Jump Flooding in GPU with Applications to Voronoi Diagram and Distance Transform

Guodong Rong[†] Tiow-Seng Tan[†]
School of Computing, National University of Singapore

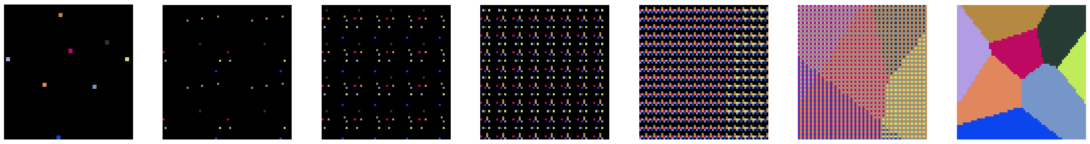


Figure 1: With the eight seeds (highlighted and shown in different colors) in a 64×64 grid on the leftmost picture, the progress of each round of the jump flooding algorithm is shown in the other six pictures, with the rightmost being the computed Voronoi diagram.

Abstract

This paper studies jump flooding as an algorithmic paradigm in the general purpose computation with GPU. As an example application of jump flooding, the paper discusses a constant time algorithm on GPU to compute an approximation to the Voronoi diagram of a given set of seeds in a 2D grid. The errors due to the differences between the approximation and the actual Voronoi diagram are hardly noticeable to the naked eye in all our experiments. The same approach can also compute in constant time an approximation to the distance transform of a set of seeds in a 2D grid. In practice, such constant time algorithm is useful to many interactive applications involving, for example, rendering and image processing. Besides the experimental evidences, this paper also confirms quantitatively the effectiveness of jump flooding by analyzing the occurrences of errors. The analysis is a showcase of insights to the jump flooding paradigm, and may be of independent interests to other applications of jump flooding.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture – Graphics Processors; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling – Geometric algorithms, languages, and systems; I.4.1 [Image Processing and Computer Vision]: Digitization and Image Capture – Image Geometry.

Keywords: Digital geometry, Interactive application, Programmable graphics hardware.

1. Introduction

The rapid advances in computational power and programmable capability of GPU in PC have prompted many research works in using GPU for general purpose computations. In these works, the parallel nature of GPU computation is exploited to gain huge speedup at a low cost; see [Owens et al. 2005]. In essence, one utilizes texture units in GPU like memory units in CPU to facilitate computations, while achieving speedup from working with texels (texture elements) in parallel via multiple fragment processors in GPU. In another view, the speedup with GPU computation is a result of the communications, i.e. passing of data, in parallel among texels. To date, speedup is mainly achieved with simple patterns of communication; little is known on quantifying computational capability with different patterns of communication. This paper studies a novel pattern of communication called *jump flooding* with its applications to computing Voronoi diagram and distance transform.

[†] {rongguod | tants}@comp.nus.edu.sg

In 2006 ACM Symposium on Interactive 3D Graphics and Games, 14-17 March, Redwood City, CA, USA, pp. 109-116, pp. 228.

Let $\mathcal{G}$ be a space consisting of a set of points, and $\mathcal{S}$ be a subset of points in $\mathcal{G}$ that are designated as *seeds*. A *Voronoi diagram* of $\mathcal{S}$ is defined to be a partition of $\mathcal{G}$ into subspaces, called *Voronoi cells*, where each subspace corresponds to one seed s and contains all points of $\mathcal{G}$ that are closer to the corresponding seed s than to any other seed, with respect to some fixed distance function. For simplicity, points of $\mathcal{G}$ that are of equal distance to two or more seeds are assigned arbitrarily to any of these seeds. A point of $\mathcal{G}$ that is closest to three or more seeds is a *Voronoi vertex* of these seeds.

In this paper, $\mathcal{G}$ is a discrete space that can be stored in a texture unit in the GPU. For example, $\mathcal{G}$ is a 2D grid of 512×512 grid points where each is a texel. As $\mathcal{G}$ is a discrete space, there may be none or more than one grid point with equal (closest) distance to three or more seeds. So, we adapt the usual definition of a *Voronoi vertex* to refer to any grid point whose (at most eight) neighboring grid points (that sharing a side or a corner with the grid point) belong to two or more other Voronoi cells.

Voronoi diagram is known for a long time due to its various applications in geometric and graphics problems among others; see [Aurenhammer 1991; Okabe et al. 1999]. A closely related notion called *distance transform* was introduced by Rosenfeld and Pfaltz [1966] for image processing applications. Distance transform is an operation that takes a binary image to compute a grayscale output image. The pixels in the binary image with value 0 are *seeds*, and 1 are background; the pixels in the output image have real values where each indicates the distance from the pixel to its closest seed.

To date, distance transform has found many uses in applications beyond image processing such as in computer graphics, pattern recognition, etc. Some example applications are: skeleton computation [Montanari 1968; Danielsson 1980], mathematical morphology operation [Ragnemalm 1992a], and displacement mapping [Wang et al. 2003; Donnelly 2005]. Though there are already many approaches to compute distance transform [Cuisenaire 1999], there are still unresolved issues: many approaches compute only an approximation to the distance transform in order to be efficient in applications, and they are sequential in nature that have no good avenue for speedup with parallel computation. All these limit their uses in, for example, interactive applications that require real-time responses. The major contributions of this paper are thus:

1. Presenting jump flooding as an efficient and effective computational paradigm for GPU; and
2. Proposing, for an input set of seeds in a 2D grid, the first parallel algorithm in GPU to compute in constant time (i.e. independent of the number of seeds) a highly accurate Voronoi diagram and distance transform.

The rest of the paper is organized as follows. Section 2 reviews related work in computing Voronoi diagram and distance transform. Section 3 presents our jump flooding algorithm and its variants to computing (an approximation to) Voronoi diagrams. These algorithms with minor adjustment can also compute distance transform. Section 4 analyses paths (patterns of communication) in jump flooding due to a single seed, and Section 5 quantifies grid points with errors (i.e. grid points where our computed Voronoi diagram and the actual one have different values) due to many seeds working in parallel during jump flooding. Section 6 describes our experimental results to confirm the effectiveness of jump flooding in computing Voronoi diagram as well as distance transform. To illustrate the potential of the jump flooding in higher dimensions, Section 7 presents our simulation results of computing 3D Voronoi diagram. Finally, Section 8 concludes the paper with potential future works.

The video accompanying this paper can be obtained from our project webpage at: http://www.comp.nus.edu.sg/~tants/jfa.html.

