem, which is then extended to solve the more general problem.

Given a set S of n point sites in the plane, it is a well known fact, that the requested circumscribing circle C is determined either by the diameter of the set, thus by two sites, or by three of the point sites. In the first case, the center c of C lies on an edge of fVoD(S), the furthest point Voronoi diagram of S, and in the second case, it lies at a vertex of the fVoD(S).

In our pixel-based approach to compute the fVoD(S), we render circle–based cones from above, i.e. compute the upper envelope of these cones. In doing so, we allow only the highest fragments to pass to the frame buffer and make an update of the depth buffer. Eventually, the depth buffer resembles the fVoD(S).

Fixing an arbitrary pixel, let us ask for the smallest circumscribing circle located at the pixel's position. Then the radius of the circle corresponds to its depth buffer value in the fVoD(S). Thus the pixel with the lowest depth buffer value gives us the center c of the requested circle, and the radius equals that of the depth buffer value. Moreover, let p_c be the pixel with the lowest depth buffer value h. Then rendering a cone upside down at position p_c and height h will draw the desired circle (provided a cutting plane at $z = 0$).

Stepping towards star–shaped polygons

Let P be the given star–shaped polygon, and m_P be a point of the kernel of P, i.e. an interior point such that all the boundary points of P are visible from m_P.

In principle, the procedure remains the same. We compute the upper envelope of an arrangement of cones. Each cone is a translation of a cone the base of which corresponds to the polygon P. However, we have to take care about the shape of the cone we put at every site in order to compute the upper envelope. A problem arises if the given polygon P is not centralized symmetrically about the kernel point.

To gain some insight into the problem, examine the following example. Let P be a star–shaped polygon such that the kernel point is at $(0,0)$. P is assumed to be not centralized symmetric. Let s be point site and P positioned at s. If we consider the set of all translations of P, that contain s on their boundary, then the corresponding set of kernel points form the boundary of the $\tilde{P} = -P$. This corresponds to the concept of the Minkowski difference between s and P.

In the two dimensional case, $\tilde{P}$ can also be gained as rotation of P about π centered around its kernel point. The finial procedure to compute the smallest enclosing homothet with regard to a star–shaped polygon P starts with the computation of the upper envelope of the arrangement of (identical) cones, where the base of the cones is the polygon P rotated about π around its kernel point. Any further step remains the same as before, in the computation of the MEC.

Computing the smallest *k*–enclosing homothet

This problem is first stated by Efrat et al. [11]. Given a set of n points in the plane, the smallest k–enclosing circle problem is defined to be the smallest circle enclosing at least a set of k points.

Adopting the algorithms for higher order Voronoi diagrams, mentioned in section 2 in the most natural way, enables us to use this approach to compute the smallest k–enclosing circle (homothet), too. The key idea is to use two depth buffers. Although there is no second depth buffer, we can emulate it in hardware deploying shadow buffering (cf. [5]). With the aid of the second depth buffer, we peel the order k Voronoi diagram (i.e. the k-th lower envelope) from the $k-1$ diagram, one after another. After we compute the appropriate depth buffer values, we continue with our algorithms as before.

4.1.3. Analysis

For the upper hull computation based on n points and a polygon P with v vertices we need n triangular fans consisting each of v triangles. In case of P being a circle, this is the computation of the furthest point Voronoi diagram, which is also the worst case for the number of triangles send to the graphics engine.

The next step is the crucial part in the time consumption analysis. To find the center of the smallest enclosing homothet we have to read back the depth buffer values of the entire frame buffer. Based on the hardware at our disposal, it takes about 30 milliseconds to read back 4 bytes for each of the 1000×1000 pixels.

Compared to the computation of Voronoi diagrams, we do not use any color information but only the depth buffer. Hence, except for the errors made by the graphic engine, our computation is exact, and can be rendered in real-time. Even the computation of the minimum enclosing circle is only affected by vertical approximation errors such that 85 triangles per cone are sufficient.

4.2. Extremal polygon containment

Let us recall our example of the alliance of communities. But now, instead of establishing a transmitting station for their citizens, they are faced with the problem of finding the *best place* for a waste disposal site. Of course, nobody wants to live near a bad smelling waste disposal. Accordingly, the best place is furthest away from its nearest city, i.e. the distance between the waste disposal and the next city should be as large as possible. Additionally, the choice for a place is limited by the area the cities have at their disposal. This leads to the definition of the *extremal polygon containment* problem.

