Algorithmic Geometry via Graphics Hardware

Dissertation
zur Erlangung des Grades
des Doktors der Ingenieurswissenschaften (Dr.-Ing.)
der Naturwissenschaftlich–Technischen Fakultät I
der Universität des Saarlandes

von

Markus Oswald Denny

Saarbrücken
14. März 2003

Datum des Kolloquiums: 31.10.2003

Dekan der technischen Fakultät:
Professor Dr. Philipp Slusallek

Gutachter:
Professor Dr. Raimund Seidel, Universität des Saarlandes, Saarbrücken
Professor Dr. Philipp Slusallek, Universität des Saarlandes, Saarbrücken

Kurzzusammenfassung

Wir entwickeln ein *pixelbasiertes* Berechnungsmodell, dem die Möglichkeiten moderner Grafikhardware zu Grunde liegt. Auf der Basis dieses Modells untersuchen wir exemplarisch verschiedene Probleme aus dem Gebiet der algorithmischen Geometrie. Unser Hauptbeispiel ist die Berechnung des Voronoi-Diagrammes einer planaren Punktmenge mittels der unteren Einhüllenden eines Arrangements von Kegeln. Wir führen eine detaillierte Analyse des Fehlers durch, der durch die Verwendung von pixelbasierten Algorithmen auftreten kann. Wir stellen dann eine Methode vor, um die Berechnung des Voronoi-Diagrammes erheblich zu beschleunigen. Dieser generische Algorithmus wird auch dazu benutzt, eine Vielzahl von Verallgemeinerungen des Voronoi-Diagrammes zu berechnen, zum Beispiel Diagramme bzgl. allgemeiner Distanzfunktionen, Diagramme basierend auf Liniensegmenten und Kreisbögen sowie Diagramme höherer Ordnung. Außerdem können auch Kombinationen dieser Verallgemeinerungen berechnet werden. Für einige dieser Verallgemeinerungen ist kein schneller klassischer Algorithmus bekannt. Des Weiteren stellen wir einige Anwendungen vor, insbesondere eine Lösung für das folgende Problem: Gegeben sei ein Voronoi-Diagramm einer planaren Punktmenge. Bestimme die Position eines neu einzufügenden Punktes, dessen Voronoi-Region im daraus resultierenden Voronoi-Diagramm maximale Fläche besitzt. Soweit uns bekannt ist, existiert keine klassische Lösung für dieses Problem. Weiterhin präsentieren wir pixelbasierte Algorithmen, um die Alphahülle einer planaren Punktmenge S zu berechnen, sowie um das flächenminimale und flächenmaximale Dreieck bzw. Viereck zu bestimmen, dessen Ecken Punkte aus S sind. Schließlich stellen wir pixelbasierte Algorithmen vor, die das kleinste S umschließende Homothet eines sternförmigen Polygons P berechnen, sowie das größte leere Homothet von P bezüglich S, das vollständig in einer beliebigen polygonalen Region enthalten ist. Alle in dieser Arbeit vorgestellten Algorithmen sind schnell und einfach zu implementieren, Kenntnisse in der Grafikprogrammierung und in OpenGL vorausgesetzt.

Abstract

We develop a *pixel-based* model of computation relying on the power of modern graphics hardware. It provides the foundation on which we exemplarily discuss various problems arising in the field of algorithmic geometry. Our prime example is the computation of the Voronoi diagram of a planar point set via the lower envelope of an arrangement of cones. We give a detailed analysis of the error occurring due to the nature of pixel based algorithms. Furthermore, we present a method to achieve a significant speedup for the computation. This generic algorithm is further used to compute various generalizations of the Voronoi diagram like diagrams based on generalized distance functions, line segments and circular arcs as Voronoi sites, as well as higher order Voronoi diagrams. Additionally, we can compute combinations of these generalizations as well. For some of these generalizations no fast traditional algorithm is known. We further give some applications, in particular we present a solution to determine the position for a point p such that, included in a Voronoi diagram of a planar point set, the resulting Voronoi region of p has maximal area. As far as we know there exists no traditional solution for this problem. Moreover, we present pixel based algorithms to compute the alpha hull of a planar point set S, and to determine the minimum and maximum area triangle and quadrangle spanned by points of S. Finally, we demonstrate how to determine the smallest homothet of a star shaped polygon P enclosing S and how to find the largest empty homothet of P completely contained inside an arbitrary polygonal region. All algorithms presented in this work are fast and easy to implement, assuming knowledge about graphics programming and OpenGL.

Zusammenfassung

Nehmen wir an, ein Manager einer Schnellrestaurantkette will den Markt in einer Stadt erschließen, in der er noch nicht präsent ist. Natürlich sollte das Einzugsgebiet des neuen Restaurants so groß wie möglich sein. Um diesen optimalen Platz zu finden, konstruiert er ein Voronoidiagramm der momentan vorhandenen Filialen seiner Konkurrenten. Der beste Platz für seine Filiale ist derjenige Punkt, der, eingefügt in das Voronoidiagramm, die neu entstehende Region maximiert. Überraschenderweise ist kein traditioneller Algorithmus zur Lösung dieses *Settle Point* Problems bekannt. Wir können es in einer diskretisierten Version lösen. Unser Ansatz basiert auf einem Berechnungsmodell, welches die Leistungsfähigkeit moderner Hardware ausnutzt.

Sogar die billige und weitverbreitete PC-Grafikhardware ist mächtig genug, komplexe Szenen völlig unabhängig vom Hauptprozessor darzustellen. Diese Leistungsfähigkeit wollen wir uns zu Nutze machen, um Probleme der algorithmischen Geometrie zu lösen.

Der Einsatz der Grafikhardware für nicht vorgesehene Aufgaben birgt allerdings Hindernisse. Die Information fließt hauptsächlich in eine Richtung, nämlich vom Hauptprozessor zum Grafikprozessor. Der Hauptprozessor übergibt eine Beschreibung der darzustellenden Szene an den Grafikprozessor und lässt diesen seine Arbeit ohne Rückmeldung verrichten. Ein weiteres Merkmal von Grafikhardware ist, dass sie sich wie ein Parallelcomputer verhält, bei dem jedes Pixel konzeptionell von einem eigenen Prozessor verwaltet wird. Allerdings können wir den Parallelismus nur ähnlich einer SIMD-Instruktion (Single Instruction Multiple Data) verwenden. Diese Eigenschaften finden im Modell Beachtung.

Unser Modell ist so gestaltet, dass es die zugrunde liegende Hardware möglichst genau wiedergibt. Gleichzeitig ist es aber auch abstrakt genug, um auf verschiedene Typen von Grafikhardware angewandt zu werden, sowohl auf hochgezüchtete Grafik-Workstations als auch auf billige PC Grafikkarten. Deshalb ist unser Berechnungsmodell an OpenGL angelehnt, der populärsten herstellerunabhängigen Grafikbibliothek.

Mit Hilfe dieses *pixelbasierten* Berechnungsmodells entwickeln wir schnelle Algorithmen für Probleme der algorithmischen Geometrie. Eine der vielseitigsten Strukturen in der algorithmischen Geometrie, das Voronoidiagramm, dient uns als Paradebeispiel. Wir berechnen es mittels der unteren Einhüllenden eines Arrangements von Kegeln. Wir präsentieren eine detaillierte Analyse des Fehlers, der durch die Verwendung von pixelbasierten Algorithmen auftreten kann. Zudem liefern wir korrekte obere Schranken für die Anzahl der benötigten Dreiecke für die Approximation der unteren Einhüllenden. Des Weiteren stellen wir eine Methode für eine erhebliche Beschleunigung der Berechnung vor. Anders als Manocha et al. [HKL+99] sind wir in der Lage, das Voronoidiagramm in fast konstanter Zeit zu berechnen, Gleichverteilung der Punkte vorausgesetzt.

Dieser generische Algorithmus wird des Weiteren dazu benutzt, verschiedene Verallgemeinerungen des Voronoidiagrammes zu berechnen, zum Beispiel Diagramme bzgl. allgemeiner Distanzfunktionen, Diagramme basierend auf Liniensegmenten und Kreisbögen sowie Diagramme höherer Ordnung. Außerdem können auch Kombinationen dieser Verallgemeinerungen berechnet werden. Es ist zum Beispiel möglich, das Voronoidiagramm k-ter Ordnung einer Menge von additiv gewichteten Liniensegmenten und Kreisen zu berechnen, wobei die zugrunde liegende Distanzfunktion die $L_{1/2}$ Funktion ist. Für einige dieser Verallgemeinerungen ist kein schneller klassischer Algorithmus bekannt.

Alle Algorithmen sind schnell und sie sind unkompliziert zu implementieren, einen erfahrenen Programmierer vorausgesetzt. Die Berechnung einer der oben erwähnten Verallgemeinerungen für eine gleichverteilte Punktmenge von bis zu 2000 Punkten benötigt etwa 50 Millisekunden bei einer Bildgröße von 1000×1000 Pixels. Für Voronoidiagramme k-ter Ordnung braucht man allerdings etwa k-mal länger, vorausgesetzt k ist ausreichend klein.

Die Berechnung von Voronoidiagrammen und deren Verallgemeinerungen ist ohne Zweifel schon für sich gesehen eine interessante Aufgabe. Voronoidiagramme sind oft der erste Schritt einer Lösung eines komplexerern Problems. Auf der Grundlage unseres *pixelbasierten* Modells, zeigen wir exemplarisch *grafische* Lösungen für Anwendungen von Voronoidiagrammen auf, die als pixelbasierte Algorithmen realisiert werden. Diese sind die Berechnung der Hausdorff-Distanz zweier Punktmengen, der Durchmesser einer Punktmenge sowie die Berechnung der maximalen Distanz zweier Punktmengen.

Wie oben bereits erwähnt, dient unser Modell als Fundament, mit dessen Hilfe Algorithmen aus ganz verschiedenen Gebieten der algorithmischen Geometrie realisiert werden können. Um diese Vielfältigkeit zu untermauern, untersuchen wir pixelbasierte Implementationen zur Berechnung der Alpha-Hülle einer planaren Punktmenge. Die Alpha-Hülle wurde entwickelt, um das Aussehen, die Form, einer Punktmenge widerzuspiegeln.

Weiterhin entwickeln wir Lösungen, die zu einer planaren Punktmenge S das flächenkleinste und flächengrößte Dreieck bzw. Viereck berechnen, deren Ecken Punkte aus S sind.

Schließlich stellen wir pixelbasierte Algorithmen vor, die bzgl. einer planaren Punktmenge S das kleinste S umschließende Homothet eines sternförmigen Polygons P berechnen, wobei die Menge S auch aus Kreisbögen und Liniensegmenten bestehen kann. Wir können als Spezialisierung ebenfalls den kleinsten Kreis berechnen, der die Menge S enthält. Das duale Problem, die Berechnung des größten leeren Homothets von P bezüglich S, das vollständig in einer beliebigen polygonalen Region enthalten ist, kann sogar noch ein wenig schneller berechnet werden, da wir auf die für Voronoidiagramme vorgestellte Methode zur Beschleunigung der Berechnung zurückgreifen können.

Die in dieser Arbeit vorgestellten Algorithmen erlauben nicht nur allgemeinere Problemstellungen im Vergleich zu traditionellen Ansätzen, sondern sie sind effizient sowie schnell und unkompliziert zu implementieren, Kenntnisse in Grafikprogrammierung und OpenGL vorausgesetzt. Für einige der hier vorgestellten Probleme ist kein schneller klassischer Algorithmus bekannt. Zu guter Letzt erhalten wir, bedingt durch die Art der pixelbasierten Algorithmen, eine Visualisierung der Lösung gratis dazu.

Summary

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 in. Most naturally, the area to attract customers 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. Your store is best located on that point, that when inserted into the existing diagram as a new site, will possess the maximum area. Surprisingly, no algorithm was known to solve this *settle point* problem. We can solve it in a discretized version. Our approach is based on a model of computation relying on the power of modern graphics hardware.

Even the cheap and widespread graphics hardware used in modern PCs is powerful enough for rendering complex graphic scenes completely independent of the CPU. We want to harness the power of such a graphic processing unit (GPU) to solve problems in algorithmic geometry.

Abusing graphics hardware for such an unintended purpose has some obstacles. The information flow is almost unidirectional, namely from the CPU to the GPU. The CPU hands over a description of the scene to be rendered and relies on the GPU to carry out its duty without further feedback. This complicates the development of algorithms. Another aspect of graphics hardware is, that although it behaves like a parallel computer where conceptually a processor resides on each pixel, we can only use the parallelism in SIMD-like fashion (single instruction multiple data). Of course, these properties exert influence on the development of the model.

Our model is designed to reflect the underlying hardware as well as possible, however, abstract enough to be applicable for various different types of graphics adapters, including high end graphics workstations as well as cheap PC graphics hardware. For that, our model of computation is designed following OpenGL, the most popular vendor-independent graphics library.

In this *pixel based* model of computation, we give fast algorithms for a number of problems in the field of algorithmic geometry. As one of the most versatile structures in computational geometry, the Voronoi diagram serves as our prime example. It is computed via the lower envelope of an arrangement of cones. We give a detailed analysis of the error occurring due to the nature of pixel based algorithms. We provide correct upper bounds on the number of required triangles for the approximation of the lower envelope. Furthermore, we present a method to achieve a significant speedup for the computation. In contrast to the approach of Manocha et al. [HKL$^+$99], we are able to render the Voronoi diagram in almost constant time, assuming a uniform distribution of the sites.

This generic algorithm is further used to compute various generalizations of the Voronoi diagram, like diagrams based on generalized distance functions, line segments and circular arcs as Voronoi sites, and higher order Voronoi diagrams, as well as combinations of these generalized Voronoi diagrams. For instance, we can compute the order k Voronoi diagram of a set of additively weighted line segments and circles with $L_{1/2}$ as underlying distance function. For some of these generalizations no fast traditional algorithm is known.

All these algorithms are fast and straightforward to implement for experienced programmers. For a uniform distribution of up to 2000 sites, about 50 milliseconds are sufficient to compute any of the before mentioned generalization of the Voronoi diagram assuming a picture size of 1000×1000 pixels (the computation of the order/degree k Voronoi diagram takes k times longer on moderate k).

Certainly, the computation of the Voronoi diagram and its generalization is an interesting task on its own. Besides, Voronoi diagrams are often used as a first step of an algorithmic solution for more complex problems. Based on our model of computation, we exemplarily give *graphical solutions* for applications of the Voronoi diagram realized as pixel based algorithms. These are the computation of the Hausdorff distance between a red and a blue set of points, the diameter of a set of points, and the maximum distance between a red and a blue set of points.

As mentioned before, our model serves as a foundation to provide solutions for problems of quite different areas of computational geometry. To prove the versatility, we investigate pixel based implementations for the computation of the alpha hull of a planar point set. The alpha hull was invented to capture the notion of the shape of a set of points.

Furthermore, given a planar point set S, we present solutions for determining the minimum and maximum area triangle and quadrangle spanned by points of S.

Finally, we consider the problem of finding the smallest homothet of a star shaped polygon P enclosing a set S consisting of points, line segments, and circular arcs. As a specialization, we can also compute the mimimal circle enclosing these sites. The dual problem – given a star shaped polygon P, find the largest empty homothet of P completely contained inside an arbitrary polygonal region – has even a slightly faster solution, because we can make use of the speedup idea for Voronoi diagrams once more.

As a last remark, our pixel based implementations do not only allow for much more generalized problem settings, but also all algorithms presented in this work are fast and straightforward to implement, assuming knowledge about graphics programming and OpenGL. For some of the discussed problems there were no or no fast traditional solutions known. Last but not least, due to the nature of pixel based algorithms, a visualisation of the problem under consideration is delivered for free.

Contents

Chapter 0

Introduction

Since the ancient days of Euclid's straightedge-and-compass constructions, algorithmic geometry has always been a fascinating field of studies. The invention of the computer turned over a new leaf, melting algorithmic design, analysis and geometry into the new research discipline christened computational geometry.

Meanwhile, a new branch of scientific research called computer graphics came to life. At the very beginning, computer graphics appeared as a rather esoteric specialty involving expensive display hardware, substantial computer resource, and idiosyncratic software (cf. [FvDFH96]). Spurred by the enormous commercial success activated by the invention of the personal computer and especially the graphical user interfaces, computer graphics has grown into a recognized research discipline that a large community of researchers is engaged in. In the course of this development, computer graphics engines have emerged to powerful systems capable to compute and display complex graphical tasks on their own. To some extent, this success is owed to the game industry, which up to now is still an important catalyzer of this research branch. Nonetheless, this has the effect that the present personal computers command cheap and yet powerful graphics hardware.

This thesis gives the answer to the following question: How can computational geometry benefit from this powerful graphics hardware?

The RAM, the standard model of computation used in computational geometry, has very little to do with the peculiarities of graphics hardware. Thus, our first step is the development of a

suitable model of computation that is a realistic abstraction of current graphics hardware. This is the topic of investigation in the first part of the thesis. In the first chapter we present our *pixel based* model of computation, and in the second chapter discuss the relationship between our model and the real world.

There are two design criteria for our model. On the one hand, there needs to be a sufficient high level of abstraction to ensure that a wide range of actual graphics hardware is covered. On the other hand, the model has to reflect the underlying hardware as well as possible.

Hence, in the first part of this thesis, we investigate what is special in programming graphics hardware. A major feature is the flow of information, directed from the main processor, the CPU, towards the graphic processor, the GPU. Usually, the CPU hands over just a description of the scene to be rendered. Thereafter, it relies on the GPU to carry out its duty without further feedback. There is no need to establish more than a one-way communication directed from the CPU to the GPU. This is the reason, why retrieving data from the GPU is rather time expensive. The other aspect of graphics hardware is that although it behaves like a parallel computer where conceptually a processor resides on each pixel, we can only use the parallelism in SIMD-like fashion (single instruction multiple data). Our pixel based model of computation inherits these aspects as it is designed following OpenGL the most popular vendor-independent graphics library.

Now, that we have an appropriate model of computation, what is it good for? This question is answered by several pixel based algorithms in the second part of this thesis. Based on our model of computation, we give fast solutions for a number of problems in the field of algorithmic geometry.

As one of the most versatile structures in computational geometry, the Voronoi diagram, discussed in chapter 3, serves as our prime example. It is computed via the lower envelope of an arrangement of cones. We give a detailed analysis of the error occurring due to the nature of pixel based algorithms. We provide correct upper bounds on the number of required triangles for the approximation of the lower envelope. Furthermore, we present a method to achieve a significant speedup for the computation. Compared to the approach of Manocha et al. [HKL+99], we are able to render the Voronoi diagram in almost constant time, assuming a uniform distribution of the sites.

This generic algorithm is further used to compute various generalizations of the Voronoi diagram, like diagrams based on generalized distance functions, line segments and circular arcs as Voronoi sites, and higher order Voronoi diagrams, as well as combinations of these generalized Voronoi diagrams. For some of these generalizations no fast traditional algorithm is known. For instance, we can compute the order k Voronoi diagram of a set of additively weighted line segments and circles with $L_{1/2}$ as underlying distance function.

All these algorithms are fast and straightforward to implement for experienced programmers. Assuming a uniform distribution of up to 2000 sites, about 50 milliseconds are sufficient to compute any of the before mentioned generalization of the Voronoi diagram assuming a picture size of 1000×1000 pixels (the computation of the order/degree k Voronoi diagram takes k times longer on moderate k). This result is achieved on an Intel-Pentium™ 800 equipped with a NVIDIA™GForce 2 graphic adapter.

Certainly, the computation of the Voronoi diagram and its generalization is an interesting task on its own. Besides, Voronoi diagrams are often used as a first step of a more complex algorithmic solution. Based on our model of computation, we exemplarily give *graphical* solutions for applications of the Voronoi diagram realized as pixel based algorithms. These are the computation of the Hausdorff distance between a red and a blue set of points, the diameter of a set of points, and the maximum distance between a red and a blue set of points.

One of the most exciting aspects of our model is that we can give an answer to problems for which as far as we know no traditional solutions exist. In particular, we present a solution for the *settle point* problem, i.e. determine the position for a point p such that, included in a Voronoi diagram of a planar point set, the resulting Voronoi region of p has maximal area.

As mentioned before, our model serves as a foundation to provide solutions for problems of quite different aeras of computational geometry. To prove the versatility, we investigate the following problems:

In chapter 4 we delve into the problem of finding the *Smallest Enclosing Homothet*. This is a generalization of the well known smallest enclosing circle problem in the sense that our approach allows to determine the smallest homothet of a star shaped polygon

P enclosing a set S consisting of points, line segments, and circular arcs.

The dual problem – given a star shaped polygon P, find the largest empty homothet of P completely contained inside an arbitrary polygonal region – is considered in chapter 5. As before, we can allow the sites to be points, line segments, and circular arcs. In contrast to traditional approaches, we allow the polygonal region the homothet has to be contained in, to be be quite arbitrary, i.e. it may also contains holes.

In chapter 6 we present pixel based implementations for the computation of the alpha hull of a planar point set. Presented 1983 by Edelsbrunner, Kirkpatrick, and Seidel, alpha hulls were invented to capture the notion of the shape of a set of points.

Furthermore, in chapter 7 we develop efficient solutions for determining the minimum and maximum area triangle and quadrangle spanned by points of a planar point set.

Part I

The model

Chapter 1

Pixel based Model of Computation

1.1 Introduction

Before defining a model of computation for the pixel based algorithms we will first have a look at the usual model used to classify algorithms in computational geometry.

First and foremost the purpose of a model is to list and explain the available operations. On the basis of this data it is then possible to compare competing algorithms in (asymptotic) running time or space consumption depending on the size of the input.

Although there exist different philosophies concerning the structure and organization of a computer and a CPU, counting the number of arithmetic operations and comparisons of the algorithm in consideration gives a pretty good measure for the resources eventually consumed. This is a consequence of the fact that the time cycles allocated by the different instructions of a CPU differ by only a rather small constant factor. This is to bear in mind when deriving an appropriate model for pixel based algorithms.

1.2 Overview

Motivated by the speed at which modern graphic engines can generate pictures, we transform problems arising in computational geometry into the computer graphics framework.

The graphic engines compute a picture on the basis of triangles. From our point of view, a picture is a two dimensional array of pixels, each consisting of seven buffers: z, z′, stencil, red, green, blue, and alpha (see figure 1.2).

Based on OpenGL, a widespread free graphic library, our model can be coarsely sketched as follows (see figure 1.4).

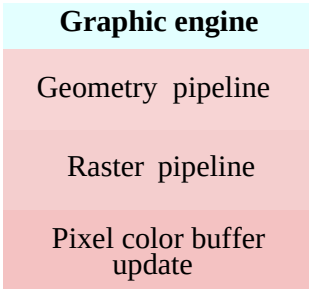


Figure 1.1
Structure of a graphic engine

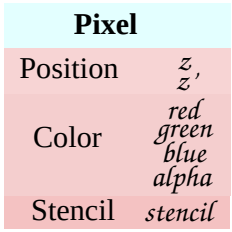


Figure 1.2
Structure of a pixel

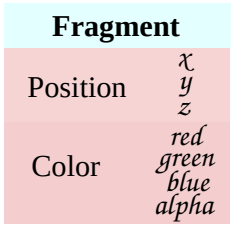


Figure 1.3
Structure of a fragment

Geometry pipeline

One at a time, the triangles enter the graphic engine at the geometry pipeline (see section 1.3). After a triangle is colored, and affinely transformed, parts of the triangle are clipped away with the aid of six clipping planes.

The coloring function, and the coefficients for the affine transformation and clipping planes define the state of geometry pipeline. In general, state values remain valid unless explicitly changed by the CPU, but at least for the complete lifetime of the triangle in process.

At the end of the geometry pipeline any remaining portion of the triangle is pushed through a sieve producing a set of fragments. Each fragment consists of seven values derived from the originating triangle. There are four color values, namely red, green, blue, and alpha, and three position values, x, y, and z (see figure 1.3).

Raster pipeline

The next stage is the raster pipeline (see section 1.4). From here on the graphic engine behaves like a SIMD[1] parallel computer where conceptually a processor resides on each pixel. With respect to the x and y values, the fragments are distributed to the pixels. Before

[1] SIMD refers to an architecture with a **s**ingle **i**nstruction stream that is capable of operating on **m**ultiple **d**ata items concurrently (cf. [FR97]).

the buffers of a pixel can be updated, four tests must be passed. As soon as a test fails the processing for this fragment terminates.

In the scissor test, the fragment's x and y values are compared to predefined state values.

Next, the alpha value of the fragment is compared to a reference value. The compare function and the reference value are again part of the state of raster pipeline.

In the stencil test, a comparison between the stencil buffer value of the pixel and a reference value is executed. The state of the pipeline concerning this test, is composed of the compare function, the reference value, and additionally three update operations, applied to the pixel's stencil buffer. The first is applied if the stencil test fails, the second if the stencil test succeeds but the depth tests (see below) fails, otherwise the third is applied.

The depth test consecutively compares the fragment's z value to the values of the z and z' buffer of the pixel. If a comparison is successful the corresponding buffer is overwritten with the fragment's value. As in the former tests, the kind of applied comparisons is selected by two state variables.

Color buffer update

After all tests are passed, the four color buffers of the pixel are updated (see section 1.5). There are two mutual exclusive alternatives applicable. The fragment's color values can be blended with the color buffer values of the pixel or they can be logically combined. There are several different methods applicable for blending, as well as a bunch of logical operations (see section 1.5). The selection between blending and logical operations, and the number of the applied method respectively logical operation is dictated by state variables of the raster pipeline.

Reading back information

The current contents of the pixel's buffers can be read back into the main memory of the CPU (see section 1.7). One method is to read back the specified buffers, e.g. stencil buffers, of a rectangular area of the picture. Alternatively, there are two methods to gather

statistical information about the picture. The histogram function returns the occurrences of a color buffer value. The minmax function computes the minimum and maximum values for the color buffers of the picture.

