JUMP FLOODING ALGORITHM ON GRAPHICS HARDWARE AND ITS APPLICATIONS

RONG GUODONG

NATIONAL UNIVERSITY OF SINGAPORE

2007

JUMP FLOODING ALGORITHM ON GRAPHICS HARDWARE AND ITS APPLICATIONS

RONG GUODONG
(Bachelor of Engineering, Shandong University)
(Master of Engineering, Shandong University)

A THESIS SUBMITTED
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
DEPARTMENT OF COMPUTER SCIENCE
NATIONAL UNIVERSITY OF SINGAPORE
2007

to my wife Yang Xia

Acknowledgement

During the past four years of my Ph.D. research, I owe special thanks to many people for their guidance, cooperation, help and encouragement. First and foremost, I would like to thank my supervisor, Associate Professor Tan Tiow Seng, for his kindly guidance in both research and life. During the past few years, from his advice and his attitude, I have learnt the right approach, and more importantly, the right attitude to do research. I will benefit from it throughout my life. He has worked together with me all the way throughout my research. His foresight is always able to find the problems in my algorithms and programs. His constructive feedback in the writing of the technical papers has helped to improve my writing skills. Without his help, this thesis will never be completed.

I am also grateful to other members in graphics group, Assistant Professor Anthony Fang Chee Hung, Assistant Professor Low Kok Lim, Assistant Professor Alan Cheng Holun, Dr. Huang Zhiyong and Dr. Golam Ashraf, for their helpful discussion in the G3 Seminar.

During my stay in the graphics lab, I have enjoyed friendships with a lot of people. Special thanks to Martin Tobias for his hard shadow codes and the fantasy scene, which my soft shadow program is based on. Special thanks to Stephanus and Cao Thanh Tung for their cooperation in the Delaunay triangulation project;

without their effort, the Delaunay code will not be as efficient and reader-friendly as it is now. Calvin Lim Chi Wan seated besides me for all the four years. Thank you for the numerous and enlightening discussions between us. I would like also thanks the other members in the graphics lab: Ng Chu Ming, Zhang Xia, Shi Xinwei, Ouyang Xin, Ashwin Nanjappa and Zheng Xiaolin. You have made this office a nice place to stay and study. Outside the graphics lab, I will also thank my former apartment mate Zhang Hao for his many helpful discussions in the area of mathematics and statistics.

Last but not least, I would like to give my appreciation and thanks to my parents Rong Yun and Li Yunlan, for their love and support throughout my life. I am deeply grateful to my wife Yang Xia, for her endless love, selfless support, perpetual encourage and strong confidence to me. Your support and love will always be the most important thing in my life.

Contents

Summary

The graphics processing unit (GPU) has been developing at a very fast pace these few years. More and more researches have been done to utilize the ever increasing computability power of the GPU on general-purpose computations. This thesis proposes a new GPU algorithm – jump flooding algorithm (JFA). JFA is a new paradigm of communication between pixels on the GPU. It can quickly propagate the information of certain pixels to the others. The speed of JFA is exponentially faster than that of the standard flooding algorithm, and is approximately independent to the input size.

In this thesis, we explain the details of JFA and its variants. Some properties of JFA are proven in order to help us to understand this new algorithm better. Using JFA, we present a novel algorithm to compute the Voronoi diagram and the distance transform. This new algorithm is faster than previous ones, and its speed is mainly dependent on the resolution of the texture instead of the input size. According to our analysis and experiments, the error rate of the new algorithm is low enough for most applications.

JFA is also applied on the computation of real-time soft shadows. Two purely image-based algorithms, JFA-L and JFA-E, are proposed. Inherited from JFA, the speeds of both JFA-L and JFA-E are similarly dependent on the resolution of the

texture instead of the complexity of the scene. This makes them very useful for real-time applications such as games.

Based on the discrete Voronoi diagram generated by JFA, we propose a new algorithm to compute the Delaunay triangulation in continuous space. This is the first attempt to use the GPU to solve a geometry problem in continuous space. The speed of the new algorithm exceeds that of the fastest Delaunay triangulation program to date.

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

When researchers design new algorithms, it is always one of the goals to make them faster. Using parallel computation is one approach to greatly increase the speed of the algorithms. In traditional parallel computation, the focus is on parallel machines or clusters. In recent years, the quick development of the graphics processing unit (GPU) provides a new approach of achieving higher speed on a PC with moderate price.

Compared to the CPU, the GPU has many advantages. One important advantage is the speed of its development. The development of the CPU roughly follows the famous Moore's Law – the number of transistors in the CPU, and accordingly the speed of the CPU doubles every 18 months [Int07]. The speed of the GPU grows much faster than that of the CPU as it doubles every 6 months. This phenomenon is sometimes known as "Moore's Law Cubed". Thus, even if the current GPU is weaker than the current CPU in some areas, it is expected to exceed the CPU in the near future. The parallel architecture is another advantage of the GPU. A GPU can be seen as a processor composed of many small processors, like

many smaller CPUs (although simplified version). These "smaller CPUs" work in parallel, which lead to an extremely high throughput of the GPU compared with the CPU.

Due to these advantages of the GPU, researchers are interested in exploring the GPU to perform tasks other than graphics processing, which is the original function that the GPU is designed for. Such general-purpose computation on the GPU is usually know as GPGPU [OLG+07, GPG07]. Even before the term "GPU" is coined by NVIDIA in 1999, researchers have already attempted to use graphics hardware to solve problems other than graphics processing. Hoff et al.'s work on Voronoi diagram [HCK+99] is a good example of such pre-GPU researches. To date, there are numerous GPGPU applications in many different areas, including physically based simulation, signal and image processing, global illumination, geometric computing, database and data mining, etc. A good survey of GPGPU can be found in [OLG+07]. Many other up-to-date applications can also be found at the GPGPU website [GPG07]. In the next section, we briefly review some previous work that is closely related to our work in this thesis.

1.1 Previous Work on GPGPU

Generally speaking, there are two different approaches in GPGPU algorithms. One approach uses the small processors in the GPU separately. Every small processor is used like the processor in a parallel machine. So, the GPU is used in a similar fashion as the traditional parallel machines. This kind of algorithm does not consider the information of the positions of the pixels, and the texture is used only as normal random-access memory. In this approach, the positions of the pixels are

just used as the address of the memory. For example, Purcell et al.'s ray tracing on the GPU [PBMH02] and Carr et al's Ray Engine [CHH02] both belong to this type.

The computation of the other approach of GPGPU algorithm does include the information of the positions of the pixels, and is more suitable towards the architecture of the GPU. Furthermore, some of them utilize the communication between pixels. Next, we briefly review some of such algorithms.

Bitonic sorting is one of such algorithms. Bitonic sorting is an old sorting algorithm that is first introduced in parallel computation. When GPGPU becomes popular, it is implemented on the GPU [PDC+03, KW05]. In their implementation, the sequence of the number to be sorted is stored into a 2D texture, using an indexing mapping function from 1D to 2D. To sort n numbers, the algorithm uses $\log n$ stages. Stage i ($0 \leq i < \log n$) consists of $i + 1$ passes. In every pass, a pixel can communicate with another pixel at a certain distance (called *step length*) away from it. These pixels are *neighbors* of each other in this pass. The step lengths of these $i + 1$ passes start from 2^i and are halved in every pass until the step length reaches 1. In every pass, pixels are compared to their neighbors later in the sequence. Those are not in the correct order are exchanged in this pass. At the end of the algorithm, all the numbers in the sequence are sorted. The total number of the passes is $O(\log^2 n)$. On the other hand, on the CPU, it has been proven that the optimal time complexity of sorting is $O(n \log n)$. Hence the CPU version is much slower if n is big.