2. Related Work

For our case of working with grid and GPU, it is straightforward to obtain distance transform through Voronoi diagram. In the following, we present the previous works as they were originally discussed either for Voronoi diagram or distance transform. On the former, we discuss in Section 2.1 known approaches using GPU to compute Voronoi diagram in linear time. These are to compare with our proposed algorithms that run in constant time once the input seeds are stored in a texture unit. On the latter, we outline in Section 2.2 the development of fast-approximate to slow-exact distance transform algorithms, and some conventional approaches adapted to parallelize these sequential algorithms. A comprehensive survey on the distance transform approaches can be found in [Cuisenaire 1999].

2.1. Voronoi Diagrams

Hoff et al. [1999] computes a 2D Voronoi diagram by drawing (right-angle) cones with their apexes at the positions of seeds. The image of the cones as viewed orthogonally from the space of the seeds is a Voronoi diagram of the set of seeds. The algorithm thus relies on the GPU to efficiently rasterize the cones in time linear to the total number of triangles used to represent the cones. As mentioned in [Denny 2003], one needs many more triangles per cone than suggested in [Hoff et al. 1999] to compute the correct Voronoi diagram in some extreme case. This is not favorable for a large number of seeds. Denny [2003] presents a variant of the above algorithm by drawing a quad with depth texture, instead of cone, at each seed; see also [Strzodka and Telea 2004] for a similar method. This approach is faster but still remains in time linear to the number of seeds.

The above approaches require to know the nature of each seed, such as line segment or curve (not just regarded as a collection of points) to construct quads or cones. They can be cumbersome in processing general seeds that are complex objects as these must first be discretized. Also, they cannot work directly with inputs that are images as they must first extract the “seeds” in the images to construct quads or cones. This limits their uses in many image processing applications.

2.2. Distance Transforms

The algorithms for distance transform (DT) can be divided into two categories: approximate DT algorithms and exact DT algorithms. Generally, approximate DT algorithms, while having some errors in the results, are much faster than exact DT algorithms.

Approximate DT algorithms are usually using scan schemes to achieve computational costs linear to the number of pixels (or grid points). Two widely used methods are Chamfer distance transform [Borgefors 1984] and sequential Euclidean distance mapping [Danielsson 1980]. Both of these use a mask to perform two passes scanning over the grid points of the image. In the first pass, the mask moves from left to right, top to bottom. In the second pass, the mask moves recessively from right to left, bottom to top. When the center of the mask passes through a pixel, the corresponding value of the pixel is updated.

There are many kinds of exact DT algorithms, such as storing and sorting the front of the propagation [Ragnemalm 1992b; Eggers 1998], storing several closest seeds instead of just storing the closest one [Mullikin 1992], using big mask [Cuisenaire 1999] etc. All the above algorithms are sequential in nature and rely on the order of the scanning where the value of a pixel is determined by the value of those scanned before. They are thus difficult to be parallelized.

Another kind of exact DT algorithm repeatedly applies a mask on every pixel until no pixel has changed its value [Yamada 1984; Shih and Mitchell 1992; Huang and Mitchell 1994]. This kind of algorithm may be executed on all the pixels in parallel, but is not computationally efficient.

A parallel method proposed by Embrechts and Roose [1996] divides the screen into several sub-regions and then uses multi-processor computers to process them simultaneously. This algorithm needs to address the influence due to neighboring sub-regions. As it is not parallel on the pixel level, it is not suitable for GPU.

Interestingly, a truly massively parallel algorithm on distance transform was mentioned by Danielsson [1980]. At that time, no practical hardware was available to execute the presented algorithm. Though our proposed algorithms are the same in spirit to that proposed by Danielsson, they were discovered independent of the work of Danielsson as we were researching on parallel computation for Voronoi diagram on GPU. Danielsson stated in his paper that the work was merely a curiosity with no practical use at that time, whereas we have in this paper advanced the understanding and feasibility of jump flooding in GPU.

3. Flooding in Logarithmic Steps

To compute the Voronoi diagram for a 2D grid of size $n\times n$ with a given set of *seeds* at some grid points, we are interested to propagate the content (in particular, position information) of each seed s to each grid point so that each grid point can decide which seed is its closest one.

To achieve this for s, one standard algorithm is to flood the content in increasing distance from s outward, similar to a ripple effect starting at s. In the first round (of flood), we pass the content of s to its (maximum) eight neighboring grid points. From the second round onward, we pass on the content to grid points neighboring those which have just received the content in the previous round. This is repeated, in the number of rounds linear to n, till all grid points have received the content.

The above algorithm does not flood efficiently, with the currently available GPUs, as each grid point participates in just one round of passing on (non-repeating) content throughout the process. Section 3.1 discusses our proposed Voronoi diagram algorithm that avoids the above inefficiency. Section 3.2 provides some details on implementing our algorithm with GPU, and Section 3.3 presents more variations to the basic algorithm in Section 3.1.

3.1. Jump Flooding Algorithm

The above flooding can be seen as one where each round is a flood of *step length of* 1, i.e. each grid point (x, y) passes on its content to other grid points at $(x+i, y+j)$ where $i, j\in \{-1, 0, 1\}$. In contrast, Figure 2 shows two efficient ways one can flood the content of a seed (shown shaded) in $\log n$ rounds to all the other grid points, by varying the step length in each round, either doubling as in Figure 2(a) or halving as in Figure 2(b). Formally,

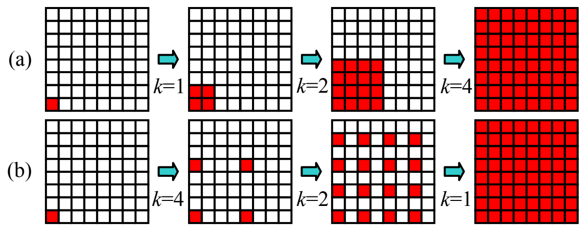


Figure 2: Jump flooding to propagate the content of a seed at the lowest left corner by (a) doubling step length, and (b) halving step length.

a flood of step length of k in a round, or simply *a round with step length of k*, is such that the grid point (x, y) passes on its content to other grid points at $(x+i, y+j)$ where $i, j \in \{-k, 0, k\}$. It is easy to show that (see Section 4) the content of a seed can reach all other grid points in such a *jump flooding* scheme of log$n$ rounds.

With the above process on a single seed, we can extend it straightforwardly to design an algorithm, called *jump flooding algorithm* (JFA) to compute the Voronoi diagram for a given set of seeds in an $n \times n$ grid as follows. Without loss of generality, we can assume n is a power of 2.