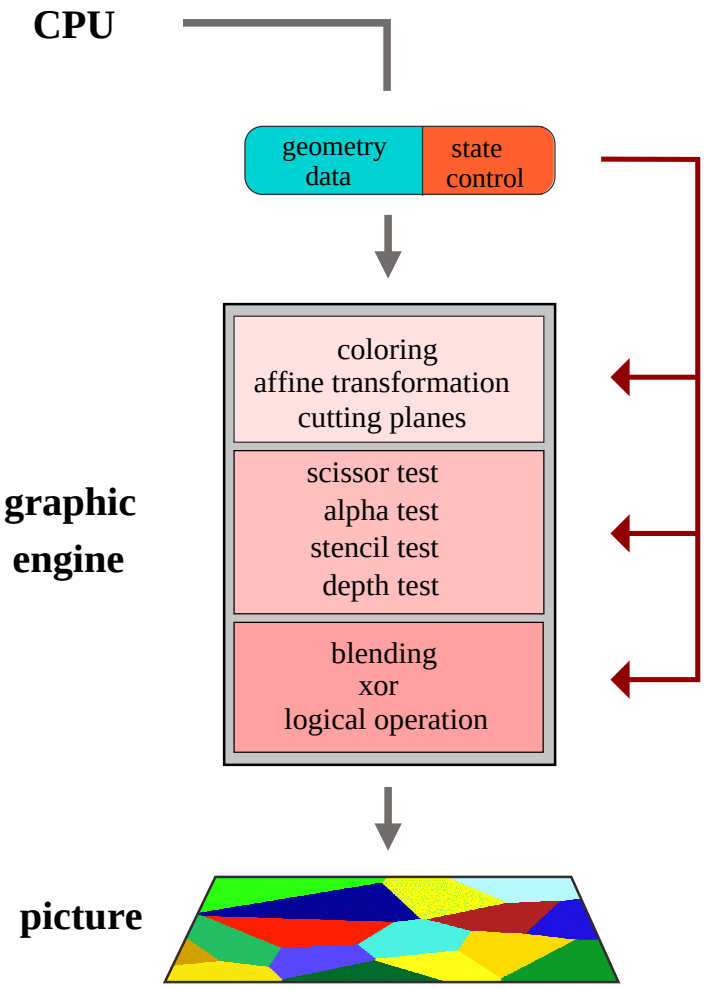


Figure 1.4
Model of computation

1.3 Geometry pipeline

We now give a more detailed description of the operating sequence of the graphics engine (see figure 1.6). The first one, the geometry pipeline, allows geometric computations on triangles. The input

for the geometry pipeline is made up of data to generate geometric objects as well as instructions to control the state of graphic engine.

Geometric primitives

Each object is composed of geometric primitives. The graphic generates these as fed by the CPU with three dimensional vertices and a keyword to select between the different types (see figure 1.5). As illustrated in the figure, only convex polygons are allowed.

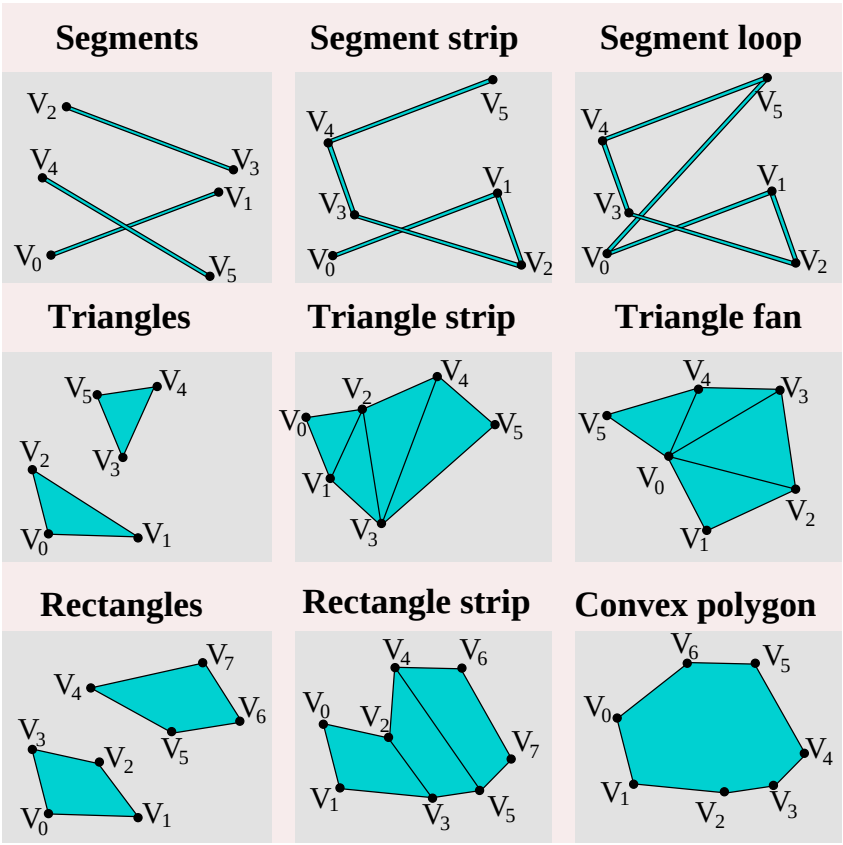


Figure 1.5

Geometric primitives

For the sake of simplicity, we refer to the geometric primitives as set of triangles, although line segments, rectangles, and polygons are also allowed geometric primitives.

Coloring

For each set of triangles there are four additional values determined by state variables, namely the red, green, and blue as well as the so called alpha value. The latter is a fourth color component conventionally used to describe the transparency of the triangle under consideration.

Beside the standard way to assign a color for the triangles, it is possible to apply texturing.

A *texture* is a two dimensional array of color values. These values are equidistant supporting points defining a two dimensional color function which can be mapped upon a geometry primitive.

Using textures it is possible to define quite arbitrary shaped object.

Affine transformation

The next step to take place, is an affine transformation of the just generated object.

Clipping planes

Now that the triangles are placed as desired, obviously superfluous parts overlapping the viewing volume are automatically clipped away. To further reduce the amount of eventually generated fragments up to six clipping planes can be declared. Each of these is specified as state variables in terms of $A_i x + B_i y + C_i z + D_i = 0$.

Any part that survived the final clipping is rastered into small squares called fragments, as defined above, in such a way that each fragment corresponds to a pixel. Each of the thus generated set of fragments inherits the same color values from its triangle. This set is passed through as input to the next stage.

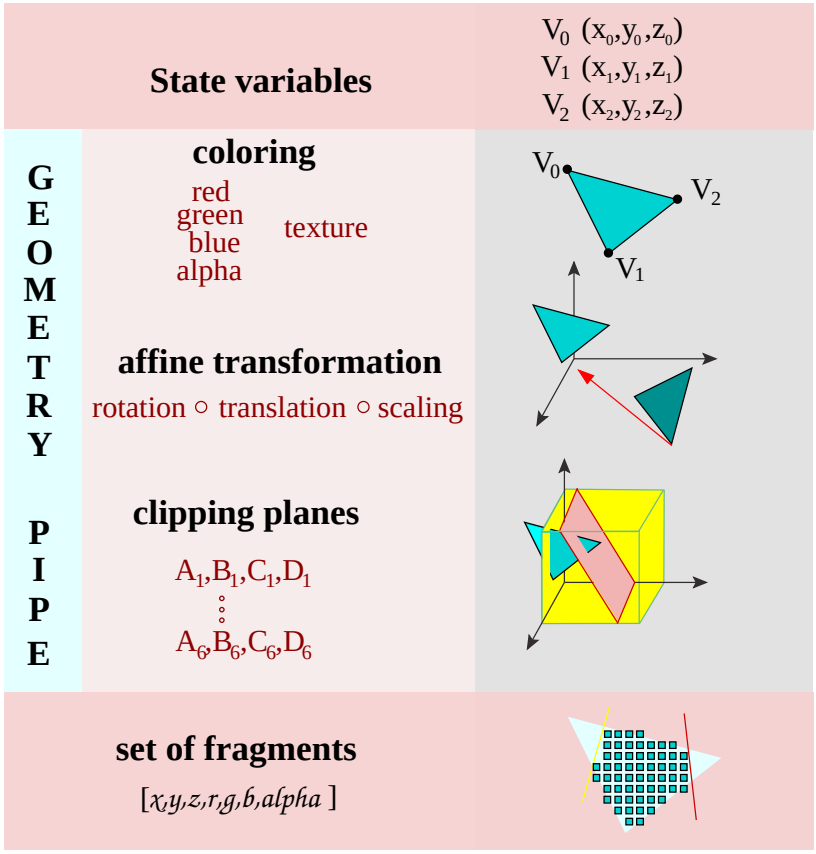


Figure 1.6
Geometry pipeline

1.4 Raster pipeline

In contrast to the geometry pipeline, the input of the raster pipeline is a set of fragments.

Like the triangles in the geometry pipeline, the fragments from the input set are pushed through the pipeline, separately and independently of each other (see figure 1.7). In a way the processing of the fragments can be compared to a SIMD parallel processor field with a rather simple processor residing on each pixel.

According to the x and y values of a fragment, it is attached to the appropriate pixel.

Based on the position, depth, and color values, a fragment has to undergo four tests until it eventually alters the buffers of the associated pixel. These tests are the scissor, alpha, depth and stencil test.

The order of these tests as stated above is predetermined and cannot be altered. As a result of passing the depth or stencil tests, buffers of the corresponding pixel are changed.

As soon as a fragment fails either of the first two tests, it vanishes from the pipeline without any further side effects.

The values and parameters of the per fragment tests define the state of the raster pipeline. These settings are valid for the entire set of fragments. A change of parameters can only be caused by a new set of fragments.

Scissor test

The first test to pass is the scissor test. Here it is possible to define a rectangular portion of window containing the picture. If a fragment resides inside this area, it passes the scissor tests. Else, the fragment is rejected without any side effects on the pixel buffers.

Alpha test

The alpha value of a fragment is compared to the value of the corresponding state variable. Allowed comparison function is any out of $<, \leq, =, \geq, >, \neq$. Additionally it is possible to always accept or always reject a fragment.

Again, if the fragment does not succeed this test, it vanishes from the pipeline without any further side effects.

Stencil test

The scissor and the alpha tests apply solely on the values of the fragment. These tests reduce the portion of the input eventually used for computation.

In contrast to these tests, the stencil test is applied to the pixel

the fragment is attached to.

Its stencil buffer value is compared to a reference value. As comparison function one can use one out of the set of functions as above.

Furthermore any result of the comparison, even the negative one, causes a side effect on the stencil buffer of the pixel. A negative outcome of the test will erase the fragment.

In the same manner as the reference value and compare function are specified, it is possible to execute a predefined action on the stencil buffer. There are three cases to consider. The fragment fails the stencil test, second it passes this test but fails the subsequent depth test and finally it passes both tests. For each possible result one of the following actions can be executed:

- keep the current value of the stencil buffer
- replace the stencil buffer value with zero
- replace the stencil buffer value with the reference value
- increment or decrement the stencil buffer by 1 (with or without wrap around)
- bitwise invert the buffer

Depth test

The depth test splits up into two consecutive depth units. The depth test is declared to be passed if and only if both test units are successfully passed by the fragment. An unsuccessful test will cause the fragment to be erased.

The first of the two tests operates on the z buffer of the pixel, whereas the second one relies on the z' buffer. Besides this difference they behave in exactly the same manner. For this reason we restrict our description on the first unit of the depth test.

Similar to the stencil test the allowed compare function might be any out of $<, \leq, =, \geq, >, \neq$. Beside these, it is possible to always accept or always reject a fragment.

Up to now all tests compare a reference value either against a value of a fragment or against a buffer of the corresponding pixel.

The depth test is the only test in which data from the fragment is compared directly to data belonging to the pixel.

If an incoming fragment passes the depth test, the z buffer value of the pixel is replaced by the fragment's value.

As described earlier the result of the depth test has side effects on the stencil buffer.

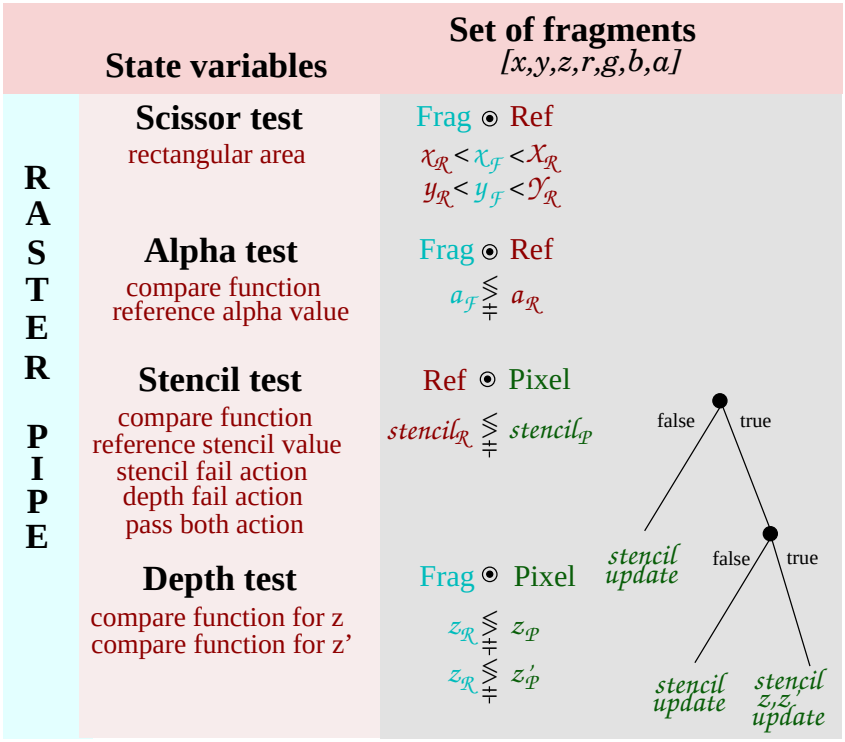


Figure 1.7

Raster pipeline

1.5 Pixel color buffer update

Any fragment which passed all the per fragment tests is finally allowed to appear as visual impression on the screen. Which means that the red, green, blue, and alpha color buffers of the corresponding pixel are updated.

The simplest method to accomplish an update is to overwrite the existing values with the incoming ones. But besides this there

are two other methods. Either the values can be combined using logical operations or the fragment data can be blended with that of the pixel. Whatever method is specified, it is again valid for the entire set of the remaining fragments.

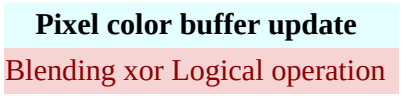


Figure 1.8
Pixel color buffer update

Blending

Instead of just overwriting the previously stored values, blending allows the combination of the color values from an incoming fragment with that of the attached pixel. The tradeoff between incoming and already existing values can be made to depend on the corresponding alpha values.

The new value for each color buffer is computed in the following way. All values are assumed to be between zero and one. Let R_F, G_F, B_F, α_F, R_P, G_P, B_P, α_P be the fragment and pixel values. I_R, I_G, I_B, I_α, E_R, E_G, E_B, E_α denote the blending factors of the incoming fragment and the existing pixel. Then the new pixel color values eventually compute to:

$$(R_P E_R + R_F I_R,\ G_P E_G + G_F I_G,\ B_P E_B + B_F I_B,\ \alpha_P E_\alpha + \alpha_F I_\alpha)$$

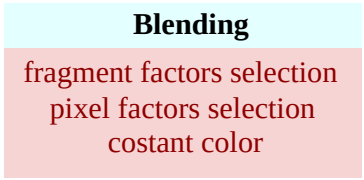


Figure 1.9
Blending state variables

For each color the new value is computed separately and independently. Each resulting value of this quadruplet is rounded in the mathematical correct sense and is clamped to 1, to lie inside the interval $[0, 1]$. The table below contains all possible blending factors, whereas *min* denotes $\min(\alpha_P, 1 - \alpha_F)$ and C_R, C_G, C_B, C_α a predefined constant color. The entries for the fragment's and the pixel's blending factors can be chosen independently from each other.

E_R, E_G, E_B, E_α	I_R, I_G, I_B, I_α
$0,0,0,0$ $1,1,1,1$	$0,0,0,0$ $1,1,1,1$
R_F, G_F, B_F, α_F $1-R_F, 1-G_F, 1-B_F, 1-\alpha_F$	R_P, G_P, B_P, α_P $1-R_P, 1-G_P, 1-B_P, 1-\alpha_P$
$\alpha_F, \alpha_F, \alpha_F, \alpha_F$ $1-\alpha_F, 1-\alpha_F, 1-\alpha_F, 1-\alpha_F$ $\alpha_P, \alpha_P, \alpha_P, \alpha_P$ $1-\alpha_P, 1-\alpha_P, 1-\alpha_P, 1-\alpha_P$	$\alpha_F, \alpha_F, \alpha_F, \alpha_F$ $1-\alpha_F, 1-\alpha_F, 1-\alpha_F, 1-\alpha_F$ $\alpha_P, \alpha_P, \alpha_P, \alpha_P$ $1-\alpha_P, 1-\alpha_P, 1-\alpha_P, 1-\alpha_P$
C_R, C_G, C_B, C_α $1-C_R, 1-C_G, 1-C_B, 1-C_\alpha$	C_R, C_G, C_B, C_α $1-C_R, 1-C_G, 1-C_B, 1-C_\alpha$
$C_\alpha, C_\alpha, C_\alpha, C_\alpha$ $1-C_\alpha, 1-C_\alpha, 1-C_\alpha, 1-C_\alpha$	$C_\alpha, C_\alpha, C_\alpha, C_\alpha$ $1-C_\alpha, 1-C_\alpha, 1-C_\alpha, 1-C_\alpha$
———	min, min, min, min

We can for example achieve a write protection of the pixel color buffers, if we take the $(I_R, I_G, I_B, I_\alpha)$ vector to be all zero and the $(E_R, E_G, E_B, E_\alpha)$ vector to be all one.

Logical operations

Logical operation

Logical operation selection

Figure 1.10
Logical operation state variables

As an alternative to blending we can also combine the fragment's color values with the one of the pixel using a bitwise logical operation.

Any applicable operation is listed in the following table where P denotes the pixel color quadruplet and F that of the fragment.

Parameter	Operation	Parameter	Operation
Clear Buffers	all 0's	Set Buffers	all 1's
F-values	F	negated F-values	$\neg F$
P-values	P	negated P-values	$\neg P$
AND	$F \wedge P$	NAND	$\neg(F \wedge P)$
OR	$F \vee P$	NOR	$\neg(F \vee P)$
XOR	$F \mathrm{xor} P$	Equivalent	$\neg(F \mathrm{xor} P)$
OR-reversed	$F \vee \neg P$	OR-inverted	$\neg F \vee P$
AND-reversed	$F \wedge \neg P$	AND-inverted	$\neg F \wedge P$

1.6 General settings valid for all buffers

Beside the above mentioned settings for the per fragment operations there are some additional common features.

Any pixel buffer can be write protected. So although the fragment succeeded all tests, a change of the buffer values can be omitted.

1.7 Retrieving information

Reading back rectangular area

In order to transfer the current contents of the pixels buffers back to the CPU, a rectangular area of the picture and the kind of buffer has to be specified.

Unless the exact values are needed, but only information about their distribution, there are two alternatives to the above.

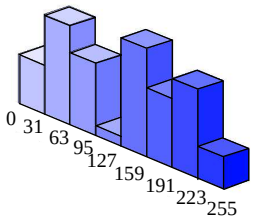


Figure 1.11
Histogram on blue with eight intervals

Histogram

The first function gets as arguments a color buffer type and a positive integer n. It examines for each pixel the specified color buffer and returns a histogram of the distribution of these values. The number of intervals for the histogram is determined by n, which has to be a power of two (see figure 1.11).

Min	Max
0	255
12	55
16	255
0	255

Figure 1.12
Minmax example

Minmax

The second method to collect statistical information is the minmax function. It returns for each of the four color buffer types the minimum and maximum color buffer values, computed over all pixels (see figure 1.12).

Chapter 2

A model meets reality

In the previous chapter we declared functions and operations providing a tool set for the construction of pixel based algorithms. But does it make sense to develop algorithms based on this model, or is it just a theoretical phantasy?

In what follows, the answer to that question is given.

2.1 OpenGL's history

The relationship between the model and what is feasible in reality is established by OpenGL, an open graphics library.

As soon as a processor only for graphic demands existed, there was a need for a language to establish a communication with it. But it was not before 1992 that SGI™ released OpenGL, an open so called vendor independent application programming interface (API) fulfilling this task.

Although vendor and hardware independence was an important aspect for OpenGL, it is designed to reflect the underlying hardware as well as possible to ensure fast actions.

As a consequence, commands concerning windowing tasks, and user input as well as high level commands for describing models of three-dimensional objects, respectively, aren't supported at all. These are delegated to more sophisticated libraries built on top of OpenGL and the appropriate operating systems.

Nowadays OpenGL consists of about 250 commands as well as numerous extensions of different hard and software developers proving OpenGL to be a vivid and most important graphics library. The importance of OpenGL is best demonstrated by the fact that any graphically demanding game uses OpenGL.

2.2 Practical aspects

OpenGL is structured similarly to the previously described model. There is also a geometry and a raster pipeline. However, since the library is built for programming actual graphic engines, OpenGL has a lot more to offer than our model demands for. However, additional features differ in size and complexity from one to the other graphic adapter, we rely on only a basic subset of the OpenGL functionality, with the only exception of the z' buffer (see below).

Our algorithms are implemented using a NVIDIA GForce graphic adapter, a low-cost solution providing everything we need. Compared to *real* graphic stations, the main drawback are the restricted buffer sizes.

Buffer size management

In general, the buffer size of the red, green, blue, alpha, and stencil buffers is limited to eight bits each. The z buffer is 24 bits wide. In the case that there is no alpha buffer needed, the z buffer can be enlarged to 32 bits.

What happens if a buffer value is increased beyond the maximum value or decreased below zero? Regarded from the graphic scientist's viewpoint, it is desirable that brightening an absolute white picture shouldn't change it. The same holds for darkening an absolute black picture. So, nothing happens at all. The only exception is the stencil buffer, which could be advised to wrap around in case of an overflow and underflow, respectively.

The z' buffer

To be true, there is no second z buffer in the current version of OpenGL. But there are a lot of computer graphic scientists supporting the idea of a second z buffer to be realized in OpenGL. Besides, with the SGIX shadowing OpenGL extension developed by NVIDIA™(cf. [cas02][1]), a second z buffer can be realized in hardware with full depth buffer precision.

Display lists

OpenGL commands can be merged together in display lists. The contents of such lists is preprocessed and might be executed more efficiently. Additionally, display lists reduce the amount of data send to the graphic adapter, since it is possible to execute a command sequence by only referring to the appropriate display list index. Geometric object composed of numerous geometric primitives are good candidates for display lists.

Textures

As mentioned before, a texture is a two[2] dimensional array of color values, representing equidistant supporting points defining a two dimensional color function.

The effect textures are invented for is to generate an impression of grain or some sort of ornamentation avoiding quite complex and elaborate set of graphic primitives (see figure 2.1).

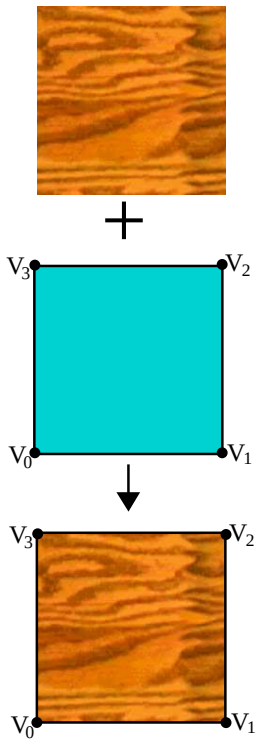


Figure 2.1
Standard use of textures

Textures are subject to certain conditions with respect to the size. Any texture has to be quadratic with a side length of a power of two. Depending on the underlying hardware it is obvious that there are only a limited number of texture elements feasible.

On the other hand with the concept of textures it is for example possible to realize a better approximation of *exact* circles. Using standard OpenGL commands drawing a circle results in just a square. Since a square is much easier and thus faster to draw.

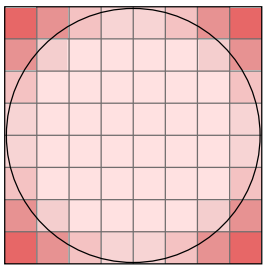


Figure 2.2
Texture circle

[1] Interactive Order-Independent Transparency

[2] Besides the two dimensional textures, OpenGL even knows one and three dimensional textures.

Regarding standard applications in computer graphics, it is not that important to have a truly round circle. In a texture we can specify for each element a color coding the portion the element belongs to the circle (see figure 2.2). Besides for the construction of *exact* circles, applying the concept of textures in the just described way, it is easy to define quite arbitrary shaped object, as well.

Depth textures

A depth texture is a special kind of textures, which stores depth values instead of color values. In the same way as a normal texture affects the color values of the geometric primitive (respectively the values of the generated fragments) it is applied to, the depth texture changes the depth values of the object under consideration.

Depth textures are realized as an OpenGL extension. Regarding the NVIDIA™ production family, the *GL_ARB_depth_texture* extension can be accessed by GForce 3 or higher GPU's.

A fourth stage

In effect, there is a fourth stage as the computed pixels have to be displayed. Although converting the digital frame buffer data into analog video signals is in itself not a trivial task it is not a matter of investigation in this work.

Reading back

Graphic adapters are primary constructed to display the geometry provided by the CPU as quick as possible. Optimal performance is achieved if most of the data just resides on the graphic adapters memory such that the amount of data transmitted by the CPU is small. As long as our goal is to just display our graphical computation, everything is fine.

It might happen that a buffer size is to small for the desired computation. Then intermediate backups of the buffers contents to main memory is unavoidable. But reading back the entire picture into the CPU's main memory is rather time consuming. To illustrate this statement consider the following example.

Assume a picture size of 1000 times 1000 pixels of which just the red, green, blue, and alpha values – each 8 bit wide – are to be read back in main memory. Moreover assume the CPU possesses a modern advanced graphics port to communicate with the GPU. Nonetheless the time spent for just reading back the color buffer equals the time required for rendering about half a million triangles. So it is obviously a good idea to avoid these operations and to use the histogram or minmax functions wherever possible.

Memory restrictions

The next problem to cope with is the restricted amount of memory on the graphic adapter. This memory is used for example to store display lists and texture elements. In contrast to display lists texture elements can be very space demanding. Insufficient memory on the graphic board results in a time punishment comparable to a CPU forced to swap data to a hard disk. Consequently the size of the texture elements in use is an important factor. But even nowadays low-end GPUs possess enough memory of about 64 byte per pixel such that *swapping* to the CPU's main memory is rather seldom, as far as we are concerned.

Part II

Implementations

Chapter 3

Voronoi Diagram

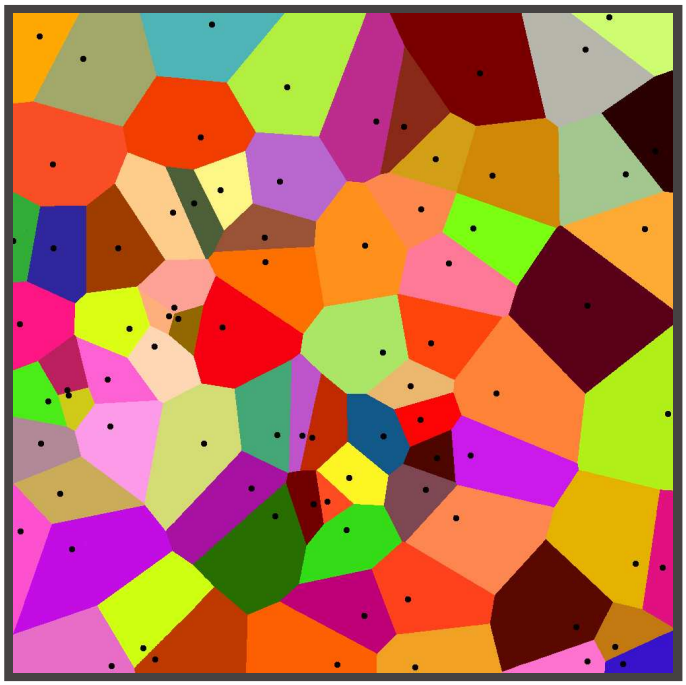


Figure 3.1
Voronoi diagram

3.1 Introduction

In this chapter we present a pixel based computation of various Voronoi diagrams.