In bitonic sorting, although the numbers are stored in a 2D texture, the algorithm is in fact a 1D algorithm. Every pixel communicates with its neighbors in 1D space only. Since the GPU is specifically designed for 2D computation, it is

more efficient to use communication in 2D space on the GPU.

One example of the use of 2D communication is parallel reduction algorithm on the GPU [BP04]. Reduction is a simple process using an operation to reduce many values into a single value. For example, if the operation is addition, the result of the reduction is the sum of all the values. In parallel reduction, there are similarly many passes. In every pass, selected pixels communicate in parallel with their three neighbors in 2D space (the pixels to the right, above, and above-right of it). For every pixel, the values of these three neighbors, together with its own value, are reduced using the specified operation into a single value, which is stored at the position of this pixel for the usage of the later pass. After every pass, four values are reduced to a single value. Thus the resolution of the texture is halved in every pass. When the resolution of the texture becomes 1×1, the only value stored in the texture is the result of the reduction. Here, the number of the passes is $O(\log n)$, where n is the original resolution of the texture. Compared with the reduction algorithm on the CPU which requires $O(n^2)$ time, the GPU version is much more efficient.

In this parallel reduction algorithm, many passes are performed to get only one single value from n^2 values. Although it is more efficient than the CPU version, it does not take full advantage of GPU. Ideally, we can make every pixel communicate with its (maximum) eight neighbors in 2D space, and generate n^2 values in the end. This will make the maximum use of the computability of the GPU.

Décoret's N-Buffer [Déc05] is a good example of such work. N-Buffer is a hierarchy structure that is built from the standard depth buffer. The building process is similar to that of the parallel reduction algorithm above. It also consists of $\log n$ passes, where n is the resolution of the depth buffer. The step lengths in this

algorithm start from 1 and are doubled in every pass until the step length reaches $n/2$. In every pass, one pixel communicates with its three neighbors (defined the same as in the parallel reduction above). The maximum of the four depth values stored in these fragments is selected and stored into all the four pixels. So, after the last pass with step length of $n/2$, a hierarchy structure (the N-Buffer) with $\log n$ levels is constructed. Every pixel in every level stores the maximum depth value of a square area with it as the left-bottom corners. This N-Buffer can be generated in the pre-processing step, and can be used in different applications later, such as occlusion culling, particle culling and shadow volume clamping.

Our jump flooding algorithm proposed in this thesis is similar to N-Buffer. Both of them use various step lengths in $\log n$ passes. However, the jump flooding algorithm perform a lot more operations on every pixel, and thus has many different properties to N-Buffer. The algorithm is further explained in detail in Chapter 3.

1.2 Contributions

In this thesis, we propose a new GPGPU algorithm – jump flooding algorithm. Jump flooding algorithm makes use of a new paradigm on the communication between pixels, and is very useful to many different applications. We provide proofs on some properties of this algorithm. The algorithm is applied to three different applications in this thesis. These include the Voronoi diagram and distance transform, real-time soft shadows and the Delaunay triangulation in continuous space. The main contributions of this thesis are as follows:

- Propose a new paradigm, jump flooding algorithm, on general-purpose computation on the GPU. This algorithm utilizes a new way of communication

among pixels to quickly propagate the information from certain pixels to the others. The speed of the new algorithm is exponentially faster than that of the standard flooding algorithm. The speed is approximately independent to the input size.

- Apply the jump flooding algorithm on the computation of Voronoi diagrams and distance transforms in discrete space. The speed of the new algorithm is faster than the previous algorithms, and the error rate is low enough for most practical applications. [RT06a, RT07]

- Apply the jump flooding algorithm on the generation of real-time soft shadows. Two purely image-based algorithms are developed based on the jump flooding algorithm. JFA-L can generate outer penumbra, while JFA-E can generate both outer and inner penumbra. Both of them achieve good frame rates with the current GPU for complex scenes of over hundreds of thousands triangles. [RT06b]

- Propose a new algorithm to compute the Delaunay triangulation in continuous space. This is the first attempt to utilize the GPU to solve a geometry problem in continuous space, instead of discrete space. The speed of the algorithm exceeds that of the fastest CPU Delaunay triangulation program to date. [RTCS08]

1.3 Outline of the Thesis

The rest of this thesis is organized as follows: Chapter 2 introduces the foundations of GPU programming, which is used throughout the thesis. Chapter 3 explains

the details of the jump flooding algorithm. Some properties of this algorithm are proven in this chapter. Next, Chapter 4 apply the jump flooding algorithm on the computation of Voronoi diagrams and distance transforms. The error rate is analyzed in this chapter. Chapter 5 utilizes the jump flooding algorithm to generate real-time soft shadows. And Chapter 6 introduces a new algorithm based on the jump flooding algorithm to compute the Delaunay triangulation in continuous space. We also provide the proof of the correctness of the algorithm. Finally, the conclusion and future work are given in Chapter 7.

Chapter 2

GPU Programming

The Graphics Processing Unit (GPU) has been applied in many areas other than traditional graphics, such as physically based simulation, signal and image processing, global illumination, geometric computing, database and data mining, etc. Before introducing more details of our new algorithms on the GPU, we first introduce GPGPU programming, or, in other words, how to use the GPU to solve non-graphics problems. The GPU has its own unique structure, and thus the programming on the GPU is quite different from the programming on the CPU. This chapter briefly introduces some fundamental knowledge of GPU programming.

2.1 Graphics Pipeline

A *pipeline* is a sequence of several stages, where each stage takes its input from the previous stage, performs some operations on it, and then sends the output to the next stage. In order to display a geometry on the screen, the data have to go through such a pipeline on the GPU; see Figure 2.1. All the geometries are

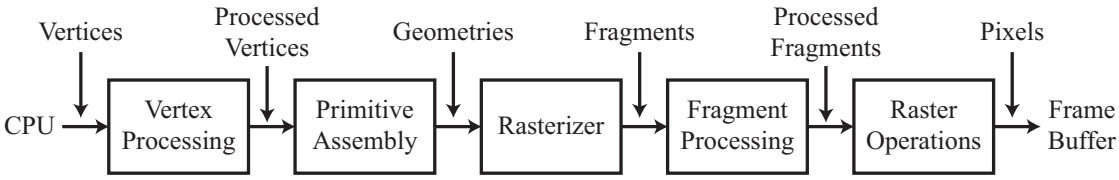


Figure 2.1: Graphics pipeline.

represented by triangles, and every triangle contains three vertices. These vertices, along with some of their attributes (original position, color, texture coordinate, etc.), are sent from the CPU to the GPU, and processed through the first stage of the pipeline – Vertex Processing. In this stage, the vertices are transformed and some new attributes such as illuminated color, transformed normal, etc. are also computed. The processed vertices are then sent to the second stage – Primitive Assembly. The vertices are assembled here into triangles, and the triangles are connected into meshes. These triangles are then rasterized into many *fragments*. Note that fragments are in some sense similar to pixels, but they are different concepts. One fragment corresponds to one pixel on the screen, but one pixel can have more than one fragment. The fourth stage in the pipeline is Fragment Processing, where the attributes of these fragments are computed. The processed fragments are then sent to the final stage – Raster Operations. In this stage, they must go through some common graphics tests, such as stencil test, depth test, etc. Only the surviving fragments can affect the contents of the corresponding pixels. The updated pixels are sent to the frame buffer and finally displayed on the screen.