There are log$n$ rounds of flood with step lengths of $n/2$, $n/4$, …, 1. At the beginning, each grid point with a seed s records $\langle s, \text{position}(s)\rangle$ to indicate its closest seed found so far is s at the indicated position (which is itself), whereas the other each records $\langle \text{nil}, \text{nil}\rangle$. In a round with step length of k, each grid point (x, y) passes its closest seed to other grid points at $(x+i, y+j)$ where $i, j \in \{-k, 0, k\}$. In a symmetrical view, each grid point (x', y') receives (a maximum of) eight seeds (plus its current closest seed) from grid points at $(x'+i, y'+j)$ where $i, j \in \{-k, 0, k\}$. Among those received, grid point (x', y') decides which seed, say s', is its closest found so far, and updates $\langle s', \text{position}(s')\rangle$, if needed. Figure 1 shows the progress of the six rounds of flood to obtain a Voronoi diagram for a 64×64 grid with eight seeds.

A few notes are in order here. First, for a grid point s_k to record a seed s_0 as its closest seed found (at the end of some particular round of flood), the information of $\langle s_0, \text{position}(s_0)\rangle$ has traveled through a sequence of grid points $s_1 s_2 \dots s_k$ where each is from a different round of flood of progressively smaller step length and s_i passes $\langle s_0, \text{position}(s_0)\rangle$ to s_{i+1} till it reaches s_k. Such a sequence forms a *path* from s_0 to s_k. We note that at any one round, there can be more than one grid point passing the same information of $\langle s_0, \text{position}(s_0)\rangle$ to another grid point s_{i+1}; such case results in more than one path from s_0 to s_{i+1}.

Second, as stated before, the JFA does not always compute the correct Voronoi diagram. In other words, some grid points do not record the actual closest seeds. Figure 3 is an example where grid point p does not record the closest seed r but g (or b). This is because for r to reach p, we need a path to pass through either p' at (10, 6) or p'' at (10, 8), but both do not record r as its closest seed and thus such a path does not exist. However, in practice (see Section 5 and Section 6), JFA with Euclidean distance metric makes very few and only very special errors and thus remains very attractive to many applications.

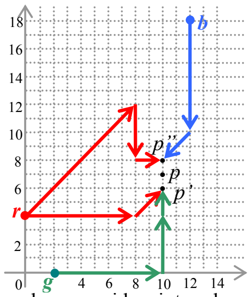


Figure 3: An example where a grid point p does not receive the correct closest seed r with jump flooding algorithm.

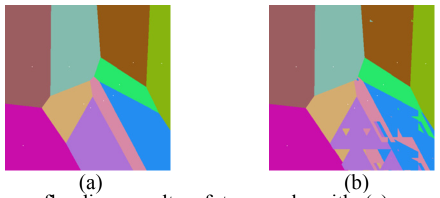


Figure 4: Jump flooding results of ten seeds with (a) no error when halving step length in each round, and (b) many errors when doubling step length in each round.

Third, Figure 2(a) alludes to an alternative to JFA as a jump flooding starting with step length of 1 and then doubling the step length in each round. This alternative, however, does not work as well as JFA; it generates many more errors in recording the closest seeds (see, for example, Figure 4). This can be explained qualitatively as follows.

For a grid point p to record the correct seed s, there must exist a path from s to p passing through a sequence of grid points p' such that each p' must regard s as the closest seed so far when it receives s as a seed. This is, however, very demanding as many grid points (especially those closer to seeds and possibly our required p') already recorded the correct seeds in some early rounds and thus do not permit other seeds (such as s) from passing on to other grid points. In contrast, our proposed JFA tends not to finalize the closest seeds for all grid points until much later rounds. This means each grid point permits many other seeds to temporarily be its closest seeds in order to pass on to other grid points, and JFA thus makes less number of errors.

3.2. Jump Flooding Algorithm in GPU

It is straightforward to implement JFA using OpenGL with fragment program support. A typical implementation for a given input set of seeds to output the results on the screen is as follows.

Let T_1 be a texture with resolution $n \times n$ that is equal to the resolution of the output grid. First, we draw the seeds into T_1 using its red and green channels to record the coordinates of the seeds. Second, we execute the log$n$ rounds of JFA with step lengths of $n/2$, $n/4$, …, 1. For each round with step length of k, we draw a screen-size quad to activate the execution of a fragment program on every pixel (i.e., grid point). In the fragment program for a pixel at (x, y), we refer to textures T_1 to gather the (maximum) nine seeds stored with pixels $(x+i, y+j)$ where $i, j \in \{-k, 0, k\}$ to decide which is the closest seed known so far to the pixel. The closest seed found for each pixel is written into T_1 for the next round. Note that T_1 is implemented as a so-called ping-pong buffer with two buffers alternating as input and output in consecutive rounds.

At the end of log$n$ rounds, T_1 has the closest seed for each grid point. To generate the Voronoi diagram of the seeds, we need another texture T_2 to record the colors of the seeds. For efficiency, T_2 is written together with T_1 at the first step using ARB_draw_buffers extension. With this, we can use the coordinates stored in T_1 to refer to T_2 to obtain the color for each grid point. At the same time, the distance transform, if needed, can be calculated by the fragment program for each pixel.

3.3. Variants to Jumping Flooding Algorithm

As mentioned, JFA may not produce correct closest seeds for all grid points. We summarize in this subsection variants to JFA to further reduce the occurrences of errors, at the expense of more computational time.

3.3.1. Additional Rounds

We observe in our experiments that most grid points with errors are grid points clustered around Voronoi vertices, or around

Voronoi edges intersecting the boundary of the grid. Moreover, many grid points with errors are each a single error and itself a Voronoi vertex. As such, a simple yet effective way to remove many errors is to execute JFA for one additional round with step length of 1, i.e. a total of (log*n*)+1 rounds where both of the last two rounds are with step length of 1. We call this variant JFA+1.

It is easy to realize that one can then use JFA+2 which is the usual JFA plus two additional rounds, with step length of 2 and 1 respectively. Likewise, one can do as many additional rounds as needed to achieve good accuracy (while converging to no better than the standard flood). We have also used JFA plus log*n* additional rounds, i.e. JFA+log*n* or simply JFA^2. Our experiments as reported in Section 6 confirm the effectiveness of these variants. In particular, JFA^2 produces correct results in most cases, though it fails for the difficult case as shown in Figure 5. (This is also a case where [Borgefors 1984] cannot correctly handle with a 3×3 mask.) In this case, if the center grid point *p* is incorrect at the end of JFA, it will still remain incorrect at the end of JFA^2. This is so as the Voronoi cell (shown as a wedge) does not "cover" any grid points along the dotted lines that *p* refers to during the additional log*n* rounds, i.e. *p* thus never receives the correct seed from any grid points of its Voronoi cell.