Definition

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 (cf. figure 3.1).

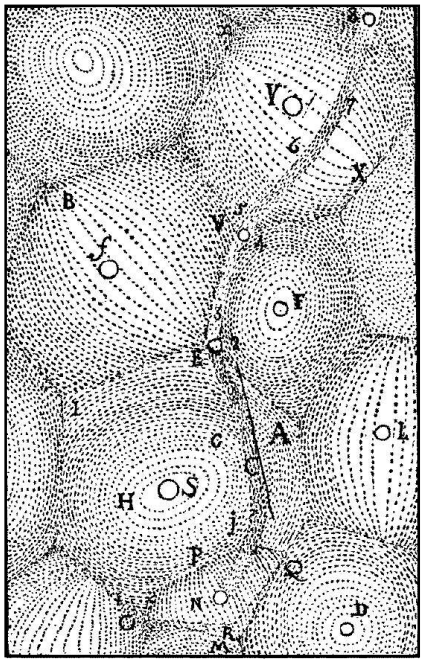


Figure 3.2
Descartes' decomposition of space into vortices

Historical note

Voronoi diagrams have been known for at least four centuries, first appearing in Descartes' treatise "Le Monde de Mr. Descartes, ou Le Traite de la Lumière", published in 1644 [Des44]. Descartes uses Voronoi-like diagrams to claim that the solar system consists of vortices (cf. [AK00]). He thought of space as a decomposition into convex regions, each consisting of matter revolving round one of the fixed stars (figure 3.2).

Illustration

The concept of Voronoi diagrams can be illustrated by the following example. Imagine a tourist on a survival tour across a desert. Equipped with a GPS navigation system she always knows about her global position. A map of the desert reveals the position of the surrounding oases to her. Assume a plot of the Voronoi diagram based on the oases is also inscribed into the map. Finding the nearest oasis is without any doubt of vitally importance. The nearest oasis is determined by the Voronoi region her current position belongs to.

Real world applications

The above mentioned illustration is a more modern variation of the post-office problem introduced by Knuth [Knu73, page 555] and solved as one application for Voronoi diagrams in computational geometry by Shamos and Hoey [SH75].

But there are also many other fields besides computational geometry in which the Voronoi diagrams and related structures are utilized to derive a description of certain phenomena.

Voronoi diagrams are used in anthropology and archeology to identify the parts of a region under the influence of different neolithic clans, chiefdoms, ceremonial centers, or hill forts. In metallurgy they are used for modeling grain growth in metal films and even in zoology Voronoi diagrams are deployed for modeling and analyzing the territories of animals (cf. [Epp]).

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. [Aur91], [AK00], [AA02], [OBS92]).

Algorithmic properties and computational complexity

There are many different algorithms for constructing Voronoi diagrams. Nearly any algorithmic design paradigm known nowadays in computer science may be applied to this task. There are algorithms based on the sweep line paradigm ([For86]), divide and conquer([SH75]), randomized incremental construction (cf. [OIM84], [GKS90]) as well as algorithms exploiting the relationship between Voronoi diagrams and three dimensional convex hulls (cf. [Bro79], [ES85]).

The lower bound for the computational complexity for the construction of the Voronoi diagram of n sites in the computation-tree model proves to be $\Omega(n \log n)$ (cf. [Sha75], [PS95, 192-195]). This holds, because the *element uniqueness* problem – given n numbers, decide if any two are equal – is reducible to computing an adequate Voronoi diagram.

3.2 Computing lower envelope

The relationship between the Voronoi diagram of n sites and the lower envelope of the arrangement of a set of n cones, first observed by Edelsbrunner and Seidel (cf. [ES85]), emerges as particularly interesting for us. It is as follows:

Let $S = \{s_1, \ldots, s_n\} \subset \mathbb{R}^3$ denote the set of points 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. $C = \{(x, y, z) \in \mathbb{R}^3 | z = \sqrt{x^2 + y^2}\}$. We now observe what happens if we position such a cone on top of every site $s \in S$.

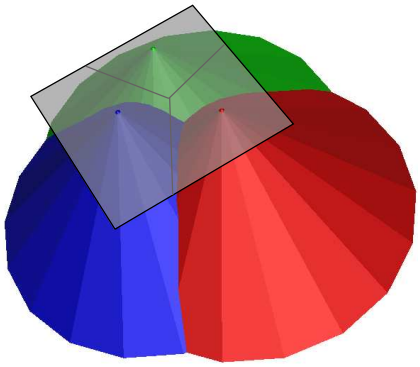


Figure 3.3
Construction of a Voronoi diagram employing cones

Theorem 3.1 *The orthogonal projection of the lower envelope of the arrangement of the cones is exactly the partition of the plane into Voronoi regions.*

Proof: Let $p = (x_p, y_p, 0)$ be an arbitrary point of the xy-plane, and $r = \{(x_p, y_p, z) \in \mathbb{R}^3 | z \geq 0\}$ denote an orthogonal ray starting at p. We now examine the intersection points of the ray with each cone. Due to the construction of the cones, we know that the height of the intersection point equals the distance between p and the apex of the cone. As a consequence the first cone hit by r is located at the site nearest to p. Since the cone is the first intersection, it is also part of the lower envelope at that position. That concludes our proof. □

Computing the lower envelope of the arrangement is exactly the way we compute Voronoi diagrams by using graphic adapters (cf. [HKL+99], and section 3.3 resp.). This observation can also be extended to derive a relationship between higher order Voronoi diagrams and the orthogonal projection of the parts of the arrangement of the cones, which is the subject of section 3.7. Moreover, Voronoi diagrams based on quite arbitrary distance functions can be computed in this way by changing the shape of the bases of the cones. In section 3.6 we investigate non euclidean distance functions, and in section 3.8 we examine weighted Voronoi diagrams, exploiting the idea of different cone shapes.

3.3 A first graphical implementation

This implementation is due to Manocha et al [HKL+99] initiated by a note in the OpenGL 1.1 Programming Guide (cf. [WNDO99, pages 587-589]). It approximates the Voronoi diagram for point sites in the plane.

Given a point site s, the distance from s induces a function the graph of which is a right circular cone with its apex in s. Such a cone is created for each site, such that each cone resides in the half space $z \leq 0$ touching the site's origin with its apex. As shown in the last section, the orthogonal projection of upper envelope of the arrangement of these cones gives us the Voronoi diagram of the sites.

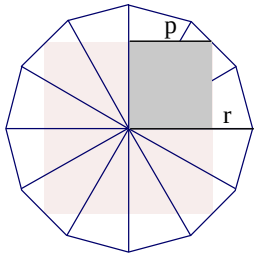


Figure 3.4
Cone size compared to picture size

Pixel based adaptation

The cone is approximated as a triangle fan (see figure 1.5 on page 11). The number of triangles required for the approximation of the cone depends on the error we are willing to accept and the size of the picture we generate a Voronoi diagram for. If not stated otherwise, we assume the picture to be a square of size $p \times p$ pixels.

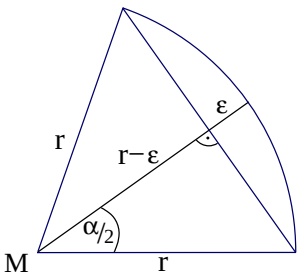


Figure 3.5
Maximal error in approximation of the cone

Manocha et al restrict their implementation to Voronoi diagrams based on the euclidean norm. In this case, the distance function graph is a right circular cone. Since the cone is approximated as triangle fan, the base of the approximation is a regular polygon with T vertices. Let M denote the center of the polygon, r the distance between M and a vertex of the polygon, and α the angle $2\pi/T$ (see figure 3.5). The maximal radial distance between the triangle fan and the correct cone is denoted by ε. Since a site may reside on a corner of the picture, r has to be at least as great as the diameter of the picture (see figure 3.4).

Hence the number of required triangles T computes as:

$$\begin{aligned} \cos(\alpha/2) &\leq \frac{r-\varepsilon}{r} \\ r &= p\sqrt{2} \end{aligned}$$

$$\Rightarrow \qquad T \qquad \geq \left\lceil \frac{\pi}{\arccos \frac{p\sqrt{2}-\varepsilon}{p\sqrt{2}}} \right\rceil \tag{3.1}$$

Demanding ε to be less than 1, and assuming a picture size of 1024×1024 pixels, 85 triangles are sufficient.

Each cone is assigned a unique color, e.g. the site's index as color value. Once generated as a display list, it is translated such that the apex of the cone points at the appropriate site's position. The depth buffer is enable in order to extract the upper envelope of the arrangement of the cones. To do so, it has to be configured such that only the highest fragments for each pixel are allowed to pass to make a pixel color buffer update. This concludes the proposal due the Manocha et al.

3.4 The error caused by the approximation

As the above calculation proves, using a triangle fan of 85 triangles guarantees, that the distance from a pixel to the apex of the cone (i.e. the site under consideration) is at least overestimated, in the worst case by ε. This holds, because the correct distance graph is a right circular cone bearing the property that the error made in horizontal direction is the same as that made in vertical direction.

We denote by $\mathrm{dbv}(p)$ the depth buffer value of pixel p. Let S be the set of point sites. Then, for each pixel p of the final picture holds that

$$\min_{s \in S} d(s,p) \leq \mathrm{dbv}(p) \leq \min_{s \in S} d(s,p) + \varepsilon.$$

Unfortunately, an ε-approximation of the minimal distance does not imply an equally good coloring of the lower envelope of the cones. Thus, the assignment of the pixels to their nearest site might be wrong.

To prove this assertion, we examine the deviation in the bisector occurring for two sites which lie at the left lower corner of the picture at (1,0) and (0,1).

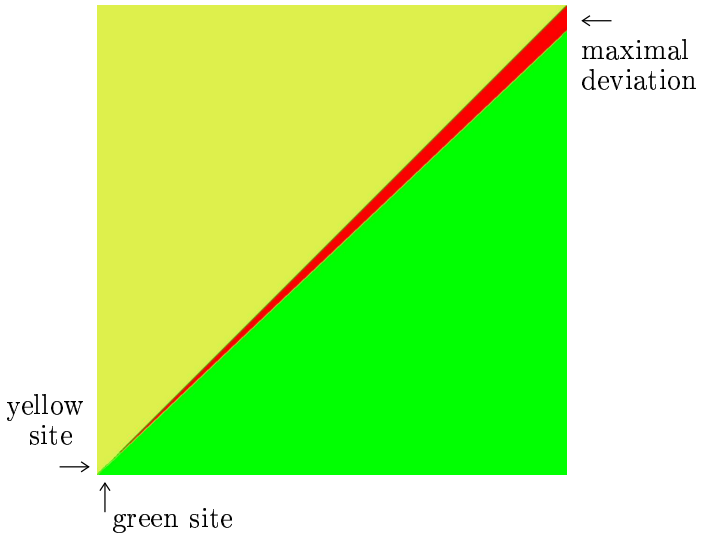


Figure 3.6
Worst case deviation (in red) of correct and computed bisector of two sites in the lower left corner

Although we constructed the triangle fan as recommended above, i.e. within a deviation of $\varepsilon = 1$, there is a deviation in the computation of the bisector of more than 50 pixels in the right upper corner for a picture size of 1024×1024 pixels. This rather unexpected behavior is worthwhile of further investigation.

In order to distinguish the two notions for an acurate approximation, we introduce the following terms.

Definition 3.1 (distance diagram versus site diagram) *In case of we aim at a good distance approximation for each pixel, we call the desired diagram, a distance diagram. Otherwise, if we are interested to bound the maximum deviation, the diagram under consideration is called site diagram.*

With regards to this terminology, Manocha et al. compute the number of required triangle to satisfy a ε distance diagram.

Worst case analysis

In order to derive an upper bound on the number of triangles required to compute a site diagram, we calculate the deviation in the case that the approximation is always as bad as possible for one site and at the same time as well as possible for the other site.

For this reason, we investigate the intersection of two circular cones. The first cone is the biggest possible cone which just fits inside the triangle fan. Thus, the base of the first cone (at height r) is a circle with radius $r - \varepsilon$. The second cone is the smallest possible cone containing the triangle fan, i.e. nothing else but the cone we intented to approximate by the triangle fan. This is illustrated in figure 3.7. In dependence on figure 3.6, the base of the first cone is colored green and the base of the second cone is colored yellow. The ploygonal chain in between is the base of the approximation, i.e. the triangle fan.

In lemma 3.1, we prove that the orthogonal projection of the intersection of two cones (as just described) is a circle. This knowledge can now be used to argue that the smaller the distance between two sites is, the larger is the deviation with regards to the actual (euclidean) bisector.

Lemma 3.1 (Intersection of cones) *Given two right circular cones with different apex angle. If the apices of both cones lie on the plane $z = 0$ and their axes are perpendicular to that plane, then the orthogonal projection of their intersection is a circle.*

Proof: For the ease of calculation we assume the apex of the first cone to be at $(0, 0, 0)$ and the apex angle to be $\pi/2$. Then the cone satisfies the equation

$$x^2 + y^2 - z^2 = 0.$$

Without loss of generality, the second cone is assumed to have a smaller apex angle, represented by the constant $c > 1$. Furthermore, its apex is assumed to be at position $(u, v, 0)$. Then, the cone satisfies the equation

$$c((x - u)^2 + (y - v)^2) - z^2 = 0, \quad c > 1.$$

The intersection of the two cones is represented by the quadric Q

given a the set of zero of the equation 3.2

$$(x,y)\underbrace{\begin{pmatrix}1-c & 0\\ 0 & 1-c\end{pmatrix}}_{A}\begin{pmatrix}x\\ y\end{pmatrix}+2\underbrace{(cu,cv)}_{\mathfrak{a}^T}\begin{pmatrix}x\\ y\end{pmatrix}-c(u^2+v^2)=0. \tag{3.2}$$

We apply the principal axis transformation. Because of $\det(A) = (1-c)^2 \neq 0$, the equation

$$A\begin{pmatrix}x\\ y\end{pmatrix}=-\mathfrak{a}. \tag{3.3}$$

is solvable, which proves that Q is a central quadric. The solution $(cu/(c-1), cv/(c-1))$ of equation 3.3 is the center of Q. Translating the coordinate system about the center, we obtain a simplified formula

$$x^2+y^2=c\frac{u^2+v^2}{(1-c)^2}. \tag{3.4}$$

The principal axis transformation is already completed, since A is diagonal. In equation 3.4 all coefficients are positive and equal. Furthermore, $c\frac{u^2+v^2}{(1-c)^2} > 0$. Hence, the orthogonal projection of the intersection of the two cones is a circle with center $(cu/c-1, cv/c-1)$ and radius $\sqrt{c\frac{u^2+v^2}{(1-c)^2}}$. This concludes our proof. □

With this knowledge in mind, we can now derive an upper bound on the number of required triangles per cone. It depends on the annulus in which the polygon chain has to be. For that, we investigate the maximal deviation of the bisector of two sites occurring by approximating the correct cone by a triangle fan.

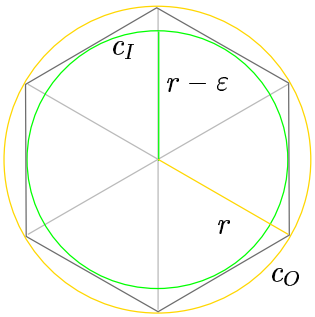


Figure 3.7
Worst case study of the approximation error

Let r denote the radius of the cone, large enough to cover the entire picture. We assume c_I to be a right circular cone with radius $r-\varepsilon$, touching the triangle fan from the inside, and c_O to be right circular cone with radius r, touching the approximation from the outside (see figure 3.7). In doing so, the worst case is always included in our consideration.

$$c_O := \{(x,y,z) \in \mathbb{R}^3 | z = \sqrt{x^2+y^2}\ \}$$

$$c_I := \{(x,y,z) \in \mathbb{R}^3 | z = \sqrt{x^2+y^2}\ \frac{r}{r-\varepsilon}\}$$

Let s_I be the position of the apex of the cone c_I and s_O be the position of the apex of the cone c_O. We denote by a the distance

between s_I and s_O. For the ease of calculation, we assume s_O to be at $(-\frac{a}{2}, 0)$ and s_I to be at $(\frac{a}{2}, 0)$. Then the bisector B is given as (cf. figure 3.8)

$$B := \{(x,y) \in \mathbb{R}^2 \mid \quad ((x+\frac{a}{2})^2 + y^2) = ((x-\frac{a}{2})^2 + y^2)\,(\frac{r}{r-\varepsilon})^2\,\}. \tag{3.5}$$

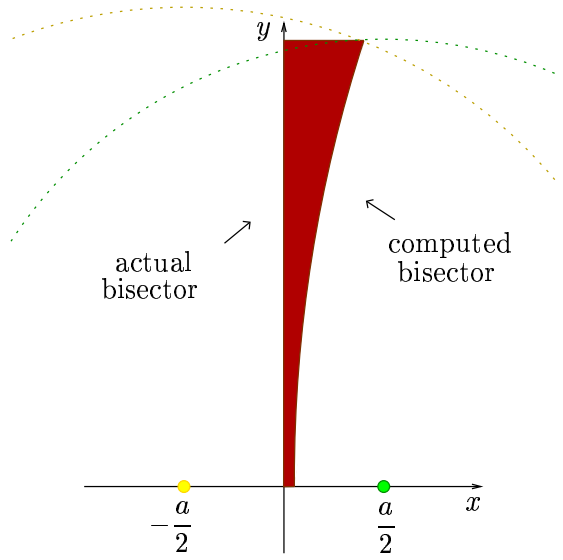


Figure 3.8
Bisector deviation

We know from lemma 3.1 that the bisector is a circle. Furthermore, it is easy to see that the smaller the distance between the two sites is, the smaller is the circle defining the bisector. That means, that the derivation between the actual and the computed bisector increases the nearer the sites are.

If we want a maximal deviation of at most 1 pixel in the worst case and assume a picture size of 1000×1000 pixels, then the number of triangles per cone suffices our worst case study is 2642, about 30 times more triangles than assumed by Manocha et al. [HKL+99].

Overcoming the approximation problem using depth textures

Although only a few triangles are sufficient to compute a good approximation for the *minimal distance picture*, a huge number of

triangles is required if our goal is an appropriately good approximation of the Voronoi regions. Instead of using a triangle fan to approximate the cone, we can also make use of depth textures to compute the Voronoi diagram overcoming the before mentioned approximation error.

In a precompute step, we calculate a picture wide depth texture, which contains for each entry of the texture (for short texel) the appropriate depth value (with regard to a *virtual* site at $(0, 0)$). The former triangle fan is now substituted by a square of twice the length of the picture.

Each quarter of the square is rendered with the depth texture applied, such that for any position of a site the square covers the entire picture. Additionally, it provides the exact depth value for each pixel as it has adopted the depth values of the applied depth texture.

The use of depth textures guarantees not only the most accurate approximation, but it has also the effect that we can compute other kinds of diagrams, which formerly were not feasible. For instance, the power diagram can be realized with depth textures. The power diagram is a generalization of the Voronoi diagram based on the *power distance*, $d(p, s_i) = ||p - s_i||^2$. The graph of distance function is a circular paraboloid and as such almost impossible to be approximated by triangle fan.

Of course, we do not get all these advantages without paying for it. One critical aspect of depth textures, is that up to now they are only accessible as OpenGL extension. Thus, it might be the case, that they are not accessible at all. A second disadvantage is, that rendering a texture is considerably time consuming, especially if we consider the fact that any texture has to be quadratic with a side length of a power of two. Furthermore, using triangle fans is sometimes more flexible, e.g. a Voronoi diagram (respectively the distance picture) of line segments and circular arcs can be computed in straightforward fashion using triangle fans (cf. section 3.9). Last but not least, the construction step of the distance matrix is also time consuming.

Our conclusion is, that depth textures are the best choice, whenever accurate coloring is demanded. In case of only the distance for each pixel to its nearest site is required, triangle fans are still a good alternative.

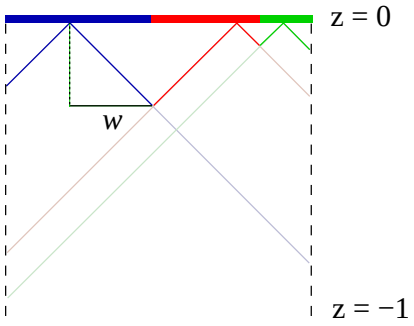


Figure 3.9
Width w of the blue region

In the applications presented here, we just rely on the distance of a pixel to its nearest site, and not in the correct coloring. Thus, in the sequel of the chapter, we limit our considerations on cones, constructed as triangle fans.

3.5 Speedup

Assume the picture to be a square of size $p \cdot p$. As soon as the size n of the input set for the Voronoi diagram is greater than 1, there are ineffective fragments generated due to unnecessarily large cones. The rendering of each cone generates a fragment for each pixel, thus $p \cdot p \cdot n$ fragments are created although $p \cdot p$ fragments are sufficient.

As long as the number of sites is rather small this approach works sufficiently well. Thus, the above described approach due to Manocha et al. [HKL+99] works sufficiently well for a rather small set of input.

Our idea to accelerate the computation, reduces the overhead of needlessly generated fragments. For that reason, we define the width of a Voronoi region and the width of a Voronoi diagram.

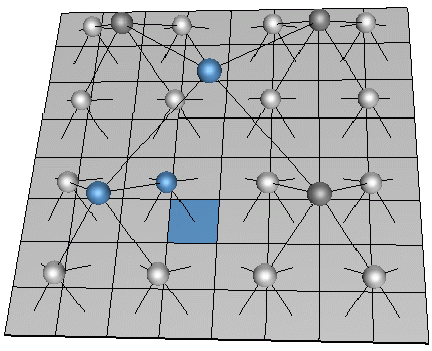


Figure 3.10
Level three quadtree with 4^3 leaves

Definition 3.2 (width of a Voronoi region) *The width of a Voronoi region with respect to a bounded area is the maximal distance from its site to any point belonging to that region (cf. figure 3.9).*

Definition 3.3 (width of a Voronoi diagram) *The width of a Voronoi diagram is the maximum of all widths over all regions.*

Hence, the width of a Voronoi diagram with respect to a bounded area is the radius of the largest empty circle such that the center of the circle is inside the area. 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 shading stage decreases.

Idea

Employing a quadtree structure gives us a fast approximation for the width of the Voronoi diagram. A quadtree is a spatial data structure, derived by recursively subdividing the picture in both dimensions into equal sized squares. This results in a tree structure with a branching factor of four (cf. figure 3.10). As a consequence of the construction of the quadtree, it follows, that the leaves represent the smallest squares. An internal node represents the union of the area of the squares the leaves in its subtree are responsible for.

For each level of the initially empty tree and for each site, we mark the nodes representing the area in which the site resides as being visited (cf. figure 3.10). In terms of the quadtree, we determine the path from the leaf representing the position of the site under consideration to the root of the quadtree.

Our intention is an upper and 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.

We will make use of the following theorem to derive an upper bound of the width.

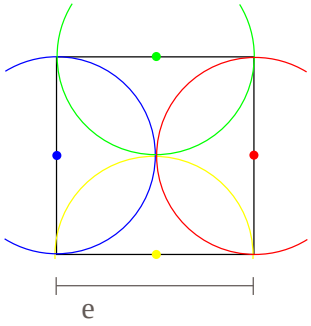


Figure 3.11
Lower bound for radius

Theorem 3.2 (Reduction of the height of the cones) *Let S be a set of points in the plane and $B \subset \mathbb{R}^2$ be a bounding area, containing S. Assume, there is a covering of B into rectangular cells such that each cell contains at least one point of S. let e denote the largest diagonal of all cells. Let c be a right circular cone as described in section 3.2, but with finite height (respectively radius) e, i.e. truncated at height e. Then the Voronoi diagram of S restricted to B is the orthogonal projection of the lower envelope of the arrangement of cones identical to c, set on top of each point site.*

Proof: It is easy to see, that the width of the corresponding Voronoi diagram is at most e, since the underlying area is completely covered by at the arrangement of cones. Each cell is completely covered by a at least one corresponding cone, such that the maximal height of the lower envelope is at most e. □

Lemma 3.2 (Adaption to quadtree partition) *Let S and B as in theorem 3.2. Assume there is a partition of B into quadratic cells of diameter e such that each cell contains at least one point of S. Let c be a cone truncated at height e, as before. Then the Voronoi diagram of S restricted to B is the orthogonal projection of the lower envelope of the arrangement of cones identical to c, set on top of each point site.*

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 (cf. figure 3.11).

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 picture will be covered. That is our upper bound.

Construction

The quadtree is filled as follows. For each site we determine the corresponding leaf of the quadtree and mark the path from the root to the leaf as visited.

This can be accelerated in that we avoid marking nodes serveral times. The idea is that 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.

Time consumption

The time consumed by the approximation procedure depends directly on the number of nodes marked as visited. This number can be bounded from above as follows. Let n be the number of sites and h be height of the quadtree. Assume n is less or equal the number of leaves of the quadtree. Then the number of visited

nodes v is at most (cf. figure 3.12):

$$v \leq \sum_{i=0}^{\lceil \log_4 n \rceil - 1} 4^i + n + n(h - \lceil \log_4 n \rceil).$$

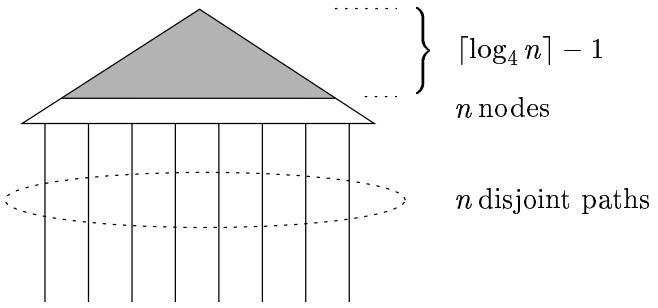


Figure 3.12
Upper bound for the number of visited nodes in the quadtree

More easily, in the worst case any node of the tree as well as any site is visited at most once. Thus, $n + \sum_{i=0}^{h} 4^i$ operations are sufficient.

We use a quadtree of height six for a picture of size 1024×1024. On an Intel-Pentium™ 800 the processing of 10000 sites requires 1.5 milliseconds.

