Construction of Voronoi Diagrams

David Pritchard

April 9, 2002

1 Introduction

Voronoi diagrams have been extensively studied in a number of domains. Recent advances in graphics hardware have made construction of these diagrams relatively cheap, for a limited set of applications. In this paper, we discuss the implementation of two Voronoi diagram construction algorithms. The first uses an algorithm introduced in [HKL+99], which finds a Voronoi diagram using standard polygon rasterization hardware. The second algorithm is the classic $O(n \log n)$ sweepline approach introduced by Fortune. We restrict the problem to only consider point sites, not arbitrary line or polygonal sites. The goal is to compare construction times and implementation issues for these two techniques.

2 Hoff's Algorithm

Implementation of the hardware-accelerated algorithm from [HKL+99] proved quite straightforward. The approach is every bit as simple as the paper describes, and required only about two hours to implement. The algorithm was written in C++ using the OpenGL graphics library, and runs on Linux-based PCs. The core algorithm required 300 lines of code.

As described in the paper, the graphics hardware is set up with an orthographic projection where the viewer is at $z = -\infty$ looking in the positive z direction. Each site is rendered as a right cone, with the tip positioned in the $z = 0$ plane, and the circular base in the $z = 1$ plane. Each cone is shaded uniformly with a unique colour. The graphics hardware uses a depth buffer to determine which cone is closest to the viewer at each pixel, leaving a discretized representation of the Voronoi diagram in the framebuffer. For a more complete description of this interpretation of the Voronoi diagram, refer to [GS87].

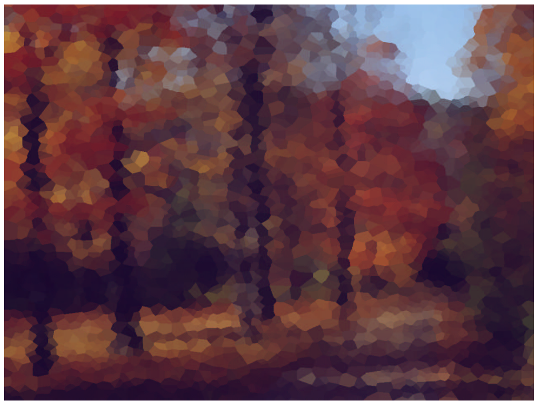
Figure 1: An example of the mosaic effect.

From there, it is easy to perform point-in-site tests, which can be done in $O(1)$ time, although typical graphics hardware often has a high cost associated with framebuffer readback. Other operations which are amenable to solution with graphics hardware can also be done with reasonable efficiency—for example, the mosaic technique described by [HKL+99]. An example of this is shown in Figure 1.

For some applications of Voronoi diagrams, however, this discrete representation is clearly unsuitable. For example, if site connectivity information is needed, then the discrete plot must be scanned using image analysis techniques to obtain a conventional planar subdivision data structure. The time required for such an operation is much larger than the construction of the discrete diagram. It would be interesting to see how such an approach would compare to a pure software algorithm, but this is beyond the scope of this project.

As observed by [HKL$^+$99], graphics fillrate is usually the bottleneck in the performance of this algorithm. In situations where an upper bound on the distance between points is known, the radius of the cone base can be reduced, which consequently reduces the number of pixels that the graphics card must fill for each site. For example, if the sites are chosen from a uniform random distribution, then the maximum distance between points is likely to be reasonably small, saving substantial time. From my observations, this is the single largest parameter affecting the algorithm's speed. For a nonuniform random distribution of points, it could be valuable to adjust the cone size according to the probability density function.

3 Fortune's Algorithm

The software implementation of the Voronoi diagram proved much more challenging than anticipated. The algorithm follows the procedure given in [dBvKOS00] chapter 7, which provides a high-level overview of the necessary work. Implementation was done in C++ on a Linux-based PC. Approximately fifty hours were spent implementing and debugging the software implementation. The code directly related to the Fortune's algorithm totalled 1600 lines.

3.1 Data Structures

The C++ Standard Template Library provides a number of standard containers for structures such as linked lists, arrays, sorted lists and heaps. It also provides a few tree structures targeted at specific applications, such as the set and associative array structures. Some of these could be adapted for use in this project, but many new data structure classes needed to be written to accomodate my needs. The planar subdivision representing the Voronoi digram was stored in a doubly-connected edge list, as described in [dBvKOS00] chapter 2. In a postprocessing step, the infinite edges of the Voronoi diagram were truncated using a surrounding box.

