Variants of Jump Flooding Algorithm for Computing Discrete Voronoi Diagrams

Guodong Rong Tiow-Seng Tan
School of Computing, National University of Singapore
{rongguod | tants}@comp.nus.edu.sg

Abstract

Jump flooding algorithm (JFA) is an interesting way to utilize the graphics processing unit to efficiently compute Voronoi diagrams and distance transforms in 2D discrete space. This paper presents three novel variants of JFA. They focus on different aspects of JFA: the first variant can further reduce the errors of JFA; the second variant can greatly increase the speed of JFA; and the third variant enables JFA to compute Voronoi diagrams in 3D space in a slice-by-slice manner, without a high end graphics processing unit. These variants are orthogonal to each other. In other words, it is possible to combine any two or all of them together.

1. Introduction

Graphics hardware has been utilized to efficiently and effectively compute Voronoi diagrams in discrete space. Such a discrete Voronoi diagram, and the related concept of discrete distance transform, have many real-time interactive applications, such as motion planning, object selection, mosaics, skeleton, feature preserving evolution etc. (see [4], [13]). Moreover, discrete Voronoi diagrams can be utilized to achieve robustness in the computation of continuous Voronoi diagrams [6].

Notable works using graphics hardware on Voronoi diagrams include Hoff et al.'s paper [4] that builds a cone for every input (point) site and renders these cones to obtain the Voronoi diagram as the lower envelop of these cones. Denny [2] presents a similar method using a pre-computed texture in place of the cone. This method is faster and produces more accurate results; see also Strzodka and Telea's work [13]. Fischer and Gotsman [3] use planes tangent to a paraboloid and thus avoid the errors caused by the tessellation of the cones. All these algorithms run in time linear to the size of the input. In other words, their speeds reduce with the increase in the number of sites.

Recent advances in the graphics processing unit (GPU) allow programmed control of the graphics pipeline; see [8] for a comprehensive survey. For our purposes here, a GPU can be imagined as a collection of processors working in parallel on all the pixels (information) in a texture with resolution of $n \times n$ where n is typically 512, 1024, or up to 4096 with current state-of-the-art GPU. Such capability is used by Sigg et al. [12] to compute distance transform (a concept closely related to Voronoi diagram) in 3D space; see also Sud et al.'s work [14]. These algorithms need significant CPU effort to compute the bounding volume of Voronoi cells, and their speeds are still dependent on the size of the input sites.

Rong and Tan [9] introduce the jump flooding algorithm (JFA) to compute Voronoi diagrams and distance transforms in GPU. Unlike all previous work, the speed of JFA is almost independent to the size of the input sites. Specifically, given a collection of m sites in a texture with a fix resolution, the time to compute the Voronoi diagram of these sites is almost a constant regardless of m. But, JFA is an approximate algorithm and errors at pixels can occur; that is, some pixels may not record the correct nearest sites after the JFA computation. On the whole, the nice property of being fast while not incurring many errors makes JFA suitable to compute discrete Voronoi diagrams.

A similar idea extended to 3D space is proposed by Cuntz and Kolb [1]. They solve the problem by packing a 3D texture into a 2D one. Due to texture size limit in the current GPU, they can handle only small resolutions of 3D space.

In this paper, we propose three new variants of JFA. They focus on different aspects of JFA: The first variant can greatly decrease the rate of errors of JFA with exactly the same computational cost of a previous variant in [9]. The second variant can improve the speed of JFA by more than 25% while maintaining very low rate of errors. The last variant extends the algorithm into 3D space in a slice-by-slice manner. This is more efficient than the prior work of Sud et al. [14] as the speed of JFA is almost independent to the

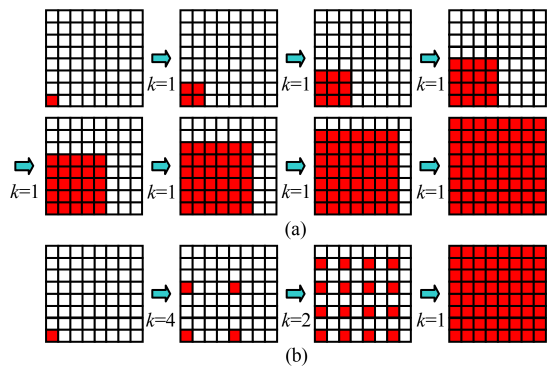

Figure 1: Propagation of the content of a site at the bottom-left corner by (a) standard flood filling, and (b) JFA.

number of sites. All of these three variants are orthogonal, so any two or all of them can be combined with each other into a more powerful flooding solution.