Now that we know an upper and a lower bound, we can combine these values. In a first attempt, we render cones with a small radius greater than the lower bound, e.g. $e\sqrt{2}$, where e denotes the edge length derived of the lower bound computation. It might happen, that we draw the cones with a too small radius. Parts of the picture remain uncolored. If we choose the color for the cones such that their red, green, and blue color values are all greater than zero, then we can reveal this case with the aid of the OpenGL minmax function (see page 19). If any pixels remain uncolored, we repeat the rendering of the cones with a greater radius, stepping towards the upper bound.

Algorithm for computing the Voronoi diagram

```
clear buffers
enable depth test
determine radius approximation



LOOP:
     generate cone as display list
     forall sites s do
             set color(s)
             translate cone to position(s)
             execute display list

     test for uncolored pixels == true
             increase radius
             goto LOOP
```

Timings

To prove the effect of the speedup, we summarize some test series in the diagram 3.13. 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 Manocha et al. [HKL$^+$99]. 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. This behavior is reflected by the red line in the diagram.

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

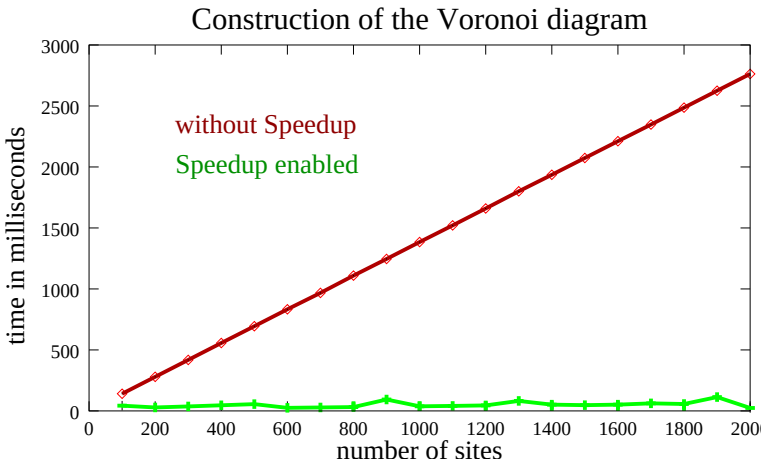


Figure 3.13

Rendering time with and without the quadtree

Even for 16000 sites the running time for the algorithm with the quadtree is about 170 milliseconds, assuming a uniform distribution of the sites.

The worst case setting for which the speedup method fails its purpose is a distribution for that one of the four quarters of the picture does not contain any points at all. Albeit, executing the quadtree procedure increases the running time by only 1.5 milliseconds per 10000 sites. For all other distribution, the quadtree proves to be a success.

Besides these, there is yet another aspect, we can benefit from. Since we know the maximal radius of the cones, we thereby also know the maximal height. This has the advantage, that we can adapt the depth buffer range appropriately. Thus in this way, applying the speedup method also decreases the probability of errors caused by the limited number of bits available for the depth buffer.

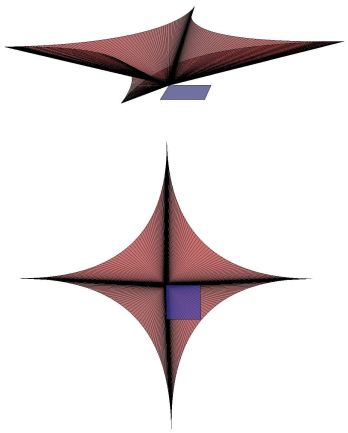


Figure 3.14

Graph of the distance function for $\ell = 0.5$, blue represents picture area

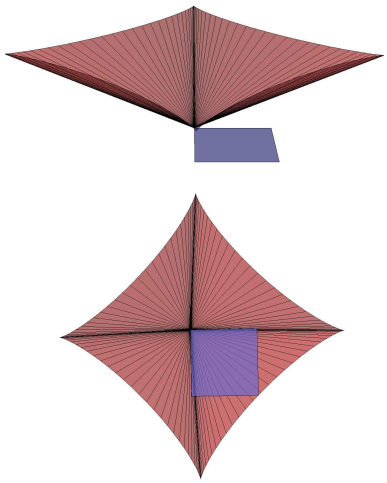


Figure 3.15

Graph of the distance function for $\ell = 0.8$, blue represents picture area

3.6 Voronoi diagrams based on non euclidean distance functions

In the same fashion as we constructed Voronoi diagrams for the euclidean norm, we can compute Voronoi diagrams for other, non euclidean, distance functions. In the sequel of this section, we in-

vestigate the number of required triangles to compute an ε distance diagram.

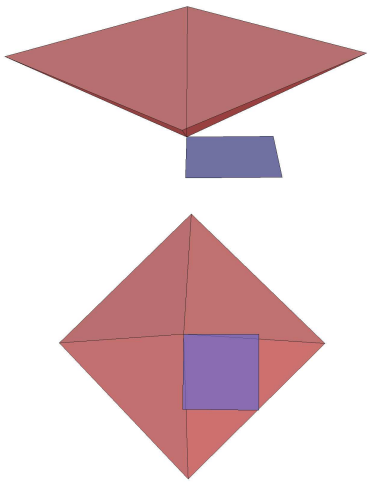


Figure 3.16
Graph of the distance function for $\ell = 1$, blue represents picture area

We start with distance functions induced by the Minkowski norms L_ℓ. Next we inspect what happens if ℓ lies in the open interval (0;1). We conclude our investigation with special cases of the Minkowski norms L_ℓ in which $\ell = \infty$ respectively $\ell = 1$, the max-norm and the Manhattan-norm. Both have the special property, that equidistant points may form areas and not just lines.

3.6.1 Distance functions based on the Minkowski norms

A vector norm, called Minkowski norm L_ℓ and denoted $||\cdot||_\ell$, is defined as

$$||u||_\ell = (|u_1|^\ell + |u_2|^\ell)^{1/\ell} \qquad 1 \leq \ell < \infty,\ u \in \mathbb{R}^2.$$

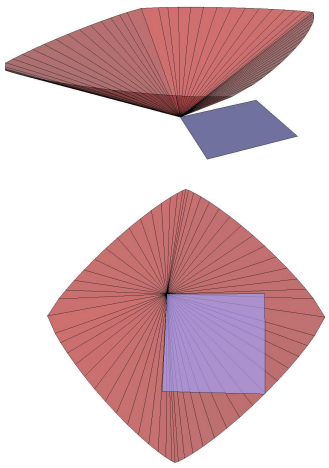


Figure 3.17
Graph of the distance function for $\ell = 1.2$, blue represents picture area

Commonly used Minkowski norms are the Manhattan norm and the euclidean norm, i.e. the L_1-norm and the L_2-norm.

We denote by $\text{dist}_\ell : \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}$ the distance function induced by the L_ℓ norm, i.e. $\text{dist}_\ell(u, v) = ||u - v||_\ell$.

Approximating the graph of the distance function

The graph of the distance function for $\text{dist}_\ell(0, v)$ is a cone with apex in $\mathbf{0}$. The cone's base $\mathcal{B}$ is a circle with regard to the L_ℓ-norm.

Our goal is to achieve the same error bounds as for euclidean distance diagrams (cf. page 34). In order to approximate the correct cone with a maximal error of ε, we define two functions f and g. The demanded polygonal chain lies inside the annulus defined by f and g (see figure 3.18).

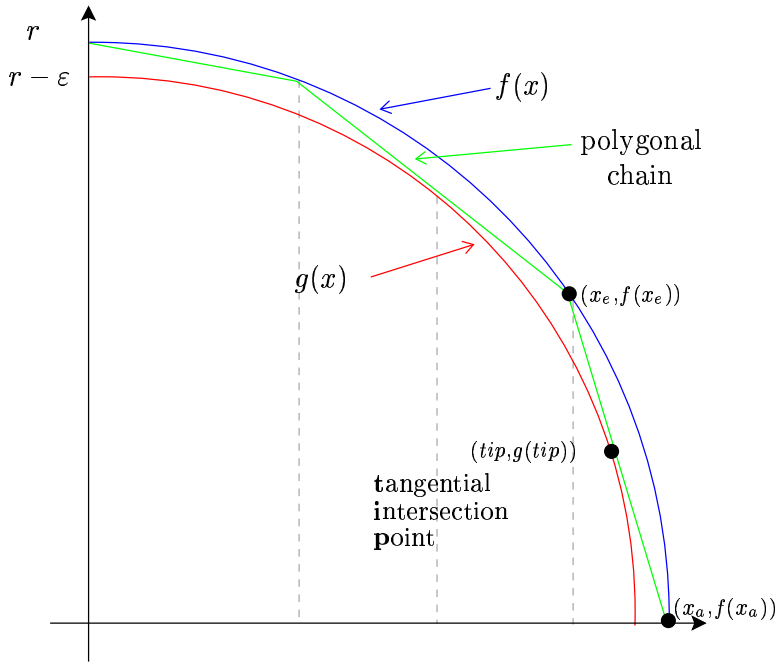


Figure 3.18
Construction of the polygonal chain

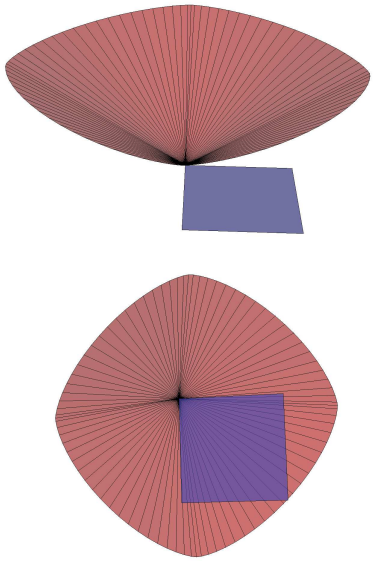


Figure 3.19
Graph of the distance function for $\ell = 1.5$, blue represents picture area

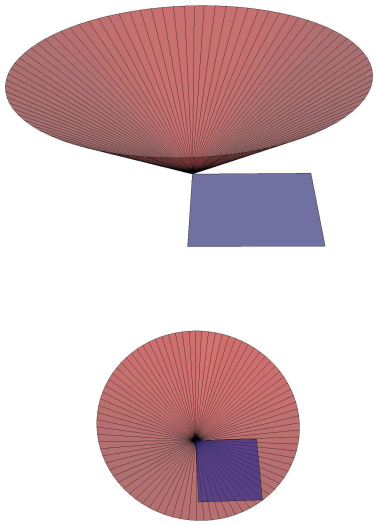


Figure 3.20
Graph of the distance function for $\ell = 2.0$, blue represents picture area

We denote by r the radius of the circle. Let f be defined as $f : [0,r] \to [0,r], x \mapsto (r^\ell - x^\ell)^{1/\ell}$. Thus f describes the border of the quater circle in the first quadrant. We denote by $g : [0,r] \to [0,r], x \mapsto ((r-\varepsilon)^\ell - x^\ell)^{1/\ell}$ a smaller quarter circle with radial distance ε to f.

The radius r depends on the picture size, as the cone has to cover the entire picture. As before, in the euclidean case, we assume the picture to be a square of size $p \times p$ pixels. Then r has to be at least $2^{1/\ell}p$. To compute the polygonal chain, we proceed as follow.

We start at position $(x_a, f(x_a))$. We look for the the point $(x_e, f(x_e))$, that maximizes the length of straight line segment starting at $(x_a, f(x_a))$. For that reason we compute the tangential intersection point $(tip, g(tip))$. The slope of the straight line through $(x_a, f(x_a))$ and $(x_e, f(x_e))$ is just the derivative $g'(tip)$.

This computation is accomplished deploying the maple™ procedure shown below. We use a pseudo maple™ code to ease the readability. As soon as we know about tip we can compute x_e, and restart the

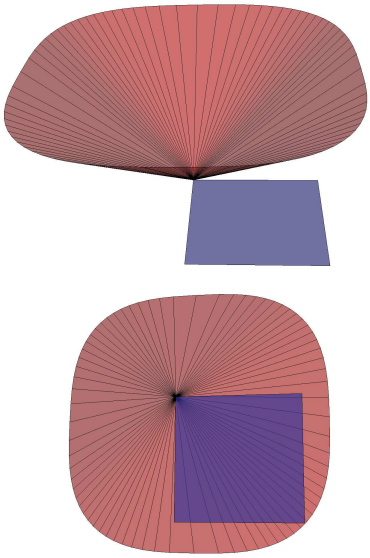


Figure 3.21
Graph of the distance function for $\ell = 3.0$, blue represents picture area

computation with x_e as the new value for x_a. The initial value for x_a is r. We terminate the computation if $x_e \leq 0$, since then the remaining value can be retrieved by reflection on the x and y axis.

```
proc(ℓ)   %polygonal-chain
      ε  := 1
      p  := 1000
      r  := 2^(1/ℓ) * p
      f  := (r^ℓ − |x|^ℓ)^(1/ℓ)
      g  := ((r − ε)^ℓ − |x|^ℓ)^(1/ℓ)
      dg :=diff(g, x)
   slope:= (g(x1) − f(x0))/(x1 − x0) − dg(x1)
      x_e := r
      repeat
         x_a    := x_e
         tip    :=fsolve(slope(x_a), x1, 0..x_a)
         tang   := dg(tip) * (xx − x_a) + f(x_a) − f(xx)
         x_e    :=fsolve(tang, xx, 0..tip)
      until x_e ≤ 0
end proc
```

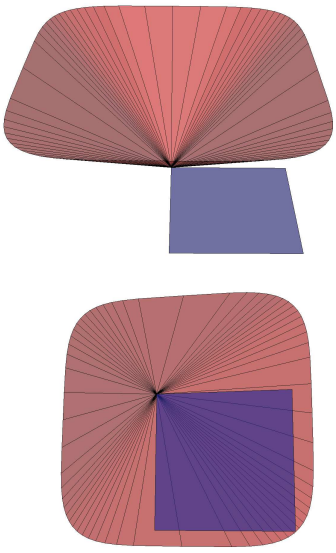


Figure 3.22
Graph of the distance function for $\ell = 5.0$, blue represents picture area

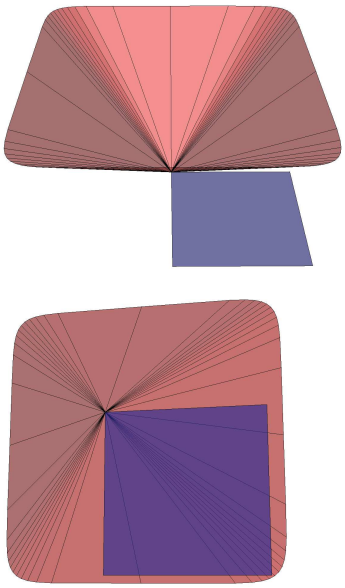
The radii and the number of vertices required for an $\varepsilon = 1$ approximation for some exemplary L_ℓ norms are listed in the table below. As mentioned in the beginning of the section, these values refer to an ε distance diagram.

L_ℓ norm	1.0	1.2	1.5	1.8	2.0	3.0	4.0	5.0	10.0
vertices	4	60	84	88	84	76	68	64	48
radius	2000	1782	1587	1470	1414	1260	1189	1149	1072

The error caused by the approximation

In contrast to the euclidean norm, the cones don't have in general constant slopes. Nonetheless, our argument for the error estimation (cf. page 34) still works. It relies only on the fact that the height, at which a perpendicular ray on top of a point p intersects the cone, is related to the distance between p and the apex of the cone by an increasing function.

3.6.2 The case of $\ell < 1$



We now examine what happens, if we allow ℓ to lie between 0 and 1.

Accordingly, the distance function $\text{dist}_\ell : \mathbb{R}^2 \to \mathbb{R}$ under investigation is:

$$\text{dist}_\ell(u) = (|u_1|^\ell + |u_2|^\ell)^{1/\ell} \qquad 0 < \ell < 1,\ u \in \mathbb{R}^2.$$

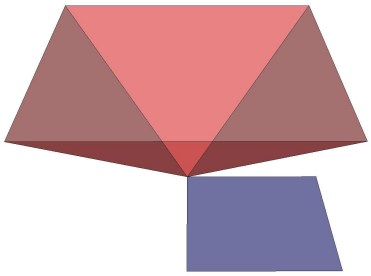
As before, the graph of the distance function is a cone with apex in $\mathbf{0}$ and the base of the cone is denoted by $\mathcal{B}$. Contrary to the first case, the distance functions are no longer founded on norms, as the triangle inequality is violated. This can easily be verified since for $0 < \ell < 1$

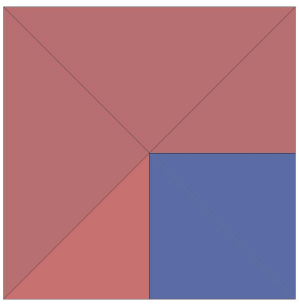
Figure 3.23
Graph of the distance function for $\ell = 10.0$, blue represents picture area

$$\text{dist}_\ell\left(\binom{0}{1} + \binom{1}{0}\right) = 2^{1/\ell} \quad > 2$$

and

$$\text{dist}_\ell\binom{0}{1} + \text{dist}_\ell\binom{1}{0} \quad = 2.$$





Approximating the graph of the distance function

This result can also be derived by analyzing the shape of B. We denote by f the function $f : [0, r] \to [0, r], x \mapsto (r^\ell - x^\ell)^{1/\ell}$. For $0 < \ell < 1$ the function f is concave. To determine the appropriate ε approximation, we proceed as in the convex case. We denote by $g : [0, r] \to [0, r], x \mapsto ((r - \varepsilon)^\ell - x^\ell)^{1/\ell}$. The only difference is that the polygonal chain starts at $(x_a, g(x_a))$ and we search for a tangential intersection point at f (see figure 3.25).

Figure 3.24
Graph of the distance function for $\ell = \infty$, blue represents picture area

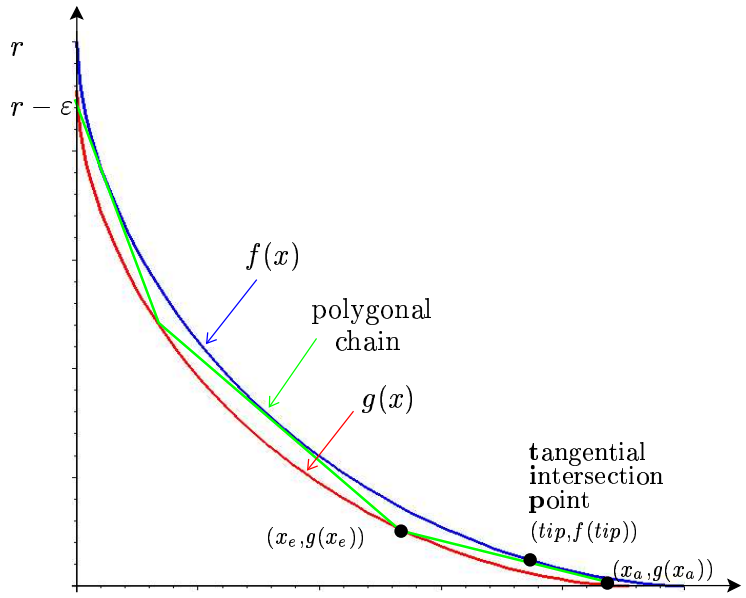


Figure 3.25
Construction of the polygonal chain

Compared to the case for $\ell \geq 1$, we need notably more triangles to guarantee a sufficiently good approximation, as listed in the table below.

L_ℓ norm	0.5	0.8	1.0	2.0
vertices	392	120	4	84
radius	4000	2378	2000	1414

The concavity of the base describing function is reflected in the fact, that in general, the resulting Voronoi diagrams are no longer connected.

Examples of Voronoi diagrams for different values of ℓ are given below. Although we present solutions for ε distance diagrams ($\varepsilon = 1$), these diagrams are site exact to allow the reader to get an impression of what such a diagram looks like.

3.6.3 Adapting the speedup method

In theorem 3.2 we deduced a maximal height for the cones in the arrangment. This theorem can be adapted to fit our current needs.

Although the shape of the base of the cones has changed, we still can use the quadtree structure to derive an upper bound on the maximum height of the cones, required to cover the entire picture.

Assume the apex of an appropriate cone truncated at height h is positioned at one of the vertices of a square. If the othogonal projection of the cone covers the diagonal opposing vertex, then the entire square is covered. Thus, the height h of the cone is sufficient large.

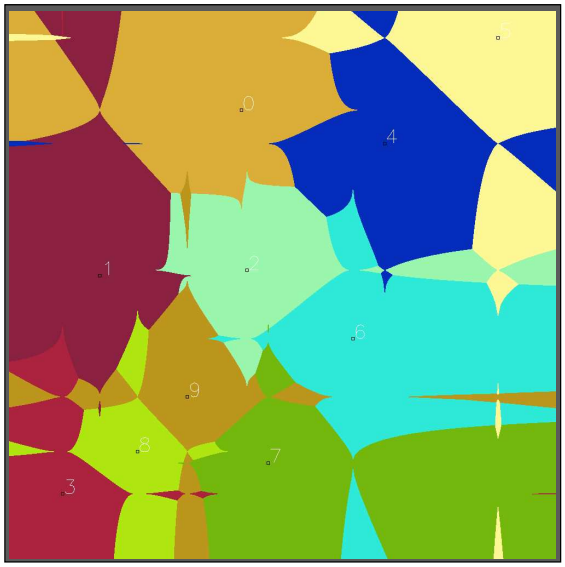


Figure 3.26
Voronoi diagram for $\ell = 0.5$

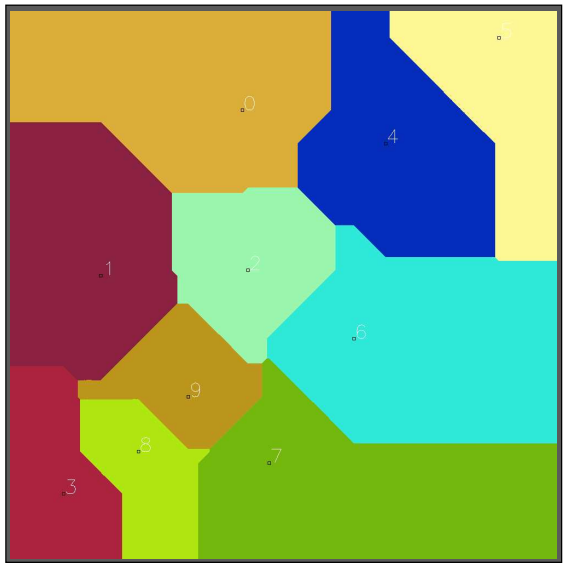


Figure 3.27
Voronoi diagram for $\ell = 1.0$

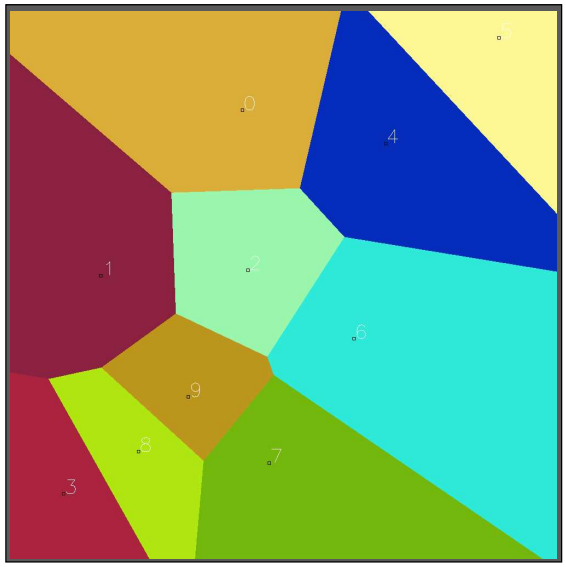


Figure 3.28
Voronoi diagram for $\ell = 2.0$

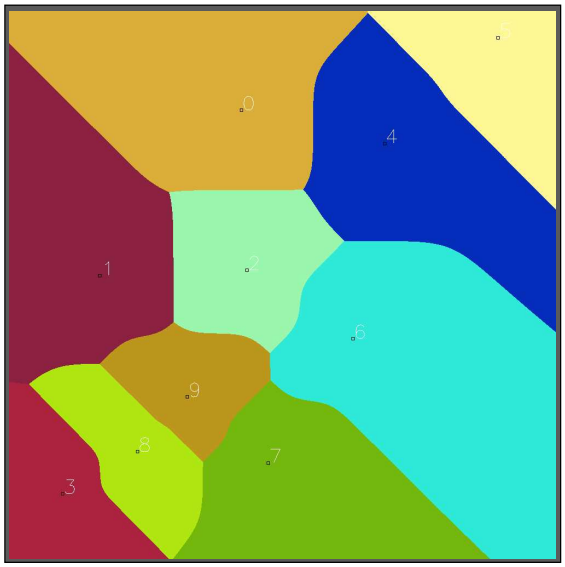


Figure 3.29
Voronoi diagram for $\ell = 6.0$

3.6.4 Areas of equidistant points

Although the max-norm and the Manhattan-norm, are mathematically well defined, we run into trouble, if we try to construct a Voronoi diagram based on either of these norms.

Regarding *ordinary* norms, equidistant points reside on a (1-dimensional) line. In case of the max-norm and the Manhattan-norm this equidistant zone can extent to be a 2-dimensional area of equidistant points.

The max-norm is defined as follows:

$$|| u ||_\infty = \max(|u_1|, |u_2|) \ , \quad u \in \mathbb{R}^2.$$

The above mentioned effect is best illustrated by the following example. Assume the input to built a Voronoi diagram for is made up of two sites s_1 and s_2. Both sides lie on a line parallel to the x-axis, i.e. have same y-coordinate. The shaded area in figure 3.30 is the union of points which are as far away from s_1 as from s_2.

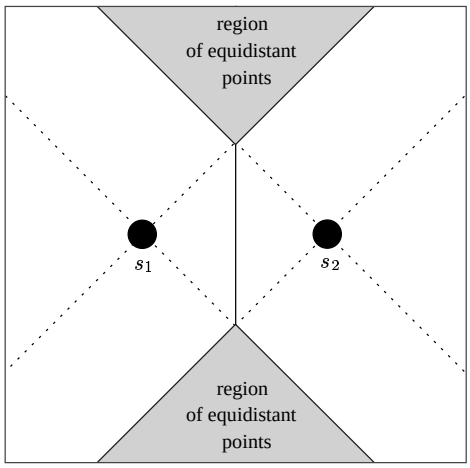


Figure 3.30
Equidistant area as an peculiarity of the max-norm

Similar holds for the Manhattan-norm, defined as

$$|| u ||_1 = |u_1| + |u_2| \ , \quad u \in \mathbb{R}^2.$$

In this case, the effect appears if the sites lie on a diagonal line.