Figure 2.2 uses a simple example of two triangles to illustrate the pipeline clearer. Figure 2.2(a) shows the input vertices of the two triangles. In the Vertex Processing stage, they are transformed to their final positions and their attributes,

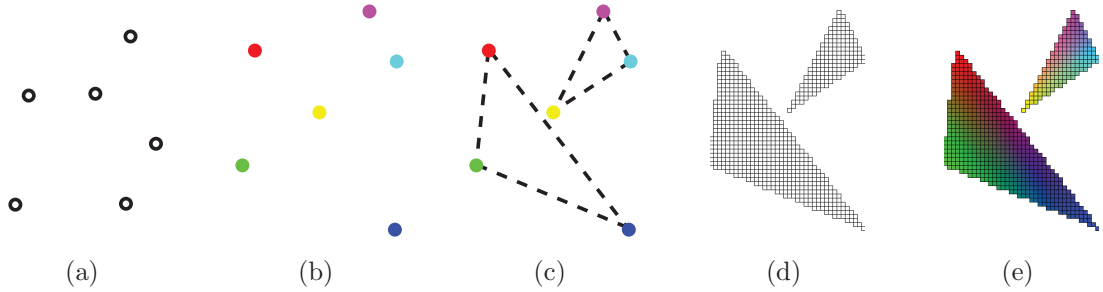


Figure 2.2: Visualizing the graphics pipeline using two triangles. (a) Input vertices; (b) Processed vertices; (c) Assembled triangles; (d) Rasterized fragments and (e) Processed fragments.

such as illuminated colors, are computed. The processed vertices are shown in Figure 2.2(b). These processed vertices are then assembled into the geometries – the two triangles, as shown in Figure 2.2(c). These triangles are rasterized into many fragments (Figure 2.2(d)). The fragments are processed in the Fragment Processing stage, and the processed fragments (Figure 2.2(e)) are then send to the Raster Operations stage for displaying on the screen.

2.2 Evolution of GPU

The term GPU is introduced by NVIDIA to refer to a powerful graphics processor that is comparable to the CPU. The GPU performs most of graphics operations on hardware, and thus frees the CPU for other tasks. Furthermore, programmers can write their own programs on the GPU. There is not a widely accepted taxonomy of the generations of the GPU. In this section, we briefly introduce the evolution of the GPU, and divide the generations following the taxonomy in *Cg Tutorial* [FK03].

Pre-GPU

In the early beginnings of computer graphics, there is no special graphics hardware. All operations in the pipeline shown in Figure 2.1 are performed by software on the CPU. However, there are already some solutions for producing high quality graphics. Silicon Graphics (SGI) and Evans and Sutherland (E&S) both provide special graphics cards. These cards are specially designed for graphics purpose, and there are some basic graphics components in them which can perform simple operations, such as vertex transformation and texture mapping. These systems are important to the development of computer graphics, but have not become popular because of their high specialities and high costs.

First Generation

With the evolution of the technology and the increasing requirements of higher quality of graphics, there are more and more new graphics hardware being produced. This generation includes NVIDIA's TNT2, ATI's Rage 128, 3dfx's Voodoo 3, etc. However, this generation is not strictly considered as the GPU. The functions of these graphics cards are very limited. They implemented a few graphics operations, such as rasterization, texture mapping, in hardware, so that the CPU can be freed from them, and the frame rate is thus greatly increased. However, this generation of graphics cards lacks of the ability to do vertex transformation. These operations are still performed on the CPU. So, this generation is better called graphics accelerator rather than GPU.

Second Generation

The word GPU is coined by NVIDIA to refer to their GeForce 256 card. By using this name, NVIDIA means that GeForce 256 is no longer a merely graphics accelerator, but a real processor like the CPU. Besides GeForce 256, there are many other GPUs belong to this generation which includes NVIDIA's GeForce 2, ATI's Radeon 7500, S3's Savage3D etc.

This first generation of "real" GPU can do vertex transformation and lighting calculation in hardware. This feature is called "hardware T&L (Transformation & Lighting)", a term occurring in almost all the advertisements of the this generation of GPUs. However, all the vertex operations in hardware are still fixed, i.e. they cannot be changed by programmers.

Third Generation

This generation includes NVIDIA's GeForce 3 and GeForce 4 Ti, ATI's Radeon 8500, etc. This is the first generation of GPUs having programmability functions. For the first time, programmers can write their own programs to control the operations in the Vertex Processing stage. This greatly increases the flexibility of the possible applications of the GPU. Such programs are called *Vertex Programs*. Writing vertex programs instead of using the build-in fixed functions to perform vertex processing is a milestone for general-purpose computation on the GPU. With the help of vertex programs, programmers can do their own general-purpose computation in the Vertex Processing stage. However, this generation of GPU lacks the programable ability for the Fragment Processing stage. This is only a transitional generation.

Fourth Generation

This generation includes NVIDIA's GeForce FX, ATI's Radeon 9700, 9800, etc. This generation provides fully programmability for both Vertex Processing and Fragment Processing stages. In this generation, programmers can write programs which are executed on every fragment (called *Fragment Programs*) to control the Fragment Processing stage. More importantly, this generation fully supports the IEEE standard of 32-bit float numbers. Therefore, the values stored in texture can be arbitrary float number, instead of that in $[0, 1]$. This feature is very important to general purpose computations, since most of the applications use real numbers which are smaller than 0 or larger than 1. The vertex program in this generation supports branching and looping commands. The accessing of textures is more flexible in this generation. Programmers can use variables as texture coordinates to access the texture, and thus precalculated look-up table becomes possible for the general-purpose computations.

Fifth Generation

This generation includes NVIDIA's GeForce 6800, 7800, ATI's Radeon X800, X1800, etc. Many new features are introduced to this generation of GPU. One of the most important features is the vertex program in this generation, similar as the fragment program, can access texture. This expands the possibility of the general-purpose computation. Another important feature is the multiple rendering target (MRT). With MRT, a fragment program can output to up to four textures at the same time. So programmers can write more result values simultaneously. Previously, any computation that requires more than four values must be calculated

in separated passes instead.

Dynamic branching in the vertex program is another important new feature. It can greatly increase the speed of the program. For a vertex program executed on all the vertices, when there is a branching command in it, according to different conditional values on different vertices, some vertices may choose the "true" branch, while the others choose the "false" branch. In the former generations, where only static branching is supported, both branches must be executed on all the vertices, and every vertex then chooses only one result according to the their conditional values. With the new dynamic branching in this generation, only the corresponding branch is executed on every vertex, just like what happened on the CPU. However, the fragment program in this generation still supports static branching only.

