Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware

Kenneth E. Hoff III, Tim Culver, John Keyser, Ming Lin, Dinesh Manocha

University of North Carolina at Chapel Hill, Dept. of Computer Science

Abstract: We present a new approach for computing generalized Voronoi diagrams in two and three dimensions using interpolation-based polygon rasterization hardware. The input primitives may be points, lines, polygons, curves, or surfaces. The algorithm computes a discrete Voronoi diagram by rendering a three dimensional distance mesh corresponding to each primitive. The polygonal mesh is a bounded-error approximation of a non-linear distance function. The algorithm divides the space into regular cells. For each cell it computes the closest primitive and the distance to that primitive using polygon scan-conversion and Z-buffer depth comparison. We present efficient techniques to detect Voronoi boundaries and compute Voronoi neighbors. The algorithm has been implemented on SGI workstations and PCs using OpenGL and applied to complex 2D and 3D datasets. We also demonstrate the applications of our algorithm to fast motion planning in static and dynamic environments, and improving the performance of continuous Voronoi diagram computation.

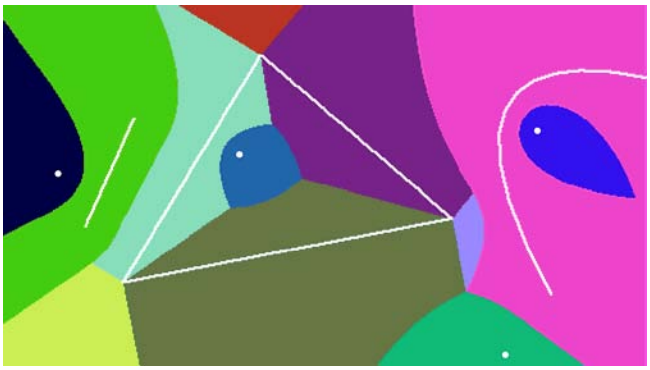
Key Words and Phrases: Voronoi diagrams, graphics hardware, polygon rasterization, interpolation, motion planning, proximity, medial axis, OpenGL, framebuffer techniques.

1 Introduction

Given a set of primitives, a Voronoi diagram partitions space into regions, where each region consists of all points that are closer to one primitive than to any other. Voronoi diagrams have been widely used in a number of applications including visualization of medical datasets, proximity queries, spatial data manipulation, shape analysis, computer animation, robot motion planning, modeling spatial structures and processes, pattern recognition, locational optimization, and selection in user-interfaces. The concept of Voronoi diagrams has been around for at least four centuries, and since the 1970s, algorithms for computing Voronoi diagrams of geometric primitives have been developed in computational geometry and related areas.

The set of input primitives may include points, lines, polygons, curves, polyhedra, curved surfaces, etc. Good theoretical and practical algorithms are known for computing Voronoi diagrams of points in any dimension. For higher order primitives like lines, curves, and polyhedra, the boundaries of the *generalized Voronoi diagrams* are composed of high-degree algebraic curves and surfaces, and their intersections. However, current algorithms used for representing and computing the Voronoi boundaries suffer from efficiency and accuracy problems. As a result, no efficient and numerically robust algorithms are known for constructing a topologically consistent and exact representation of generalized Voronoi diagrams.

Given the practical complexity of computing an exact generalized Voronoi diagram, many authors have proposed algorithms for computing an approximation or a discrete representation. Some approaches are based on computing the Voronoi diagram of a point-sampling of the primitives. Other approaches adaptively subdivide space into rectangular or tetrahedral cells and compute the boundary of the Voronoi diagram up to a pre-determined precision [Laven92, Teich97, Vleug95, Vleug96]. In practice, these approaches take considerable time and memory on large numbers of input primitives. Furthermore, this makes it difficult to use them in dynamic environments.



Cover Plate: Discrete approximation of the generalized Voronoi diagram of four points, a line, a triangle, and one cubic Bézier curve computed interactively on a PC.

Main Contributions: In this paper, we present an approach that computes discrete approximations of generalized Voronoi diagrams to an arbitrary resolution using polygon rasterization hardware. Our contributions include:

1. Efficient methods to approximate the distance function for lines, polygons, polyhedra, Bézier curves and surfaces, and other higher order primitives using a mesh that is interpolated by graphics hardware.
2. Adaptive techniques for generating the distance mesh of polygonal elements so that the error is bounded by a specified precision.
3. Efficient algorithms for detecting Voronoi boundaries and neighbors, which are used for accurate visualization and Delaunay triangulations.
4. Ability to construct weighted and farthest-site generalized Voronoi diagrams in 2D and 3D.
5. Demonstration of the effectiveness of our approach to the following applications:
   - Computing generalized Voronoi diagrams of complex 3D polygonal data sets.
   - Improving the efficiency of computing the exact and continuous Voronoi boundaries by using *neighbor-pair pruning*.
   - Fast motion planning in static and dynamic environments using discretized Voronoi diagrams.

The resulting techniques have been effectively implemented on PCs and high-end SGI workstations using the OpenGL graphics library. A 2D example computed in real-time is shown in the cover plate. Our techniques improve upon the state of the art in following ways:

- **Generality**: We make no assumption with respect to input primitives. We only need to compute the distance to the primitive from a point in space. For maximum effectiveness, we need to be able to efficiently mesh its distance function.
- **Efficiency**: We show that our approach is quite fast. Its speed arises from using coarse polygonal approximations of the distance functions while still maintaining the required error bound, using polygon rasterization hardware to reconstruct the distance values, and using the Z-buffer depth comparison test to perform distance comparisons. We demonstrate the 2D approach on models composed of nearly 100K triangles in

real-time in a motion planning application through a complex dynamic scene. We derive efficient meshing strategies for polygonal models in 3D, and show the results of a prototype implementation that demonstrates its potential.

- **Tight Bounds on Accuracy**: Although our approach produces a discretized Voronoi diagram, all sources of error are enumerated and techniques are given to produce output within any specified tolerance.
- **Ease of Implementation**: The approach can be easily implemented on current graphics systems to generate the distance mesh. The special cases are limited and the problem reduces to simply meshing a distance function for any new primitive.

Organization: We survey related work on Voronoi diagrams in Section 2. Section 3 gives an overview of our approach and highlights the features of graphics hardware used by our algorithm. We present algorithms for computing generalized discrete Voronoi diagrams in Section 4. Section 5 provides a detailed analysis of our algorithms. Section 6 discusses issues in efficient implementation. In Section 7, we use our approximate algorithm to improve the performance of a continuous Voronoi diagram computation algorithm. Based on approximate Voronoi diagrams, we demonstrate its application to motion planning in static and dynamic environments in Section 8.

2 Related Work

The concept of Voronoi diagrams has been around for at least four centuries. In his treatment of cosmic fragmentation in *Le Monde de Mr. Descartes, ou Le Traite de la Lumière*, published in 1644, Descartes uses Voronoi-like diagrams to show the disposition of matter in the solar system and its environment. The first presentations of this concept appeared in the work of [Diric50] and [Voron08]. Although the concept has been around for a long time, algorithms for computing Voronoi diagrams did not start appearing until the 1970's. See the surveys by [Auren91] and [Okabe92] on various algorithms, applications, and generalizations of Voronoi diagrams.

2.1 Voronoi Diagrams of Points

Among the algorithms known for computing Voronoi diagrams of points in 2D, 3D, and higher dimensions are the divide-and-conquer algorithm proposed by [Shamo75] and Fortune's sweepline algorithm [Fortu86]. Numerically robust algorithms for constructing topologically consistent Voronoi diagrams have been proposed by [Inaga92, Sugih94]. A number of implementations in exact and floating-point arithmetic are also available.