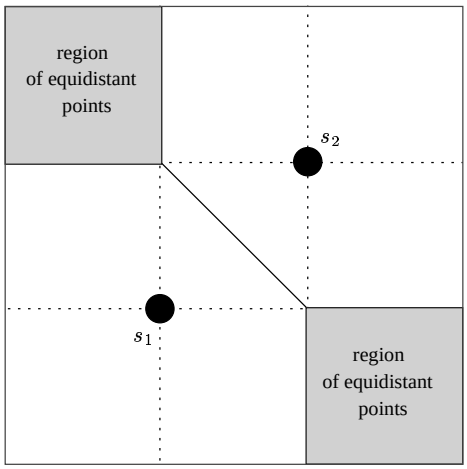


Figure 3.31
Equidistant areas for the Manhattan-norm

With the use of blending, we can reveal these regions. The idea is to assign site s_i the color 2^i, such that each pixel can identify all of its nearest sites.

The Voronoi diagram based on either norm is computed in two passes. In the first pass, we proceed as before.

We built a cone the apex of which is located at each site's position, whereas the color of the cone is adjusted appropriately. The cone for the max-norm consists of just four triangles (see figure 3.24). The cone for the Manhattan-norm is similar simple. It is that of max-norm, rotated by $\pi/4$ (see figure 3.16). Based on the polygonal shape of the base of either cone, our diagram does not suffer from an approximation error as in the euclidean case.

As before, we extract the lower envelope by setting the compare function for the depth buffer to '$\leq$'. At the end of the first pass we clear the color buffers to be all zero.

In the second pass, we rerender the entire scene with the depth buffer compare function set to equality. Additionally we enable blending, such that the color values of successful fragments are added to these of the yet existing pixels.

Figure 3.32 is an example of a max-norm based Voronoi diagram with 10 sites. The pure white area in the middle of the picture represents the set of points which are equidistant to all 10 sites.

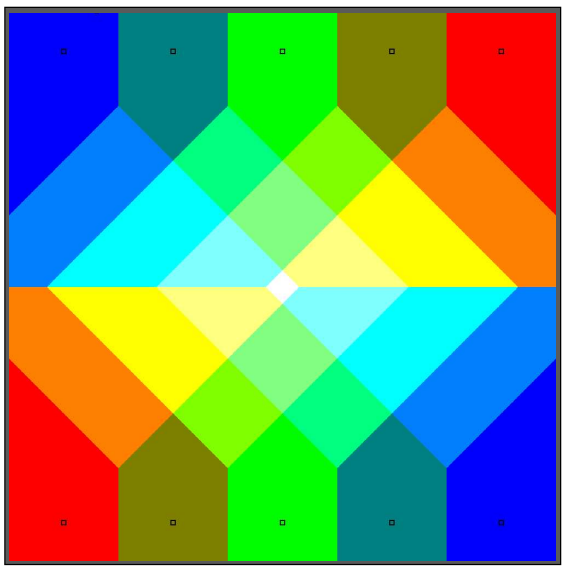


Figure 3.32
Max-norm based Voronoi diagram

A similar example for the Manhattan-norm is shown in figure 3.33.

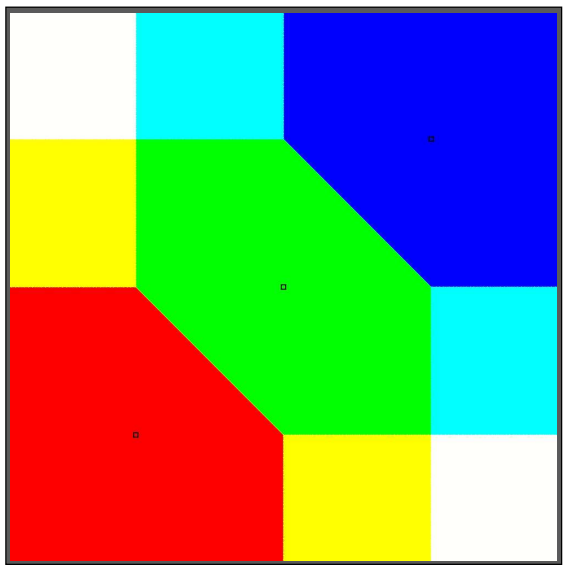


Figure 3.33
Manhattan-norm based Voronoi diagram

3.6.5 Polyhedral distance function

Boissonnat et al. [BSTY98] investigate Voronoi diagrams in higher dimensions under certain polyhedral distance functions. Polyhedral distance function are distance functions for which the base of the cone is a polygon and as such they were meant to be intergrated in our approach. Diagrams based on these distance functions can easily and acurately be computed provided the corresponding polygon is star–shaped.

3.7 Higher order Voronoi diagrams

The ordinary Voronoi diagram is defined to be a partition of the plane into regions according to the closest site. Higher order Voronoi diagrams are a generalization in the sense that the points are assigned to regions according to the k closest site, respectively the set of k closest sites, for some $1 \leq k \leq n-1$. There are two closely related kinds of higher order Voronoi diagrams.

In the sequel of this section, we allow any L_ℓ as underlying distance function for $\ell > 0$.

A *degree k* Voronoi diagram for $1 \leq k \leq n$ of a set S of n sites is defined to be a division of the plane into regions according to n distance functions $f_s : \mathbb{R}^2 \to \mathbb{R}$. Each region R_s consists of all points $x \in \mathbb{R}^2$, such that $f_s(x)$ has rank k in the ordered sequence $[f_t(x)|t \in S]$. Hence, a point p is assigned to the region R_s if s is the kth nearest site. The *degree k* Voronoi diagram is also called the *kth nearest* Voronoi diagram.

In the *order k* Voronoi diagram of a set S a region contains all points which have the same set of k nearest sites. Thus there are $\binom{n}{k}$ possibly empty regions. For any subset $U \subset S$ of cardinality k the corresponding region contains all points p such that U is the set of the k nearest sites.

Properties

Both the degree 1 and the order 1 diagram are just the well known Voronoi diagram. The degree n diagram is the same as the order $n-1$ diagram. It is called *furthest point Voronoi diagram* and is of its own special interest. In general, the regions of the degree k diagram are not connected any more in contrast to the regions of the order k diagram (if the euclidean norm is the distance function under consideration). Compared to the Voronoi diagram, the higher order diagrams are deployed for answering queries about the k nearest neighbors. Besides this, they are part of many different algorithmic solutions (cf. [OBS92]).

Computing the degree k diagram

In the computation of the Voronoi diagram, we position a cone on each point site. For each point $(x, y, 0)$ the height of a cone at position (x, y) is the distance between the point $(x, y, 0)$ and the apex of the cone, respectively the corresponding site. Besides translation, all cones are equal. Consequently, the height of the lower envelope of the arrangement gives us the correct minimal distance for each point.

This idea can be reused to compute the second nearest site for each point. Namely, we cut off the lower envelope from the arrangement and consider the *new* lower envelope of the remaining part. Orthogonal projection of this second lower envelope gives us

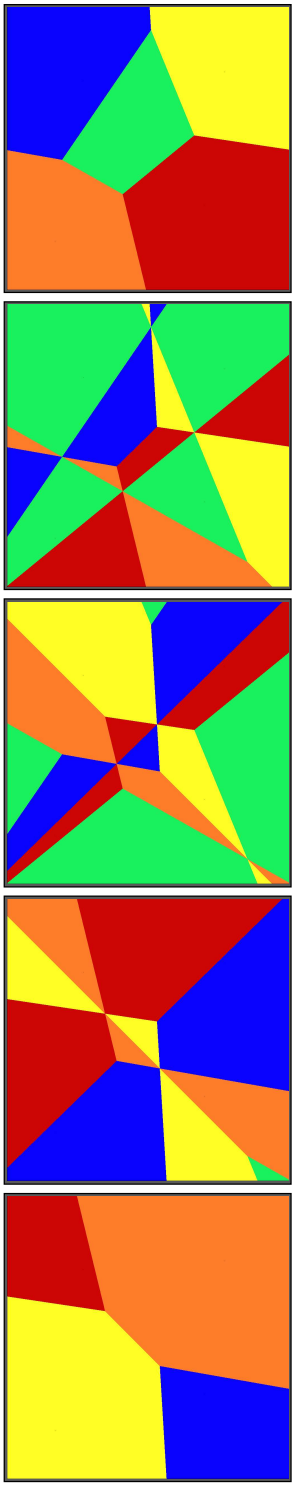


Figure 3.34

All degree k diagrams of 5 sites with underlying L_2 distance function

the Voronoi diagram of degree 2.

This can be extended to compute the kth nearest Voronoi diagram. For each point, we cut off the $k-1$ lowest values out of the arrangement. The lower envelope of the remaining part gives us the degree k Voronoi diagram. (cf. figure 3.35).

This procedure is reflected in our pixel based implementation, where we use both depth buffers, z and z' in combination.

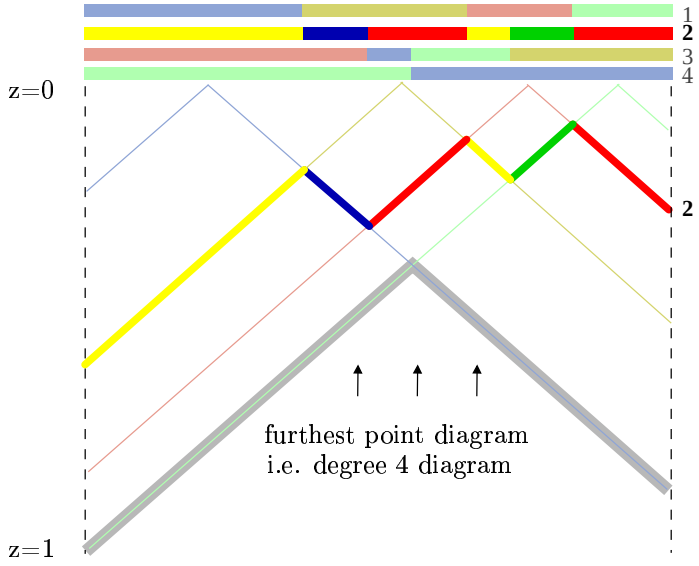


Figure 3.35
Degree k Voronoi diagrams, the bold lines represent the degree 2 diagram, the shaded lines represent the furthest point diagram

In a first pass, we compute the minimal distance for each pixel rendering the arrangement, cone by cone. A fragment is allowed to make an update, if its depth value is less than the current value stored in the z buffer of the corresponding pixel. Eventually, each pixel knows about the distance to its nearest site.

In a second pass, we redo the rendering of the cones. Before that, we set the z buffer to be write protected and change the depth test compare functions such that only those fragments are allowed to pass the depth test, if their depth value is greater than the value of the z buffer and less or equal than that of the z' buffer. At the end of the second pass the degree 2 diagram is computed and the z' buffer of each pixel contains the appropriate distance.

Repeating this procedure $k-1$ times eventually results in the degree k Voronoi diagram, provided we interchange the functionality of the z and the z' buffers for each iteration.

Equidistance treatment

In principle, we are done. Unfortunately, under certain circumstances, this simple and nice procedure fails to compute the correct kth nearest point Voronoi diagram.

If a pixel is equidistant to two or more sites, then this distance will only be noticed once, although it appears as multiple distance with regards to the pixel.

As each cone is rendered independent of each other, we store the *multiplicity* of a distance for each pixel in its stencil buffer. We maintain the following invariant.

Provided we have just terminated the computation of ith nearest Voronoi diagram, then we do not only know about the current distance for each pixel, but also the number the remaining multiplicity of this distance with regard to the pixel, i.e. the biggest number j, such for the $(i+j)$th nearest Voronoi diagram this distance remains the same with regard to the appropriate pixel.

The key to this adjusted procedure is the use of the stencil buffer to store the remaining multiplicity. To compute the degree k diagram, we execute k times the following algorithm, which splits up into four steps. We assume the stencil buffers initially to be zero.

cone by cone	
if	*stencil == 0* *depth > z* *depth < z'*
then	*z', color update*

1. Step: *Register the correct distance*
Cone by cone is pushed through the graphics pipeline, and each generated fragment has to pass several tests, before it eventually is allowed to make an update. At first, the stencil buffer of the corresponding pixel is tested. If it is greater than zero, then the current z buffer value remains correct. Nothing is to be done. Otherwise, the same depth buffer tests as before are applied.

picture wide rectangle	
if	*stencil > 0*
then	$z := \infty$

2. Step: *Preserve current multiplicity counter*
In order to prevent an unwanted overwriting of current stencil buffer values, and thus a change in the number of remaining multiplicity of the present depth value, we render a picture wide rectangle, changing the z buffer values to ∞ for each pixel which has a stencil buffer value greater than 0.

cone by cone	
if	*depth > z* *depth == z'*
then	*stencil ++*

3. Step: *Count the multiplicity of newly determined distances*
Again each cone is rendered. But now, a fragments depth value must be greater than z and equal to z'. If both tests are passed, then the stencil buffer is incremented by 1. Of course, pixels affected by the second step, are now out of question.

picture wide rectangle	
if	*stencil >= 0*
then	*stencil − −*

4. Step: *Reestablish the invariant*
As in the second step, we render a picture wide rectangle. But now, we decrement the stencil buffers of all pixels by 1, accounting the fact that the current distance value is once used for the present diagram.

   Since we interchange the functionality of the two depth buffers, z and z', there is no need to change any of these buffers.

Running time

Compared to the total running time, the time spend on the two picture wide rectangles per iteration is negligible. Thus, the running time of the complete algorithm is $2k$ times the time required for the computation of the ordinary Voronoi diagram. A further reduction of the running time can be achieved exploiting the speedup idea of section 3.5, appropriately.

In theorem 3.2 we deduced a maximal required height for the cones in the arrangment. This theorem, appropriately adapted, can also be applied, if the underlying distance function is L_ℓ for $\ell > 0$ (see

subsection 3.6.3). For the computation of the degree k Voronoi diagram, we can make use of a similar theorem.

In case of the degree k Voronoi diagram, we have to ensure that each pixel is covered by the cones corresponding to its k nearest sites. Theorem 3.3 guarantees us, that the result of the computation of the degree k Voronoi diagram remains the same, if we execute the algorithm with appropriately truncated cones.

Theorem 3.3 *Let S be a set of points in the plane and $B \subset \mathbb{R}^2$ be a bounding area, containing S. Assume, there is a covering of B into rectangular cells such that each cell contains at least k points of S. Let e denote the largest diagonal with regards to all cells. Let c be a right circular cone as described in section 3.2. Let c_t be a the cone c truncated at height e. Let $\mathfrak{A}$ be the arrangement of cones indentical to c, set on top of each point site, and $\mathfrak{A}_t$ be the same arrangement but with cones indentical to c_t. Let l be a line perpendicular to B. Then the set of the first (lowest) k intersections of l with $\mathfrak{A}$ is the same as the set of the first (lowest) k intersecions of l with $\mathfrak{A}_t$.*

Proof: Fix a cell in the covering. Any truncated cone corresponding to a point $s \in S$ covers the entire cell provided s is inside the cell. Thus, there are least k intersections for any line l perpendicular to the plane with regard to these cones. A lower intersection point of l with a cone regarding the original arrangement remains unaffected by the truncation, since the intersection occurs at a height less than e. □

Thus we have to adapt the quadtree structure mentioned in section 3.5 to count the number of sites inside a cell. The lowest level of the tree such that all nodes are marked for at least k times, determines the maximum required height of the cones.

Analogous to the analysis in section 3.5, the time spent on the construction and processing of the quadtree depends directly on the number of nodes visited, and can be bounded as follows. Each site is visited once and each node is visited k times in the worst case.

Compared to the ordinary Voronoi diagram, we present timings

for the computation of the order 10 (distance) diagram. Again, the sites are chosen uniformly at random from $\{0, \ldots, 1023\}$. As before, the use of the quadtree cause a sizable reduction in the running time.

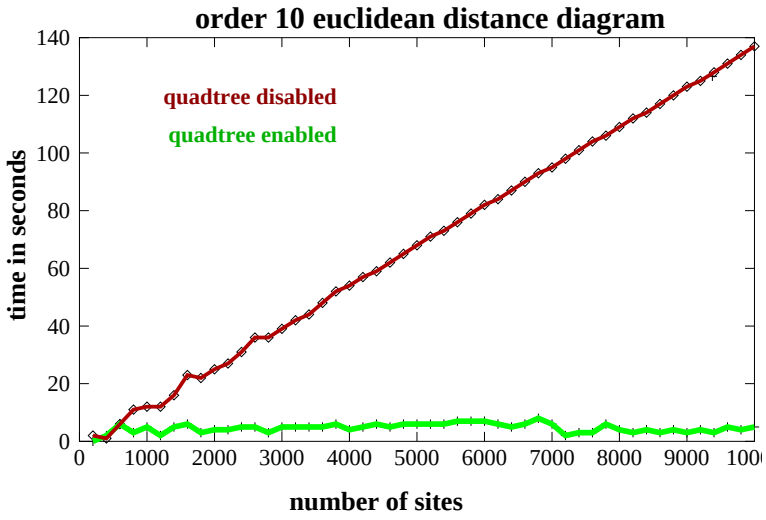


Figure 3.36

Order 10 euclidean distance diagram with and without quadtree

Computing the order k diagram

Our algorithm for the order k diagram is based on the following observation. Assume we have just computed the degree k diagram, such that the z buffer values resemble the desired distances. Then, we render the scene of the cones once more, but now we allow a fragment to pass the depth test if its depth value is at most z. In order to prevent a change of the z buffer, we enable the write protection for it. Additionally, we enable blending to collect all fragments passing the depth test.

Furthermore, assume that each site $s_i \in S$ is assigned the color value zero but the ith bit set to 1. Then the color value of a pixel p is 1 at position i if and only if there are at most $k-1$ sites nearer to p than site s_i.

3.7.1 Furthest point Voronoi diagram

Regarding the furthest point Voronoi diagram, the plane is partitioned in regions according to the furthest sites. We just saw, that it can be computed by peeling all but the last lower envelopes. In doing so, we needed an $n-1$ pass algorithm. But this can also be done in a single pass. Instead of *looking* from above. i.e. from $z = 0$ towards negative z direction, we change the viewing direction to look upwards from $z = -1$ to $z = 0$. The same effect can also be achieved by turning upside down the cones. The result remains the same. Now, the lower envelope is the furthest point Voronoi diagram, and as this, computable in a single pass.

3.8 Weighted Voronoi diagrams

Computing the Voronoi diagram of points by rendering a three dimensional scene of cones gives us the ability to change the distance function in a quite arbitrarily fashion. The fundamental concept remains invariant, only the shape of the cones alters.

Up to now, all sites are treated equally. They only differ in their position. Besides this, it might be desirable to assign different weights to the sites.

Subsequently, we present the multiplicatively and the additively weighted Voronoi diagram. A more detailed and comprehensive description can be found in [OBS92].

3.8.1 Multiplicatively weighted Voronoi diagrams

Let $\mathrm{dist}_s : \mathbb{R}^2 \to \mathbb{R}$ be one of the so far discussed distance function sets. The *multiplicatively weighted Voronoi diagram* (cf. [OBS92]) of a set of sites $\{s_i \mid 1 \leq i \leq n\}$ with associated weights $\{w_i \mid 1 \leq i \leq n, w_i > 0\}$ is characterized by the weighted distance given by

$$\mathrm{dmw}_{s_i}(x) = \frac{1}{w_i}\,\mathrm{dist}_{s_i}(x).$$

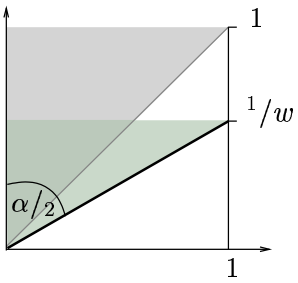


Figure 3.37
Computing the apex angle α of a *weighted* cone

Euclidean norm

In order to ease the description we consider at first only the euclidean case. In the unweighted (or equal weighted) case, all the cones have the same apex angle. This degree of freedom can now be used to realize different weights for each site.

Our goal is to maintain the one-to-one correspondence between the height of a cone at a point $p = (x, y, 0)$ and the distance between p and the apex of the cone. As a consequence, we choose the apex angle to be π for a cone the corresponding site of which has an associated weight of 1. From that we can conclude that the apex angle α_w corresponding to a weight w is (cf. figure 3.37)

$$\alpha_w = 2 \cdot \arctan w.$$

The pixel based implementation remains the same as before. The only difference is that the appropriate cone has to be selected from a set of cones, each realized as a display list of its own. Each cone could also be computed separately, with regard to the proper weight. Moreover, we can make use of the scaling functionality available in OpenGL. This is the more interesting as the cones could also be replaced by a depth texture scaled appropriately.

Even if we use the ordinary euclidean distance as underlying distance function, we can observe some surprising properties of the diagram (cf. [OBS92]).

- The regions need not to be connected, they may have holes.
- If the maximal weight is uniquely associated to one site, then its region is the only infinite one.
- An edge is a straight line if the weights of the neighboring regions are the same. Otherwise the edge is a circular arc (cf. lemma 3.1).

An illustration of an weighted Voronoi diagram with underlying euclidean distance is given below. The white numbers represent the weights associated to the sites at that position.

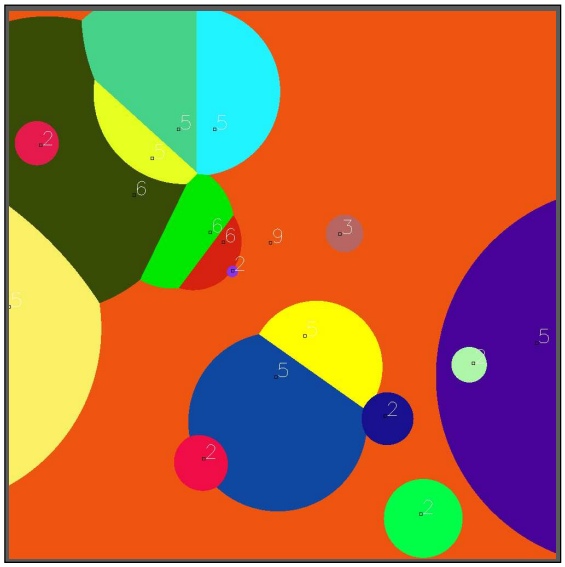


Figure 3.38
Multiplicatively weight Voronoi diagram with underlying euclidean distance. The white number represents the weight of a site.

Generalized distance functions

Weighted distance diagrams based on any of the so far discussed distance functions can easily be realized using the scaling functionality available in OpenGL.

3.8.2 Additively weighted Voronoi diagrams

In contrast to the former diagram, we can also combine the weights additively. As before, we denote by $\{s_i \mid 1 \leq i \leq n\}$ the set of sites and by $\{w_i \mid 1 \leq i \leq n, w_i \geq 0\}$ the set of associated weights. Let dist denote an already known distance. The *additively weighted Voronoi diagram* is characterized by the distance

$$d_{aw}(x, s_i) = \text{dist}(x, s_i) + w_i.$$

Up to now, the rendered scene consists of cones the apex of which resides on the plane $z = 0$. To realize a additive weight of w_i, we translate the position of the apex of each cone from $(x, y, 0)$ to (x, y, w_i) (cf. figure 3.39).

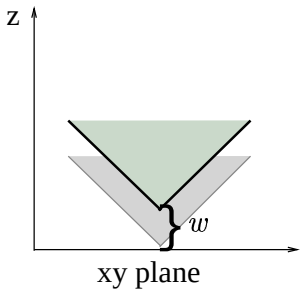


Figure 3.39
An additive weight w causes a translation of the cone in z direction

Compound weight Voronoi diagram

The *compound weight Voronoi diagram* is a composition of the multiplicatively and the additively diagram. Each site s_i has an associated pair of weights (m_i, a_i), such that the distance is measured as

$$d_{cw}(x, s_i) = \frac{1}{m_i} \operatorname{dist}(x, s_i) + a_i.$$

Since both, the apex angle (i.e. the scaling factor) and the z coordinate of the apex of a cone are independent from each other, the pixel based implementation is a straightforward combination of the last two algorithms.

Running time

Actually, the algorithm to compute the weighted Voronoi diagram remains the same as computing the unweighted. The only difference is that we have to adopt the speedup method, too, comprising the multiplicative and/or additive weights. For that, we assume for all cone the worst case, i.e. the smallest multiplicative respectively biggest additive weight.

3.9 Voronoi diagrams of a set of line segments and circular arcs

Up to now, we restrict the generalization of Voronoi diagrams to point sites. In this section, we inspect what happens if we allow line segments and circular arcs as sites.

Figure 3.40
Distance graph of a line segment site

Line segments

Considering the euclidean space, the shortest distance between a line site l and a point site p is determined by a line segment which starts at p and is perpendicular to l. As a result, the distance function graph for a line is a saddle roof on top of the line, whereas the apex angle is that of a point site of equal weight.

Almost the same holds for the distance function graph of a line segment site. It is combined out of a saddle roof and two halfcones matching the endpoints of the segment (see figure 3.40). For a pixel based implementation, the only thing to take care of is that all three parts, the saddle roof and the two cones, get the same color, as they belong to the same site.

This idea could also be exploited to compute the Voronoi diagram based on arbitrary distance function L_ℓ for $0 < \ell \leq \infty$. A change in the distance function cause the base of the corresponding cone to be rather arbitrarily shaped. Nevertheless, we can still compute the graphical representation of the graph of the distance function.

Let C be the cone representing the distance function graph for a point site. Let P be the (star-shaped) polygon, defining the base of C. We denote by e_s and e_t the two endpoints of the line segment under consideration. Then, we position a copy of C at both endpoints e_s and e_t. Furthermore, we render a rectangle for each vertex v of P. The corresponding 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 3.41).

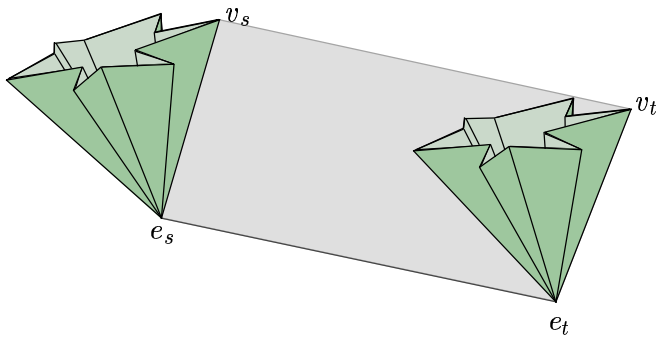


Figure 3.41
Line segment computation for arbitrary distance function L_ℓ

This idea could also be exploited to compute the Voronoi diagram of areas which can be described by a polygonal chain.

Circles

For circles it is even easier. The shortest distance between a point and a circle is determined by a line radial through p. This remains true for ℓ chosen arbitrary, $0 < \ell \leq \infty$.

Let M be the center of a circle C and r its radius. The distance function graph for C is a combination of a cone and cone frustum, residing in the half space $z > 0$. The cone is positioned such that its apex has the same x and y coordinates as M and its base coincides with C. The smaller base of the cone frustum also coincides with C, such that the combination of both resembles a straw hut (see figure 3.42).