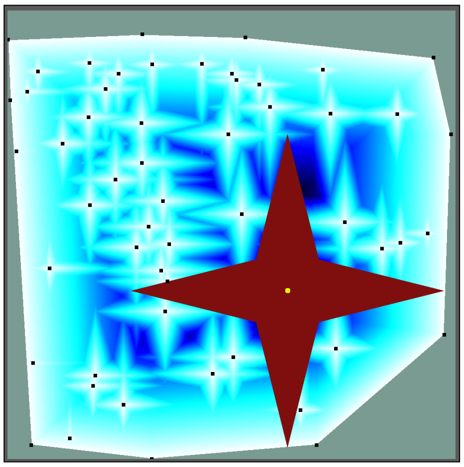

Figure 10:

Biggest empty star–shaped polygon (in red with yellow kernel point) constrained to be completely inside the convex hull. The blue area represents the lower envelope computation.

Let S be a set of n points in the plane, A be a subset of the plane, and P be a simple star–shaped polygon. Determine the biggest homothetic scaling H of P such that no point of S is contained inside H and H is contained in A. Thus, we search for the pair $(\mathfrak{t}, \lambda)$ with $\mathfrak{t} \in \mathbb{R}^2$ and $\lambda \in \mathbb{R}^+$ with

$$\max\{\lambda | (\mathfrak{t} + \lambda P) \cap S = \emptyset \text{ and } (\mathfrak{t} + \lambda P) \subset A\}.$$

A variation of the problem asks for the largest empty circle the center of which is inside a predefined bounding box, commonly the convex hull of the points in S (see [23]pp. 256 et sqq.).

4.2.1. Previous work

Restricted to the largest empty circle, Shamos and Hoey present an $O(n \log n)$ time algorithm based on Voronoi diagrams in [25]. Some years earlier, in 1986, Lee and Wu [18] had just settled a lower bound of $\Omega(n \log n)$ for the algebraic decision tree model proving optimality of the preceding algorithm. Sharir and Toledo [26] developed an algorithm for placing the largest copy of a convex polygon P with k–vertices inside a bounded two dimensional environment consisting of a collection of polygonal obstacles having altogether n corners. The copy is not allowed to intersect any of the obstacles and may have arisen from P by translation, rotation and scaling. The execution time is bounded by $O(k^2 n\lambda_4(kn)\log^3(kn)\log\log(kn))$ ($\lambda_q(r)$ is the maximum length of an (r,q) Davenport Schinzel sequence, i.e. almost linear in r for fixed q). Fortune [14] and Leven and Sharir [19] attended the problem to find the largest homothetic copy of a polygon P inside an arbitrary polygonal environment. One result is an $O(kn\log(kn))$ time algorithm provided P is a convex polygon with k vertices and the environment consists of at most n vertices. The more general question – nesting two non–convex, possibly non–connected polygons – is answered by Avnaim and Boissonnat [4] with an $O(k^3 n^3 \log(kn))$ time algorithm.

4.2.2. Pixel–based realization

Let $S = \{s_1, \ldots, s_n\}$ be a set of n points in the plane, and P a star–shaped polygon. Our aim is to compute the largest empty homothetic scaling H of P. In a basic approach we do not impose any constraint on the position of the kernel point m_H of H but to be inside the rectangular area of the screen.

The way we determine the kernel point of H is quite similar to the one for determining the smallest enclosing homothet. In contrast to there, we now use the lower envelope of the arrangement of cones to discover m_H. As before, all cones are translations of a cone the base of which has the shape of P rotated about π around m_P. The kernel point m_H of the requested homothet is determined by the pixel with the highest depth buffer value, which is also a measure for the scaling factor of H.

4.2.3. Restricting the position of the homothet

There are a several ways to extent the basic algorithm. A frequent one is to demand the kernel point to be contained inside a predefined polygon. In the pixel based world, it is rather easy to comply with this requirement: Before we determine the highest depth buffer value, we just reset the depth buffer values of all *forbidden* pixel. This can be accomplished by drawing a polygon corresponding to the desired region without altering the depth buffer values but the stencil buffer values. After that we reallow altering of the depth buffer, and limit a frame buffer update to these pixels the stencil buffer value of which is 0. Eventually we render a screen size wide rectangle at height $z = 0$.