2.2 Generalized Voronoi Diagrams

Algorithms have been proposed for constructing Voronoi diagrams of higher order primitives like the lines, polygons, and polyhedral and curved-surface models. Two broad approaches based on incremental and divide-and-conquer techniques have been summarized in [Okabe92]. The set of algorithms includes divide-and-conquer algorithms for polygons [Lee82, Held97], an incremental algorithm for polyhedra [Milen93b], algorithms based on 3D tracing for polyhedral models [Milen93, Sherb95, Culve98], curved primitives [Chian92], and CSG objects [Dutta93, Hoffm94]. In all these cases, the computation of generalized Voronoi diagrams involves representing and manipulating high degree algebraic curves and surfaces and their intersections. As a result, no efficient numerically robust algorithms are known for computing them.

2.3 Approximate/Discrete Voronoi Diagrams

Previously proposed algorithms to compute approximations of generalized Voronoi diagrams are based on sampling points on the surface of the object and computing the Voronoi diagram of the points [Sheeh95]. However, deriving any error bounds on the output of such an approach is difficult, and the overall complexity is not well understood.

[Vleug95] and [Vleug96] have presented an approach that adaptively subdivides the space into regular cells and computes the Voronoi diagram up to a given precision. [Laven92] uses an octree representation of objects and performs spatial decomposition to compute the approximation. [Teich97] computes a polygonal approximation of Voronoi diagrams by subdividing the space into tetrahedral cells. All these algorithms take considerable time and memory for large models composed of tens of thousands of triangles, and cannot easily be extended to directly handle dynamic environments.

The idea of using polygon rasterizing hardware is suggested in the OpenGL 1.1 Programming Guide for displaying Voronoi regions of 2D points [Woo97].

2.4 Graphics Hardware

Polygon rasterization graphics hardware has been used for a number of geometric computations, such as visualization of constructive solid geometry models [Rossi86, Goldf89] and interactive inspection of solids, including cross-sections and interferences [Rossi92]. Algorithms for real-time motion planning using raster graphics hardware have been proposed by [Lengy90].

3 Overview

In this section, we briefly give an overview of generalized Voronoi diagrams, our approach for computing discrete approximations of Voronoi diagrams, and polygon rasterization hardware.

3.1 Generalized Voronoi Diagrams

Let us denote the set of input primitives as $A_1,A_2,\ldots,A_k$. For any point p in the space, let $dist(p,A_i)$ denote the Euclidean distance from the point p to the primitive A_i. Let us define the *bisector* of A_i and A_j by

$$b(A_i, A_j) = \{p \mid dist(p, A_i) = dist(p, A_j)\}$$

and the dominance region of A_i over A_j by

$$Dom(A_i, A_j) = \{p \mid dist(p, A_i) \leq dist(p, A_j)\}$$

For a primitive A_i, we define the Voronoi region for A_i by

$$V(A_i) = \bigcap_{j \neq i} Dom(A_i, A_j)$$

The partition of space into $V(A_1),V(A_2),\ldots,V(A_k)$ is called the *generalized Voronoi diagram*. The (ordinary) Voronoi diagram corresponds to the case when each A_i is an individual point. When the primitives are linear elements (points, lines, polygons), the bisectors are algebraic curves or surfaces.

3.2 Discrete Voronoi Diagrams

To compute a discrete Voronoi diagram, we start with a uniform subdivision of a bounded region of space into rectangular cells. In this bounded region, we approximate the Voronoi diagram by determining, for a sample point in the center of each cell, the closest primitive to the cell and its distance. The resulting diagram is a table of IDs and distance values, one for each cell. All points in the bounded region of space belong to only one cell, so for any point we know the closest primitive and the distance to within half the diagonal length of a cell.

A simple brute-force approach to find the closest primitives to each cell is to iterate through all cells, computing for each cell the distances to all primitives, and recording the closest primitive. The algorithm can be rearranged to iterate through the primitives: for each primitive, check all cells, updating the current closest primitive for each cell. The second arrangement is amenable to an implementation in graphics hardware.

Our approach is inspired by an interesting sidenote in the OpenGL 1.1 Programming Guide [Woo97]. In the Section "Now That You Know" on "Dirichlet Domains", the authors briefly discuss a simple method to construct discretized 2D Voronoi diagrams for points using OpenGL graphics hardware. The authors mention the use of cones for Voronoi diagrams of points in 2D, but warn that the technique "might require thousands of polygons." We show that we can render cones using fewer than 100 polygons for a 1K×1K resolution grid and achieve the same level of accuracy. In addition, we generalize this approach to higher-order primitives in both two and three dimensions.

The main idea of our approach is to render a polygonal mesh approximation to a primitive's distance function. We make use of polygon rasterization hardware to reconstruct distance values to all pixels (cell centers). The distance comparison is performed by the Z-buffer depth test. A pixel will only be updated with a primitive color ID if the depth comparison passes (less than current value). In order to maintain an accurate Voronoi diagram, we bound the error of the mesh to be smaller than a pixel's width.

3.3 The Distance Function

We define the *distance function* with respect to a single primitive over a region of space as the distance from the primitive to all points in the region. For 2D Voronoi diagrams, the region is the entire plane. For 3D Voronoi diagrams, the region is a planar slice of space. In both cases, the distance function is a function of two variables (x,y).

3.4 Polygon Rasterization Hardware

Our approach makes use of standard Z-buffered raster graphics hardware for rendering polygons. The frame buffer stores the attributes (intensity or shade) of each pixel in the image space; the Z-buffer, or depth buffer, stores the *z* coordinate, or depth, of every visible pixel. We assume that the Z-buffer has *l* bits of precision for each pixel (on most current graphics systems, $l \geq 24$). Given only the vertices of a triangle, the rasterization hardware uses linear interpolation to get depth values across the triangle's surface. All raster samples covered by a triangle have an interpolated *z*-value.

We make use of two components of the graphics hardware: linear interpolation across polygons and the Z-buffer depth comparison operation. When rendering a polygonal distance mesh, the polygon rasterization reconstructs all distances across the mesh. The Z-buffer depth test compares the new depth value to the previously stored value. If the new value is less, we update the Z-buffer with the new distance and the pixel with the new color.

4 Discretized Voronoi Diagrams

Our goal is to generate a discrete approximation to the Voronoi diagram for a group of primitives. The polygon rasterization hardware computes the approximation by rendering a three-dimensional *distance mesh* corresponding to each primitive. Rendering the distance mesh is equivalent to computing the distance from a primitive *A* to each pixel location (*x*,*y*). Geometrically, the distance mesh is a piecewise-linear approximation to the graph of the distance function $d=dist((x,y),A)$. Each primitive is assigned a different color, and the corresponding distance mesh is rendered in that color using a parallel projection along the *d*-axis. For each pixel, the Z-buffering hardware automatically selects the color for the primitive with the smallest *d* value—the color of the nearest primitive. In this way, each pixel in the frame buffer will have a color indicating which primitive it is closest to, and the Z-buffer will have the distance to that primitive.

Current graphics systems are optimized to render dense triangular meshes at fast rates using hardware polygon scan-conversion. The exact distance functions for many primitives are non-linear geometric shapes, and to render them on graphics systems, we approximate the distance structures with polygonal meshes. In this section, we present algorithms for meshing the distance function, and derive tight bounds on the error generated by this approximation.

4.1 2D Voronoi Diagrams

The four types of 2D primitives are points, lines, curves, and polygons. Their corresponding distance functions are shown in the following table. In this section, we present algorithms for computing distance meshes for each of them.

2D primitive	Shape of Distance Function	Figure
Point	Right circular cone	1a
Line segment	"Tent"	1b
Curve	Polyline of cones and tents	4
Polygon	Cones and tents	5

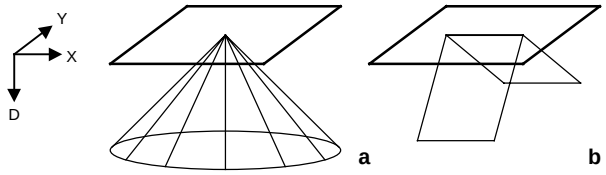


Figure 1: The distance meshes used for a point (left) and a line segment (right). The XY-plane containing the primitive is shown above each mesh.