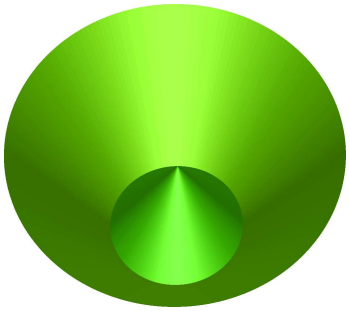


Figure 3.42
Distance graph of a circle site

Circular arcs

The just presented idea to realize circle sites can be extended such that circular arc sites can be admitted, too, provided the distance function is the euclidean norm.

Let e_s and e_t be the two endpoints of the circular arc under consideration. As before, we construct a *straw hut*. But now, we slice it radially at e_s and e_t and remove this piece, such that the removed part resembles a piece of cake. Additionally, we set two *ordinary* cones at both endpoints e_s and e_t. In figure 3.43 we present such a distance graph. For the sake of simplicity, we omit the two cones at the endpoints.

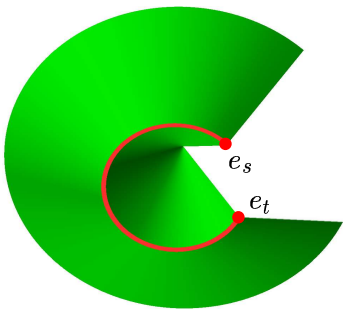


Figure 3.43
Distance graph of a (red) circular arc site (to ease perception, without the two *ordinary* cones at e_s and e_t)

To prove our assumption consider the distance between a point p and an arc, as sketched in figure 3.44.

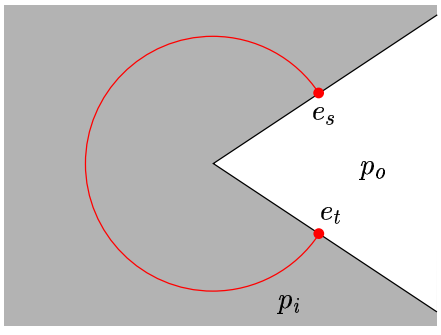


Figure 3.44
Distance between a point and and circular arc

If p is outside the shaded area, e.g. $p = p_o$, then its nearest distance to the arc is determined by either point e_s or e_t. Otherwise, if p is inside the shaded area, then neither e_s nor e_t will be the nearest arc point for p.

Below is an Voronoi diagram of a set of points, line segments and circles, based on the euclidean distance.

Figure 3.45
Voronoi diagram of a set of points, line segments and circular arcs based on the L_2 norm

3.10 Yet another fast food in town

In the last sections we presented a fast implementation of Voronoi diagrams exploiting the power of graphic adapters. Now, we turn our focus to a closely related problem.

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 in. Most naturally, the area to attract customers 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 on that point, that when inserted into the existing diagram as a new site, will have the biggest region. More precisely,

the problem states as follows:

Definition 3.4 *Given a set S of n sites within a bounding box 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 has maximum area.*

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

3.10.1 Previous work

Cheong et al [CHLM02] describe a related problem. Consider the following game, in which a first player chooses an n-point set $\mathcal{R}$ inside a square $\mathcal{Q}$. Thereafter, a second player places another n-point set $\mathcal{B}$ inside $\mathcal{Q}$. The payoff for the second player is the fraction of the area of $\mathcal{Q}$ occupied by the regions of $\mathcal{B}$ in the Voronoi diagram of $\mathcal{R} \cup \mathcal{B}$. Cheong et al [CHLM02] 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 the convex case, Dehne et al [DKS02] describe a method to maximize a Voronoi region. But up to now, we are not aware of any previous solution delivering the settle point in a general setting.

3.10.2 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 edges of the Voronoi diagram nor with one of the furthest point Voronoi diagram.

Figure 3.46 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 the rest of the points the brightness of the color of a point represents its

potential area, i.e. the area of the corresponding region if the point would be included in the diagram.

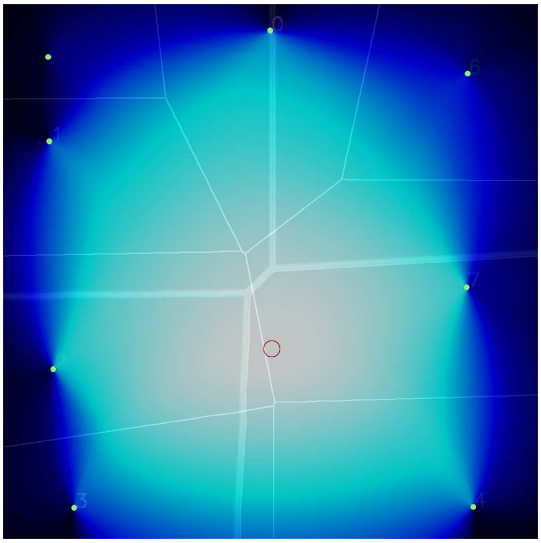


Figure 3.46
Voronoi and furthest point Voronoi diagram of eight sides with superposed settle point computation

We can derive another property from the following observation. Consider a point p at distance d from its site. If p changes its affiliation as a new site t is inserted, then the distance between t and p is at most d. That means, that the width of the Voronoi diagram is monotonically decreasing if new sites are inserted.

Nevertheless, the area of the biggest region can grow although a new site enters the diagram. Consider the following example. Regarding the euclidean space, place n sites on the greatest possible circle such that the position for each site coincide with a vertex of a regular n-gon. The circle is chosen to fit just inside a square of with edge length d.

If we place a new site at the center of the circle, then its corresponding area will be at least $\pi/16\,d^2$, independent of n. Provided the first n sites are equidistant on the border of the circle, then the biggest region will always be less than $(\pi(\sqrt{2}/2\,d)^2 - \pi(1/4\,d)^2)/n$. Thus, for $n > 7$ we have the desired result (cf. figure 3.47).

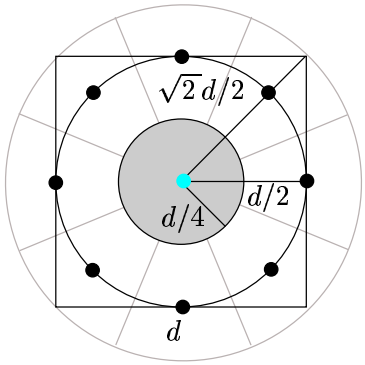


Figure 3.47
Increase in the maximum region

3.10.3 Pixel based adaptation

The easiest method to compute the settle point is the following. 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.

One of the drawbacks is, that there is no counter in OpenGL, which logs the number of successful fragments. Consequently, after each inserted point, the entire picture had to be read back into CPU memory to count the number of newly colored pixels.

Inspired by John F. Kennedys key note "Ask not what your country can do for you, ask what you can do for your country"[1], we investigate for which potential new sites the pixel p under consideration would contribute to their regions (cf. figure 3.48).

For that reason, we draw a circle with radius r around p, such that r equals the distance between p and its corresponding 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.

For each pixel p we do the following:

1. Determine the site s_p in which region p resides.

2. Draw a circle around p with radius r equal the distance between p and s_p.

3. p contributes 1 to the potential area of any pixel inside the circle (e.g. q_{in}). For this purpose, we administer for each pixel an *area counter*[2] accumulating these events.

[1] Inaugural address of president John F. Kennedy, Washington, D.C. January 20, 1961

[2] The area counter is realized deploying the color buffers as explained later.

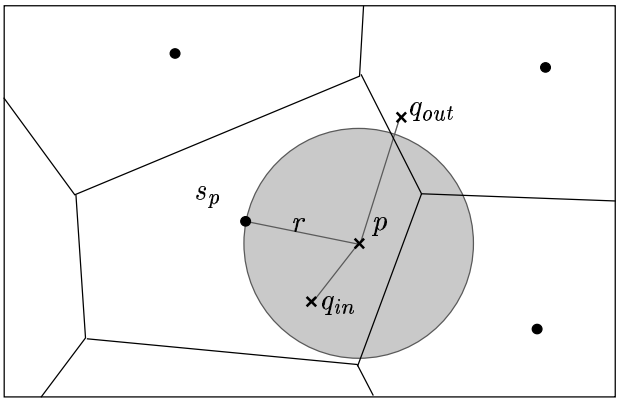


Figure 3.48
Contribution area of p

After all pixels are processed, the area counter of any pixel represents the size of its region, if the pixel were inserted into the existing Voronoi diagram.

Correctness

Let $\mathcal{Q}$ be the pixel grid under consideration, i.e. the picture to be drawn. Let $S = \{s_1, \ldots, s_n\}$ be the set of sites, and dist be the distance function on which the Voronoi computations are based on. To prove the correctness of the above applied method, we define for each $t \in \mathcal{Q}$

$$\text{zone}(t) = \{p | p \in \mathcal{Q} \text{ and } \forall s \in S : \text{dist}(t,p) < \text{dist}(s,p)\}.$$

Then by definition, the settle point is the point with the largest zone. For any pixel p we denote by s_p its nearest site. Let $C(p)$ be the circle with center p and radius $\text{dist}(p, s_p)$. We are done, if we can prove, that for every $p, t \in \mathcal{Q}$:

$$p \in \text{zone}(t) \text{ iff } t \in C(p).$$

$p \in \text{zone}(t)$ implies $dist(t,p) < dist(s_p, p)$ implying $t \in C(p)$, and vice versa. As a consequence it follows that:

$$\text{zone}(t) = \{p | p \in \mathcal{Q} \text{ and } t \in C(p)\}.$$

Sketch of algorithms

At first, we have to determine for each pixel its nearest site in order to draw a circle of the appropriate size. Therefore, we compute the Voronoi diagram of the given set of sites. The depth buffer value of each pixel encodes the distance between the pixel and its corresponding site.

After this step, we know for each pixel the radius of the circle we have to draw around it.

How do we realize the area counter? We abuse the pixels' color and stencil buffers to count the number of covering circles. Whenever a circle overlaps a pixel its stencil buffer increments by 1. When an overflow in the stencil buffer occurs, the fragment is allowed to pass the stencil test. This has the effect, that one of the four color buffers value increases by 1, provided that blending is appropriately enabled.

Drawing color	
red, green, blue, alpha:	[1,0,0,0]
Stencil	
initialize to:	1
compare function:	== 0
actions:	incr_wrap, incr_wrap, inrc_wrap
Blending	
frag factors:	[1,1,1,1]
pixel factors:	[1,1,1,1]

Figure 3.49
OpenGL settings for the settle point computation

This is, why for all pixels we initialize the color buffers with zero and the stencil buffers with 1. We select the blending factors to be all one (cf. blending on page 17). 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[3] 1 and green, blue and alpha equal zero.

Then we draw a circle with radius r around each pixel p with $r = \mathit{distance}(\mathrm{color}(p), \mathrm{position}(p))$. This can be accomplished by either a triangle fan similar to the cone approximation, or by using a texture with inscribed circle (cf. figure 2.2 on page 23).

Whenever a circle overlaps a pixel, its stencil buffer increments. Every 256 times[4] the stencil buffer wraps around, and due to the blending settings, the red color buffer increments by 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 picture and reset the color and stencil

[3] For the sake of simplicity, we interpret the contents of a byte as integer between 0 and 255.

[4] Only the first wrap around happens after 255 hits since the stencil buffer is initialized to 1.

buffers to continue counting. Finally, we add up all intermediate pictures together with the last picture in the CPU's main memory to get the final picture. Thus, for a picture of 1024x1024 pixels, we have to read back the entire picture five times. Any pixel with the greatest value is the requested candidate.

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 disregarded, i.e. no new site can profit from them.

3.10.4 Speedup

The idea to speed up the algorithm is based on the observation that the computation made for a set of pixels repeats exactly for other sets.

To gain a better insight, we trace the computation for a horizontal row of pixels $p_1, \ldots, p_\delta$ all belonging to the site s_p. Pixel p_i is at distance i away from s_p.

In the course of the computation, 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 circles. T is a rectangle sufficiently large to enclose all circles. This is illustrated in figure 3.50. The gray shaded area resembles the computation, respectively the circles, done so far for the pixels $p_1, \ldots, p_\delta$.

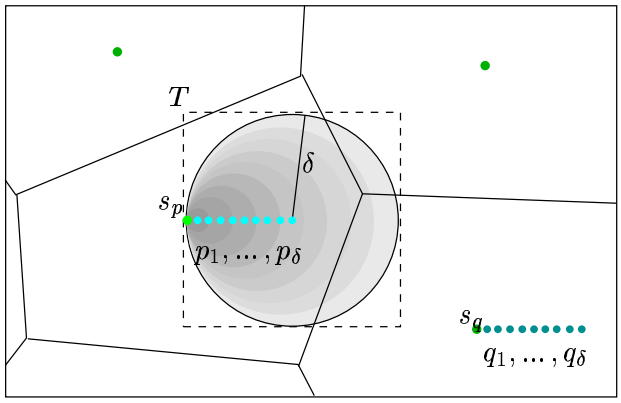


Figure 3.50
Repeated computation

This stamp can now be used in the course of the computation. For any row of pixels $q_1, \ldots, q_\delta$ 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.

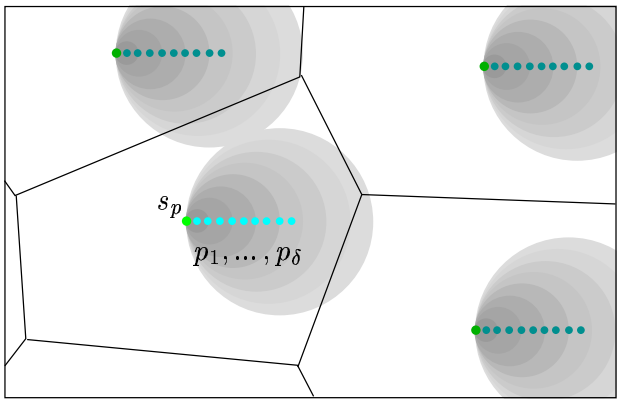


Figure 3.51
Repeated computation

This could be exploited in the following way. We precompute the circles for various groups 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. Hence, for every member of the

group, we draw an appropriate circle. Every pixel triangle yields a corresponding pixel patch including all the influenced pixels of the surrounding (e.g. in figure 3.50 the gray shaded area), such that for each pixel triangle the resulting patch consists of any pixel the area counter of which is greater than zero.

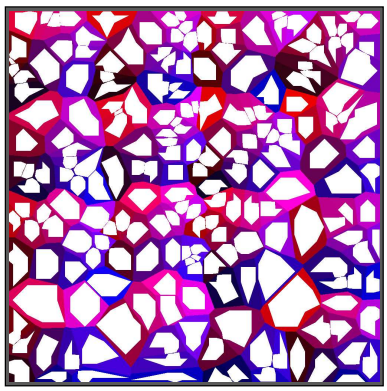


Figure 3.52
Voronoi diagram with blank patches of precomputed areas

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 fit inside the sites region, such that the virtual site's position maps onto the real sites position. Thereafter, any pixels inside the triangles are marked (see figure 3.52). For the rest of the pixels we proceed as before, hence we draw circles of appropriate radius and color around these. At the end of the ordinary algorithm we just add the corresponding patches to the picture in order to get the complete result.

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.

In our implementation, the number of pixels which remain to process is reduced by 60 percent on average exploiting an underlying sample space of about 600 MBytes of precomputed data.

3.10.5 Running time anomaly

Interestingly, there is an apparent anomaly concerning the running time. The greater the input set of sites, the quicker the settle point can be computed. This is because the area and width of the Voronoi regions decrease the more sites share the same bounded area. Circles from points far away from their corresponding site cause a lot more fragments than these points which are near the site (see figure 3.53). Consequently the running time increases the fewer sites are involved. The rendering time varies between 10 seconds for set of about 100 sites and up to four minutes for set of two sites.

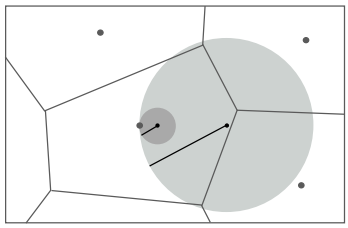


Figure 3.53
Running time anomaly

3.11 Applications

In the last sections we presented pixel based algorithms for a variety of different Voronoi diagrams. This is an interesting task on its own. Besides, Voronoi diagrams are often used as a first step of a more complex algorithmic solution. At a first glance, it seems to be unavoidable to reload the picture into main memory in oder to use the result of the pixel based computation.

To prove the contrary, we present three examples how to benefit from the Voronoi diagram picture in a more complex solution. We start with the computation of the Hausdorff distance between two finite sets of points in the plane.

3.11.1 Hausdorff distance between a red and a blue set of points

The Hausdorff distance owes its name Felix Hausdorff (1868-1942), a mathematician known for his work on topology and set theory. The Hausdorff distance is of important role in the field of image comparison, i.e. pattern matching. It may be described as *the maximum distance of a set to the nearest point in the other set* and vice versa [Rote91].

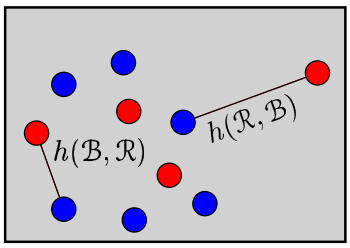


Figure 3.54
Directed Hausdorff distance between the set of red and blue points, and vice versa

More formally, let $\mathcal{B}$ be a set of *blue* points and $\mathcal{R}$ be a set *red* points. The *directed Hausdorff distance* $h(\mathcal{B},\mathcal{R})$ from $\mathcal{B}$ to $\mathcal{R}$ is defined to be the function

$$h(\mathcal{B},\mathcal{R}) = \max_{b\in\mathcal{B}} \min_{r\in\mathcal{R}} ||b-r||.$$

In general, this function is not symmetric (cf. 3.54). Let $b_h \in \mathcal{B}$ the point that is farthest away from any point $r \in \mathcal{R}$, then $h(\mathcal{B},\mathcal{R})$ is the distance from b_H to its nearest neighbor in $\mathcal{R}$.

The *Hausdorff distance* $H(\mathcal{B},\mathcal{R})$ is defined to be

$$H(\mathcal{B},\mathcal{R}) = \max(h(\mathcal{B},\mathcal{R}), h(\mathcal{R},\mathcal{B})).$$

Intuitively, $H(\mathcal{B},\mathcal{R})$ is a measure for the degree of divergence of the two sets. If we place a circle with radius $h(\mathcal{B},\mathcal{R})$ around each $r \in \mathcal{R}$, then any $b \in \mathcal{B}$ lies inside of at least one of these circles, and the same holds vice versa.

We solve the problem of finding the pair of points determining the Hausdorff distance with the aid of Voronoi diagrams.

In a first step we compute the directed Hausdorff distance $h(\mathcal{B}, \mathcal{R})$. For that we build the Voronoi diagram of the red points whereas each red point receives its own unique color. We then inspect for each blue point its depth and color buffer value. The blue point with the maximal depth buffer value and the corresponding red point, encoded in the color value of the blue point, is the pair of points we are looking for.

In a similar way, we determine $h(\mathcal{R}, \mathcal{B}))$ and eventually $H(\mathcal{B}, \mathcal{R})$.

3.11.2 Largest inter point difference of a set of points

The next problem we turn our focus to is the *largest inter point difference of a set of points*, also known as the *diameter of set of points*.

Given a set S of n points in the plane. The *diameter of* S is defined to be the maximal distance between any two points of S:

$$\text{diameter}(S) = \max_{s_1, s_2 \in S} ||s_1 - s_2||.$$

To compute the diameter of S we exploit the following property. Let $p, q \in S$ be the pair of points such that the distance between p and q is the diameter of S. Then both p and q are convex hull points of S (cf. [HY61], [PS95, p. 177], [Lee80], [TB81]).

This leads to the following solution. Compute the furthest point Voronoi diagram of S. The diameter is determined by the point with the largest depth value. Again the color value of this point reveals the missing second point.

Considering the algebraic computation tree model, this method is still optimal, since computing the diameter of set of points in the plane can be used to solve the *Disjointness* problem any solution of which needs $\Omega(n \log n)$ time (cf. [PS95, pp 176-177], [BO83]).

A practical approach for computing the diameter of a point set $S \subset \mathbb{R}^d$ is presented by Har-Peled [Har01] in 2001.

3.11.3 Maximum distance between a red and a blue set of points

As a last example we investigate the *maximum distance* between two sets of points $\mathcal{B}$ and $\mathcal{R}$. It is defined to be:

$$d_{\max}(\mathcal{B},\mathcal{R}) = \max_{b\in\mathcal{B}} \max_{r\in\mathcal{R}} ||b-r||.$$

The maximum distance between two sets of points is closely related to the diameter of the union of the sets.
More precisely, diameter$(\mathcal{B}\cup\mathcal{R}) \geq d_{\max}(\mathcal{B},\mathcal{R})$, since a partitioning of $\{\mathcal{B}\cup\mathcal{R}\}$ into two set only reduces the number of possible pairs of points, i.e. distances, to be considered. Actually, it is quite easy to construct two sets of points such that diameter$(\mathcal{B}\cup\mathcal{R}) > d_{\max}(\mathcal{B},\mathcal{R})$. Figure 3.55 illustrates such an example invented by G. Toussaint et al. [TB81], [BT83], and [TM82].

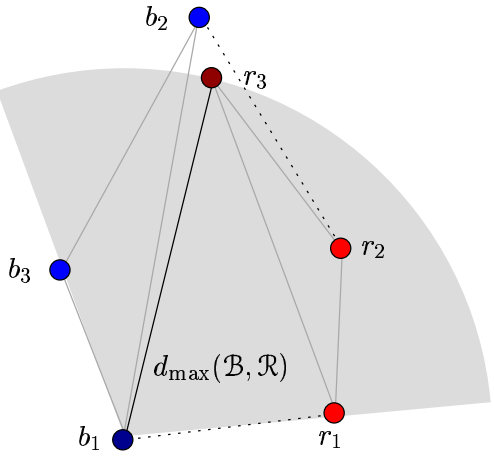


Figure 3.55
An exemplary configuration in which diameter$(\mathcal{B}\cup\mathcal{R})$, given by b_1 and b_2, is greater than $d_{\max}(\mathcal{B},\mathcal{R})$ defined by b_1 and r_3

They were the first to come up with an $(n \log n)$ algorithm. Interestingly, they explicitly abandon the idea to use the furthest point Voronoi diagram to solve this task. And that is just the way, we solve this problem.

Having computed the furthest point Voronoi diagram of the blue points, we look for the red point p_{red} with highest depth buffer

value. The point corresponding to the color of p_{red} and p_{red} are the points defining the maximum distance.

3.12 Pixel based computations executed without a graphic processor

Nowadays graphic adapters are quite cheap and powerful. Anyhow, there are still enough computers not supplied with a OpenGL capable graphic processor, for short GPU. Against this background, we present pixel based algorithms solely relying on the main CPU.

3.12.1 Generic approach for Voronoi diagrams

Our algorithm consists of several cycles. In each cycle we inspect pixels at distance d away from their nearest site, where d is increased in every cycle.

1. In a round robin fashion, we determine for each site s_i the pixel p_i at distance d.
2. If p_i is not yet colored, we assign p_i the color of s_i.
3. Eventually the distance d is increased, such that the next nearest pixel can be inspected.

We start our algorithm with the direct neighbors of the sites and grow circular around the sites. To resolve ambiguities, we inspect the pixels distance in counter clockwise order.

The algorithms has to terminate as soon as all pixels are colored. For this reason, we increase a global counter whenever a pixel is colored, and check this counter as a first step in each cycle.

To find the next nearest pixel, we use a precomputed array containing the positions of all possible pixels sorted by ascending distances. In doing so, the distance function is implicitly given as an order relation. As a consequence, quite arbitrary distance functions can be realized. This is the same array, we have to compute if we use depth textures instead of triangle fans (cf. section 3.4).

Running time

The runnning time is determined by the width w of the Voronoi diagram, i.e. the largest empty circle. As soon as $d \geq w$, all pixels are visited. Thus, the running time is bounded by $O(n * w)$. This resembles the runnning time for the algorithms using the GPU.

3.12.2 Higher order Voronoi diagrams

As in the OpenGL algorithm for computing the order/degree k Voronoi diagram, we can here use a similar idea to draw these diagrams.

We proceed as describe above with the following differences. We store for each pixel the first k colors in an individual array. Compared to the above described algorithm, we increase the global counter only for pixels which are colored k times.

Running time

Any pixel is colored at least k times. For $p \cdot p \cdot k$ is a trivial lower bound assuming a picture of $p \cdot p$ pixels. Similar to analysis in the case of the computation with a GPU, we can bound the time from above using theorem 3.3. Let e be the width of the best possible covering, then the running time of the algorithm can be bounded from above by $O(e \cdot n)$.

3.12.3 Settle point computation

As stated before the problem is to place a new site such that it possess the biggest possible region. Again the straightforward method - insert a new site, compute its region and delete it again - is obviously a bad choice. Since Voronoi regions can be disconnected, e.g. based on the $L_{0.5}$ distance, every pixel would have to be tested for membership. This can be circumvented using the same idea as before. Draw a circle around every pixel p choosing the distance between p and its corresponding site as radius.

For our current needs, this idea can even be improved. Instead of increasing the area counter for the entire set of pixels inside

the circle, it is sufficient to walk around the border of the circle. For each row we add 1 to the area counter of the first pixel inside the circle and subtract 1 from the area counter of the first pixel outside the circle. At the end of the entire circle drawing part of the algorithm, a prefix summation over each row yields the correct size of the area for all pixels.

Running time

Assuming a picture of $p \cdot q$ pixels, the time consumed is bounded from above by $O(p \cdot q \cdot (p + q))$ operations, as drawing a circle involves no more than $O(p + q)$ pixels.

As a matter of fact, precomputed patches reduce in the same way as before the number of circle to be *drawn*, hence the time decreases similarly.

Figure 3.56 illustrates a settle point computation of 18 sites based on the $L_{0.5}$ distance.