The rest of the paper is organized as follows. Section 2 reviews the basic idea of JFA. Sections 3 to 5 introduce our three new variants. Section 6 concludes the paper with possible directions for future work.

2. Review of JFA

Suppose we have a site s at pixel (x, y) in a texture with resolution of $n \times n$, and we want to pass its information to all the other pixels in the texture. The most straightforward way is to use the standard flood filling algorithm. In the first pass, we pass the information of s to its (maximum) eight neighbors at positions $(x+i, y+j)$ where $i, j \in \{-1, 0, 1\}$. These neighbors continue to pass the information to their neighbors in the later passes. This process continues until all the pixels in the texture receive the information of s. Figure 1(a) demonstrates this process in an 8×8 texture with a site at the bottom-left corner.

Now suppose we have many sites in the texture, and the information of every site is just its coordinate. When all the pixels in the texture get the information of the sites, each can use this information to choose the nearest site to it. Thus, the Voronoi diagram can be computed using this standard flood filling algorithm. A simpler version of this idea is also presented in *Algorithm 4.14* in [7]. However, the number of passes and thus time needed by standard flood filling algorithm is linear to the resolution of the texture.

In the standard flood filling algorithm, the *step length* in all the passes is always 1. So every pixel effectively only propagates information of a site once in the entire process. This is wasteful in the computing power of GPU. To rectify this, we can vary (i.e. jump) the step lengths in different passes to have the *jump flooding algorithm*. In JFA, the step length in the first pass is equal to half of the resolution. (In our discussion, we always assume n is a power of 2; in more general form, the step length of the first pass is $2^{\lceil \log n \rceil - 1}$ for a resolution of $n \times n$.) Then the step length is halved in every pass until the step length of 1. Formally, in a pass with step length of k, a pixel at the position (x, y) passes its information to the pixels at the positions $(x+i, y+j)$ where $i, j \in \{-k, 0, k\}$. This process is shown in Figure 1(b) for the same configuration of input site as in Figure 1(a). By using different step lengths k, the number of passes needed by JFA is reduced to logarithmic of n.

Same as the standard flood filling algorithm, JFA can compute Voronoi diagram. Although JFA may cause errors at pixels in the final result, as presented in [9], the rate of such errors is very low, and not noticeable to the naked eye. Besides Voronoi diagram and distance transform, JFA has also been successfully used to compute real-time soft shadows [10].

With this basic idea of JFA, we next present its three new variants in the following three sections.

3. Variant 1: 1+JFA

During the process of JFA, an error occurs when the correct site (the nearest site) is *killed* by other sites en route during the flooding. According to Property 4 of JFA in [9], the necessary condition for two sites meeting at a same pixel at step length of 2^l is that the l last bits (in binary encoding) of their x- and y-coordinates are exactly the same. So if the very last bit of the x- or the y-coordinate of a site s is already different to that of another site t, these two sites do not meet each other in any pixel in any pass before the pass with step length of 1, and thus one cannot kill the other.

This leads to the idea of the first variant, called 1+JFA. Before performing the standard JFA, we pass the information of a site s at position (x, y) to its (maximum) eight neighbors at positions $(x+i, y+j)$ where $i, j \in \{-1, 0, 1\}$. This is equivalent to performing a pass with step length of 1 before the standard JFA, and leads to the name of 1+JFA. In effect, we have made information of s available at pixels where their last bit patterns of x- and y-coordinate cover all possible combinations of 1s and 0s. This reduces the chances of all the copies of s being killed by other sites during flooding. The process of 1+JFA as compared to that of the standard JFA is shown in Figure 2.