4.1.1 Points in 2D

The distance function for a point in the plane is a right circular cone. We approximate cones as a triangle fan proceeding radially outward from the apex (Figure 1a). A point's Voronoi region can potentially extend to any portion of the region of interest, and thus the radius at the cone's base must be of size $M\sqrt{2}$ if the scene is contained in an $M \times M$ square. The error in the distance mesh will be greatest at the far edges of the cone; along any radial edge of the triangle fan, the computed distance is correct. To ensure that our cone mesh has at most ε deviation, we examine a single triangle of the fan as viewed from above. The maximum distance of the triangle from the continuous cone is at the center of the triangle's outermost edge. Because this is a right circular cone, the error in approximating the circular base as viewed from above is the same as the error in distance.

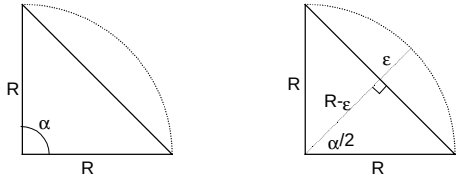


Figure 2: The left image shows a single triangle of the meshed cone. α is the angle we wish to maximize, R is the radius of the cone, and ε is the prescribed max error.

From this formulation (see Figure 2), we compute the maximum angle as:

$$\cos(\alpha/2) = \frac{R-\varepsilon}{R} \;\rightarrow\; \alpha = 2\cos^{-1}\left(\frac{R-\varepsilon}{R}\right)$$

For example, for a maximum distance error of no more than one pixel's width, a cone mesh for a 512×512 grid will require only 60 triangles, and the one for a 1024×1024 grid will require 85 triangles. Bounding the potential Voronoi region of a point to less than the diameter of the scene will reduce the number of triangles and pixel fill-rate required. However, it is difficult to determine a sufficient radius to ensure coverage without knowing the primitive's Voronoi region.

4.1.2 Line Segments in 2D

The distance function for a line segment is composed of three parts: one for the segment itself and one for each endpoint. The endpoints are treated the same way as points. The distance function for the line segment (excluding the endpoints) is just a "tent" (Figure 1b); its distance mesh is composed of two quadrilaterals. These represent the distance function exactly, so there is no error in the distance mesh representation. The only error for the line segment is in the cone mesh for the endpoint distance functions, as described in the previous section.

4.1.3 Curves in 2D

The exact distance function for a curved primitive can be rather complicated, and for Bézier or algebraic curves is a high degree algebraic function. We simplify this by creating a linear tessellation of the curved primitive, and then creating a mesh of the distance function of this approximation. We can use algorithms such as in [Filip87] to obtain bounded-error tessellations. Figure 3 shows the mesh for a Bézier curve. Since the mesh for a linear segment is exact, the distance error for any of the linear segments is just the error in the deviation of the original line to the curve. The endpoints of the curve must be treated as points, just as for the line segment. The distance mesh for the "joints" between linear segments is a portion of the radial mesh of triangles. An overall maximum error bound of ε can be obtained for the entire curve by:

- tessellating the curve into linear segments with maximum error bound of ε;
- rendering the distance mesh for the linear segments; and
- treating the endpoints and joints as points, and rendering each point distance mesh with maximum error bound of ε.

A picture of the generalized Voronoi diagram generated by a curve and five points is given in Figure 3.

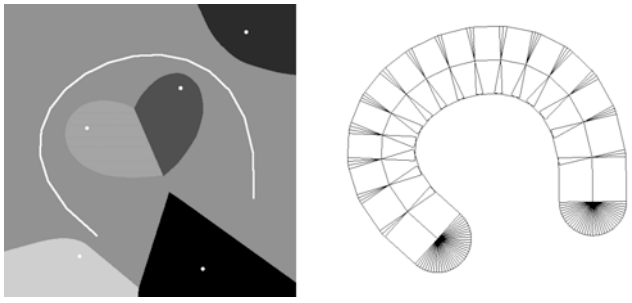


Figure 3: The Voronoi diagram of a Bézier curve and 5 points (left). The distance mesh for the Bézier curve that has been tessellated into 16 segments (right).

4.1.4 Polygons and Per-feature Voronoi Diagrams

It is often useful to consider primitives as a collection of features, rather than as a single entity. For example, a line segment would be considered as three features: the two endpoints and the linear edge between them. By rendering the distance meshes for different features in different colors, we obtain a discrete approximation of a *per-feature Voronoi diagram*. Such diagrams are useful in several contexts: for example, the computation of a medial axis of a polygon. A picture of a per-feature Voronoi diagram for a polygon is given in Figure 4.

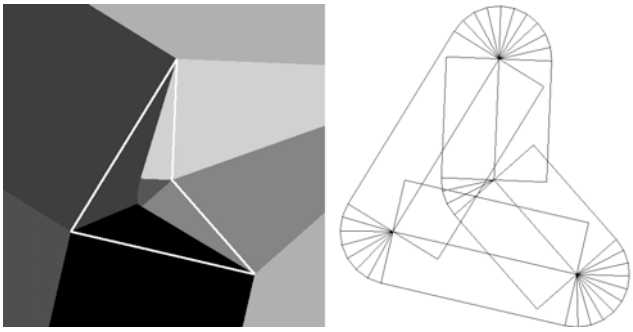


Figure 4: The per-feature Voronoi diagram of a quadrilateral (left). The corresponding distance mesh (right).

Polygons are rendered as a series of linear segments connected at the vertices. Each edge and vertex is a feature. For the vertices, rendering a triangle fan connecting two adjacent edges, rather than a full point distance mesh cone, saves on the total number of triangles computed and ensures that the distance meshes for adjacent features join smoothly. See Figure 4 for an illustration.

4.2 3D Voronoi Diagrams

The discrete Voronoi diagram in three dimensions is computed as a sequence of 2D planar slices. As in the 2D case, the distance function of a primitive is defined over the set of points in a rectangular bounded region of the slice; we approximate this with a polygonal mesh. The distance structure is then rendered in graphics hardware, yielding a slice of the 3D discrete Voronoi diagram in the frame buffer, and a slice of the distance function in the Z-buffer.

In this section, we show how to construct distance meshes for 3D primitives with respect to a 2D planar slice. Our goal is to minimize the number of mesh polygons while ensuring that the error incurred in interpolating the distance function over each tile is within ε. Separate algorithms generate meshes for three primitives: the polygon, the line segment, and the point. As part of pre-computation, all curved patches are tessellated into polygons. For notational convenience, a slice is assumed to be of the form $z=z_0$. We denote the distance function from a primitive to the point (x,y,z_0) in this slice by $dist(x,y)$.

3D primitive	Shape of distance function	Figure
Polygon	Plane	5
Line segment	Elliptical cone	6
Point	1 sheet of a hyperboloid of 2 sheets	7

4.2.1 Polygonal Primitives

The influence of this primitive in 3D is confined to the region formed by sweeping the polygon orthogonally through space, since points outside this region are considered to be closer to an edge or vertex of the polygon. In the slice, this region is a polygon, and $dist(x,y)$ is linear within this region, as illustrated in Figure 5. The distance to the primitive is computed at the vertices of the region, and a distance mesh composed of a single polygon is rendered. No meshing error is incurred. If the polygon intersects the slice, the intersection is computed and the polygon is decomposed into two sub-polygons. Each sub-polygon is treated as above.

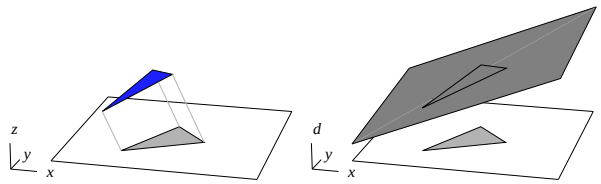


Figure 5: A polygonal primitive and its region of influence in a slice (left). The corresponding linear distance function (right).