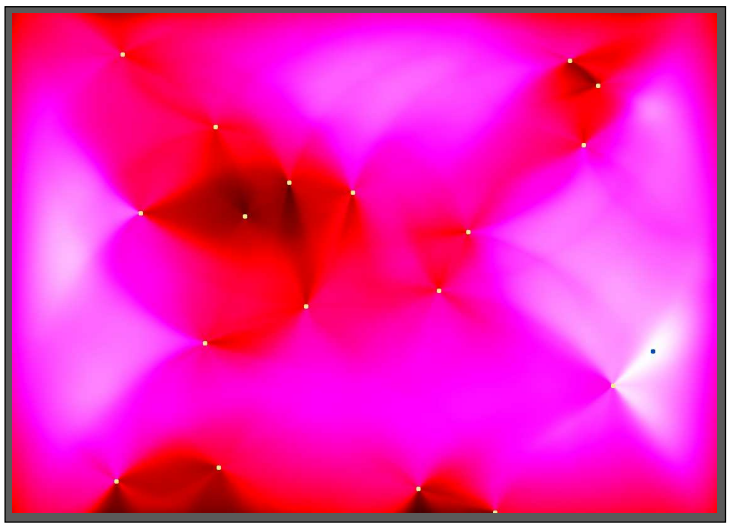


Figure 3.56
Settle point computation based on the $L_{0.5}$ distance, the settle point is colored in blue

Chapter 4

Smallest Enclosing Homothet

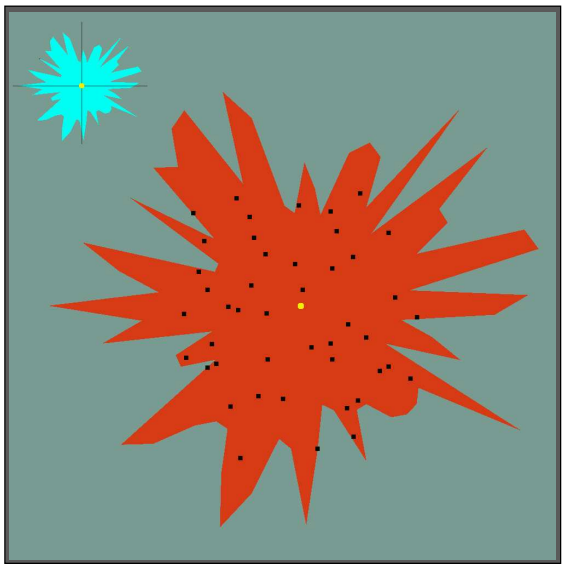


Figure 4.1
Smallest Enclosing Homothet Computation

4.1 The mission

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 gives a lower bound on the transmitting power which has to be deployed to cover all the communities.

In the sequel of this chapter, we identify, for the sake of simplicity, a circle with a regular n-gon with *infinitely* many vetrices. Thus, if we state Q to be a star–shaped polygon, we also allow Q to be a circle.

The *minimal enclosing circle*, for short MEC- problem as introduced above is a special case of the *smallest enclosing homothet* problem, defined as follows:

Definition 4.1 (Homothet) *Given a polygon Q. The polygon H is called a homothet of Q, if H can be generated by scaling and translating Q.*

Definition 4.2 (Smallest enclosing homothet) *Let S be a set of n points in the plane and Q be a simple star–shaped polygon. Determine the smallest homothet H of Q such that all points of S are contained inside H.*

This problem can also be stated as a facility location problem. Given the set S and the star-shaped polygon Q, such that $(0,0)$ is contained in the kernel. we search for a translation vector $\mathfrak{t} \in \mathbb{R}^2$ minimizing the scaling factor $\lambda \in \mathbb{R}^+$ such that $s \in (\mathfrak{t} + \lambda Q)$ for all $s \in S$.

A straightforward 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.

4.2 Previous work in the traditional model

Choosing the euclidean norm as the underlying distance function, leads us back to the former mentioned MEC–problem, where the polygon Q takes the shape of a circle, i.e. a convex n-gon with $n = \infty$. For that special case the prune-and-search techniques for linear programming developed by Nimrod Megiddo in 1983 can be adapted to solve this problem in linear time [Meg83].

It was not before 1991, that a further enhancement for the computation of the minimum enclosing circle was presented by Welzl ([Wel91]). 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. [GS98]).

A related problem is considered by Schwarz et al.[STWE94], 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 [FS75]. In 1985 Aggarwal, Chang and Yap [ACY85b] 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 [DA84], who on his part proposed solutions for the minimal enclosing equiangular or regular k–gon polygon.

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

4.3 Pixel–based solution

In a primary step, we develop an algorithm to solve the MEC–problem in our model of computation. Then this algorithm is 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 the furthest point Voronoi diagram of S, fVod(S) (cf. 3.7.1), 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, resembling the fVod(S) at that position. 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 Q be the requested star–shaped polygon, and k_Q be a *kernel point* of Q, i.e. an interior point such that all the boundary points of Q are visible from k_Q.

In principle, the procedure remains the same. We compute the lower envelope of an arrangement of cones. Each cone is a translation of a cone the base of which corresponds to the polygon Q. However, we have to be careful about the shape of the cone we put at every site in order to compute the upper envelope. A problem arises if the requested polygon Q is not centrally symmetric about the kernel point.

In order to ease the understanding, one might think of Q as the *unit circle* of a distance function $\text{dist}_Q : \mathbb{R}^2 \mapsto \mathbb{R}$ such that the kernel point k_Q is mapped to center of the unit circle.

To gain some insight into the problem, examine the following example. Let Q be a star–shaped polygon with kernel point k_Q. Moreover, Q is assumed to be not centrally symmetric about k_Q. Let p be a point on the boundary of Q. Let Q_p be the polygon Q translated by $p - k_Q$. Then k_Q is in general not part of the

boundary of translated polygon Q_p (see figure 4.2).

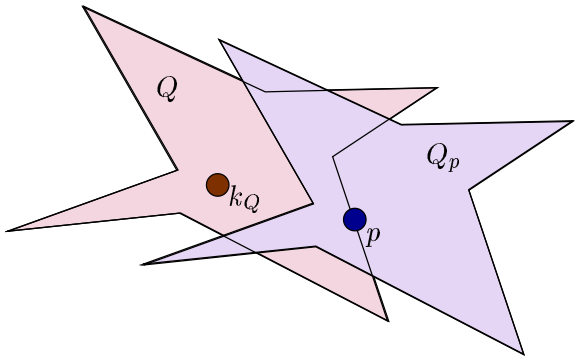


Figure 4.2
Asymmetrical star–shaped polygon

Mathematically stated, let Q be a star-shaped polygon. Without loss of generality, we assume $(0,0)$ inside the kernel of Q. Let k_Q be a point. Our goal is to find the polygon $\tilde{Q}$ such that for all points p on the boundary of the homothet $k_Q + \lambda Q$ holds that k_Q is on the boundary of $p + \lambda\tilde{Q}$, for $\lambda \in \mathbb{R}^{\geq 0}$.

$$\begin{aligned} & p \in \mathrm{bd}(k_Q + \lambda Q) \\ \iff \quad & k_Q \in \mathrm{bd}(p - \lambda Q) \\ \iff \quad & \tilde{Q} = -Q \end{aligned}$$

We obtain $-Q$ as a central reflection of Q.

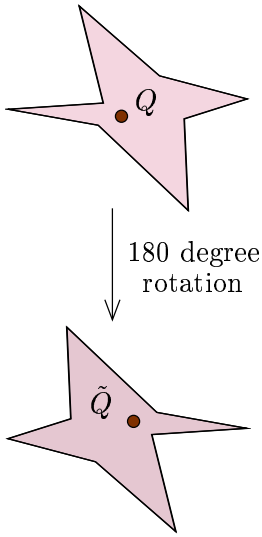


Figure 4.3
Construction of the base of the cones

The finial pixel based procedure to compute the smallest enclosing homothet with regard to a star-shaped polygon Q starts with the computation of the upper envelope of the arrangement of (identical) cones, where the base of the cones is the centrally reflected polygon Q. Thereafter, we determine the lowest pixel of the upper envelope. The position of the pixel is the center of the desired homothet, and the depth buffer value encodes the scaling factor.

4.4 Generalizations

Smallest enclosing homothet of line segments and circles

Let Q be the polygon we search a smallest enclosing homothet for. If we consider $\tilde{Q} = -Q$ to represent a unit circle of a distance func-

tion, we can apply the algorithm to compute the furthest Voronoi distance diagram for a set line segments and circles based on a non euclidean distance function. Thereafter, we proceed as before.

As a consequence, we can use our algorithm to compute the minimal enclosing circle of circles in the plane.

4.4.1 Computing the smallest k–enclosing homothet

Let us consider the following problem, first stated by Efrat et al. [ESZ93].

Definition 4.3 (smallest k–enclosing circle) *Given a set of n points in the plane, the smallest k–enclosing circle is the smallest circle enclosing at least a set of k points.*

This can be generalized in the same sense as before, asking for the smallest k–enclosing homothet.

Adopting the algorithms for higher order Voronoi diagrams, presented in chapter 3 in the most natural way, enables us using this approach to give the appropriate answer.

4.5 Analysis

For the upper envelope computation based on n points and a polygon Q with v vertices we need n triangular fans consisting each of v triangles. In case of Q being a circle, this is the computation of the furthest point Voronoi diagram.

Furthermore, in oder 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 in chapter 3, we do not use any color information, only the depth buffer values matter. Hence, except for the errors made by the graphic engine, our computation is exact. Even the computation of the minimum

enclosing circle is only affected by vertical approximation errors caused by the computation of the furthest point Voronoi diagram, respectively the corresponding distances.

Chapter 5

Extremal polygon containment

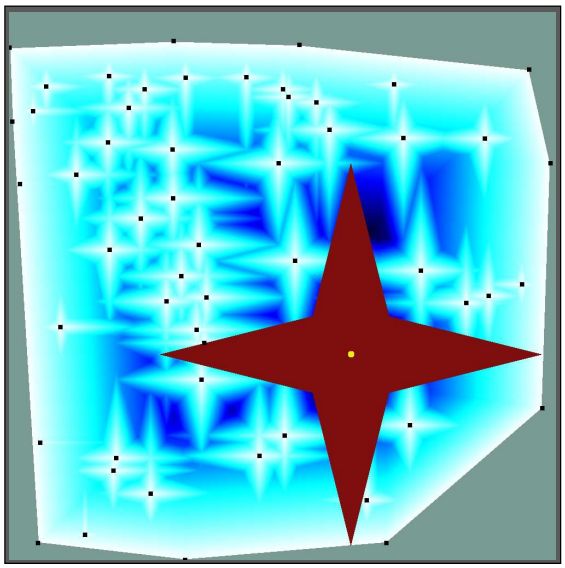


Figure 5.1
Biggest empty star–shaped polygon (in red with yellow kernel point) constrained to be completely inside the convex hull, the blue area represents the lower envelope computation

5.1 The mission

Recall our example of the alliance of communities from chapter 4. 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 further 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 following definition.

Definition 5.1 (Extremal polygon containment) *Let S be a set of n points in the plane, A be a subset of the plane, and Q be a simple star–shaped polygon. Determine the biggest homothet H of Q such that no point of S is contained inside H and H is contained in A.*

The problem is also called *largest empty homothet* emphasizing the duality with the *smallest full homothet* of the last chapter.

A variation of the problem asks for the largest empty circle the center of which is inside a predefined bounding box, which normally is the convex hull of the points in S (e.g. see [PS95, pp. 256 et sqq.]).

This can also be stated as a facility location problem. Given the set S of sites. Let A be a subset of the plane and Q be a star-shaped polygon such that $(0,0)$ is contained in the kernel. We search for a translation vector $\mathfrak{t} \in \mathbb{R}^2$ maximizing the scaling factor $\lambda \in \mathbb{R}^+$ such that $\mathfrak{t} + \lambda Q \cap S = \emptyset$ and $\mathfrak{t} \in A$, respectively $\mathfrak{t} + \lambda Q \subset A$.

5.2 Previous work in the traditional model

Restricted to the largest empty circle, Shamos and Hoey present an $O(n \log n)$ time algorithm based on Voronoi diagrams in [SH75]. In 1986, Lee and Wu [LW86] settled a lower bound of $\Omega(n \log n)$ for the algebraic decision tree model proving optimality of the preceding algorithm.

Sharir and Toledo [ST94] developed an algorithm used 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 running time for that algorithm is $O(k^2 n \lambda_4(kn) \log^3(kn) \log\log(kn))$ ($\lambda_q(r)$ is the maximum length of an (r, q) Davenport Schinzel sequence, i.e. almost linear in r for fixed q).

Fortune [For85] and Leven and Sharir [LS87] 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 [AB88] with an $O(k^3 n^3 \log(kn))$ time algorithm.

5.3 Pixel–based realization

Let $S = \{s_1, \ldots, s_n\}$ be a set of n points in the plane, and Q a star–shaped polygon. Our aim is to compute the largest homothet H of Q such that no point of S is contained in Q. In a basic approach we do not impose any constraint on the position of kernel point k_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 from chapter 4 for determining the smallest enclosing homothet. In contrast to there, we use now the lower envelope of the arrangement of cones to discover k_H. As before, all cones are translations of a cone the base of which has the shape of Q rotated about π around k_Q. The kernel point k_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 λ.

5.4 Extensions

5.4.1 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 (even if the polygon has holes).

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 re-allow 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 5.1).

We can just add the boundary of the constraint polygon as an additional site, deploying the idea of site in the shape of line segments for the compuation of generalized Voronoi diagrams.

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.

5.4.2 Line segments and circular arcs obstacles

As a consequence of the before mentioned idea to realize the constraint polygon, we can also compute the largest empty homothet in the case that the obstacles are in the shape of line segments and/or circular arcs.

5.4.3 Weighted Facilities

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

Definition 5.2 (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 $w_i > 0$, and A be a subset of the plane. Find the point $c \in A$ such that

$$c = \max_{p \in A} \min_{i} \; w_i * d(p, s_i).$$

Follert et al. [FSS95] give a subquadratic $O(n \log^4 n)$ time algorithms, 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 (cf. chapter 3).

Chapter 6

Alpha Hulls

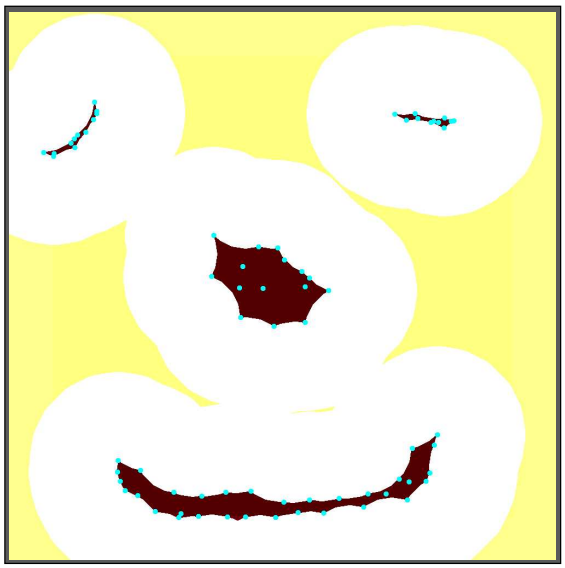


Figure 6.1
The alpha hull (in red) of a (blue) point set where the associated α is negative, the yellow area represents a cutting plane that arose from the pixel–based construction

6.1 The shape of a point set

In 1983, Edelsbrunner, Kirkpatrick, and Seidel [EKS83] presented a new structure christened to the name α–shapes. They were in-

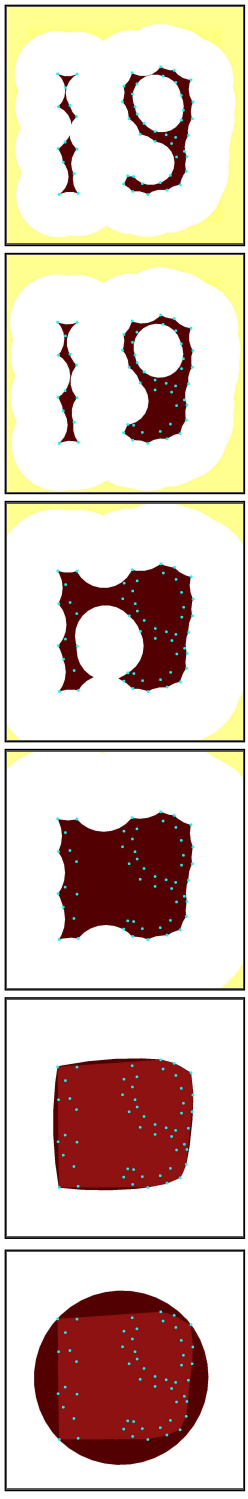


Figure 6.2
A sequence of alpha–hulls with increasing α value

vented to grasp the notion of the shape of a set of points. For example, one representative for such a description is the (ordinary) convex hull, a special case for the α–hull for $\alpha = 0$.

In a sense, α–shapes are intended to reflect the *distance* between the point set under consideration and an observer. From a sufficiently large distance, the single points are not recognized individually but as an agglomeration. The more you approach the set of points, the more points emerge *visible* out of the melted set. For an appropriate small value of α nothing but the points remain as the α–hull (cf. figure 6.2).

For the definition of the α–hull of a set of points we refer to the initial paper presented by Edelsbrunner, Kirkpatrick, and Seidel [EKS83].

Definition 6.1 (**α–hull of set of points** ($\alpha \geq 0$)) *Let α be a positive real. The α–hull of a set S of points is the intersection of all closed discs with radius $1/\alpha$ that contain all the points of S.*

In case of $\alpha = 0$, the closed discs degenerate to half spaces (discs with infinite radius), such that the resulting hull is nothing else but the convex hull.

Definition 6.2 (**α–hull of set of points** ($\alpha < 0$)) *For an arbitrary negative real α, the α–hull of a set S is the intersection of all closed complements of discs (of radius $-1/\alpha$) that contain all the points of S.*

For negative values of α the α–hull can also be regarded as the entire plane subtracted by the area of the union of empty discs of radius $-1/\alpha$.

A sequence of six different alpha hulls based on the same set of points is shown in figure 6.2. The corresponding values for α is increasing from top to bottom, whereby the first four values are negative. Integrated in the fifth and sixth picture is the convex hull, i.e. the $\alpha = 0$ hull. The smallest enclosing circle is represented in the sixth picture as another special case for an α hull.

As stated above, the entire plane, the minimal enclosing circle, the convex hull, and the set itself can each be viewed as special

cases of the continuous family of α–hulls. Each member of this family satisfy the relationship that if $\alpha_1 \leq \alpha_2$, then the α_1–hull is contained in the α_2–hull (cf. [EKS83]).

6.2 Previous work in the traditional model

Edelsbrunner et al.[EKS83] presented a $O(n \log n)$ time algorithm, which is optimal as the convex hull can be expressed as an α–hull. Further aspects of α–hulls can also be found in [Ede87, on page 309–315], [KF], as well as [Ede].

Concerning the three dimensional space, Edelsbrunner and Mücke introduces in [EM92] the formal notion of the family of alpha-shapes of a finite point set in R^3. Visualization and modeling, for example in molecular shape analysis [Akk96], [Mor96], [Fac96], as well as surface reconstruction [GMW97] and curve approximation [SC99] are some of the fields in which alpha-shapes prove to be useful.

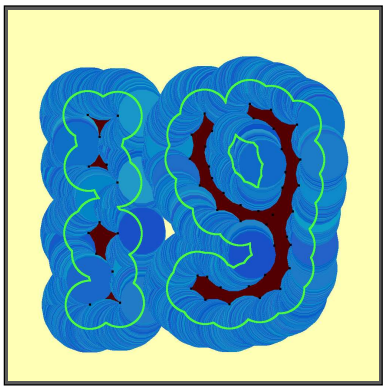


Figure 6.3
Construction of the α–hull of a point set

6.3 Pixel–based approach for negative α

In a first step, we regard the entire plane as a coarse approach to the correct hull. Given a set of point sites and a fixed radius α we could draw a circle of radius α around each pixel whenever none of the sites is covered. How can we be sure that no site is covered without testing for it?

To solve this problem, we compute the lower envelope of the union of cones positioned at the point sites. Then, we draw a circle of radius α around each pixel at height α. Clearly, none of the sites is contained in any circle. In fact each circle has at least one site on its boundary. In a sense, these circles erase the area around the sites without touching them, since the center of each circle is always at distance α away from its nearest site guaranteed by the lower envelope computation.

Despite, there might still be area not belonging to the α–hull, i.e. pixels at distance more than two times α away from its nearest site. Accordingly, these are not covered by any circle. This problem can be tackled in drawing a picture wide rectangle at depth α erasing

the remaining *wrongly colored* pixels.

In figure 6.3 the construction of the α–hull is made visible. The light yellow area is caused by the picture wide rectangle at depth α. The α circles are drawn in blue whereas the center of the circles are marked in green. Eventually, the dark red part of the picture is the actual α–hull.

In case that the points are too near to the border of the picture we run into trouble. It might happen that some of the circles, respectively the centers of them, might fall outside of the picture, and as such remain undrawable. It is because the initially computed lower envelope might be too high at the border. As a result we compute an overestimation of the desired α–hull. However, we can determine the part of the picture (the α–hull), which might be wrong. At least all pixels at distance α away from the border of the picture are correctly colored.

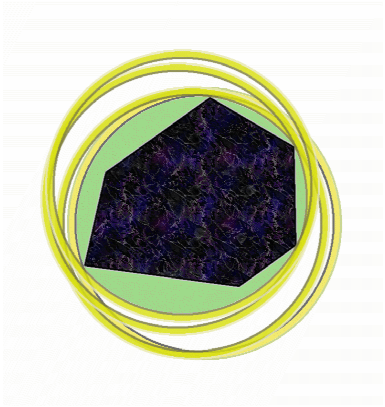


Figure 6.4
A boulder with a moving ring around it, the α–hull is made up of the boulder and the green area, as it cannot be reached by the ring

6.4 The α–hull for positive α

In order to get a notion of the shape of the positive α–hull of a set of points imagine we have a massive boulder in the shape of the convex hull of the point set. Around this boulder there is a ring which can be moved freely but such that the boulder is always inside it. However, in whatever position you move the ring, there will be always some space between the ring and the boulder since the boulder has straight edges whereas the ring is round. In our example, the positive α–hull is the union of the boulder and the space which cannot be reached by the ring for whatever position it is moved t (see figure 6.4).

6.4.1 Implementation

Before we start the actual computation of the α–hull, we first verify that the demanded radius is big enough, i.e. at least as big as the minimal enclosing circle (cf. chapter 4.1). In the computation of the hull, we use LEDA [MN99] to precompute the convex hull points, as it is by far the fastest method.

We process the convex hull points subsequently in counterclockwise order. Let s, and t be two subsequent points of the convex hull.

The first step is to draw a circle with radius α such that s and t lie at the border of the circle. The second step is to test whether this circle covers all convex hull points. If that is the case, we redo the first step with t and t', the convex hull point succeeding t. If the test fails, we undo the drawing and continue the first step with s and t' until any point of the convex hull was processed (cf. figure 6.5).

During the course of the algorithm, we maintain the following invariant. Let s and t be a pair of points for which the corresponding circle denoted by C_{st} covers all other convex hull points, i.e. C_{st} is a *valid* circle. Then any other *valid* circle covers the part of the area of C_{st} on the left to the line segment between s and t. As a consequence from the definition, this area determines the α–hull limited to the left of the segment.

The test whether a circle covers all convex hull points is accomplished by enabling the stencil buffer. We set the stencil buffer function such that for any modified pixel the stencil buffer value is increased by one. Next, we check for each pixel on the convex hull if it was hit by the circle in testing the color buffer values. Unless all the convex hull defining pixels are covered, we redo the drawing of the circle by rendering the same circle again, but now with stenciling enabled decreasing the stencil buffers each by one.

The intersection of all *valid drawn* circles is given by the pixels with the highest stencil buffer values, here denoted by k. Thus, the α–hull is gained by rendering a picture wide rectangle allowing a color update only for pixels the stencil buffer value of which is k.

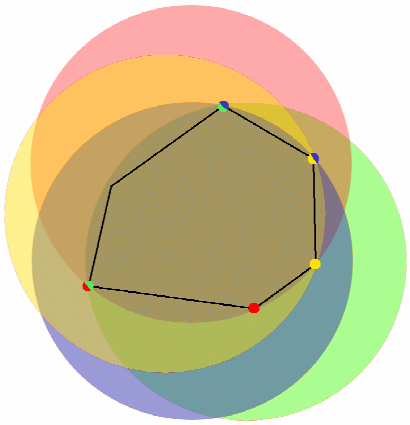


Figure 6.5
Visualization of the construction of the α–hull of a point set

Running time

The running time is made up of the time required to compute the convex hull and the time spent on the rendering of the circles. In the worst case we draw at most two circles for each point of the convex hull, as the *undo* operation might demand a redraw of a circle.

6.4.2 Alternative implementation

The above presented implementation has the drawback, that we rely on LEDA (or any comparable tool) to precompute the set of

convex hull points. If we do not want to include any other tools, than the one OpenGL provides, we can still compute the alpha hull for positive values of alpha. In a sense, we map the definition of the alpha hull into graphics.

Let C be the circle of appropriate size, corresponding to the alpha under consideration. Our idea for the pure pixel-based algorithm subdivides into two parts.

First, we determine the set of points, where we can place the center of the circle C covering all points. This is accomplished by drawing a circle for each site, such that the center of the circle is at the position of the site. The intersection of all circles is the desired set of points.

Second, for each point q of the just computed set of points, we draw a circle centered at q. The intersection of these circles is the desired alpha-hull.

Both intersections are realized by counting for each pixel, the number of times it undergoes an update. This can be implemented either be enabling blending in the same fashion as used for counting in section 3.10 of chapter 3. Provided the number of sites is small, we can also use stenciling to reckon for each pixel the number of times it is covered by a circle. Using solely the stenciling mechanism is slightly faster and less complicated to use for counting than blending.

Compared to the computation in chapter 3, we now only have to pay attention to the pixels the stencil buffer of which are equal to the number of just processed sites. Thus, we do not have to read back any stencil values but we can render a picture wide rectangle to reset the stencil buffer values correctly. Hence, to prevent a buffer overflow, we render a first rectangle resetting the stencil buffer of all pixels to zero, which have a stencil buffer value of less than the number of currently processed sites. A second (picture wide) rectangle reduce the stencil value to 1 of all the remaining pixels, which are still candidates to be contained in the final intersection set.

In the second part, we do not know the maximum number we have to count to. Thus, the *misuse* of blending is the first choice for counting.

Running time

For the advantage of relying solely on OpenGL, we have to pay for by an increase in the running time. This is founded in the fact, that the set of pixels computed in the first step can be very large. Consequently, the number of rendered circles in the second part, i.e. the number of processed triangles, may be very large, which can cause a considerable increase in the running time compared to the first implementation. This unwanted behavior appears every time, the radius of the circle C is sizeably larger than the diameter of the point set.

An example of such a worst case is shown in figure 6.6.

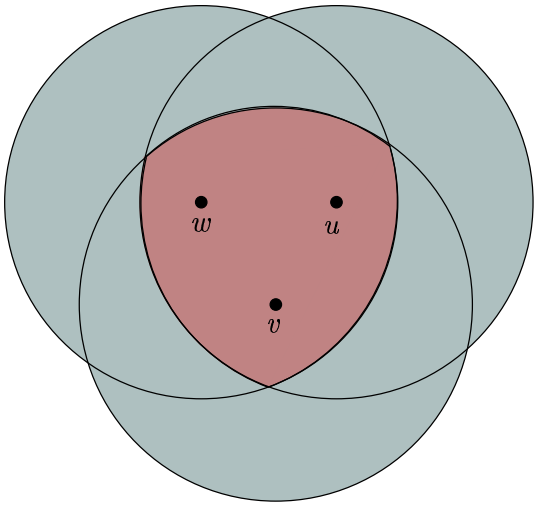


Figure 6.6
Worst case for alternative alpha hull computation

Speedup

Considering the before mentioned algorithm, we actually do not need to draw all the circles of the part. Since the intersection of convex sets is also convex, it is sufficient to draw circles only for

the points on the boundary of the intersection resulting from the first part. This can be accomplished by choosing only the first and the last pixel of each row to draw a circle for. This reduce the number of rendered circles in the worst case to be bounded by $2(w + h)$, assuming a rectangular picture of size $w \times h$.