Sixth Generation

NVIDIA's GeForce 8800 belongs to this generation. GeForce 8800 is a breakthrough of the structure of the GPU. Before this generation, vertex programs and fragment programs are executed by separate parts of the hardware. From this generation on, *unified structure* is introduced in the GPU. Now there is no longer the vertex processing part and the fragment processing part. Both of them are replaced by the new unified processing part, which can execute vertex programs, fragment programs and geometry programs. *Geometry programs* are programs for the Primitive Assembly stage. They allow programmers to write their own programs to control this stage. With the help of geometry programs, new vertices can be generated while some input vertices can be discarded. Geometry programs may

lead to many more new GPGPU applications. There are many other new features in this generation, such as integer texture, float depth buffer, etc.

In this thesis, we only employ the new functionalities of NVIDIA GeForce 8800 in Chapter 6, since all the other work is done prior to the introduction of the GeForce 8800 by NVIDIA.

2.3 GPU Programming Languages

To utilize the GPU for general-purpose computations, we need a programming language to write the vertex program and the fragment program (and the geometry program for GeForce 8800). Similar to the CPU, we can directly use the assembly language for the GPU to write these programs. However, assembly language is too complicated to use. Fortunately, there are many high-level programming languages which are much easier to use than the assembly language.

Currently, there are three widely used high-level languages on the GPU: HLSL [Mic05], GLSL [Kes06] and Cg [MGAK03], . Since they are all designed for graphics purposes, they are sometimes called *shading language*. The main functions of all these three languages are similar. However, they work differently for the two main graphics libraries – DirectX and OpenGL. HLSL (High Level Shading Language) is developed by Microsoft, and has been a part of DirectX. It is only compatible with DirectX and Windows operating system. GLSL (OpenGL Shading Language) on the other hand, as the name suggested, can only work with OpenGL. The advantage of it is that it can work on different operating systems as long as OpenGL is supported. It has been a part of OpenGL 2.0. Cg (C for graphics) is developed by NVIDIA. Cg is highly similar to HLSL, but it can work together with both DirectX

and OpenGL, and thus can work on different operating systems (with OpenGL only). In this thesis, all of the experiments are done by Cg. The details of Cg can be found in *Cg Tutorial* [FK03]. Besides these three main shading languages, there are some other shading languages such as Sh [MTP+04] and ASHLI [BP03].

There are also many programming languages on the GPU which are no longer designed for graphics purposes. Some of them are similar to the programming language on the CPU and do not have any relationship to graphics at all. These programming languages include Brook [BFH+04], Scout [MIA+04], Accelerator [TPO06], CGiS [LFW06], etc. Most recently, NVIDIA also releases its own GPGPU language – CUDA [NVI07].

2.4 Typical Usage of GPU

In this thesis, the GPU is used as an SIMD (**S**ingle **I**struction **M**ultiple **D**ata) machine. SIMD machine is a type of parallel machine. It is able to execute same instructions on different data in parallel. Similarly, the GPU can execute the same program (vertex program or fragment program) on different data (stores in the vertices or fragments) in parallel.

When we program on the GPU, the texture behaves as the RAM for programs on the CPU. The values can be read from the texture, and the results be written into the texture. The processors in the GPU (called *stream processors*) work as many small independent CPUs. The instructions in the fragment program are executed by the stream processors in parallel on all the pixels at the same time[1]

[1]This is only conceptually true. In the real GPU, the number of the pixels that can be processed at the same time depends on the number of stream processors in the GPU. For example, in NVIDIA GeForce 8800, there are 128 stream processors. So maximum of 128 pixels can be processed in parallel. However, this is transparent to the programmer. Programmers can just

to process the data.

A simple way to execute the fragment program on all the pixels is to render a screen-size quad, which covers the whole texture. The quad is rasterized into many fragments, one fragment for one pixel. The fragment program is thus executed on all the pixels. The fragment program reads data from the texture, performs the computation, and writes the results back into the texture. This process can be repeated many times, which is called *multi-pass* process. These passes can use different fragment programs to perform different tasks. Every pass takes input from the previous pass, and outputs results as the input for the next pass. At the end of the last pass, the results stored in the texture can be read back to the CPU for further processing.

In this thesis, we mainly use the fragment program as described above, while the vertex program is seldom used. We only perform some simple computations in vertex program, and use the rasterizer to interpolate the values on the four vertices of the quad to all the fragments. For example, when we want to compute an offset value on all the pixels, we can compute this value only on the four vertices of the quad. The fragments on all the pixels can get their values by interpolation. However, since the Rasterizer stage can only do bi-linear interpolation, only very simple values, such as the offset value in the above examples, can be computed in this way. Since the newest geometry program occurs later than most of the work in this thesis, and the current version of Cg does not support it[2], we have not used the geometry program in this thesis.

imagine that all the pixels are processed in parallel.

[2]A beta version of Cg included NVIDIA SDK 10.0 has already provided limited support for the new features of NVIDIA GeForce 8800, including the geometry program. However, at the writing time of this thesis, this version is not officially released yet.

Chapter 3

Jump Flooding Algorithm

The propagation of information is a very common task in many applications. For example, when using the "Paint Bucket" tool in an image processing software (e.g. Photoshop), we click a point and give it a certain color. The purpose is to propagate the information of this color from this point to all the other points in the region containing the point. Or, in other words, the purpose is to fill the region containing this point with the specific color. Generally speaking, given one or more points containing initial information, we want to propagate the information from them to all or some of the other points in the space.

The standard flooding algorithm is a naive way to perform this task. The information is propagated outwards in a way similar to the ripple effects. However, the standard flooding algorithm is slow and is thus not suitable for many real-time applications. In this chapter, a novel algorithm – Jump Flooding Algorithm (JFA) is proposed. JFA is exponentially faster than the standard flooding algorithm. So it fits many real-time applications much better than the standard flooding algorithm. Several properties of JFA are proven in this chapter. These properties

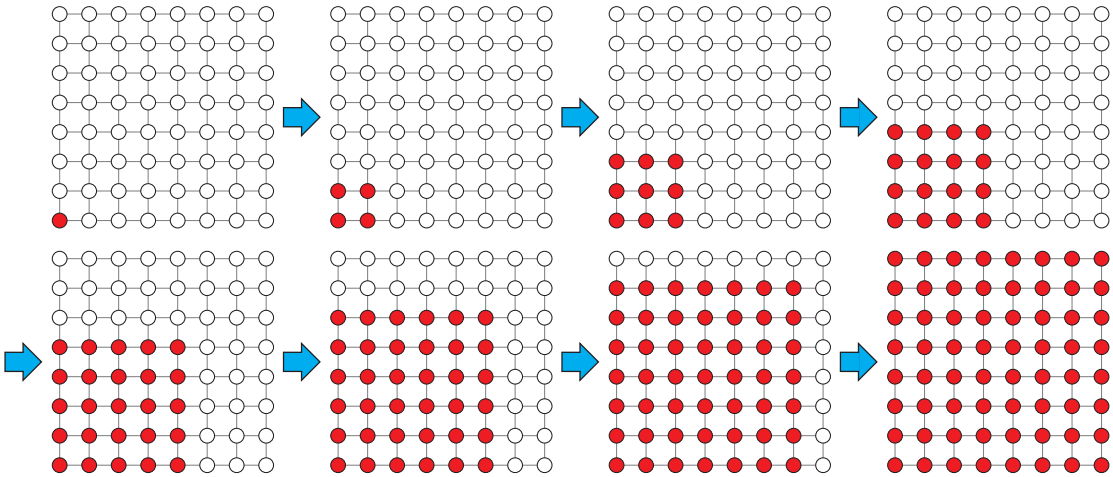


Figure 3.1: The process of the standard flooding process. The resolution of the grid $n = 8$.

form a theoretical foundation of the proposed algorithm.