4.2.2 Line Segment Primitives

The graph of the distance function for a line segment primitive is an elliptical cone (Figure 6). The apex of the cone lies at the intersection of the segment's line with the slice, and the eccentricity is determined by the relative angle of the line and the slice. The 3D region of influence of a line segment lies between two parallel planes through the endpoints, since a point outside these planes is closer to one of the endpoints than to the segment. The portion of the slice between these two planes is called a "slab."

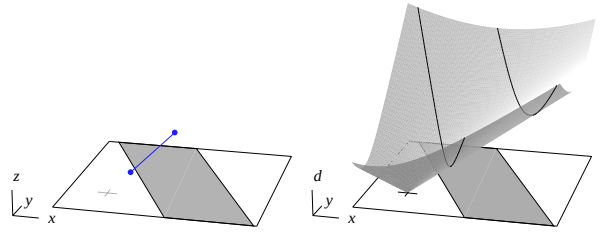


Figure 6: A line-segment primitive and its region of influence in a slice (left). The corresponding conical distance function (right).

A mesh for this slab is shown in Figure 8(left). The distance function is linear along the radial lines containing the cone vertex, so we use these lines in the mesh. The interpolated distance is correct along these lines, and all meshing error occurs between them. Thus the mesh consists of irregular quadrilaterals. The maximum error in each quadrilateral occurs along the *far line* of the slab, so it suffices to analyze the error along this line.

The distance function, restricted to the far line, is simply a hyperbola. The equation is of a simple form: $d = \sqrt{q^2 + t^2}$ where q is the distance from the far line to the line segment primitive, and t is a unit-speed parameter along the line. The hyperbola's shape is determined entirely by the quantity q.

We tessellate this hyperbola with line segments, and this one-dimensional tessellation determines the rest of the mesh. The vertices of this tessellation are placed as far apart as possible, so that the maximum error ε is attained at some point along each edge. This results in a very coarse mesh that can be rendered efficiently. Since the shape of the hyperbola is determined by the single number q, we precompute an optimally coarse tessellation for a family of hyperbolas corresponding to many values of q, and store them in a table. The computation of this table is straightforward, and the details are given in [Anon99]. The error incurred by discretizing q translates directly into error in the distance function, and can be taken into account when choosing ε.

We illustrate the table size using a typical example. Assume that the model's bounding box is the unit cube, so that $M=\sqrt{2}$. For a 1000^3-voxel array, a meshing error of 1/1000 in the distance function is acceptable. Allowing half of this error to be committed by meshing and half to be committed by discretization of q, we set $\varepsilon=1/2000$. In this case, the table has $M/\varepsilon \approx 2800$ rows—that is, a library of about 2800 tessellated hyperbolas. In computing this table, we find that the most densely-tessellated hyperbolas have 36 vertices (by symmetry, only 18 need be stored). The storage required for this table is about 280 kilobytes, for a system which is not otherwise memory-intensive. A Mathematica program can precompute this table in 20 minutes. The table is reusable for any model with the same ratio M/ε.

The clipping effect illustrated in Figure 8(left) is accomplished within the context of the table, and allows fewer, smaller quadrilaterals to be rendered (again, see [Anon99] for details). If the line segment intersects the slice, it is broken into two segments, each of which is treated as above.

4.2.3 Point Primitives

The distance function for a point primitive is shown in Figure 7. Its graph is one sheet of a hyperboloid of revolution of two sheets. The slice is meshed into quadrilaterals by radial lines and concentric circles, as shown in Figure 8(right).

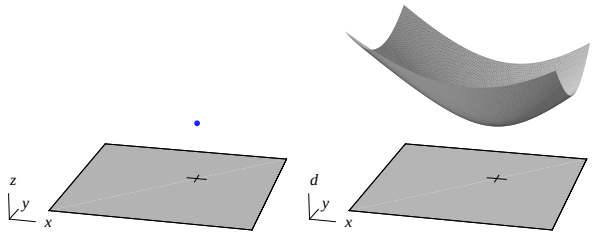


Figure 7: A point primitive and its region of influence in a slice (left). The corresponding hyperbolic distance function (right).

As in the line-segment case, a coarse mesh with bounded error is constructed by table lookup. The radial lines are spaced by an angle α which is chosen once and for all (we use $\pi/12$). The radii of the concentric circles are stored in a table indexed by the distance q from the point to the slice. The maximum error of ε is attained at some point between each pair of adjacent circles. The table's size and method of computation are very similar to those in the line-segment case. See [Anon99] for details.

If the point lies in the slice, the distance function is a cone rather than a hyperboloid. In this case, and when the point is close to the slice, a simple cone is drawn as in the 2D case.

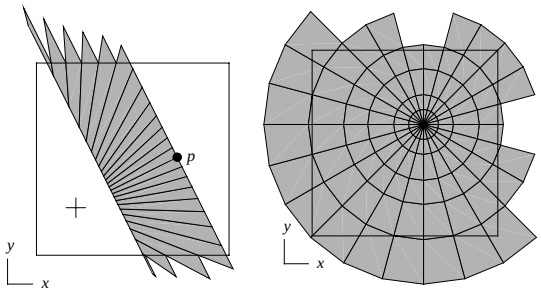


Figure 8: A bounded-error distance mesh for the line-segment primitive (left) and the point primitive (right).

4.3 Weighted and Farthest Voronoi Diagram

A *weighted* Voronoi diagram is one for which the distance function is an additively or multiplicatively weighted distance function (see [Okabe92]). Additive weights can be incorporated by simply translating the entire distance mesh along the d axis. Multiplicative weights are incorporated in our approach by linearly scaling the distance mesh along the d axis. In 2D, this is equivalent to changing the angle of the cone or tent. Keep in mind that scaling the distance mesh along d also scales the meshing error.

In a *farthest site* Voronoi diagram, the farthest primitive from each point is found. We can compute a farthest site Voronoi diagram

with our approach by using the previously defined distance mesh reflected about the distance function domain plane and then renormalized. The mesh will monotonically *decrease* in *d* as you move away from the primitive. For example, when forming the cones for 2D points, the apex of the cone should be the farthest point, and the edges of the cone the nearest point. Both weighted and farthest site methods work in 2D and 3D.

4.4 Voronoi Boundaries and Neighbors

In this section, we present algorithms to compute Voronoi boundaries and neighbors from the discretized representation. We present first a brute-force approach, followed by an efficient algorithm that takes advantage of spatial coherence. These algorithms compute a discrete approximation of the Voronoi boundary.

The brute-force approach simply examines each pair of adjacent cells in 2D or 3D. If the colors are different, the location is registered as a point on a Voronoi boundary. In 2D, there are four types of edges between adjacent pixels: vertical, horizontal, positive-slope, and negative-slope. Each of these edges is associated arbitrarily with one of its endpoints: the upper, left, upper-right, and upper-left pixels, respectively. The brute-force search considers each pixel and examines the four edges associated with it, and registers a boundary point there if one of these four edges spans different colors. This avoids checking each adjacent pair twice. This approach extends to 3D, where there are 13 types of edges instead of four.

The more efficient approach uses a *continuation* method that finds the boundary locations by looking only at locations near known boundaries. The overall approach is output sensitive. It examines a number of pixels which is at most a constant factor times the number of boundary locations it finds. The correctness of the continuation method depends on whether the Voronoi diagram is connected. The generalized Voronoi diagram of a collection of convex sites is always connected, so the method is correct for inputs consisting of point, line-segment, or convex polygonal primitives in a polyhedral model. The method may fail in the presence of curves or curved-surfaces, where the generalized Voronoi diagram may have isolated components.

In this approach, at least one boundary point must be known as a "seed" value. Assuming convex sites, some Voronoi boundary must pass through the edge of the bounding box, so the method begins by examining every pixel along the edge of the discrete Voronoi diagram. When all Voronoi boundaries are connected, as is often the case in 2D, only one seed point is needed since all others can be reached from that first point.