Chapter 7

Minimum and maximum area triangle and quadrangle

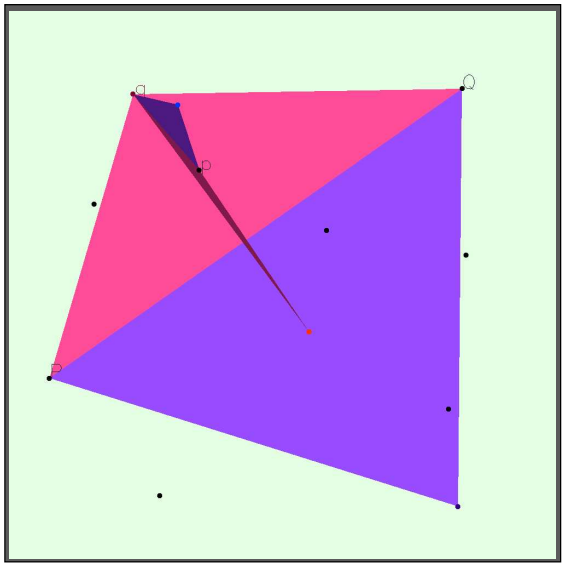


Figure 7.1
Minimum and maximum area triangle and quadrangle

7.1 Introduction

In this chapter we are primarily interested in the problem of finding these three points out of the input set that form the minimum area triangle. The pixel–based approach is based on an elementary geometric property concerning the area of triangles. This idea is further exploited to compute the maximum area triangle, as well as the minimum and maximum quadrangle.

7.2 Previous work in the traditional model

As one application of the construction of arrangements of lines and hyperplanes [EOS82], Edelsbrunner, O'Rourke, and Seidel presented in 1982 an $O(n^2)$ time algorithm exploiting the power of duality. Further development was made by Eppstein et al. [EORW92]. They presented solutions for determining convex (empty) k–gon.

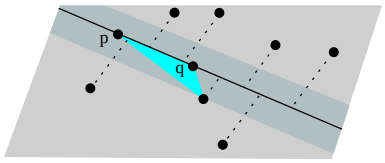


Figure 7.2
Minimum area triangle with base line through p and q

7.3 Pixel–based approach

7.3.1 Minimum area triangle

The minimum area triangle can be found by exploiting the following property. Suppose we know two of the three points that define the minimum area triangle. The remaining point is the one nearest the line through the former two as pictured in figure 7.2. Thus, the desired triple of points can be found by examining for every pair of points the minimum area triangle.

We can transform the idea in the pixel–based world as follows: For each pair of points (p, q), we put a saddle roof like object upside down with its roof edge onto the pair (p, q) (cf. figure 7.3). Then, we read back the depth buffer values for all other points. The one with the lowest value is the one we search for.

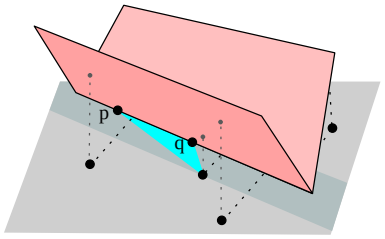


Figure 7.3
Saddle roof on top of two points p and q

7.3.2 Minimum area triangle, revised

For the computation of the minimum area triangle, we can avoid reading back the depth buffer values in every pass. More properly, we can render the $\binom{n}{2}$ saddle roof in only one pass.

For that, we exploit a yet unused degree of freedom. We can adapt the apex angle to encode the length of the segment between the two fixed points p and q, i.e. we can choose the angle such that the appropriate depth values encode the areas of the corresponding points.

Let g the length of the segment between p and q, the two points under consideration. We denote by ζ the apex angle of the saddle roof and choose it to be

$$\zeta = 2 \cdot \operatorname{arccot}(g/2).$$

Let s be an arbitrary point with distance d to the line through p and q. After the rendering of the saddle roof, the height z of s is

$$z = d \cdot \cot(\zeta/2) = d \cdot \cot(\operatorname{arccot}(g/2)).$$

Hence, z equals the area of the triangle defined by p, q and s (cf. figure 7.4).

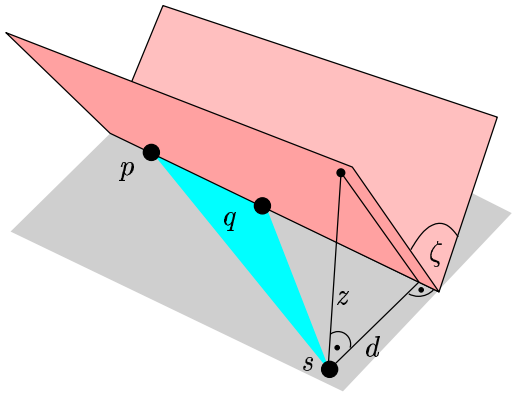


Figure 7.4
Choosing the appropriate apex angle

The minimum area triangle can now be found in a single pass, provided the depth buffer function is set to "$\leq$". To identify the triple of points defining the demanded triangle, we *color* the saddle

roofs with one of its defining points. Additionally, we have to take care that a saddle roof does not change the buffer values of its defining points. This can be accomplished with the use of the stencil buffer. Eventually, we search for the point p with the lowest depth buffer value. The point s nearest the line through p and the point determined by the color value of p completes the desired triple of points.

7.3.3 Additionally cutting down the rendering time

We can even further accelerate our algorithm sizeably exploiting the following idea. Assume we know an upper bound on the minimum area of the demanded triangle. Then we can adjust the width of the saddle roofs appropriately, such that only pixels within this bound are covered. Hence, determine the minimum area triangle of a small sample set of the points may result in a further acceleration.

Moreover, we can make use of a pigeon hole argument to derive a upper bound on the area of the minimum area triangle. Consider an arbitray triangulation of the n points, the picture is partitioned into at least $n-2$ triangles. That means, that the area of the minimum area triangle is at most $(w \cdot h)/(n-2)$, assuming a rectangular picture of size $w \times h$.

7.3.4 Minimum area quadrangle

In order to compute the minimum area quadrangle, we can make use of the same methods as before except the adaptation of the apex angle is of no use. But, the acceleration in precomputing an upper bound of the area can also be applied, although slightly modified. In determining the maximal height of the saddle roof, we have to take into consideration, that the minimum quadrangle can be almost completely on one of the both sides, i.e. thus in the red or the blue plane.

Assume we know the two points p and q, determining the diagonal of the minimum area quadrangle. Then the nearest point to the left and the right of the diagonal gives us the quadrangle we search for. We use the before mentioned construction to accomplish this task.

For n points in the plane we need $\binom{n}{2}$ rendering passes, each consists of the following steps.

- Reset the depth buffer values.
- Draw a saddle roof on the pair of points under consideration. To distinguish the points to the left from the one on the right, we color the right side of the saddle roof blue, the left one red.
- Read back the depth buffer values of the $n-2$ remaining points. The red and blue points with minimum depth buffer value are the one we searched for.

7.3.5 Maximum area triangle/quadrangle

We can use the same algorithms to compute the maximum instead of the minimum area triangle and quadrangle, respectively. The only difference is that we now have to determine the points with the maximum depth buffer values.

Concerning the algorithm to compute the minimum/maximum area triangle, we cannot make use of acceleration achieved by reducing the width of the saddle roof. The same holds for the minimum/maximum area quadrangle.

List of Figures

Bibliography

[AA02] O. Aichholzer and F. Aurenhammer. Voronoi diagrams - computational geometry's favorite. *Special Issue on Foundations of Information Processing of TELEMATIK*, 1:7–11, 2002.

[AAP02a] O. Aichholzer, F. Aurenhammer, and B. Palop. Quickest paths, straight skeletons, and the city voronoi diagram. In *Proc.* 18^{th} *Ann. ACM Symp. Computational Geometry*, Barcelona, Spain, 2002.

[AAP02b] Oswin Aichholzer, Franz Aurenhammer, and Belén Palop. Quickest paths, straight skeletons, and the city voronoi diagram. In *Proceedings of the eighteenth annual symposium on Computational geometry*, pages 151–159. ACM Press, 2002.

[AB88] F. Avnaim and J.-D. Boissonnat. Polygon placement under translation and rotation. In M. Wirsing R. Cori, editor, *Proceedings of the 5th Annual Symposium on Theoretical Aspects of Computer Science (STACS '88)*, volume 294 of *LNCS*, pages 322–333, Bordeaux, France, February 1988. Springer.

[ACY85a] Alok Aggarwal, J. Chang, and Chee Yap. Minimum area circumscribing polygons. Technical Report NYU Courant Report 160, Robotics Report 42, May, 1985, IBM-TJW/NYU-Courant, 1985.

[ACY85b] Alok Aggarwal, J. S. Chang, and Chee K. Yap. Minimum area circumscribing polygons. *The Visual Computer*, 1(2):112–117, October 1985.

[AE84] F. Aurenhammer and H. Edelsbrunner. An optimal algorithm for constructing the weighted Voronoi diagram in the plane. *Pattern Recognition*, 17(2):251–257, 1984. [IIG-Report-Series F109, TU Graz, Austria, 1983].

[AH35] P. Alexandroff and H. Hopf. *Topologie*. L. Julius Springer, Berlin, 1935.

[AI87] F. Aurenhammer and H. Imai. Geometric relations among Voronoi diagrams. In *Proc. 4^{th} Ann. STACS, Lecture Notes in Computer Science*, volume 247, pages 53–65, Passau, Germany, 1987. Springer Verlag.

[AK00] 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. [SFB Report F003-092, TU Graz, Austria, 1996].

[Akk96] Nataraj Akkiraju. Molecule surface triangulation from alpha shapes. Technical Report UIUCDCS-R-96-1942, University of Illinois at Urbana-Champaign, June 1996.

[Ata99] Mikhail J. Atallah, editor. *Algorithms and theory of computation handbook*. CRC Press, 2000 N.W. Corporate Blvd., Boca Raton, FL 33431-9868, USA, 1999.

[Aur84] F. Aurenhammer. *Gewichtete Voronoi Diagramme: Geometrische Deutung und Konstruktions-Algorithmen*. PhD thesis, IIG-TU Graz, Austria, 1984. Report B53.

[Aur86a] F. Aurenhammer. The one-dimensional weighted Voronoi diagram. *Information Processing Letters*, 22(3):119–123, 1986. [IIG-Report-Series F110, TU Graz, Austria, 1983].

[Aur86b] Franz Aurenhammer. A new duality result concerning Voronoi diagrams. In Laurent Kott, editor, *Automata, Languages and Programming, 13th International Colloquium*, volume 226 of *Lecture Notes in Computer Science*, pages 21–30, Rennes, France, 15–19 July 1986. Springer-Verlag.

[Aur91] 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].

[BKOS97] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin Heidelberg, 1997.

[BKST00] S. N. Bespamyatnikh, Klara Kedem, Michael Segal, and Arie Tamir. Optimal facility location under various distance functions. *International Journal of Computational Geometry and Applications*, 10(5):523–534, 2000.

[BMS94] C. Burnikel, K. Mehlhorn, and S. Schirra. How to compute the Voronoi diagram of line segments: Theoretical and practical results. *Lecture Notes in Computer Science*, 855:227, 1994.

[BO83] Michael Ben-Or. Lower bounds for algebraic computation trees. pages 80–86, 1983.

[Bro79] Kevin Q. Brown. Voronoi diagrams from convex hulls. *Information Processing Letters*, 9(5):223–228, December 1979.

[BSTY98] Boissonnat, Sharir, Tagansky, and Yvinec. Voronoi diagrams in higher dimensions under certain polyhedral distance functions. *GEOMETRY: Discrete & Computational Geometry*, 19, 1998.

[BT83] Binay K. Bhattacharya and Godfried T. Toussaint. Efficient algorithms for computing the maximum distance between two finite planar sets. *J. Algorithms*, 4(2):121–136, 1983.

[cas02] Cass Everitt cass@nvidia.com. Interactive order–independent transparency. Technical Report OpenGL Applications Engineering, NVIDIA, 2002.

[CD85] L. P. Chew and R. L. Drysdale, III. Voronoi diagrams based on convex distance functions. In Joseph O'Rourke, editor, *Proceedings of the Symposium on Computational Geometry*, pages 235–244, Baltimore, MD, June 1985. ACM Press.

[CD86] L. Paul Chew and Robert L. Scot Drysdale. Finding Largest Empty Circles with Location Constraints. Technical Report PCS-TR86-130, Dartmouth College, Computer Science, Hanover, NH, 1986.

[CD88] J. Canny and B. Donald. Simplified voronoi diagrams. *Discrete and Computational Geometry*, 3:219–236, 1988.

[CE87] B. Chazelle and H. Edelsbrunner. An improved algorithm for constructing K sup th -order voronoi diagrams. *IEEE Trans. Comput.*, C-36:1349–1354, 1987.

[Che93] L. Paul Chew. Near-quadratic bounds for the L_1 voronoi diagram of moving points. Technical Report TR93-1348, Cornell University, Computer Science Department, May 1993.

[CHLM02] 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.

[CY84] J. S. Chang and C. K. Yap. A polynomial solution for potato-peeling and other polygon inclusion and enclosure problems. In IEEE, editor, *25th annual Symposium on Foundations of Computer Science, October 24–26, 1984, Singer Island, Florida*, pages 408–416, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1984. IEEE Computer Society Press.

[DA84] DePano and Aggarwal. Finding restricted k-envelopes for convex polygons. In *ALLERTON: 22th Annual Allerton Conference on Communication, Control, and Computing*, pages 81–90. Allerton House, Monticello, Illinois, 1984.

[Des44] René Descartes. *Principia Philosophiae*. Ludovicus Elzevirius, Amsterdam, 1644.

[DKS02] Frank Dehne, Rolf Klein, and Raimund Seidel. Maximizing a voronoi region: The convex case. In *ISAAC, International Symposium on Algorithms and Computation*, volume 13, November 2002.

[DL91] Hristo Djidjev and Andrzel Lingas. On computing the Voronoi diagram for restricted planar figures. In F. Dehne, J.-R. Sack, and N. Santoro, editors, *Algorithms and Data Structures, 2nd Workshop WADS '91*, volume 519 of *Lecture Notes in Computer Science*, pages 54–64, Ottawa, Canada, 14–16 August 1991. Springer-Verlag.

[DWE02] J. Diepstraten, D. Weiskopf, and T. Ertl. Transparency in interactive Technical Illustrations. In *Proc. EuroGraphics '02*, volume 21 of *3*, page 11, August 2002.

[Dwy91] R. A. Dwyer. Higher-dimensional Voronoi diagrams in linear expected time. *Disc. and Comp. Geometry*, 6:343–367, 1991.

[Ede] Herbert Edelsbrunner. Alpha shapes on web. *http://www.alphashapes.org/*.

[Ede85] H. Edelsbrunner. Computing the extreme distances between two convex polygons. *J. Algorithms*, 6:213–224, 1985.

[Ede87] Herbert Edelsbrunner. *Algorithms in Combinatorial Geometry*, volume 10 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, November 1987.

[EKS83] Herbert Edelsbrunner, D. G. Kirkpatrick, and Raimund Seidel. On the shape of a set of points in the plane. *IEEE Trans. Information Theory*, IT-29:551–559, 1983.

[EM92] Herbert Edelsbrunner and Ernst P. Mucke. Three-dimensional alpha shapes. *1992 Workshop on Volume Visualization*, pages 75–82, 1992.

[EORW92] David Eppstein, Mark Overmars, Günter Rote, and Gerhard J. Woeginger. Finding minimum area k-gons. *Discrete & Computational Geometry*, 7(1):45–58, 1992.

[EOS82] H. Edelsbrunner, J. O'Rourke, and R. Seidel. Constructing arrangements of lines and hyperplanes with applications. In *24th Annual Symposium on Foundations of Computer Science*, pages 83–91, Los Alamitos, Ca., USA, November 1982. IEEE Computer Society Press.

[Epp] David Eppstein. Geometry in action. *http://www.ics.uci.edu/ eppstein/geom.html*.

[Epp92] David Eppstein. New algorithms for minimum area k-gons. In Frances, editor, *Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete*

Algorithms (SODA '92), pages 83–88, Orlando, FL, USA, January 1992. SIAM.

[ES85] 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.

[EST+95] B. L. Evans, C. Schwarz, J. Teich, A. Vainshtein, and E. Welzl. Minimal enclosing parallelogram with application. In *Proceedings of the 11th Annual Symposium on Computational Geometry*, pages C34–C35, New York, NY, USA, June 1995. ACM Press.

[ESZ93] 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.

[Fac96] Michael Allen Facello. Geometric techniques for molecular shape analysis. Technical Report UIUCDCS-R-96-1967, University of Illinois at Urbana-Champaign, October 1996.

[For85] S. J. Fortune. A fast algorithm for polygon containment by translation (extended abstract). In Wilfried Brauer, editor, *Automata, Languages and Programming, 12th Colloquium*, volume 194 of *Lecture Notes in Computer Science*, pages 189–198, Nafplion, Greece, 15–19 July 1985. Springer-Verlag.

[For86] Steven J. Fortune. A sweepline algorithm for Voronoi diagrams. In *Proc. 2nd ACM Symp. Computational Geometry*, pages 313–322. ACM Press, 2–4 June 1986.

[For92] Steve Fortune. Voronoi diagrams and Delaunay triangulations. In F. K. Hwang and D.-Z. Du, editors, *Computing in Euclidean Geometry*. World Scientific, 1992.

[FR97] Flynn and Rudd. Parallel architectures. In *Allen B. Tucker, Jr. (Editor-in-Chief), The Computer Science and Engineering Handbook, CRC Press, in cooperation with ACM, 1997.* 1997.

[FS75] 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.

[FSS95] Frank Follert, Elmar Schömer, and Jürgen Sellen. Subquadratic algorithms for the weighted maximin facility location problem. In *Proc. 7th Canadian Conference on Computational Geometry*, pages 1–6, 1995.

[FT92] Gordon Fuller and Dalton Tarwater. *Analytic geometry*. Addison-Wesley, 7th ed. edition, 1992.

[FvDFH96] J. Foley, A. van Dam, S. Feiner, and J. Hughes. *Computer graphics: Principles and practice in C*. Addison-Wesley, 2nd edition, 1996.

[GKS90] Leonidas J. Guibas, Donald E. Knuth, and Micha Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. In Michael S. Paterson, editor, *Automata, Languages and Programming, 17th International Colloquium*, volume 443 of *Lecture Notes in Computer Science*, pages 414–431, Warwick University, England, 16–20 July 1990. Springer-Verlag.

[GMW97] B. Guo, J. Menon, and B. Willette. Surface reconstruction using alpha shapes. *Computer Graphics Forum*, 16(4):177–190, 1997. ISSN 1067-7055.

[GO97] Jacob E. Goodman and Joseph O'Rourke, editors. *Handbook of Discrete and Computational Geometry*. CRC Press, 1997.

[GS98] Bernd Gärtner and Sven Schönherr. Exact primitives for smallest enclosing ellipses. *Information Processing Letters*, 68(1):33–38, 1998. a preliminary version appeared in *Proc. 13th Annu. ACM Symp. on Computational Geometry, 1997, pages 430–432*.

[Har01] S. Har-Peled. A practical approach for computing the diameter of a point-set. In *SOCG2001*, pages 177–186, 2001.

[HKL+99] 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.

[HY61] J. G. Hocking and G. S. Young. *Topology*. Addison–Wesley, Reading, Massachusetts, 1961.

[Jän01] Klaus Jänich. *Topologie*. Springer, Berlin, 2001.

[KF] kfischer@iiic.ethz.ch Kaspar Fischer. Introduction to alpha shapes. *http://n.ethz.ch/student/fischerk/alphashapes/as/as.html*.

[Kil96] Mark J. Kilgard. *OpenGL Programming for the X Window System*. Addison-Wesley, Reading, MA, USA, 1996.

[KKS95] Matthew J. Katz, Klara Kedem, and Michael Segal. Improved algorithms for placing undesirable facilities. In *Proceedings of the 11th Canadian Conference on Computational Geometry*, pages 65–67, 1995.

[KL85] V. Klee and M. Laskowski. Finding the smallest triangles containing a given convex polygon. *Journal of Algorithms, 6, 1985*, pages 359–375, 1985.

[KL94] R. Klein and A. Lingas. Hamiltonian abstract Voronoi diagrams in linear time. *Lecture Notes in Computer Science*, 834:11 et seqq., 1994.

[Kle89] Rolf Klein. *Concrete and abstract Voronoi diagrams*, volume 400 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., New York, NY, USA, 1989.

[Kle97] Rolf Klein. *Algorithmische Geometrie*. Addison-Wesley-Longman, 1997. ISBN 3-8273-1111-x.

[Knu68] D. Knuth. *The Art of Computer Programming, Fundamental Algorithms*, volume 1. Addison-Wesley, Reading, Mass., 1968.

[Knu73] Donald E. Knuth. *The Art of Computer Programming. Volume III: Sorting and Searching*. Addison-Wesley, Reading, Massachusetts, 1973.

[KW87] Rolf Klein and Derick Wood. Voronoi diagrams based on general metrics in the plane. Technical

Report report00005, Albert-Ludwigs-Universitaet Freiburg, Institut fuer Informatik, November 1, 1987.

[KW88] R. Klein and D. Wood. Voronoi diagrams based on general metrics in the plane. In M. Wirsing R. Cori, editor, *Proceedings of the 5th Annual Symposium on Theoretical Aspects of Computer Science (STACS '88)*, volume 294 of *LNCS*, pages 281–291, Bordeaux, France, February 1988. Springer.

[Le94] N.-M. Le. On Voronoi diagrams in the L_p-metric in higher dimensions. *Lecture Notes in Computer Science*, 775:711, 1994.

[Lee80] D.T. Lee. Farthest neighbor voronoi diagrams and applications. Technical Report 80-11-FC-04,, Northwestern University, November 1980.

[Lee82] Der-Tsai Lee. On k-nearest neighbor Voronoi diagrams in the plane. *IEEE Trans. Computers*, C-31(6):478–487, June 1982.

[LS87] 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.

[LW86] D. T. Lee and Y. F. Wu. Geometric complexity of some location problems. *Algorithmica*, 1:193–211, 1986.

[Meg83] Nimrod Megiddo. Linear-time algorithms for linear programming in R^3 and related problems. *SIAM Journal on Computing*, 12(4):759–776, 1983.

[Meh84a] Kurt Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*, volume 1 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, Germany, 1984.

[Meh84b] Kurt Mehlhorn. *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness*, volume 2 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, Germany, 1984.

[Meh84c] Kurt Mehlhorn. *Data Structures and Algorithms 3: Multi-dimensional Searching and Computational*

Geometry, volume 3 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, Germany, 1984.

[MN99] Kurt Mehlhorn and Stefan Näher. *LEDA: A Platform of Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, England, January 1999.

[Mor96] Patrick Joseph Moran. Visualization and modeling with shape. Technical Report UIUCDCS-R-96-1945, University of Illinois at Urbana-Champaign, March 1996.

[OAMB86] J. O'Rourke, A. Aggarwal, S. Maddila, and M. Baldwin. An optimal algorithm for finding minimal enclosing triangles. *J. of Algorithms, 1986*, 7:258–269, 1986.

[OBS92] 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.

[OIM84] Takemasa Ohya, M. Iri, and K. Murota. A fast Voronoi-diagram algorithm with quaternary tree bucketing. *Information Processing Letters*, 18(4):227–231, May 1984.

[O'R93] Joseph O'Rourke. *Computational Geometry in C*. Cambridge University Press, 1993. ISBN 0-521-44034-3.

[PS95] 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.

[Rap89] David Rappaport. Computing the furthest site Voronoi diagram for a set of discs (preliminary report). In F. Dehne, J.-R. Sack, and N. Santoro, editors, *Algorithms and Data Structures, Workshop WADS '89*, volume 382 of *Lecture Notes in Computer Science*, pages 57–66, Ottawa, Canada, 17–19 August 1989. Springer-Verlag.

[Rot91] Günter Rote. Computing the minimum Hausdorff distance between two-point sets on a line under translation. *Information Processing Letters*, 38(3):123–127, May 1991.

[SA93] Mark Segal and Kurt Akeley. The OpenGL graphics system: A specification. Technical report, Silicon Graphics Computer Systems, Mountain View, CA,USA, 1993.

[SC99] Takis Sakkalis and Ch. Charitos. Approximating curves via alpha shapes. *Graphical models and image processing: GMIP*, 61(3):165–176, May 1999.

[SH75] 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.

[Sha75] Michael Ian Shamos. Geometric complexity. In ACM, editor, *Conference record of Seventh Annual ACM Symposium on 7th Theory of Computing: papers presented at the Symposium, Albuquerque, New Mexico*, pages 224–233, New York, NY, USA, May 1975. ACM Press.

[Smi00] Michiel Smid. Closest-point problems in computational geometry. In Handbook of Computational Geometry, J.-R. Sack, and J. Urrutia, editors, *Elsevier, 2000*. 2000.

[ST94] Sharir and Toledo. Extremal polygon containment problems. *CGTA: Computational Geometry: Theory and Applications*, 4, 1994.

[STWE94] Christian Schwarz, Jürgen Teich, Emo Welzl, and Brian Evans. On finding a minimal enclosing parallelogram. Technical Report TR-94-036, Berkeley, CA, 1994.

[TB81] G.T. Toussaint and B.K. Bhattacharya. On geometric algorithms that use the furthest point voronoi diagram. Technical Report SOCS-81.3, School of Computer Science, McGill University, January 1981.

[TM82] G. Toussaint and J. McAlear. A simple o(n log n) algorithm for finding the maximum distance between two finite planar sets, 1982.

[Tol91] S. Toledo. Extremal polygon containment problems. In ACM-SIGACT ACM-SIGGRAPH, editor, *Proceedings of the 7th Annual Symposium on Computational Geometry (SCG '91)*, pages 176–185, North Conway, NH, USA, June 1991. ACM Press.

[Wel91] E. Welzl. Smallest enclosing disks (balls and ellipsoids). In Hermann Maurer, editor, *Proceedings of New Results and New Trends in Computer Science*, volume 555 of *LNCS*, pages 359–370, Berlin, Germany, June 1991. Springer.

[WNDO99] Mason Woo, Jackie Neider, Tom Davis, and OpenGL Architecture Review Board. *OpenGL programming guide: the official guide to learning OpenGL, version 1.2*. Addison-Wesley, Reading, MA, USA, third edition, 1999.