3.3.2. Additional Information on Seeds

As noted from Figure 3, the problem that *r* did not reach *p* is because *p'* and *p"* did not record *r* as their closest seeds. On the other hand, *r* is indeed the second closest seed to *p'* and *p"*. Thus, if we keep both the closest and the second closest seeds at each grid point, the above problem can be avoided. This is at the expense of additional texture memory and GPU computational time as we now need to examine a maximum of 18 values to identify the two closest seeds. We call this variant JFA2Seed. Similarly, we can have JFA3Seed, JFA4Seed etc. However, our experiments as reported in Section 6 indicate that this type of variants is inferior to that discussed in Section 3.3.1.

4. Paths in Jump Flooding

This section discusses properties of paths for a single seed *s* in an *n*×*n* grid due to jump flooding. These are exploited in JFA and are of independent interests possibly to other algorithms adapting jump flooding concept.

Property 1. Regardless of the position of *s* in the grid, JFA fills the grid with *s*.

Proof. In the following, we discuss a constructive proof (that can be extended to show the validity of Property 2).

Without loss of generality, let *s* be located at the grid point with coordinate (0, 0). We want to show that *s* can reach another grid point *p* at integer coordinate (p_x, p_y). Let us first suppose p_x and p_y are both positive integers. Let $(x_{m-1}\ x_{m-2}\ \ldots x_0)_2$ be the binary form of p_x and $(y_{m-1}\ y_{m-2}\ \ldots y_0)_2$ of p_y. Then, a path from *s* to *p* is obtained as follows: At step length of *l* (where *l* is either *n*/2, *n*/4, …, or 1) of JFA, we set *k*=log*l*, and

move the path diagonally up right if x_k=1 and y_k=1, or
move the path horizontally right if x_k=1 and y_k=0, or
move the path vertically up if x_k=0 and y_k=1, or
do not move the path if x_k=0 and y_k=0.

It is clear that each move arrives at a grid point *s'* closer to *p*, and we can pretend *s'* is our new *s* in the next move. Thus a path from *s* to *p* is constructed incrementally. If the above p_x and p_y are negative integers, we can modify the above rules to obtain a path from *s* to *p* analogously (with left in place of right, down in place of up etc.). Similarly, when only p_x or p_y (but not both) are negative integers, we modify only the relevant part of the rules involving p_x or p_y respectively. □

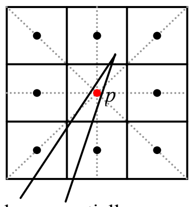


Figure 5: A Voronoi cell shown partially as a wedge can "steal" a center pixel *p* without including any of the eight neighboring pixels. Such a Voronoi cell looks disconnected when displayed on screen.

Property 2. Let *s* be a seed at (*x*, *y*), and *p* be a grid point. Suppose there exists another grid point *s'* at (*x'*, *y'*) where $|x-x'| = 2^l$ and $|y-y'| = 2^l$ for some *l*, and *p* lies within the square $2^l \times 2^l$ with *s* and *s'* as its two (out of four) vertices. Then, JFA generates more than one path from *s* to *p*.

Proof. Without loss of generality, we assume *s* is at (*x*, *y*) = (0, 0) and *s'* at $(-2^l, -2^l)$. That is, *p* is at the third quadrant with respect to the *x* and *y* axes. It suffices to construct another path from *s* to *p* that is different from that already provided in the argument of Property 1. A path from *s* to *p* can first make a move to *s'* via the round with step length of 2^l in JFA. That is, *s'* is such that *p* is in the first quadrant with respect to the vertical and horizontal lines passing through *s'*. This reduces the problem to finding a path from *s'* to *p*, with step lengths less than 2^l. For this, we re-use the argument for Property 1, and we are done.

In fact, in the above argument, we have many other choices of the first move and thus many other paths. The first move can be to $(-2^m, 0)$, $(0, -2^m)$, or $(-2^m, -2^m)$ for any $m \geq l$ as long as the move stays within the grid. Also, the above argument can be repeated analogously for *p* at the other quadrants. □

We note that the above property can be strengthened to cases where *s* and *s'* define a rectangle with one side of 2^l grid points. In the following, we classify each path from *s* to *p* into three types according to the second last vertex *p'* (i.e. the vertex before *p*) in the path. A *type-v path* is one where *p'* has the same *x* coordinate as *p* (i.e. reaching *p* vertically), a *type-h path* the same *y* coordinate (i.e. reaching *p* horizontally), and a *type-d path* otherwise (i.e. reaching *p* diagonally). See Figure 6 for an illustration.

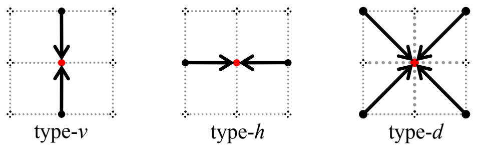


Figure 6: Three types of paths (only the last step is shown).

Property 3. The paths JFA can generate from *s* to a grid point *p* are all type-*v*, or all type-*h*, or all type-*d* (i.e. never a mixture of any two or more types).

Proof. The progress of JFA can be visualized as follows. It first partitions the given grid with vertical and horizontal lines into *n*/2×*n*/2 square grids with the exception at the boundary of the given grid that may be incomplete *n*/2×*n*/2 square grids. At this point, information of ⟨*s*, position(*s*)⟩ is known to grid points at the junctions of the vertical and horizontal lines. Subsequently, each square grid (and incomplete square grid) is further partitioned by vertical and horizontal lines into four smaller square grids each having half of the side length than before, and so on. As always, ⟨*s*, position(*s*)⟩ is known at the junctions of the vertical and horizontal lines.

So, with respect to a grid point *p*, it receives ⟨*s*, position(*s*)⟩ if, and only if, it becomes a junction of a vertical and horizontal line. There are three possibilities at the round before it becomes a junction. First, p is on some vertical line, and the grid point passing it the information is on the same vertical line. As such, the resulting paths from s to p are of type-v. Second, similarly, if p is on some horizontal line, the paths from s to p are of type-h. Third, p is at the center of some square grid, and the paths from s to p are of type-d.

Note that once p receives the information of $\langle s, \mathsf{position}(s)\rangle$, it cannot receive in subsequent rounds the same information. This is because the subsequent rounds are with smaller step lengths and p thus does not refer to any junctions that have the information. Thus, no new type of paths from s other than those mentioned in the previous paragraph can reach p, and thus all paths from s reaching p are of the same type. □

Let i (and j, respectively) be the first bit position, starting from the least significant bit, where the x-coordinates (and y-coordinates, respectively) of grid points s and p have different bit values. We define $\mathsf{order}(s, p)$ to be the minimum of i–1 and j–1. For example, with s at (0, 0) = $(0000, 0000)_2$ and p at (12, 14) = $(1100, 1110)_2$, we have $i = 3, j = 2$, and $\mathsf{order}(s, p) = 1$.