The beach line was stored in a binary tree, as described by [dBvKOS00]. Since both internal nodes and leaves need to hold data, the STL map structure (built upon a red-black tree) was not a feasible choice. Instead, a new binary tree structure had to be written from scratch. At present, this has not yet been move over to a balanced binary tree format, so the algorithm runs in $O(n^2)$ time, and not the desired $O(n \log n)$ time.

The priority queue must be stored in a data structure allowing efficient extraction of the minimum, but also efficient insertion and deletion. The STL set structure was chosen for this, since it supports the necessary operations in $O(\log n)$ time.

3.2 Numerical Methods

A few numerical routines were also needed by the algorithm. Circle events require the determination of the position and size of the circle containing three points. This fit operation was performed by taking the intersection of two bisectors, which proved adequate for the needs of the algorithm.

Calculation of the breakpoint between arcs required a quadratic root solver. This is not difficult in principle, but numeric stability proved a little more involved than anticipated.

3.3 Breakpoint Convergence

The algorithm presented by [dBvKOS00] is straightforward, except for a few steps. The authors gloss over the details of detecting whether or not two breakpoints converge. For any triple of sites, there are many different ways in which a circle event could potentially be added to the event queue. For example, suppose that the leaves of the beach front are $< a, b, c, b, a >$ after site c is inserted. Then, two different triples involving the same three sites are considered for insertion into the event queue: $< a, b, c >$ and $< c, b, a >$. If both are inserted, then both may later be extracted from the queue and a duplicate vertex could be inserted into the planar subdivision, causing a breakdown in the algorithm.

In fact, one triple corresponds to the converging breakpoints, and the other triple corresponds to diverging breakpoints. This convergence must be detected at insertion time, so that diverging breakpoints are rejected. Several different approaches were attempted to detect convergence.

- Comparing breakpoint positions at the current sweepline and a lower sweepline position. Specifically, a sweepline at the circle base was used. However, in situations where the middle site of a triple is at the base of the circle, this approach failed. It also seemed to have numerical problems.

- Testing breakpoint and circle positions relative to the sites' positions. This approach took advantage of the fact that a converging breakpoint will lie either to the right or the left of both of its sites, and will be on the same side of the sites as the circle centre. This also fails in situations where the middle site of a triple is at the base of the circle. It seemed to have other sporadic problems as well.

- Testing order of sites. For a triple $< a, b, c >$, test the two breakpoints $< a, b >$ and $< b, c >$. For $< a, b >$, if site a is lower than site b, then the circle centre should lie to the right of a; if b is lower than a, then the circle centre should lie to the left of b. If this test fails, then the breakpoint is divergent. This approach proved the most robust, but has difficulty with the degenerate situation where the sites are on a common horizontal line.

The final solution chosen is still not perfect. For many larger diagrams with more than 50 sites, the approach breaks down. It is not clear if the problem is numerical or algorithmic.

For more information, [GS87] was consulted, but did not provide any further details. Fortune's original implementation might prove useful for resolving this problem.

3.4 Planar Subdivision Construction

When a circle event occurs, two edges must be joined, and a third edge constructed. However, there are two different ways in which the edges could be connected, and it was not clear how to choose between the two types. Again, several attempts were made before arriving at an acceptable solution.

- Constructing a line from the middle site to the left site, and testing to see which side of the line the circle centre lies on. This approach proved incorrect.

- Similar construction, but testing with the right site instead of the circle centre. This also proved incorrect.

- Testing breakpoint movement direction with a descending sweepline: either away from or towards the circle centre. The breakpoint movement corresponds to the opposite of the edge's orientation. This technique is correct, but it is numerically problematic to find two locations on the line swept out by the breakpoint, especially when a site is at the bottom of the circle.

- Examining order of sites in angular sweep about circle centre, starting at middle site. If left site is found first, choose one configuration; if right site is found first, choose the other. This is correct and appears numerically sound.

The final choice of approach appears to work quite successfully. With large numbers of sites, there are sometimes topological errors in the current implementation, but this may be a problem with the breakpoint convergence test.

3.5 Degeneracies

The algorithm fails in many degenerate situations. Surprisingly, without substantial effort, it can handle many degeneracies, including some situations where the highest two sites are on the same horizontal line, and the case where three sites are collinear. However, many degeneracies cause still result in program failure.

3.6 Final results

To test robustness, the program was fed a set of sites, each of which oscillated back and forth between two nearby positions at different rates. The program still fails in a large fraction of such tests, due to remaining degeneracy bugs, and problems in the convergence test. For static site collections, the program works quite reliably for small data sets, but bugs in the convergence test make it less reliable as the number of input sites grows. Given more time, these issues could be resolved.