The method starts from a seed point and proceeds by comparing a pixel to its four (or 13 in 3D) associated neighbors. If it differs from one of these, it is registered as a boundary, and its eight (or 26 in 3D) immediate neighbors are marked for consideration and visited recursively. If it is not a boundary pixel, it is discarded.

4.4.1 Finding Voronoi neighbors

Some applications need to know the pairs of primitives whose Voronoi regions are adjacent. This information is generated during the boundary-extraction procedure described above, but each neighbor-pair is noticed many times. However, computing the Voronoi neighbors is a slightly more expensive procedure, because the neighbor relationships must be maintained in a search structure. In Section 6, an alternative approach to neighbor-finding is presented, which allows better control over the error.

5 Sources of Error

In this section we analyze our algorithm, enumerate all sources of error, and present efficient techniques to improve accuracy. We consider two broad categories: error in distance approximation and combinatorial error.

5.1 Distance Error

When rendering the distance mesh for each object, the distance computed at each pixel is assumed to be that from the center of that pixel to the primitive. The distance error can occur in the distance computation at one of those pixels when rendering one distance mesh. There are three sources of distance error:

- *Meshing error*, introduced by approximating the distance function by the distance mesh. We discussed bounds on this error in Section 4.
- *Tessellation error*, introduced when tessellating a curved primitive by a number of linear primitives. The tessellation algorithms presented in [Filip87] give tight bounds.
- *Hardware precision error* arises due to the use of fixed-precision arithmetic (integer or floating-point) during rasterization.

These errors are additive and affect the accuracy of discretized Voronoi diagrams. That is, the error from one source is not magnified by the other sources. The total distance error is at most the sum of the errors from these three sources.

To reduce the error in distance approximation, we reduce the error due to each of these sources. We have shown in Section 4 how to ensure that the meshing error is within any predetermined bound. Tessellation error can be reduced by using a finer approximation to the primitive. Hardware precision error cannot be removed without resorting to multiple-precision arithmetic, but hardware error is usually negligible compared to meshing error.

5.2 Combinatorial Error

Combinatorial error refers to the error that is qualitative rather than quantitative. For example, a pixel may be assigned the wrong color, or the algorithm reports a pair of incorrect Voronoi neighbors. There are three sources that contribute to combinatorial error:

- *Distance error*, as described in the previous section, can cause pixels to be colored incorrectly. If there is significant error in the distance computed, depth comparison at that pixel may result in incorrect visibility determination.
- *Resolution error* occurs when the cell grid is not fine enough. The raster is a sampling of space. If this sampling is too coarse, we may miss some Voronoi neighbors or find spurious neighbors. Techniques dealing with resolution error are described below.
- *Z-buffer precision error* refers to the limitations of the number of bits of precision provided by the Z-buffer. Current graphics systems have 24 bits or 32 bits precision for each pixel in the Z-buffer, which is more than the 23 bits provided in standard floating-point. If the distances between two pixels cannot be determined within that precision, the Z-buffer cannot accurately choose the correct color. This effect is relatively small compared to the other two, but can be significant at very high resolutions with very little distance error. A higher-precision Z-buffer can be simulated in software at a significant loss in efficiency.

The basic way to decrease resolution error is to adaptively compute the approximate Voronoi diagram to a higher resolution. We use *adaptive resolution* to "zoom in" on a region of interest. This involves identifying a window of interest, which may be

arbitrarily small, and applying the appropriate linear transformation for zooming into that region. Figure 9 shows an example.

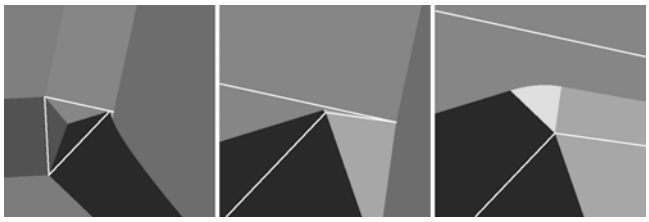


Figure 9: Adaptive resolution allows us to zoom in on features that could otherwise be missed.

Resolution error can cause a number of combinatorial problems, such as missing the entire Voronoi region of a primitive. One such example is shown in Figure 10. If none of the cells has the color of a particular primitive, we separately render the primitive itself, computing the pixels covering that primitive. By zooming around those pixels, we will find pixels in the Voronoi region. The same technique can be applied to cells in 3D. Another problem arising from resolution error, also shown in Figure 10, is incorrectly finding Voronoi neighbors. This problem (when due solely to resolution error) can be resolved by adaptively zooming in on just the boundary pixels.

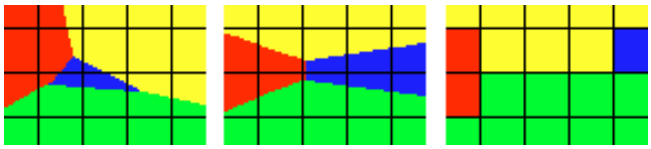


Figure 10: Problems caused by resolution error. An entire region in the center will be missed since it does not hit any pixel centers (left). The left and right regions, which should meet (middle), become disconnected after rasterization(right).

5.3 Error Bounds

In this section we will derive an exact bound on the accuracy of our algorithm. For this analysis, we will assume that there is no Z-buffer precision error. Assume that we can bound the maximum distance error by ε, as described earlier. For a pixel *P* colored with the ID of primitive *A* and with a computed depth buffer value of *D*, we know that:

$$D - \varepsilon \le dist(P, A) \le D + \varepsilon$$

Furthermore, we know that for any other primitive *B*,

$$D - \varepsilon \le dist(P, B)$$

From this information, we easily determine that

$$dist(P, A) \le dist(P, B) + 2\varepsilon$$

where $dist(X,Y)$ means the distance from the center of pixel *X* to primitive *Y*. That is, if a pixel is colored *A*, the corresponding primitive is no more than 2ε farther from the pixel center than any other primitive. The same bound also works for 3D.

6 Implementation

The 2D and 3D systems were both implemented in C++ using the OpenGL graphics library and the GLUT toolkit on high-end SGI workstations and low-end PC hardware. Any graphics API specification that uses a standard Z-buffered interpolation-based raster graphics system that supports the ability to read back entire color and depth buffers is sufficient to support the Voronoi computation.

Our system runs, without source modification, on both an MS-Windows based PC and a high-end SGI Onyx2 with InfiniteReality Graphics. Surprisingly, the performance on a 400 Mhz Intel Pentium II PC with an Intergraph Intense 3D Pro 3410-T graphics accelerator was comparable to the SGI performance. In fact, in boundary finding, neighbor finding, and particle motion planning applications, the performance exceeded the high-end SGI. This was mainly due to intense buffer readback requirements. For large numbers of input primitives, performance is bound by the graphics hardware's pixel fill-rate, and the SGI outperforms the PC.

In most of our 2D test examples, we used distance meshes that cover the entire screen. However, in many practical cases, a mesh covering only part of the screen is sufficient. Knowing a maximum radius for the Voronoi regions increases performance significantly. We exploit this observation in the 10,000-point example seen in the video and the 1,000-point example shown in Plate 1.

In our motion planning algorithm, we demonstrate interactive Voronoi diagram construction of a house floorplan composed of approximately 100K triangles. The performance could be improved by bounding the maximum distance, using coaser meshes, and considering only the internal Voronoi diagram (only one-half of all distance meshes). In addition, we could have used the silhouette boundaries of the dynamic objects as input primitives.

In 3D, we have not yet fully implemented the meshing strategies described in Section 4.2. Instead, we show a prototype implementation that uses uniform meshing and brute-force distance evaluation from each site to each 3D cell. With this initial implementation, we obtain interactive frame rates for a slice computation with hundreds of points and triangles. For large polygonal models, we compute per-feature Voronoi regions in a non-interactive preprocessing step. These models consist of around 2000 triangles (~6000 features) each, and the entire Voronoi volume was computed, in several hours, using the brute-force method for a 256x256x256 subdivision. Examples of this are seen in Color Plate 3 and the video. These volumes are used to show the correctness of the method and the need for massive speedups. We have given an analysis that shows the correctness of a mesh-based approach to achieve the same results as the brute-force approach; in addition, we clearly indicate the performance potential of adaptive meshing and graphics hardware. We speculate that the performance will be great enough to show slices of the same models interactively.