3.1 Overview of Algorithm

Suppose we have a pixel containing certain information at the lower left corner of an $n \times n$ grid (shown as red grid point in Figure 3.1), and we want to propagate its information to all the other grid points. In other words, we want to fill the whole grid with the same information as this grid point. The term *seeds* refers to such points which contains the initial information.

As mentioned above, a simple way is to use the standard flooding algorithm as shown in Figure 3.1. In every step, or *pass*, the information is passed forward by one grid point. After $n - 1$ passes, the whole grid is filled. In this standard flooding algorithm, the number of passes required is linear to the resolution of the grid.

When considering the standard flooding process carefully, we find that every

colored grid point is used effectively only once. In every pass, only those on the front of the propagation are useful, while the other internal colored grid points are not. This is not an efficient use of the computational cycles. To remedy the situation, we introduce the idea of jump flooding.

In the case of the standard flooding algorithm, a grid point (x, y) passes its information to its (maximum) eight neighbors at $(x+i, y+j)$, where $i, j \in \{-1, 0, 1\}$. We call this a pass with the *step length* of 1. The step length in the standard flooding algorithm is a constant value of 1 in all the passes. A more efficient way is to vary the step length in every pass. There are two possible ways as shown in Figure 3.2. The approach in Figure 3.2(a) starts the step length of 1 and then doubles the step length in every subsequent pass, while the approach in Figure 3.2(b) starts from a big step length and then halves the step length in every subsequent pass, until the step length reaches 1. Formally, in a pass with the step length of k, the grid point (x, y) passes its information to its (maximum) eight neighbors at $(x+i, y+j)$, where $i, j \in \{-k, 0, k\}$. In the two approaches, the numbers of passes needed to fill the whole grid are both logarithmic to the resolution of the grid.

The above discussion can be generalized to work on more than one seed, which leads to the *jump flooding algorithm (JFA)*. In a grid with the resolution of $n \times n$, JFA can propagate the information of several seeds at the same time. In the following discussion, without loss of generality, we assume n is a power of 2. There are $\log n$ passes in JFA. The step length of the first pass is $n/2$, and is halved in every subsequent passe, until the step length of 1. In a pass with the step length of k, each grid point (x, y) passes its information (if any) to the 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 information from (maximum) eight other grid points at $(x'+i, y'+j)$, where

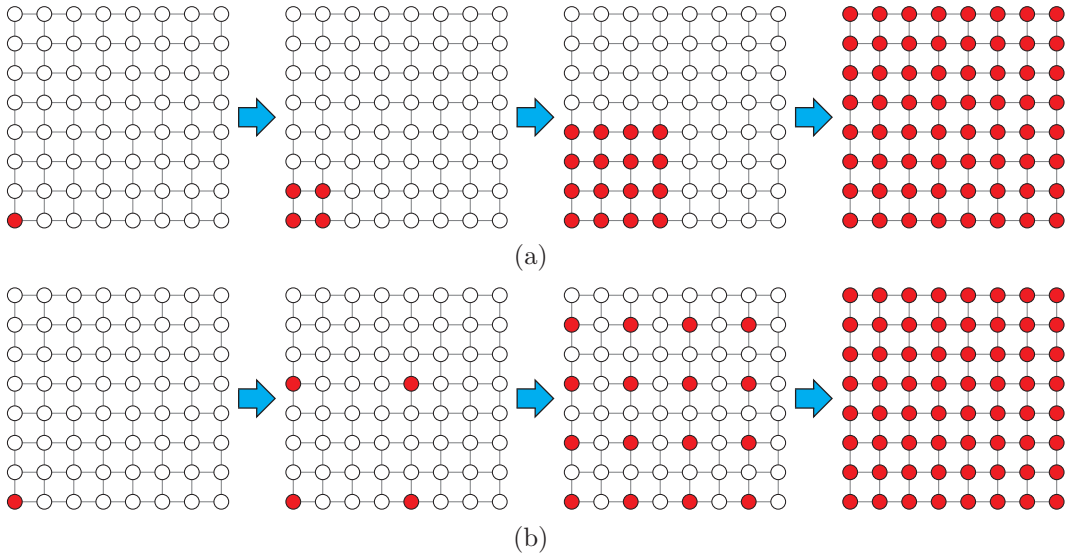


Figure 3.2: Two more efficient approaches for propagating the information of the red grid point using (a) doubling step lengths and (b) halving step lengths.

$i, j \in \{-k, 0, k\}$. Among these information, together with its own information (if any), a certain criterion is used to select the information from a "best" seed. The information of this "best" seed is stored at this grid point, and passed on in the subsequent passes. The choice of the criterion depends on the application. We explain it further in later chapters.

A few notes are in order here. First, for a grid point p to receive the information from a seed s as its best seed been found (at the end of some particular pass of the flooding process), the information of s has traveled through a sequence of grid points $p_1, p_2, \ldots p_k (= p)$, where each is from a different pass of a progressively smaller step length, and p_i passes its information to p_{i+1} till it reaches p_k. Such a sequence forms a *path* from s to p. We note that at any one pass, there can be more than one grid point passing the same information of s to p_{i+1}; such case results in more than one path from s to p_{i+1}. As a result, there is more than one path from s to p.

Second, it must be clearly pointed out that JFA cannot always guarantee the exact results. Depends on the applications, and thus the criterion used in JFA, the result may contains some errors. In other words, the results of JFA for some applications are only approximations of the exact results. An error occurring in JFA implies that some grid points are unable to obtain the information from their actual best seeds.

For a grid point p to receive the information from its best seed s, the sufficient condition is that there is at least one path where all the grid points on it having s as their best seeds. However, this condition is not always satisfied, and thus error may occur when the condition fails. Note that this condition is only a sufficient condition, but not a necessary one. In other words, even when this condition fails, p may still receive the information from s correctly.

Figure 3.3 shows an example of an error of JFA. In this example, JFA is used to computer the Voronoi diagram (see details in Chapter 4). The seeds contain their coordinates as their initial information. The best seed for a grid point is the nearest seed to it. In Figure 3.3, the nearest seed of the grid point p is the seed r. However, under this particular configuration, JFA cannot pass the information of r to p. Instead, p receives the information from g (or b) at the end of the flooding process. This is because for r to reach p, there must be a path passing through either p' at $(10, 6)$ or p'' at $(10, 8)$. But neither p' nor p'' selects r as its nearest seed and hence such a path does not exist. So, the seed r is *killed* by the seed g (or b) at the grid point p' (or p''). However, in practice (see later chapters), the results of JFA give very good approximations of the exact results, and only a small percentage of errors may occur in the final results. So, JFA is suitable for most of the real applications.