Property 4. All the paths generated by JFA from a seed s to a grid point p reaches p at the round with step length of 2^l where l=$\mathsf{order}(s, p)$.

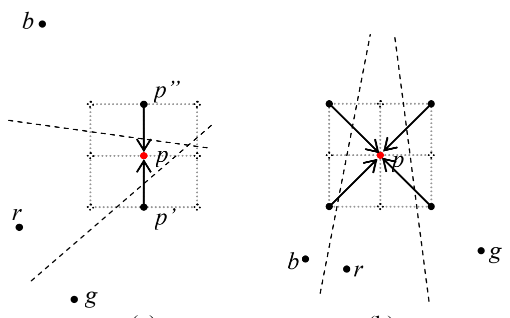
Proof. Let us visualize the progress of JFA as in the proof of Property 3. Completing the round with step length of 2^l, JFA partitions the original grid with vertical and horizontal lines into $2^l \times 2^l$ square grids. That is to say that any two adjacent vertical lines are separated with distance 2^l. So, the x-coordinates of these vertical lines have the same last l least significant bits, and alternate between 0 and 1 for their $(l+1)^{\text{th}}$ least significant bit. Similar statement holds for the y-coordinates of the horizontal lines. We note again that paths from s reach p if, and only if, when the grid is partitioned by vertical and horizontal lines such that p is on a junction. This thus happens when *both* the l least significant bits of the x- and y-coordinates of s and p are the same, and thus at the round with step length of 2^l where l=$\mathsf{order}(s, p)$. □

5. Errors in Jump Flooding

We now consider the consequence of jump flooding of many seeds at each round. Recall when two or more seeds arrive at a grid point p', JFA keeps only one piece (or finite amount) of information of a seed, say s, at p'. As such, some seeds do not survive or flood beyond and onto p' (i.e. paths from such seeds do not extend till and beyond p'), they are said to be *killed* at p' by s as of this round. As a result, a grid point may not receive, at the completion of JFA, information of all seeds. So, when a grid point p does not receive and thus cannot record its closest seed as this seed was killed at other grid points in some earlier round, we count this as an *error* (at p) with JFA. This section analyzes the extent of errors with JFA.

From our experiments on many runs of JFA with different number of seeds, we observe all errors occurring mainly along the boundaries of Voronoi cells. More specifically, for Voronoi cells along the boundary of the grid, grid points with errors can cluster around Voronoi edges; for the other Voronoi cells, all grid points with errors, with one exception out of millions of runs, are Voronoi vertices or cluster around Voronoi vertices. This is fortunate as all computed Voronoi diagrams do not have unpleasant looks of "holes" within any Voronoi cells. Additionally, this indicates the importance to analyze errors due to Voronoi vertices. To this end, we have worked out Property 5 to Property 7 as follows.

Property 5. If an error occurs at p but not at any of its neighboring grid points, then p is a Voronoi vertex.



(a) (b)

Figure 7: The grid point p has r as a closest seed but records instead either g or b as its closest seed.

Proof. Let r be the closest seed to p but not recorded by p at the end of JFA. From Property 3, we consider three scenarios. First, all paths from r to p (when there is no other seeds involved in JFA) are type-v as shown in Figure 7(a), reaching grid point p' or p'' before p. Since error occurred at p, r must have been killed at or before p' and p'' while traveling towards p.

Notice that r is not the closest seed of both p' and p''; otherwise, errors occur at all grid points along p' and p'' (inclusive of themselves, p and those neighboring grid points of p), which contradicts to the given condition. Then, let g and b, respectively, be the closest seeds of p' and p'', respectively. As the perpendicular bisector between r and g must intersect $p'p''$ below p, and that between r and b above p, we must have two distinct bisectors and thus g and b are also distinct.

So, all three grid points p, p', and p'' have distinct closest seeds. If there are no other grid points between $p'p''$, p is naturally a Voronoi vertex as required. Also, if there are other grid points along $p'p''$, none of them can have r as their closest seed by the given condition, p is thus a Voronoi vertex by definition.

Similarly, when all paths to p from r are of type-h, we adapt the above argument to show that p is still a Voronoi vertex. Lastly, when all paths to p from r are type-d as in Figure 7(b), we also adapt the above argument with the minor adjustment that r is not the closest seed to the four grid points to the left and to the right (as in the orientation given by Figure 7(b)) of p. □

Note first that we ignore the technicality in the above proof where p can be a grid point on the boundary of the given grid when p' or p'' (but not both) does not exist in the grid. Note second that our experiments (in Section 6) report over 90% of errors are single errors, which are at Voronoi vertices according to Property 5. This percentage is increasing with the increase in the number of seeds.

The next two properties attempt to provide some insights on the probability of errors due to Voronoi vertices. Suppose there are only three seeds r, g, and b in the $n \times n$ grid. To possibly do any analysis with calculus, we approximate the discrete world of the grid by a continuous one of a real plane.

Property 6. Let grid point o be a Voronoi vertex of seeds r, g, and b with r as its closest seed. The probability of error at o when all paths from r to o are of type-v or type-h is:

$$E_1 = 2\left(\frac{1}{3} - \frac{2}{\pi R} + \frac{2}{\pi^2 R^2}\right) \sum_{k=0}^{(\log n)-2} 4^{-3(k+1)}$$

where R is the distance from r to o.

Proof. We just need to prove the property for $E_1/2$ when all paths from r to o are of type-v; similar argument applies to the case of type-h.

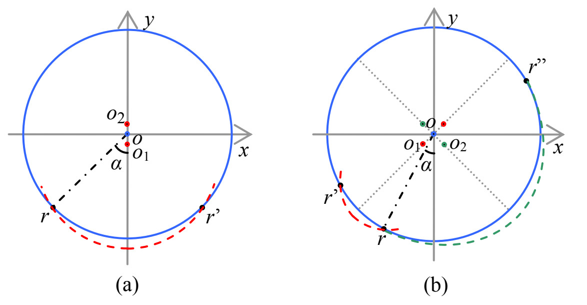


Figure 8: (a) Type-*v*. The red dashed curve indicates the circle with the center at o_1 and passing through point *r* and *r'*. (b) Type-*d*. The red dashed curve indicates the circle with the center at o_1 and passing through point *r* and *r'*. The green dashed curve indicates the circle with the center at o_2 and passing through point *r* and *r"*.