Note that this variant cannot totally eliminate all the errors, because the other sites are also flooded by one pixel to their neighbors before the standard JFA and those neighbors together may kill, for example, all the copies of s at different passes of flooding. Despite

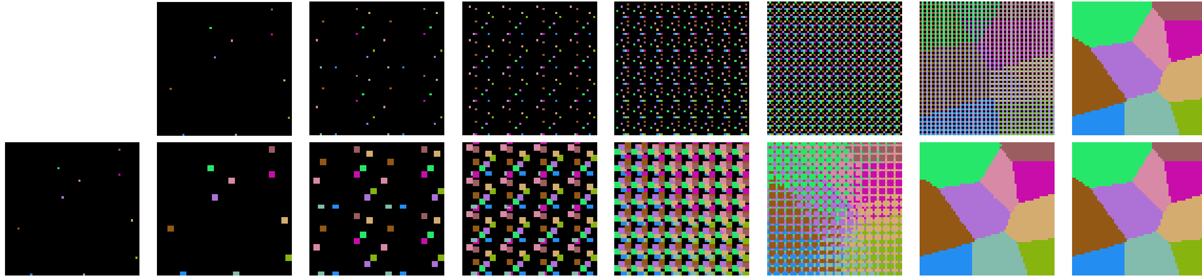

Figure 2: Comparison of processes of JFA and 1+JFA for 10 sites in a texture with resolution of 64×64. Upper row: the process of JFA. Lower row: the process of 1+JFA that has one additional pass.

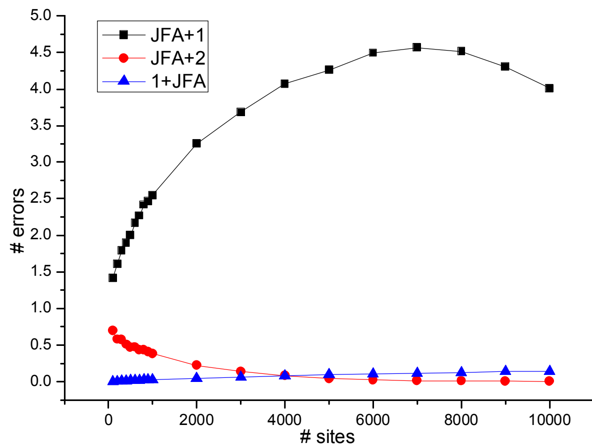

Figure 3: Errors of variants of JFA

this, we observe very few errors generated by this variant in our experiments; see Figure 3.

It is very interesting to notice that although 1+JFA is very similar to the variant JFA+1 [9], the rate of errors of 1+JFA is far less than that of JFA+1. Since the numbers of passes of 1+JFA and JFA+1 are the same, their speeds are the same too. So 1+JFA is a better algorithm than JFA+1 because of much smaller rate of errors. In Figure 3, we can also see that the rate of errors of 1+JFA is also less than that of the slower variant, JFA+2 [9] for small number of sites.

4. Variant 2: Halving Resolution

In doubling a texture from $n \times n$ to $2n \times 2n$, the standard JFA needs an additional pass with step length of n to reach among pixels further apart. Besides, each pass now deals with 4 times the number of pixels, and the speed is thus much slower than before. Turning this into a positive way, if we can half the resolution needed, the speed can improve accordingly. This is the idea of the second variant as explained next.

Before starting JFA on a texture with resolution of $2n \times 2n$, we sub-sample a square of 4 pixels into a single pixel to result in a texture with resolution of $n \times n$. In the sub-sampling, if there is one or more sites in the square of 4 pixels, we choose among them one as the representative site. Next, JFA is applied on the texture with resolution of $n \times n$ containing the representative sites. After this is completed, we expand the texture back to the original resolution of $2n \times 2n$. In doing this, one pixel (x, y) in the low resolution becomes 4 pixels in the high resolution, and each of these 4 pixels derives its nearest site from those sites represented by the representative site at (x, y). With this, the edges between two Voronoi cells generally form a staircase (zigzag) shape. So, we need another pass of flooding with step length of 1 to smooth the Voronoi edges. Figure 4 shows the process of this variant for the same configuration of input sites in Figure 2.