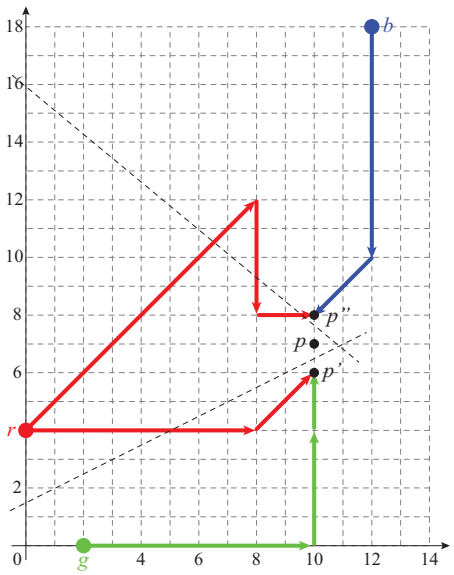
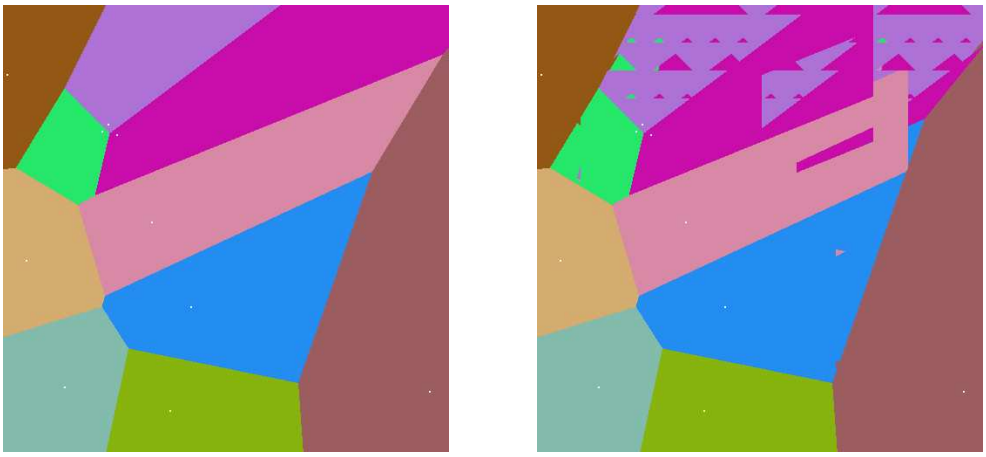


Figure 3.3: An example where a grid point p does not receive the correct nearest seed r with the jump flooding algorithm.

Third, Figure 3.2(a) alludes to an alternative to JFA as a jump flooding starting with the step length of 1 and then doubling the step length in each pass. This alternative, however, does not work as well as JFA; it generates a lot more errors than JFA. Figure 3.4 shows the results of the Voronoi diagram of ten point sites using JFA and this alternative approach. This phenomenon 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 nearest seed so far when it receives the information of s. 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 passes and thus do not permit other seeds (such as s) from passing on to other grid points. In contrast, JFA tends not to finalize the closest seeds for all grid points until much



(a) (b)

Figure 3.4: Voronoi diagram of ten point sites with (a) no error in the result of JFA and (b) many errors in the result of the doubling step lengths approach.

later passes. This means each grid point permits many other seeds to temporarily be its nearest seeds in order to pass them on to other grid points, and JFA thus makes fewer errors.

In the next section, we discuss some important properties of the paths in JFA. The corresponding proofs are also given.

3.2 Paths in JFA

As defined before, when the information of a seed s is passed to a grid point p, it travels through a sequence of grid points: $p_0(=s), p_1, p_2, \ldots p_k(=p)$, and these grid points form a *path* from s to p. This section discusses some of the important properties of the path. These properties are exploited in JFA and are of independent interests, possibly to other algorithms adapting the jump flooding concept. For simplicity, within this section, we assume there is only one seed s in a grid with the resolution of $n \times n$.

Property 3.1 *Regardless of the position of s in the grid, JFA fills the grid with the information of s.*

Proof. In the following, we discuss a constructive proof (that can be extended to show the validity of Property 3.2).

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 any integer coordinate (p_x, p_y). We first suppose p_x and p_y are both positive integers. Let $(x_{m-1}, x_{m-2} \dots x_0)_2$ be the binary form of p_x and $(y_{m-1}, y_{m-2} \dots y_0)_2$ of p_y. Then, a path from s to p can be obtained as follows: At the step length of l (where l is either $n/2, n/4, \dots$, 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) is negative integer, we modify only the relevant part of the rules involving p_x or p_y respectively. □

This property tells us that in the result of JFA, there cannot be any grid points untouched. Even if the there is only one seed in the grid, the whole grid is guaranteed to be filled after the pass with the step length of 1. The next property discusses the number of paths from a seed s to a grid point p.

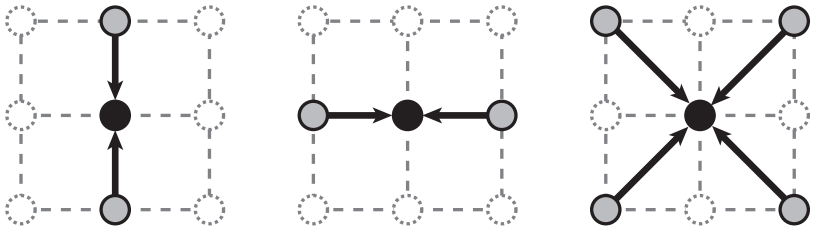
Property 3.2 *Let s be a seed at (x, y), and p be a grid point. Suppose there exists another grid point s' at (x', y') where $|x - x'| = 2^l$ and $|y - y'| = 2^l$ for a positive integer l, and p lies within the square $2^l \times 2^l$ with s and s' as its two (out of four) opposite vertices. Then, JFA generates more than one path from s to p.*

Proof. Without loss of generality, we assume s is at $(0, 0)$ 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 3.1. A path from s to p can first make a move to s' via the pass with the 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 3.1, 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 grid point p' (i.e. the grid point 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 3.5 for an illustration.

Property 3.3 *The paths JFA can generate from a seed s to a grid point p are*



(a) type-v (b) type-h (c) type-d

Figure 3.5: Three types of paths (only the last step is shown). The black point is p and the gray points are possible p'.

all of 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 $n/2 \times n/2$ square grids with the exception at the boundary of the given grid that may be incomplete $n/2 \times n/2$ square grids. At this point, the information of 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, the information of s is known at the junctions of the vertical and horizontal lines.

So, with respect to a grid point p, it receives the information of s if, and only if, it becomes a junction of a vertical and a horizontal line. There are three possibilities at the pass before it becomes a junction. First, p is on a vertical line, and the grid point passing it the information of s 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 a horizontal line, the paths from s to p are of type-h. Third, p is at the center of

some square grid (not on any lines), and the paths from s to p are of type-d.

Note that once p receives the information of s, it cannot receive in subsequent passes the same information. This is because the subsequent passes are with smaller step lengths and p thus does not refer to any junctions that have the information. So, 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. □

From the proofs of Property 3.1 and Property 3.3, we can deduce the relationship between the type of the paths from s to p and the x- and y-coordinates of s and p. Because all the paths from s to p are of the same type, the path constructed in the proof of Property 3.1 must be of the same type to all the other paths. According to the proof, the movement in the last step is decided by the last digit where x_k and y_k are not both 0. So if we shift the origin of the coordinate system to s, the new coordinate of p can tell us the type of path from s to p.

Assume the original coordinates of s and p are (x_s, y_s) and (x_p, y_p) respectively. The new coordinate of p is $(x_p - x_s, y_p - y_s)$. Assume x_k and y_k are the last digits in this new coordinate, where x_k and y_k are not both 0.