7 Continuous Voronoi Diagrams

The methods presented in this paper so far are all fundamentally discrete. For linear primitives, the continuous Voronoi diagram includes parabolic arcs in 2D and quadric surfaces in 3D. This section shows how the discrete Voronoi diagram can be used to speed up continuous Voronoi algorithms which are capable of manipulating the necessary curved objects. We restrict our attention to the problem of computing the internal part of the Voronoi diagram of a closed polygon or polyhedron, which is closely related to the medial axis.

7.1 Graph-Traversal: A Continuous Algorithm

The internal Voronoi diagram may be computed by a *graph-traversal algorithm*, described in 3D by Milenkovic [Milen93]. The algorithm traces out each curve in the Voronoi diagram by taking small steps. Such a curve is the set of points equidistant from three sites (primitive), called the *governors* of the curve. The tracing stops when a point is reached that is equidistant from a fourth site. The algorithm then forks, tracing each of the other curves (generically, there are three) incident at that vertex. The overall running time of this algorithm is $O(nm)$, where n is the input size (the number of sites) and m is the output size (the

number of curves). This time is a factor of n away from optimal $O(m)$ time since for each curve, every site must be considered as a candidate for the fourth site; a linear search through the entire input occurs at each step. All existing practical algorithms for the 3D internal Voronoi diagram (or medial axis) are based on either this graph-traversal algorithm, a discretization of the polyhedron, or a discretization of space.

We propose a preprocess for the graph-traversal algorithm, based on the discrete Voronoi diagram, which significantly reduces the total running time. The search for the fourth site can be limited to those sites which are Voronoi neighbors of all three governors of the curve. The graph-traversal algorithm can thus be sped up if it is provided with a list, for each site, of other sites which are potential Voronoi neighbors. The list must contain all actual neighbors.

The discrete Voronoi diagram is examined to produce a superset of the neighbor set for each site. The process is similar to the brute-force neighbor-finding algorithm in Section 4.4, but takes special care not to miss any neighbors. The error in the neighbor graph computed by this new method can be systematically eliminated by the graph-traversal algorithm.

7.2 A Superset of the Voronoi Neighbors

One key assumption is made by this algorithm: If a pixel's (or voxel's) color is the same as that of its eight (or 26) neighbors, then that pixel (or voxel) is colored correctly. This assumption holds if all of the Voronoi boundaries are linear; a similar theorem is proved in [Vleug95]. In the case of our 2D discrete Voronoi diagram, the assumption can fail for two reasons. First, the pixel may actually be on the other side of a parabolic boundary from its eight neighbors. This is extremely rare, and is a result of resolution error. Second, there may be an entire region where the difference in distance between the closest and second-closest sites is less than ε, the total prescribed distance error. This problem actually does not arise in the context of interior Voronoi diagrams of polygons, however, because two sites are very near each other only if they are adjacent in the polygon (and are therefore Voronoi neighbors in any case), or if they oppose each other across a narrow region of the polygon (easily detected, and resolved by zooming).

A pixel which differs from one of its neighbors is called a *white pixel*, because it is treated as though it could be in the Voronoi region of any site.

The basic idea of the algorithm is to extend each Voronoi region in turn over the white pixels until it reaches other colors, and then record the other colors reached. Figure 11 illustrates the algorithm for a single region in 2D. First, the discrete Voronoi diagram is computed, but with a slight modification: the exterior parts of the distance mesh are all rendered in black, ensuring that all pixels outside the polygon are colored black.

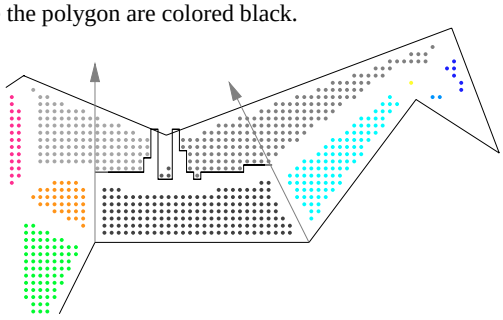


Figure 11: Starting from the horizontal edge, a Voronoi region grows upward. The growth stops whenever another Voronoi region is reached.

The second step is to consider each site (the concave vertices and all of the edges count as sites) to find a superset of its Voronoi neighbors. Starting from the site itself and moving away, the pixels in its region of influence are searched until the region is curtailed either by pixels associated with other sites, or the black exterior pixels. The region can be searched line-by-line, as in scan-conversion. White pixels are marked white only as they are discovered. The three-dimensional generalization of this algorithm is straightforward.

The algorithm examines the white pixels many times; most colored pixels are examined once. While inefficient compared to the boundary-finding algorithms in Section 4.4, this method can lend significant speedups to the graph-tracing algorithm in two or three dimensions, and perhaps to other continuous algorithms.

8 Application to Motion Planning

Motion planning is one of the fundamental problems in computational geometry and robotics. Most earlier work has focussed on the Piano Mover's problem, which can be stated as the following: Given a robot R and an environment E composed of obstacles, find a collision-free path from an initial configuration I to a final configuration F. Besides robotics, this problem also comes up in motion control and planning of digital actors or autonomous agents in computer animation, maintainability studies in virtual prototyping, and robot-assisted medical surgery. This problem has been well studied for more than two decades and a number of algorithms have been proposed, most of which can be classified into global or local methods. Some of the well-known approaches include roadmap algorithms, exact/approximate cell-decomposition, potential field methods, and other variations [Latom91].

Several algorithms have been proposed based on generalized Voronoi diagrams [Latom91]. The underlying idea is that the boundaries or skeleton curves of generalized Voronoi diagrams provide paths of maximal clearance between the robot and the obstacles. Due to the practical complexity of computing generalized Voronoi diagrams, the applications of such planners have been limited to environments composed of a few primitives.

Our discrete Voronoi computation algorithm can be applied to motion planning in static and dynamic environments. Our Voronoi algorithm computes the approximate distance function for each cell. Based on this information, it can easily locate the nearest obstacle and the distance to it quickly.

To illustrate the application of our algorithm, we have implemented a simple motion planner using our system for computing generalized Voronoi diagrams. We demonstrate its effectiveness in a rather complex environment (corresponding to the interior of a house) composed of over 100,000 polygons for both a static scene and a dynamic scene with several moving obstacles. We use the X and Y components of the polygons to give the 2D input primitives for our Voronoi diagram. The robot has three degrees of freedom: X and Y translation along the ground and rotation about the Z axis.

Our basic approach is based on the potential field method, which repels a robot away from the obstacles and towards the goal using a well-designed artificial potential function. One possible potential function $U_{art}(x)$ of the robot at the configuration x is:

$$U_{art}(x) = \frac{a}{\min_i D(O_i, R(x))^2} + b \times D(G, R(x))^2$$

where $D(O_i,R(x))$ is the shortest distance between an obstacle O_i and the robot R and $D(G,R(x))$ is the distance between the goal G and the robot R, a and b are adjustment constants. The major components of our planner include:

1. Using our Voronoi diagram computation algorithm to get the distance measurement at every grid point over the entire scene. The distance buffer was computed once for the entire house. In the dynamic scene, the distance buffer for the static part of the environment (e.g. the walls and stationary furniture) was computed once, and the distances between the "robot" and the dynamic "obstacles" (e.g. the moving furniture) are computed during each frame. We note that even computing the distance buffer for all polygons every frame had only a small effect on the frame rate.
2. For ease of implementation, our planner takes several sample points for the robot. These sample points are used to determine the center of mass and moment of inertia.
3. At each stage, we compute a force on each of the sample points. The force is a combination of the attractive force toward the sub-goals and a repulsive force based on the distance to the nearest object (obtained from the distance or depth buffer). We use bilinear interpolation of the discrete distance values to determine the exact distance, gradient in X (δX), gradient in Y (δY), and thus the gradient of the potential function for any point in the environment.
4. We decompose each force into two components: one acting toward/away from the center of mass and the other one in the orthogonal direction that contributes to the torque necessary to rotate the robot. Moreover, we sum up all the torque and force components from all sample points to obtain the overall force and torque acting on the center of mass.
5. We apply the force and torque to the target object and move it step by step. To ensure stability of our computation, a bound on the maximum linear velocity and maximum angular velocity is established and a damping effect is placed on the linear velocity.