Sites	**Hardware**	**Software**	**Software'**
10	4.3 ms	2.7 ms	1.2 ms
10	4.3 ms	2.7 ms	1.2 ms
25	9.9 ms	10.8 ms	3.4 ms
25	9.9 ms	12.6 ms	3.9 ms
50	19.1 ms	35.1 ms	9.0 ms
50	19.2 ms	36.5 ms	9.3 ms
75	28.5 ms	67.4 ms	15.6 ms
75	28.6 ms	59.7 ms	13.8 ms

Table 1: Comparison of hardware and software implementations, and an optimistic estimate of the speed of the software implementation using a balanced binary tree.

4 Comparison

Comparing the overall performance of the two approaches is difficult. The hardware algorithm can obviously perform closest-site queries in $O(1)$ time, but would take a substantial hit for many other types of queries. The software algorithm, on the other hand, may need a relatively long $O(n)$ time to perform closest-site queries, but it can provide rich information about neighbouring sites. Furthermore, the software Voronoi algorithm could be combined with a planar subdivision search data structure to provide $O(\log n)$ query time for closest-site queries.

A comparison of queries using either of these two approaches is like comparing apples and oranges. However, there is some value in comparing construction times using each approach. This can at least show a lower-bound on the cost of either approach. For certain applications, particularly those running at interactive rates, construction time is the bottleneck, since relatively few queries are performed.

The comparison is a little biased, since the software implementation did not incorporate a balanced binary tree. Consequently, the software implementation runs in $O(n^2)$ time instead of the desired $O(n \log n)$ time. However, we can get an optimistic estimate of the speed using a balanced tree by dividing the actual time by $\log n$. This estimate is included in the results table as the **Software'** column.

The hardware implementation was set to a 512×512 grid with a maximum error of half a pixel and the largest cones possible, making no assumptions about the input data. Since the software implementation only works correctly for a selected set of inputs, eight sets of working input sites were chosen and given to each implementation. Since the operation of the algorithm is fairly fast, testing was done with 100 repetitions of the data, then normalised to determine the average speed of each repetition.

The tests were run on a fast new machine, a Pentium 4 1.6 GHz processor with a GeForce3 graphics card. Results are shown in Table 1. For comparison, on an older AMD K6-2 300 MHz computer with a Matrox G400 graphics card, software speed was about 15 times slower and hardware speed was about 4 times slower.

Note that this is a pessimistic conclusion for the hardware algorithm, since the cone sizes are pessimistically large. As noted earlier, a substantial speedup can be achieved by assuming uniform random distribution of sites and reducing the cone size. By using half the cone size, which is still pessimistic, speedups of about 35% were recorded. Likewise, by using a grid size of 256×256, speedups of 140% were recorded.

It is interesting to observe that the software algorithm is comparable to the hardware algorithm for up to 25 sites. Once the algorithm is modified to use a balanced binary tree, it may even be comparable in speed for up to 100 sites. For small numbers of sites, the software algorithm is faster. Clearly, as the number of sites rises, the hardware algorithm will be able to construct its representation of the Voronoi diagram faster, since it operators in $O(n)$ time while Fortune's algorithm is $O(n \log n)$.

In terms of accuracy, Hoff's algorithm has a maximum error of 2^{-9} in the location of the Voronoi vertices by the choice of rendering resolution. The exact error of Fortune's algorithm is harder to quantify, but is at least as good as Hoff's using single-precision floating point. A comparison of the output of the two algorithms confirmed this error analysis.

5 Conclusions

Using current hardware, Fortune's algorithm can outperform the Hoff's discrete Voronoi diagram for situations where there are less than 25 sites. In situations where the number of sites is low and speed is critical, Fortune's algorithm is the best choice. However, this comes at substantial implementation cost: at least 5 times as much code, and much more programmer effort. For situations where speed is not critical or where a large number of sites are needed, Hoff's algorithm may be preferable.

References

[dBvKOS00] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications.* Springer-Verlag, 2000.

[GS87] Leonidas J. Guibas and Jorge Stolfi. Ruler, compass and computer: The design and analysis of geometric algorithms. In *Proc. the NATO Advanced Science Institute, series F, vol. 40: Theoretical Foundations of Computer Graphics and CAD*, pages 111–165, Lucca (Italy), July 1987. Springer-Verlag. Invited paper.

[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.