If $x_k = 0$ and $y_k = 1$, the path from s to p is of type-v;

if $x_k = 1$ and $y_k = 0$, the path from s to p is of type-h;

and if $x_k = 1$ and $y_k = 1$, the path from s to p is of type-d;

The pass when the information of s finally reaches p is also decided by the coordinates of s and p. 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 $\mathsf{order}(s, p) = 1$.

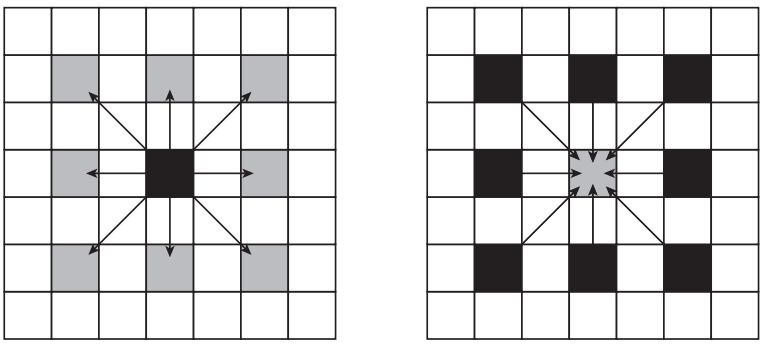
Property 3.4 *All the paths generated by JFA from a seed s to a grid point p reaches p at the pass with the step length of 2^l where $l = \mathsf{order}(s, p)$.*

Proof. Let us visualize the progress of JFA as in the proof of Property 3.3. Completing the pass with the 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 the 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 $(l + 1)^{th}$ 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 pass with the step length of 2^l where $l = \mathsf{order}(s, p)$. □

3.3 Implementation on GPU

Jump flooding algorithm is parallel in nature and thus is very suitable for GPU programming. It is inefficient to implement JFA on the CPU sequentially. When implementing JFA on the GPU, the grid is mapped to a 2D texture with the same resolution of the grid, and the grid points are mapped to the pixels. Every pass of JFA is implemented by rendering a quad of the same size as the texture, which triggers the fragment program for JFA on all the pixels.

Due to the limitation of the current GPU, we need to adapt JFA slightly to implement it on the GPU. In particular, the current GPU does not allow “scatter”



(a) Scatter (b) Gather

Figure 3.6: Illustration of scatter and gather operations. Black pixels are those having information and gray pixels are those we want to write information to. Fragment program is executed on the the center pixel.

operations (Figure 3.6(a)), and only allows "gather" operations (Figure 3.6(b)). In other words, a fragment program executing on a certain pixel p can read (gather) the information from many other positions (pixels), but cannot write (scatter) the information to any positions other than its own position. So when we want to pass the information from one pixel p to the (maximum) eight other pixels around it, we cannot run a fragment program on p and write the information to the eight pixels. Instead, we must reverse the process – run the fragment program on the eight pixels to read the information from p.

With the above understanding, JFA can be implemented as follows. We use two textures of the same size as the grid. In every pass, we read information from one texture (called *read-texture*), perform the computation, and write the results into another texture (called *write-texture*). Then we exchange the roles of these two textures in the next pass. Such a pair of textures is known as a *ping-pong buffer*. In the initialization step, the information of the seeds is written into the read-texture for the use of the first pass of JFA.

In every pass of JFA, we draw a quad of the same size as the texture. Thus a fragment program is executed on all the pixels. In the fragment program running on the pixel at (x, y), the information of different seeds is read from the read-texture using texture coordinates of $(x + i, y + j)$, where $i, j \in \{-k, 0, k\}$ and k is the current step length. Using the information read from these pixels, the criterion of JFA is computed and the best seed is determined. At last, the information of this best seed is written into the write-texture at the same position where the fragment program is executing for the use of the next pass.

In order to avoid calculating the eight texture coordinates on every pixel, we move the calculation into a vertex program. Thus such a calculation is performed only four times (on the four vertices of the quad). The results on these four vertices are correctly interpolated on all the pixels. Listing 3.1 shows the vertex program in Cg where we pack the eight texture coordinates into four `float4` variables.

Listing 3.2 shows the fragment program. This is a general prototype of the fragment program of JFA where we include pseudo-codes written in bold italic fonts. These pseudo-codes are replaced by the real codes according to the application, as explained in later chapters. The type of the variable `Information` is dependant to the application.

```
void main(float4 position      : POSITION,

          out float4 oPosition : POSITION,
          out float4 oCoord01  : TEXCOORD0,
          out float4 oCoord23  : TEXCOORD1,
          out float4 oCoord56  : TEXCOORD2,
          out float4 oCoord78  : TEXCOORD3,

          uniform float4x4 cameraModelViewProj,
          uniform int2 screenSize,
          uniform int k) // k is the current step length
{
    // Transform position from object space to clip space
    oPosition = mul(cameraModelViewProj, position);

    // map the position from [-1, 1]*[-1, 1] to
    // [0, screenWidth]*[0, screenHeight]
    float2 posRECT = (oPosition.xy+1.0)/2.0*screenSize;

    float4 pos4 = float4(posRECT, posRECT);
    oCoord01 = pos4 + float4(-k, -k, 0, -k);
    oCoord23 = pos4 + float4(k, -k, -k, 0);
    oCoord56 = pos4 + float4(k, 0, -k, k);
    oCoord78 = pos4 + float4(0, k, k, k);
}
```

Listing 3.1: Vertex program of JFA

```
void main(float4 position : WPOS,
          float4 Coord01  : TEXCOORD0,
          float4 Coord23  : TEXCOORD1,
          float4 Coord56  : TEXCOORD2,
          float4 Coord78  : TEXCOORD3,

          out float4 oColor : COLOR,

          uniform samplerRECT readTex)
{
    SomeType Information;

    Information = tex2D(readTex, Coord01.xy);  // neighbor 0
    Compute a criterion value using Information;
    Information = tex2D(readTex, Coord01.zw);  // neighbor 1
    Compute a criterion value using Information;
    Information = tex2D(readTex, Coord23.xy);  // neighbor 2
    Compute a criterion value using Information;
    Information = tex2D(readTex, Coord23.zw);  // neighbor 3
    Compute a criterion value using Information;
    Information = tex2D(readTex, position.xy); // itself
    Compute a criterion value using Information;
    Information = tex2D(readTex, Coord56.xy);  // neighbor 5
    Compute a criterion value using Information;
    Information = tex2D(readTex, Coord56.zw);  // neighbor 6
    Compute a criterion value using Information;
    Information = tex2D(readTex, Coord78.xy);  // neighbor 7
    Compute a criterion value using Information;
    Information = tex2D(readTex, Coord78.zw);  // neighbor 8
    Compute a criterion value using Information;

    float4 bestInfo;
    Using the criterion values to select the best seed, and
    store its Information into bestInfo;

    oColor = bestInfo;
}
```

Listing 3.2: Fragment program of JFA

Chapter 4

Voronoi Diagram and Distance Transform

Voronoi diagram and distance transform are two very important concepts in computational geometry. They have been used in numerous applications in many different fields, such as computer graphics, image processing, computer vision, robotics, etc. Many applications require the efficient computation of Voronoi diagrams or distance transforms in order to achieve real-time speeds. In this chapter, we discuss the use of JFA to compute Voronoi diagrams and distance transforms in real-time. Due to the discrete nature of the GPU, the computation in this chapter works in discrete space only.