One important note is in order when performing JFA in the low resolution. To compute distance from a site to a pixel (and to select the nearest site), we must still use the original coordinates in the high resolution. This is because the sub-sampling may slightly change the relative positions of the sites. Such a change, no matter how small, may result in a significant change in the final Voronoi diagram. This phenomenon is illustrated in Figure 5.

However, there are cases that such situation is not avoidable. For example, in Figure 5, suppose there is also a site at the position of a'. After sub-sampling, we only have a', b and c. Many pixels belonging to the Voronoi cell of a are in the Voronoi cell of either b or c after the computation, and these are thus pixels with errors. If the sites are uniformly distributed, this situation occurs when the density is very high. So this variant works better for cases with sparse distribution of sites. In another view, if the density is too high, all the Voronoi cells are small in size and some additional passes can fix most of these errors too. On the whole, this variant, when combined with other variants, may not generate many errors while able to accelerate the computation of flooding.

Figure 6 shows the comparison of the speed of this variant with those of JFA and its other variants. It is clear that this variant is the fastest. As we have mentioned, this variant can be combined with the other

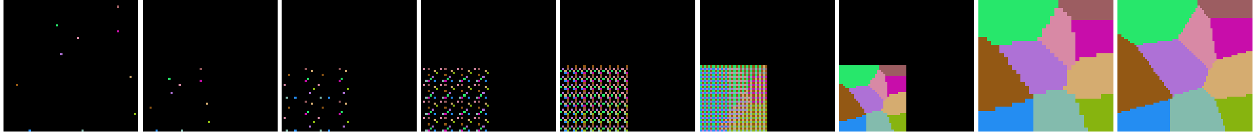

Figure 4: The process of variant 2 for 10 sites in a texture with resolution of 64×64. The input is as shown on the leftmost picture. The second picture on the left is the input after halving the resolution. The subsequent 5 pictures show the process of standard JFA working on the halved resolution, and followed by the 2 pictures on restoring to the original resolution and smoothing Voronoi edges.

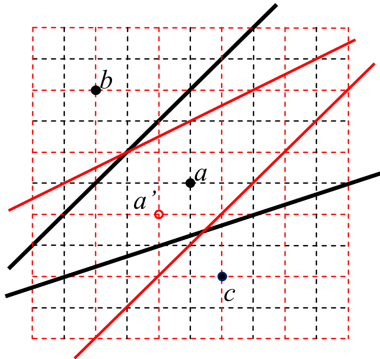

Figure 5: Small change of the position of a site can lead to a big change of its Voronoi cell. During the sub-sampling, *a* moves to *a'* while *b* and *c* remain at their original positions. The black lines show the Voronoi cell of *a* before sub-sampling, and the red lines show the Voronoi cell after. With our approach of calculating distance using the coordinates in the higher resolution, we still can obtain the correct Voronoi cell of *a*.

variants (e.g. JFA+2) to reduce the rate of errors. In our experiment on nVidia 6800 GPU, we obtain less than 3 errors on average in a texture with resolution of 512×512 while still enjoy frame rate (of around 70fps) better than that of the standard JFA (of around 55fps).

5. Variant 3: 3D Space

The idea of JFA can be easily extended into 3D space, where every voxel looks for its (maximum) 26 neighbors to choose its nearest site. But, only the latest GPU processors such as nVidia 8800 that have the ability to write into 3D texture to support the implementation of such an algorithm. Previous work of Rong and Tan [9] can only use the CPU to simulate JFA in 3D space to understand its performance. For our work here, we show a way to adapt 2D JFA to compute a 3D Voronoi diagram in a slice-by-slice manner, without the need of writing into 3D textures.