Color Plate 2 shows a sequence of motions generated by our motion planner in a static environment (as shown in the video). The piano's motion is automatically generated by the planner using nine subgoals. The color plate also shows an image of the discrete Voronoi diagram for the house.

Due to fast and reliable Voronoi diagram computation, it is also possible to apply this technique to environments with moving obstacles. Our video demonstrates the movement of a music stand through a house filled with moving furniture. The music stand has no prior knowledge about the movement of other pieces of furniture. The motion planning and the motion sequences are executed in real time, with the distance buffer dynamically computed on the fly.

9 Conclusions and Future Work

We have presented a method for rapidly finding the generalized Voronoi diagram in two and three dimensions, using graphics hardware. We have presented techniques for creating a mesh of the distance function for each primitive with bounded error, and described how this distance mesh lets us compute the Voronoi diagram rapidly. We have analyzed all sources of error, as well as how to bound or reduce the major sources of error. Finally, we have described how our approach can be used to improve the efficiency of two other applications: exact medial axis computation and motion planning.

Although we have not completed an optimized implementation of the 3D meshing discussed, we have shown a bounded-error meshing strategy that will achieve the same results as the brute force method implemented, but at interactive rates. In the very near future, we plan to complete implementation of this meshing to demonstrably prove this claim. In addition, we will apply the improved method to much larger models. We believe that the improvements will be significant enough to allow entire volume computation of high-resolution 3D Voronoi diagrams of very large medical and CAD data sets.

Other areas of our planned future work include further applications and additional acceleration techniques.

References

[Anon99] Anonymous Technical Report. 1999.

[Auren91] F.Aurenhammer.*Voronoi diagrams: A survey of a fundamental geometric data structure*. ACM Comput. Surv., 23:345–405, 1991.

[Chian92] C. –S. Chiang. *The Euclidean distance transform*. Ph. D. thesis, Dept. Comput. Sci., Purdue Univ., West Lafayette, IN, August 1992. Report CSD-TR 92–050.

[Culve98] T.Culver, J.Keyser, and D.Manocha. *Accurate computation of the medial axis of a polyhedron*. Technical Report TR98-034, Department of Computer Science, University of North Carolina, 1998.

[Dutta93] D.Dutta and C.M. Hoffmann. *On the skeleton of simple CSG objects*. Journal of Mechanical Design, ASME Transactions, 115(1):87–94, 1993.

[Diric50] G.L. Dirichlet. *Uber die reduktion der positiven quadratischen formen mit drei unbestimmten ganzen zahlen*. J. Reine Angew. Math., 40:209–27, 1850.

[Filip87] D.Filip and R.Goldman. *Conversion from Bézier-rectangles to Bézier-triangles*. CAD, 19:25–27, 1987.

[Fortu86] S.Fortune. *A sweepline algorithm for Voronoi diagrams*. In Proc. 2nd Annu. ACM Sympos. Comput. Geom., pages 313–322, 1986.

[Goldf89] J. Goldfeather, S. Molnar, G. Turk, and H. Fuchs. *Near real-time CSG rendering using tree normalization and geometric pruning*. IEEE Computer Graphics and Applications, 9(3):20–28, May 1989.

[Held97] M. Held. *Voronoi diagrams and offset curves of curvilinear polygons*. Computer-Aided Design, 1997. To appear.

[Hoffm94] C.M. Hoffmann. *How to construct the skeleton of csg objects*. In A.Bowyer and J.Davenport, editors. Proceedings of the Fourth IMA Conference, The Mathematics of Surfaces, University of Bath, UK, September 1990. Oxford University Press, New York, 1994.

[Inaga92] H.Inagaki, K.Sugihara, and N.Sugie. *Numerically robust incremental algorithm for constructing three-dimensional Voronoi diagrams*. In Proc. 4th Canad. Conf. Comput. Geom., pages 334-–339, 1992.

[Latom91] J.C. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, 1991.

[Laven92] D. Lavender, A. Bowyer, J. Davenport, A. Wallis, and J. Woodwark. *Voronoi diagrams of set-theoretic solid models*. IEEE Comput. Graph. Appl., 12(5):69–77, September 1992.

[Lee82] D.T. Lee. *Medial axis transformation of a planar shape*. IEEE Trans. Pattern Anal. Mach. Intell., PAMI–4:363–369, 1982.

[Lengy90] J. Lengyel, M. Reichert, B. R. Donald, and D. P. Greenberg. *Real-time robot motion planning using rasterizing computer graphics hardware*. In Forest Baskett, editor, Computer Graphics (SIGGRAPH '90 Proceedings), volume24, pages 327–335, August 1990.

[Milen93] V.Milenkovic. *Robust construction of the Voronoi diagram of a polyhedron*. In Proc. 5th Canad. Conf. Comput. Geom., pages 473–478, 1993.

[Milen93b] V.Milenkovic. *Robust polygon modeling*. Comput. Aided Design, 25(9), 1993. (special issue on Uncertainties in Geometric Design).

[Okabe92] A. Okabe, B. Boots, and K. Sugihara. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*. John Wiley & Sons, Chichester, UK, 1992.

[Rossi92] J. Rossignac, A. Megahed, and B. Schneider. *Interactive inspection of solids: Cross-sections and interferences*. In EdwinE. Catmull, editor, Computer Graphics (SIGGRAPH '92 Proceedings), volume26, pages 353–360, July 1992.

[Rossi86] J.R. Rossignac and A.A.G. Requicha. *Depth-buffering display techniques for constructive solid geometry*. IEEE Computer Graphics and Applications, 6(9):29–39, 1986.

[Sheeh95] D.J. Sheehy, C.G. Armstrong, and D.J. Robinson. *Computing the medial surface of a solid from a domain Delaunaytriangulation*. In Proc. ACM/IEEE Symp. on Solid Modeling and Applications, May 1995.

[Shamo75] M.I. Shamos and D.Hoey. *Closest-point problems*. In Proc. 16th Annu. IEEE Sympos. Found. Comput. Sci., pages 151–162, 1975.

[Sugih94] K.Sugihara and M.Iri. *A robust topology-oriented incremental algorithm for Voronoi diagrams*. Internat. J. Comput. Geom. Appl., 4:179–228, 1994.

[Sherb95] E. C. Sherbrooke, N. M. Patrikalakis, and E. Brisson. *Computation of the medial axis transform of 3D polyhedra*. In Solid Modeling, pages 187–199. ACM, 1995.

[Teich97] M.Teichmann and S.Teller. *Polygonal approximation of Voronoi diagrams of a set of triangles in three dimensions*. Technical Report 766, Laboratory of Computer Science, MIT, 1997.

[Vleug95] J. Vleugels and M. Overmars. *Approximating generalized Voronoi diagrams in any dimension*. Technical Report UU-CS-1995-14, Department of Computer Science, Utrecht University, 1995.

[Vleug96] J.Vleugels, V.Ferrucci, M.Overmars, and A.Rao. *Hunting Voronoi vertices*. Comput. Geom. Theory Appl., 6:329–354, 1996.

[Voron08] G.M. Voronoi. *Nouvelles applications des paramètres continus à la théorie des formes quadratiques. deuxième Mémoire: Recherches sur les parallélloèdres primitifs*. J. Reine Angew. Math., 134:198–287, 1908.