Refer to Figure 8(a). By assumption, *r*, *g* and *b* lie on a circle with center at *o*. The figure also indicates *r'* as the mirror grid point of *r* along the *y*-axis; *r* and *r'* separate the circle into two arcs, called them *major arc* and *minor arc* for simplicity.

We consider the conditions for error to occur at *o*. All paths from *r* to *o* must pass through o_1 or o_2 just before reaching *o*. From Property 4, the step length from o_1 or o_2 to reach *o* is 2^k where k = order(r, o). We consider each *k* ranges from 0, 1, 2, …, $(\log n) - 2$ to obtain the summation needed in this property. (Notice that order(r, o) = $(\log n) - 1$ is the case where $r = o_1$ or $r = o_2$ and no error occur at *o*.)

First, the probability of the path from *r* to *o* is type-*v* and with order(r, o) = *k* is equal to $4^{-k} \times ¾ \times ⅓ = 4^{-(k+1)}$. The first term 4^{-k} is the probability of order(r, o) $\geq k$. The second term ¾ is the probability of *r* and *o* having different $(k+1)$th least significant bit for their *x*- and *y*-coordinates. (These first two terms together give the probability of order(r, o) = *k*.) The third term ⅓ is the probability of the path being a type-*v* among the three types of path.

Second, we must have *g* and *b* killing *r* at o_1 and o_2 at the rounds with step lengths $\geq 2^{k+1}$. That is, order(r, g) $\geq k+1$ and order(r, b) $\geq k+1$, and *g* must lie on the minor arc (or major arc, respectively) and *b* on the major arc (or minor arc, respectively) so that each is closer to o_1 and o_2 than *r* is. The former occurs with probability $4^{-(k+1)} \times 4^{-(k+1)}$, and the latter with

$$F = \left(\frac{1}{3} - \frac{2}{\pi R} + \frac{2}{\pi^2 R^2}\right).$$

To calculate *F* as shown in the above, we first define a probability density function *f* of *α* as follows:

$$f(\alpha) = \frac{2}{2\pi}\left(\frac{2\alpha R - 2}{2\pi R}\right)\left(\frac{2\pi R - 2\alpha R - 2}{2\pi R}\right)$$

where the 2 in the first term (i.e. $2/(2\pi)$) represents the interchanging of *g*'s and *b*'s position over the full circle of 2π, the second term for one seed at the minor arc (notice the "– 2" term is to exclude grid points *r* and *r'*), and the third term for the other on the major arc. So integrating $f(\alpha)$ from 0 to 2π (alternatively, by integrating from 0 to $\pi/2$ then multiplying by 4) gives the required *F*. □

Property 7. Let grid point *o* be a Voronoi vertex of seeds *r*, *g*, and *b* with *r* as its closest seed. The probability of error at *o* when all paths from *r* to *o* are of type-*d* is:

$$E_2 = \left(\frac{1}{12} - \frac{1}{\pi R} + \frac{2}{\pi^2 R^2}\right)\sum_{k=0}^{(\log n)-2} 4^{-3(k+1)}$$

where *R* is the distance from *r* to *o*.

Proof. The same essence of the proof of Property 6 appears here. Thus, we only need to indicate the corresponding $f(\alpha)$ and *F* here for the argument to hold. Refer to Figure 8(b). In the figure, *r'* is the mirror image of *r* along $y = x$, and *r"* that of *r* along $y = -x$. Note that *g* and *b* are such that one lies in the minor arc *rr'* and the other in the minor arc *rr"*. So, the corresponding *f* and *F* are as follows:

$$f(\alpha) = \frac{2}{2\pi}\left(\frac{\left(\frac{\pi}{2} - 2\alpha\right)R - 2}{2\pi R}\right)\left(\frac{\left(\frac{\pi}{2} + 2\alpha\right)R - 2}{2\pi R}\right)$$

and $$F = \left(\frac{1}{12} - \frac{1}{\pi R} + \frac{2}{\pi^2 R^2}\right).$$ □

Notice we do integration around a circle in the proof of the above two properties. This is to sum up all the contribution of type-*v*, type-*h* or type-*d* paths from *r*, but assume only one type of paths throughout the different grid points *r* on the circle. This is obviously not true in reality. To put some confidence into the above analysis, we write a program to do explicit counting as follows. We draw different sizes of digital circles with radius of 5 to 256 pixels. For each circle, we go around each pixel on its circumference to check the coordinate of the pixel in order to mark which type of paths the pixel can generate towards the center. We find the percentages in all types of paths are quite balance for radius larger than ten pixels. This is comforting to the analysis.

For an $n \times n$ grid with $m > 3$ seeds, there are $(2m - h - 2)$ Voronoi vertices where *h* is the number of Voronoi cells on the boundary of the grid. If we assume the errors at Voronoi vertices are independent to each other and there is an estimate on *R* for all Voronoi cells, then from Property 6 and Property 7, we have the number of errors at Voronoi vertices estimated at:

$$\begin{aligned} E &= (2m - h - 2)(E_1 + E_2) \\ &\approx 2m(E_1 + E_2) \quad \text{when } h \text{ is small} \\ &= 2m\left(\frac{9}{12} - \frac{5}{\pi R} + \frac{6}{\pi^2 R^2}\right)\sum_{k=0}^{(\log n)-2} 4^{-3(k+1)} \end{aligned}$$

Note that *E* is a lower bound estimate on the number of errors at grid points. This is because *E* does not account for errors near the boundary of the grids (where some seed can be *killed* by the boundary rather than by other seed) and also errors at clusters of grid points (not necessarily involving Voronoi vertices). For the latter grid points, errors actually happen mostly because Voronoi vertices have errors at the rounds with step lengths $\geq 2^1$ and then propagate the errors to them in subsequent rounds. However, we note that such errors occur very rarely as indicated by the probabilities in Property 6 and Property 7 with $k \geq 1$. We have also verified experimentally (Section 6) that *E* can be a good estimate on the number of errors due to JFA. This means that JFA+1, JFA+2 and JFA^2 are efficient algorithms in computing Voronoi diagrams with near to 100% accuracy.

6. Experimental Results

We have implemented JFA and its variants using Visual C++.net 2003 and Cg 1.3. The hardware platform is Pentium IV 3.0GHz, 1G DDR2 RAM and nVidia GeForce 6800 GT PCI-X with 256M DDR3 video memory. For each run of our experiment, we randomly generate input seeds for a grid of 512×512. The number of seeds ranges from 100 to 1000 in increment of a hundred, and 1000 to 10000 in increment of a thousand. To obtain an exact Voronoi diagram to count the number of grid points with errors in JFA, we adapt the algorithm of [Denny 2003].