In case that the idea behind this constraint is to avoid irritations with your neighbors, we can do better. Deploying the idea of Voronoi diagrams for line segments, we can force H to lie completely inside the predefined polygonal region (cf. figure 10).

Let C be the cone under consideration, thus the base of C corresponds to the polygon P. In addition to the above described rectangle, we position for each line segment a copy of C at both endpoints e_s and e_t. Furthermore, we render a rectangle for each vertex v of P. Then the corresponding

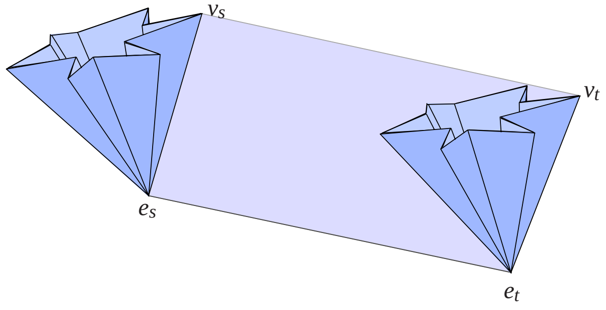

Figure 11:

Construction to force the homothet to fit entirely.

rectangle is spanned between the apices of the cones at e_s and e_t and the vertices v_s, v_t of both copies of C corresponding to v. The swept volume is given by the Minkowski sum of the two cones and the line segment (see figure 11).

It is easy to see, that after this rendering step, the depth buffer is adequately altered to yield the desired result. From there on, we execute the basic algorithm.

As a consequence of the construction, the *forbidden* area can be an arbitrarily shaped polygonal region, i.e. with holes in it, with only a marginal impact on the complexity of the algorithm.

4.2.4. Weighted Facilities

Compared to the standard problem, the *weighted maximum facility location* problem asks for the best place under the condition that each point of the input set has an associated weight.

Weighted maximum facility location
Let $S = \{s_1, \ldots, s_n\}$ be a set of n points in the plane. Let $\{w_1, \ldots, w_n\}$ be the set of associated weights, with $w_i > 0$, A be a subset of the plane, and $d : \mathbb{R}^2 \times \mathbb{R}^2 \mapsto \mathbb{R}$ a distance function. Find the point $c \in A$ such that

$$c = \operatorname{argmax}_{p \in A} \min_i \, w_i\, d(p, s_i).$$

Follert et al. [13] give a sub-quadratic $O(n\log^4 n)$ time algorithm, exploiting parametric search, which is a bit surprising as the computation of weighted Voronoi diagrams of n points is known to have quadratic complexity.

To solve this problem in the pixel based world, we maintain our basic algorithm but scale each cone appropriately in the rendering step of the arrangement of cones – the same method already applied, to compute the weighted Voronoi diagrams in the second section. If the pixel shading functionality is available, we can make use of it as follows. We just build only one depth texture and scale the values appropriately, each pixel at its own.

5. Conclusions

We presented a framework to solve problems arising in computational geometry by using graphics hardware. As an example for the versatility of our approach, we give a fast algorithm to compute the Voronoi diagram of point sites even for non euclidean distance functions. Our algorithm can also be applied, if the sites are line or circle segments.

Furthermore, our approach can be used to solve a subset of geometric Optimization problems, e.g. facility location problems, for which previously there was no solution by combinatorial algorithms.

Acknowledgements

Our special thanks go to Elmar Schömer for his great encouragement as well as Frank Lehmann for his help in this work. Furthermore, we thank the computer graphics group of Philipp Slusallek, especially Georg Demme.