[Woo97] M.Woo, J.Neider, and T.Davis. *OpenGL Programming Guide, Second Edition*. Addison Wesley, 1997.

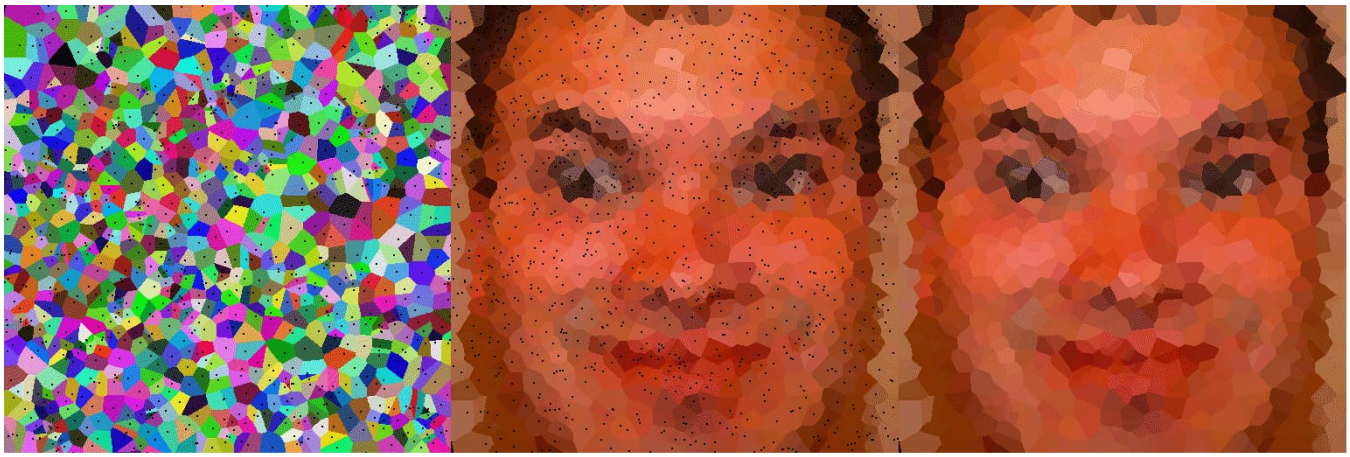


Plate 1: We can use the Voronoi diagram to create real-time mosaic effects. Left: Voronoi diagram computed for 1000 2D point sites at a resolution of 512x512. Center: Same diagram using the point locations to index into a 256x256 face texture to obtain the region colors. Right: The final effect at 512x512 with points removed. In the video, we show this same effect in real-time with 10,000 points.

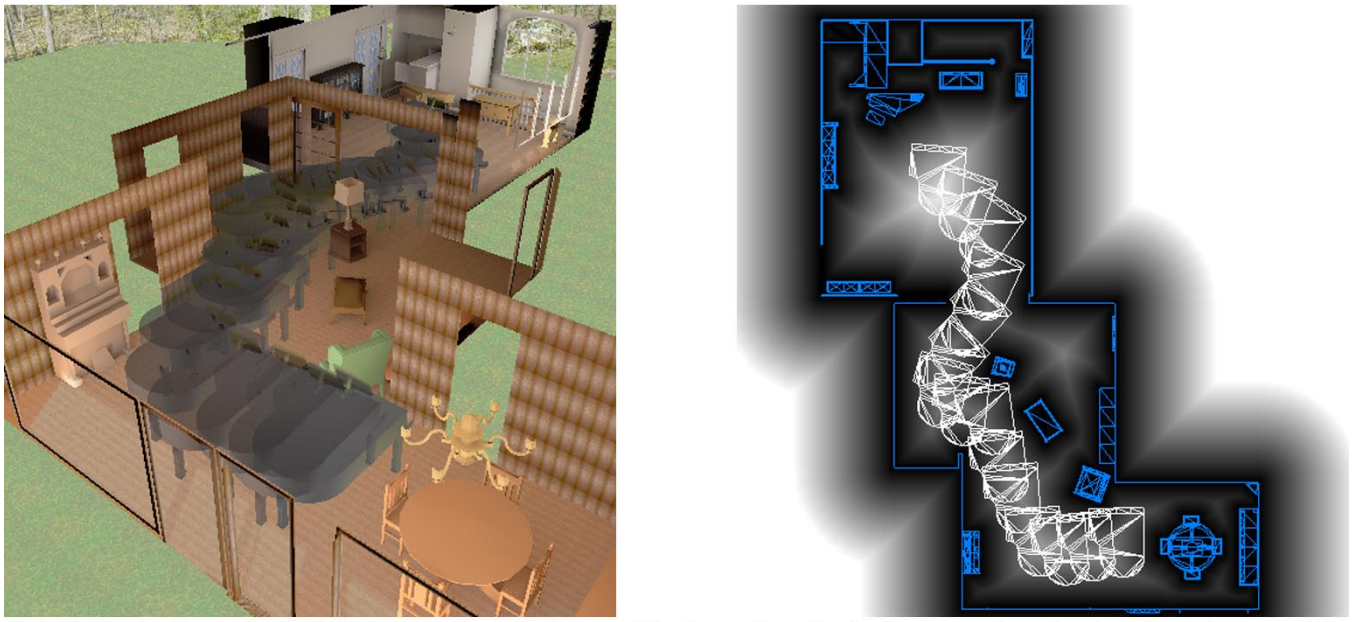


Plate 2: We create a potential field from the Voronoi diagram of the house floorplan to plan the motion of a piano through a complex static scene in real-time. Left: The piano motion sequence. Right: Motion sequence overlaying the floorplan distance function. The video demonstrates navigation through a dynamic scene in real-time.

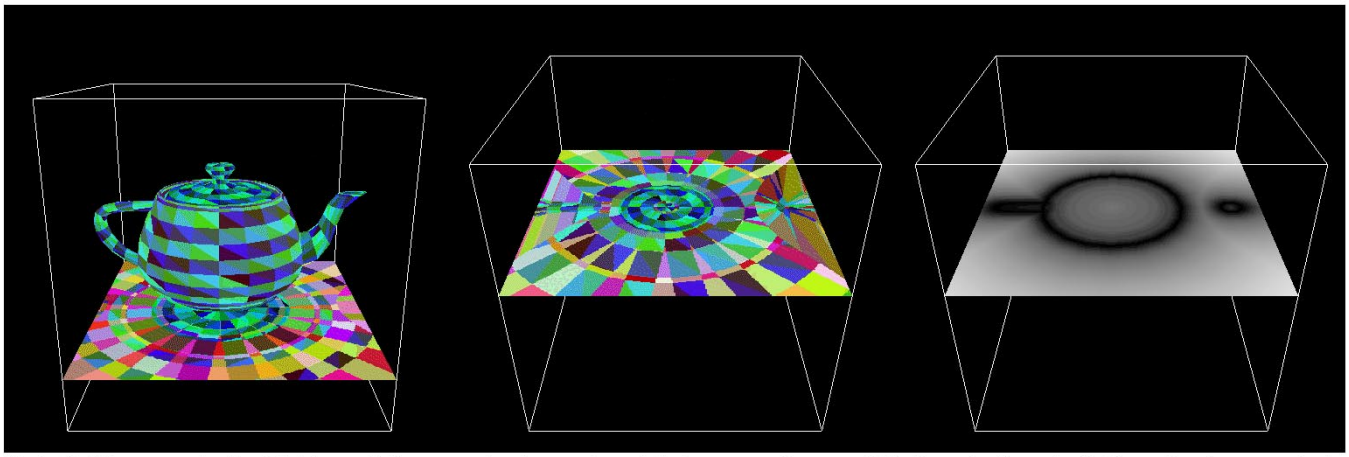


Plate 3: We compute the 3D Voronoi diagram of polygonal models by computing one 2D slice at a time. Left: One slice is computed just below the teapot. Center: Complete slice halfway through. Right: Distance image. The colors indicate the Voronoi regions for face edges, and vertices. Only the face colors are shown on the model.