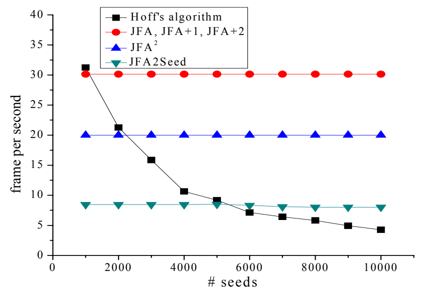


Figure 9: The comparisons of frame rates of our algorithms and Hoff's algorithm.

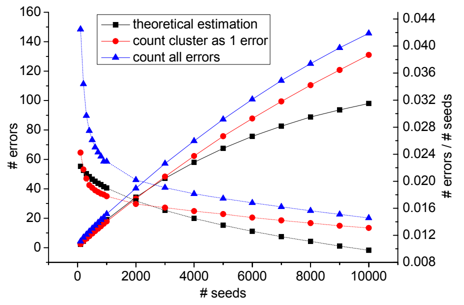


Figure 10: The actual and estimated errors in JFA.

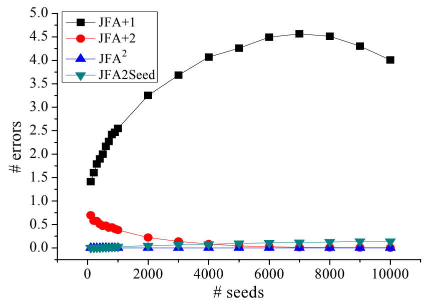


Figure 11: Errors in JFA+1, JFA+2, JFA^2 and JFA2Seed.

Each value in our following charts is obtained through averaging 10000 runs of random inputs.

Efficiency. The speed of JFA is compared with the popular algorithm by [Hoff et al. 1999]. We note that their algorithm, though linear in complexity to the number of seeds, decreases rapidly in frame rate as more and more seeds are used. On the contrary, the frame rates of our JFA implementation maintain quite consistent for different number of seeds. The result is shown in Figure 9. The black curve is for Hoff's algorithm. The red curve is for JFA, JFA+1 and JFA+2; all our three algorithms achieve about the same frame rates in our experiments. As for JFA^2, we see that it is still much faster than Hoff's algorithm for large number of seeds.

In another view, JFA is output sensitive where its running time is mainly depending on the output resolution (if no time is charged to put the seeds into a texture, as in the case of an input is already a texture).

Errors in JFA. The solid curves in Figure 10 (with *y*-axis on the left) refer to the average number of errors in JFA for different number of seeds. The blue curve counts each error at grid point as 1 to get the total number of errors, whereas the red curve counts a cluster of connected grid points with errors as 1. Each data point has a very small variance over the 10000 runs. The blue curve and the red curve do not deviate much; this is a testimony to the importance of counting errors due to Voronoi vertices as done in Section 5.

In another view, the dashed curves in Figure 10 (with *y*-axis on the right) show that the ratio of the average number of total errors to the total number of seeds decreases with increasing number of seeds. This phenomenon is also captured in E of Section 5 – when there are many seeds, the value of R decreases as each Voronoi cell decreases in size, and E/m thus decreases too. This underscores the robustness of JFA in dealing with large number of seeds.

Our estimate of errors with E is also plotted as the black curve in Figure 10. To do that, we need to estimate R for each case of different number of seeds. For that, we can either take the total area of the grid divided by the number of Delaunay triangles to estimate the radius R of the circumcircle, or take the average from the actual experiments too. We use the former but verify to be consistent with the experiments too.

The black curve compares very well to the actual errors as shown by the blue curve and red curve. This is particularly so for the number of seeds below 3000. The not-so-good estimate for the number of seeds above 3000 is because E is rather sensitive to changes in R of small value (of less than 5 pixels) as in this case with a large number of seeds.

Effectiveness of JFA Variants. As we observe, most errors in JFA are single Voronoi vertices or small clusters of grid points around Voronoi vertices. The variants of JFA as JFA+1 or JFA+2 can indeed greatly decrease the total number of errors as shown in Figure 11. The error rates of JFA+1 are just a few grid points, which are good news to most applications of Voronoi diagram or distance transform. At the extreme, JFA^2 has close to zero errors in all the 10000 runs. The main errors of JFA^2 are as shown in Figure 5 where no number of additional rounds to JFA can correct the errors. In between, the result of JFA+2 is also interesting – for large numbers of seeds, JFA+2 has very good results that approach those of JFA^2. Also, we notice in Figure 9 that JFA+2 is as efficient as JFA and it is thus the best compromised among all.

On the other hand, the error of JFA2Seed as shown in Figure 11 is comparable to (but slightly worse than) the other variants. However, its large penalty in frame rate as shown in Figure 9 does not warrant its use.

Generalized Voronoi Diagram. The seeds of the Voronoi diagram can be generalized to line segments or even curves or areas. Such generalized cases can also be computed by our algorithms; see, for example, Figure 12. Though we do not have an analysis of errors for such cases, we expect good results with little errors too. This is because our algorithms treat such generalized seeds as collections of point seeds and thus expect to inherit the good performance obtained for point seeds.

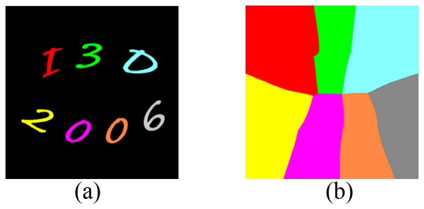


Figure 12: Applying JFA on (a) the input image, we have (b) the result of the generalized Voronoi diagram of the seven area seeds.

7. Jump Flooding in Higher Dimensions

The proposed JFA and its variants are applicable to computing higher dimensional Voronoi diagrams. However, the current GPU cannot write data into 3D textures as needed by JFA. Though one can pack and simulate a 3D texture with a 2D texture [Harris 2003], such packing currently works mainly for a small 3D texture and is thus not very useful when we need jump flooding in a large 3D space. So, to assess the effectiveness of our proposed algorithms for higher dimensions, we perform CPU simulation of JFA and its variants. In this paper, we only present the 3D case.

In our experiment, we randomly generate input seeds for a grid of 512×512×512. The number of seeds ranges from 100 to 1000 in increment of a hundred, 1000 to 10000 in increment of a thousand, and 10000 to 30000 in increment of ten thousand. Each size of seeds is performed 100 runs to obtain an average on the number of grid points with errors. We do not attempt larger number of runs because for a large number of seeds even 100 runs already take several days to complete.