References

1. Alok Aggarwal, J. S. Chang, and Chee K. Yap. Minimum area circumscribing polygons. *The Visual Computer*, 1(2):112–117, October 1985. 7
2. F. Aurenhammer. Voronoi diagrams – a survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3):345–405, 1991. Habilitationsschrift. [Report B 90-09, FU Berlin, Germany, 1990]. 2
3. F. Aurenhammer and R. Klein. Voronoi diagrams. In J. Sack and G. Urrutia, editors, *Handbook of Computational Geometry*, pages 201–290. Elsevier Science Publishing, 2000. 2
4. F. Avnaim and J.-D. Boissonnat. Polygon placement under translation and rotation. In *STACS '88*, volume 294 of *LNCS*, pages 322–333. Springer, February 1988. 9
5. Cass Everitt cass@nvidia.com. Interactive order–independent transparency. Technical Report OpenGL Applications Engineering, NVIDIA, 2002. 8
6. O. Cheong, S. Har-Peled, N. Linial, and J. Matousek. The one-round voronoi game. In *Proc. 18th Annu. ACM Sympos. Comput. Geom.*, pages 97–101, 2002. 4
7. Frank Dehne, Rolf Klein, and Raimund Seidel. Maximizing a voronoi region: The convex case. In *ISAAC*, volume 13, November 2002. 4
8. Markus O. Denny. *Algorithmic Geometry via Graphics Hardware*. PhD thesis, Universität des Saarlandes, to appear 2003. 4
9. DePano and Aggarwal. Finding restricted k-envelopes for convex polygons. In *22th Annual Allerton Conference on Communication, Control, and Computing*, pages 81–90, 1984. 7
10. Herbert Edelsbrunner and Raimund Seidel. Voronoi diagrams and arrangements. In Joseph O'Rourke, editor, *Proc. 1st ACM Symp. Computational Geometry*, pages 251–262, 5–7 June 1985. 2
11. Alon Efrat, Micha Sharir, and Alon Ziv. Computing the smallest k-enclosing circle and related problems. In *Workshop on Algorithms and Data Structures*, pages 325–336, 1993. 8
12. David Eppstein. Geometry in action. *http://www.ics.uci.edu/ eppstein/geom.html*, 2002. 2
13. Frank Follert, Elmar Schömer, and Jürgen Sellen. Subquadratic algorithms for the weighted maximin facility location problem. In *Proc. 7th CCCG*, pages 1–6, 1995. 9
14. S. J. Fortune. A fast algorithm for polygon containment by translation (extended abstract). In *Automata, Languages and Programming, 12th Colloquium*, volume 194 of *LNCS*, pages 189–198. Springer-Verlag, 15–19 July 1985. 9
15. Herbert Freeman and Ruth Shapira. Determining the minimum-area encasing rectangle for an arbitrary closed curve. *Communications of the ACM*, 18(7):409–413, July 1975. 7
16. Bernd Gärtner and Sven Schönherr. Exact primitives for smallest enclosing ellipses. *Information Processing Letters*, 68(1):33–38, 1998. 7
17. Kenneth E. Hoff III, John Keyser, Ming Lin, Dinesh Manocha, and Tim Culver. Fast computation of generalized Voronoi diagrams using graphics hardware. *Computer Graphics*, 33(Annual Conference Series):277–286, 1999. 1, 2, 4
18. D. T. Lee and Y. F. Wu. Geometric complexity of some location problems. *Algorithmica*, 1:193–211, 1986. 8
19. D. Leven and M. Sharir. Planning a purely translational motion for a convex object in a two dimensional space using generalized voronoi diagrams. *Discrete and Computational Geometry*, 2:9–31, 1987. 9
20. Dinesh Manocha. Interactive geometric computations using graphics hardware. *SIGGRAPH course notes*, 2002(31), 2002. 1
21. Nimrod Megiddo. Linear-time algorithms for linear programming in R^3 and related problems. *SIAM Journal on Computing*, 12(4):759–776, 1983. 7
22. Atsuyuki Okabe, Barry Boots, and Kokichi Sugihara. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*. Probability and Mathematical

Statistics. John Wiley & Sons, Chichester, England, September 1992. foreword by D. G. Kendall. 2, 3

23. Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction*. Texts and Monographs in Computer Science. Springer-Verlag, New York NY, 1995. corrected and expanded second printing, 1988. 8

24. Christian Schwarz, Jürgen Teich, Emo Welzl, and Brian Evans. On finding a minimal enclosing parallelogram. Technical Report TR-94-036, Berkeley, CA, 1994. 7

25. Michael Ian Shamos and Dan Hoey. Closest-point problems. In *Proc. 16th Annual Symp. Foundations of Computer Science*, pages 151–162. IEEE Computer Society, 13–15 October 1975. 8

26. Sharir and Toledo. Extremal polygon containment problems. *CGTA: Computational Geometry: Theory and Applications*, 4, 1994. 8

27. E. Welzl. Smallest enclosing disks (balls and ellipsoids). In *Proceedings of New Results and New Trends in Computer Science*, volume 555 of *LNCS*, pages 359–370, Berlin, Germany, June 1991. Springer. 7