According to Property 1 of JFA in [9], regardless of where a site is, as long as it is not killed by other sites en route, its information can reach all the pixels in

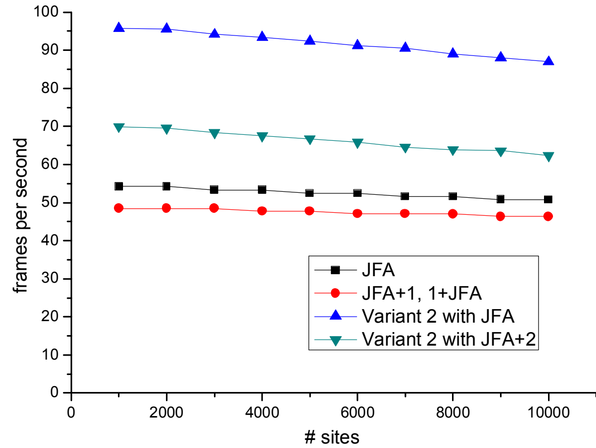

Figure 6: Speeds of JFA and its variants

the texture after the pass with step length of 1. This property suggests that it is not necessary to put the sites at their original positions to perform JFA. In fact, we can choose to put the sites at any positions convenient to an application. (This may lead to different rate of errors in the flooding.) We note that in some sense, our variant 2 discussed in the last section is of the same spirit; it can be seen as first shifting all the sites to the positions with both x- and y-coordinates are even numbers in the texture of original resolution, and then performing JFA in the original texture for these pixels only.

This understanding can help us to compute 3D Voronoi diagram using JFA, in a slice-by-slice manner. Suppose we want to compute the intersection of a slice and the 3D Voronoi diagram. We first orthogonally project all the sites onto this slice. At each pixel of the slice, we record the original 3D coordinate of the site projected to the pixel. If two or more sites projected to a same pixel, only the coordinate of the nearest site is recorded, because the Voronoi cells of the other sites do not intersect this slice and these sites thus need not appear in this slice for flooding. Next, we run JFA (or its other variants) using these projected sites in the slice while utilizing the original 3D coordinates recorded to perform distance computation to find nearest sites for each pixel.

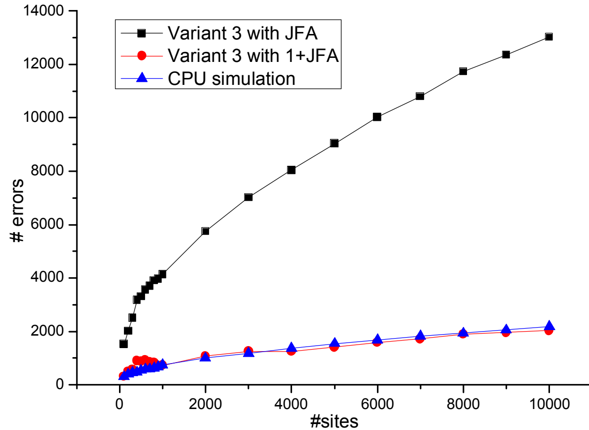

Figure 7: Errors of variant 3 (summing all errors of the 512 slices) in a space with resolution of 512×512×512.

The rate of errors of the CPU simulation of JFA in 3D reported in [9] is shown again as the blue curve in Figure 7. Compared with that, the result of this new variant (Figure 7, black curve) generate more errors. This is expected, since in this variant, JFA is performed in a 2D texture, and the number of possible paths for a pixel receiving the information of its nearest site is far less than that of a real JFA in a 3D space. So the chance of a nearest site killed by other sites is accordingly higher. On the other hand, the red curve in Figure 7 shows the rate of errors when this variant is combined with 1+JFA. Again, the rate of error is far less than that of the standard JFA. An interesting observation is that the red curve almost coincident with the blue curve now. In other words, we now have a good substitute of the real JFA in 3D but computing only in a slice-by-slice manner.

Since JFA is naturally capable of computing Voronoi diagrams of generalized sites, we have also applied this variant in computing 3D generalized Voronoi diagrams. We have tested different types of sites including points, line segments, splines, etc. One interesting type of site is sphere. Voronoi diagrams of spheres have many applications in various areas, such as biochemistry [5]. Our variant can also handle this type of sites. Figure 8 shows a screenshot of our program using 50 spheres as the sites in a cube with the resolution of 512×512×512.

6. Concluding Remarks

In this paper, we present three novel variants of jump flooding algorithm to compute Voronoi diagrams in both 2D and 3D space. All of these variants retain the good property of JFA: their speeds are almost independent to the input size.