Figure 13 shows the simulation results for JFA, JFA+1, JFA+2 and JFA^2 in 3D. The ratio of the average number of total errors to the total number of seeds for JFA is also shown as the black dashed curve. Quantitatively, JFA and its variants perform very well in generating close to the exact Voronoi diagram of a set of seeds. The percentage of grid points with errors is close to zero. We expect that JFA and its variants in 3D are even more effective than their counterparts in 2D. This is because there are many more paths from a seed to each grid points in 3D, and it becomes much harder to kill all paths to a grid point generated by its closest seed. Thus, the probability of a grid point not receiving its closest seed becomes very low.

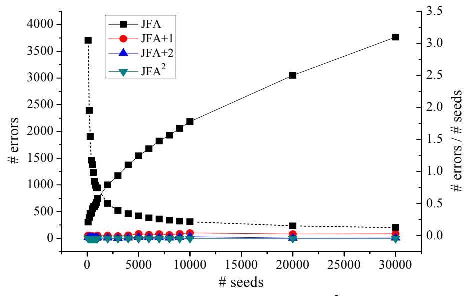


Figure 13: Errors in JFA, JFA+1, JFA+2 and JFA^2 in a 512×512×512 grid.

8. Conclusion and Future Work

This paper studies the jump flooding algorithm (JFA) for the current GPU. This is probably the first of its kind of analysis on the propagation of computation in GPU. The paper has good insights to analyzing propagation that can be useful to other studies of GPU computation.

In this work, we implement JFA and its variants for the 2D case where input is either a set of seeds or an image texture to efficiently compute Voronoi diagram and distance transform. We have also performed CPU simulation of JFA and its variants in 3D to demonstrate the effectiveness of JFA in higher dimensions in computing Voronoi diagram and distance transform.

One possible future work is to investigate the feasibility of JFA on different kinds of Voronoi diagrams. There are many distance metrics other than Euclidean metric, such as Manhattan distance, chess-board distance, and more generally the Minkowski norm, and many kinds of weighting on seeds such as multiplicative and additive.

Our JFA algorithm is a new parallel computation scheme on GPU. It may have many applications in various areas. We are currently investigating the usefulness of JFA to real-time soft shadow generation.

Acknowledgements

We would like to thank Franco Preparata and Edwin Zeng for fruitful discussions and helpful comments on this paper. This research is supported by the National University of Singapore under grants R-252-000-181-112 and R-252-000-216-112.

References

AURENHAMMER, F. 1991. Voronoi Diagrams–A Survey of a Fundamental Geometric Data Structure. *ACM Computing Surveys 23*, 3, 345–405.

BORGEFORS, G. 1984. Distance Transformations in Arbitrary Dimensions. *Computer Vision, Graphics, and Image Processing 27*, 321–345.

CUISENAIRE, O. 1999. *Distance Transformations: Fast Algorithms and Applications to Medical Image Processing*. PhD Thesis, Université catholique de Louvain.

DANIELSSON, P.-E. 1980. Euclidean Distance Mapping. *Computer Graphics and Image Processing 14*, 227–248.

DENNY, M. O. 2003. *Algorithmic Geometry via Graphics Hardware*. PhD Thesis. Universität des Saarlandes.

DONNELLY, W. 2005. Per-Pixel Displacement Mapping with Distance Functions. *GPU Gems 2: Programming Techniques for High Performance Graphics and General-Purpose Computation*. Edited by M. Pharr and R. Fernando, Addison-Wesley, 123–136.

EGGERS, H. 1998. Two Fast Euclidean Distance Transformations in Z^2 Based on Sufficient Propagation. *Computer Vision and Image Understanding 69*, 1, 106–116.

EMBRECHTS, H. and ROOSE, D. 1996. A Parallel Euclidean Distance Transformation Algorithm. *Computer Vision and Image Understanding 63*, 1, 15–26.

HARRIS, M. J. 2003. *Real-time Cloud Simulation and Rendering*. PhD Thesis. University of North Carolina at Chapel Hill.

HOFF, K. E., KEYSER, J., LIN, M., MANOCHA, D. and CULVER, T. 1999. Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware. In *Proceedings of ACM SIGGRAPH 1999*, ACM Press / ACM SIGGRAPH, New York. Computer Graphics Proceedings, Annual Conference Series, ACM, 277–286.

HUANG, C. T. and MITCHELL, O. R. 1994. A Euclidean Distance Transform Using Grayscale Morphology Decomposition. *IEEE Transaction on Pattern Analysis and Machine Intelligence 16*, 4, 443–448.

MONTANARI, U. 1968. A Method for Obtaining Skeletons Using a Quasi-Euclidean Distance. *Journal of the Association for Computing Machinery 15*, 4, 600–624.

MULLIKIN, J. C. 1992. The Vector Distance Transform in Two and Three Dimensions. *Graphical Models and Image Processing 54*, 6, 526–535.

OKABE, A., BOOTS, B., SUGIHARA, K. and CHIU, S. N. 1999. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*. John Wiley & Sons Ltd.

OWENS, J. D., LUEBKE, D., GOVINDARAJU, N., HARRIS, M., KRÜGER, J., LEFOHN, A. E. and PURCELL, T. J. 2005. A Survey of General-Purpose Computation on Graphics Hardware. *Eurographics 2005*, 21–51.

RAGNEMALM, I. 1992a. Fast Erosion and Dilation by Contour Processing and Thresholding of Distance Maps. *Pattern Recognition Letters 13*, 3, 161–166.

RAGNEMALM, I. 1992b. Neighborhoods for Distance Transformations Using Ordered Propagation. *CVGIP: Image Understanding 56*, 3, 399–409.

ROSENFELD, A. and PFALTZ, J. L. 1966. Sequential Operations in Digital Picture Processing. *Journal of the Association for Computing Machinery 13*, 4, 471–494.

SHIH, F. Y.-C. and MITCHELL, O. R. 1992. A Mathematical Morphology Approach to Euclidean Distance Transformation. *IEEE Transaction on Image Processing 1*, 2, 197–204.

STRZODKA, R. and TELEA, A. 2004. Generalized Distance Transforms and Skeletons in Graphics Hardware. *Proceedings of EG/IEEE TCVG Symposium on Visualization*, 221–230.

WANG, L., WANG, X., TONG, X., LIN, S., HU, S., GUO, B. and SHUM, H. Y. 2003. View-dependent Displacement Mapping. *ACM Transactions on Graphics. 22*, 3, 334–339.

YAMADA, H. 1984. Complete Euclidean Distance Transformation by Parallel Operation. *7th International Conference on Pattern Recognition*, Montreal, Canada, 336–338.