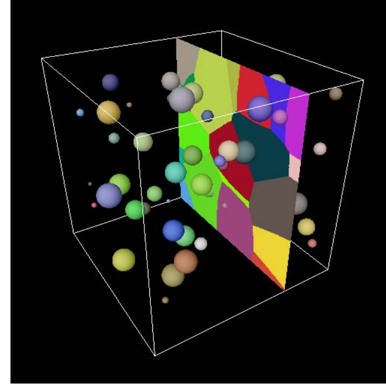

Figure 8: Screenshot of the program computing 3D Voronoi diagram using 50 spheres as sites.

The idea of putting sites at positions other than their original ones discussed in the third variant is interesting. This may help to find some special artificial patterns to further reduce the rate of errors or to obtain some interesting effects.

For computing Delaunay triangulation (the dual graph of Voronoi diagram) in the continuous space, we are investigating the use of these JFA variants to build a fast program. The challenge includes converting intermediate solution (of Voronoi diagram) in the discrete space to final solution (of Delaunay triangulation) in the continuous space. Our aim is to understand how well this approach can perform as compared to existing good sequential programs such as *triangle* [11] in a 2D continuous space. We keep further updates on JFA and its applications at http://www.comp.nus.edu.sg/~tants/jfa.html.

Acknowledgements

We would like to thank Iestyn Bleasdale-Shepherd for his suggestion and contribution to the study of Variant 2. This research is supported by the National University of Singapore under grant R-252-000-254-112.

References

[1] Cuntz, N. and Kolb, A. 2006. Fast Hierarchical 3D Distance Transforms on the GPU. *Technical report*, Institute for Vision and Graphics, University of Siegen, Germany.

[2] Denny, M. 2003. *Algorithmic Geometry via Graphics Hardware*. PhD Thesis. Universität des Saarlandes, Germany.

[3] Fischer, I. and Gotsman, C. 2006. Fast approximation of high order Voronoi diagrams and distance transforms on the GPU. *Journal of Graphics Tools*, 11(4), 39–60.

[4] Hoff, K., Culver, T., Keyser, J., Lin, M. and Manocha, D. 1999. Fast Computation of Generalized Voronoi

Diagrams Using Graphics Hardware. In *Proceedings of ACM SIGGRAPH 1999,* 277–286.

[5] Kim, D.-S., Kim, D. and Cho, Y. 2005. Euclidean Voronoi Diagrams of 3D Spheres: Their Construction and Related Problems from Biochemistry. *Lecture Notes in Computer Science*, vol. 3604, 255–271.

[6] Matuura, S and Sugihara, K. 2004. Use of Discrete Topology for the Construction of Generalized Voronoi Diagrams. In *Proceedings of International Symposium on Voronoi Diagrams in Science and Engineering (ISVD'04)*, 153–163.

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

[8] Owens, J., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. and Purcell, T. 2007. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1), 80–113.

[9] Rong, G. and Tan, T.-S. 2006. Jump Flooding in GPU with Applications to Voronoi Diagram and Distance Transform. In *Proceedings of ACM Symposium on Interactive 3D Graphics and Games*, 109–116.

[10] Rong, G. and Tan, T.-S. 2006. Utilizing Jump Flooding in Image-Based Soft Shadows. In *Proceedings of ACM Symposium on Virtual Reality Software and Technology*, 173–180.

[11] Shewchuk, J. 1996. Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator. In *Applied Computational Geometry: Towards Geometric Engineering*, M. Lin and D. Manocha, Eds., vol. 1148 of Lecture Notes in Computer Science. Springer-Verlag, Berlin, May, 203–222.

[12] Sigg, C., Peikert, R. and Gross M. 2003. Signed distance transform using graphics hardware. In *Proceedings of IEEE Visualization*, 83–90.

[13] Strzodka, R. and Telea, A. 2004. Generalized Distance Transforms and Skeletons in Graphics Hardware. In *Proceedings of EG/IEEE TCVG Symposium on Visualization*, 221–230.

[14] Sud, A., Govindaraju, N., Gayle, R., and Manocha, D. 2006. Interactive 3D Distance Field Computation using Linear Factorization. In *Proceedings of ACM Symposium on Interactive 3D Graphics and Games*, 117–124.