We first give the definition of Voronoi diagram and distance transform, and the corresponding discrete versions of their definitions. Next, JFA and some variants of JFA are introduced for the computation of the Voronoi diagram. Some new properties of JFA on errors are proven in this chapter too. Next, we show our experimental results of JFA and its variants. Finally, we discuss the extension of

JFA to higher dimension by using the CPU to simulate it and by using a new variant to compute the 3D Voronoi diagram in a slice-by-slice manner.

4.1 Definitions

In this section, we give the definitions of Voronoi diagram and distance transform, and then provide the corresponding discrete versions of their definitions.

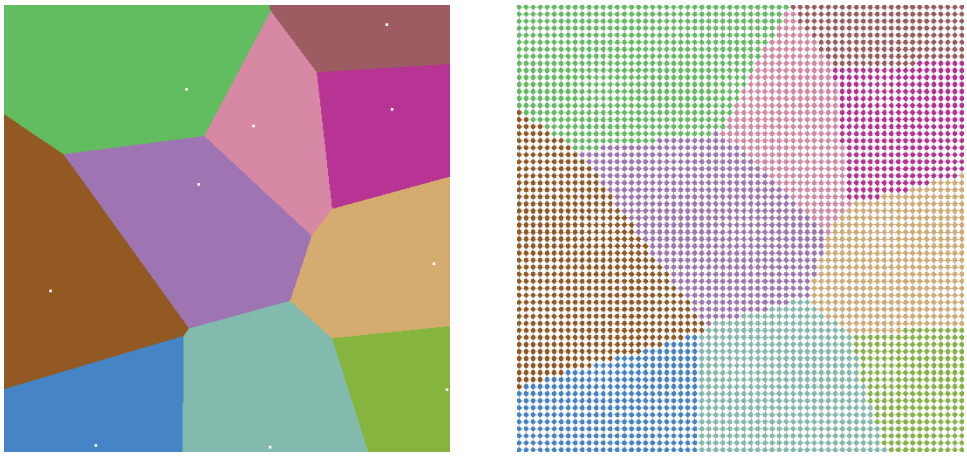
4.1.1 Voronoi Diagram

Let Ω be a bounded region in $\mathbb{R}^2$, and let $S = \{s_1, s_2, \ldots, s_n\}$, $2 < n < \infty$, be a set of mutually disjoint objects in Ω, where an object can be a point, or a set of points such as line segment, curve, or polygon. These objects are called *sites*. For any point $p \in \Omega$ and any site $s_i \in S$, denote $d(p, s_i)$ as the *distance* between p and s_i. For a site $s_i \in S$, its *Voronoi region* $R(S; s_i)$ is defined as follows:

$$R(S; s_i) = \{p \in \Omega | d(p, s_i) < d(p, s_j), \forall s_j \in \Omega, i \neq j\}$$

The collection of all the Voronoi regions $R(S; s_1), R(S; s_2), \ldots, R(S; s_n)$, forms a partition of Ω, and is called the *Voronoi diagram* of S with respect to the distance d. The curves incident to two Voronoi regions are called *Voronoi edges*, and the points incident to three or more Voronoi regions are called *Voronoi vertices*.

The simplest Voronoi diagram is the one where all the sites are points and the distance is Euclidean distance. In such a case, all the Voronoi regions are polygons and all the Voronoi edges are line segments. An example of the Voronoi diagram of ten point sites with Euclidean distance is shown in Figure 4.1(a). The



(a) (b)

Figure 4.1: (a) Voronoi diagram of ten point sites; (b) Corresponding discrete Voronoi diagram (for clarity, a low resolution of 64×64 is used here).

other types of Voronoi diagrams are usually called *generalized Voronoi diagrams*, which include Voronoi diagrams of other types of sites, and Voronoi diagrams using other distance metrics, such as Manhattan distance, weighted distance, etc. More details about other types of Voronoi diagrams and their properties can be found in [OBSC99].

In this chapter, we focus on the Voronoi diagram defined in discrete space (a grid) $G = \Omega \cap \mathbb{Z}^2$. The *discrete Voronoi diagram* is a sampling of the Voronoi diagram in continuous space, and is defined as follows. For a grid point $(j, k) \in G$, define $s(j, k) = i$ if $(j, k) \in R(S; s_i)$. For simplicity, grid points lying on Voronoi edges or Voronoi vertices are arbitrarily assigned to any Voronoi regions incident to them. So, $s(j, k)$ is the ordinal number of the site whose Voronoi region contains this grid point. The *discrete Voronoi region* is then defined as follows:

$$D(S; s_i) = \{(j, k) \in G | s(j, k) = i, s_i \in S\}$$

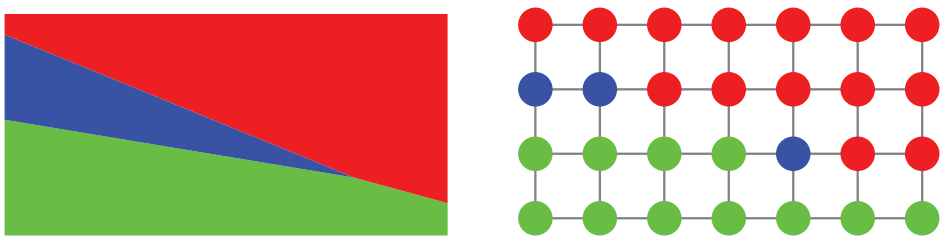
The collection of all the discrete Voronoi regions $D(S; s_1), D(S; S_2), \ldots, D(S; s_n)$ forms a partition of the grid G, and is called the *discrete Voronoi diagram* of S.

We define two kinds of neighborhood for the grid points. For a grid point p at (j, k), we define its 4-*connected neighbors* $NB_4(p) = \{(j-1, k), (j+1, k), (j, k-1), (j, k+1)\}$, and its 8-*connected neighbors* $NB_8(p) = NB_4(p) \cup \{(j-1, k-1), (j+1, k-1), (j-1, k+1), (j+1, k+1)\}$. For a grid point, if all its neighbors and itself belong to two discrete Voronoi regions, it is on *discrete Voronoi edges*; if all its neighbors and itself belong to three or more discrete Voronoi regions, it is a *discrete Voronoi vertex*. Figure 4.1(b) shows an example of the discrete Voronoi diagram corresponding to the Voronoi diagram in Figure 4.1(a).

Similar to the Voronoi diagram in continuous space, there are many kinds of *generalized discrete Voronoi diagrams* with different kinds of sites and/or different distance metrics. In the following discussion in this chapter, unless otherwise specified, we assume the sites are point sites, and the distance metric is Euclidean distance.

One important difference between the continuous Voronoi diagram and the discrete Voronoi diagram needs to be pointed out. In continuous space, a Voronoi regions is always a *connected* area – a polygon. However, this is not always the case in discrete space, even with Euclidean distance. A discrete Voronoi region may contains one or more disconnected components. A simple example of a disconnected Voronoi region is show in Figure 4.2.

Voronoi diagram is a very important data structure in computational geometry. It has found applications in numerous areas including computer graphics, image processing, bioinformatics, mechanics etc. For more details, see good surveys in [Aur91] and [OBSC99].



(a) (b)

Figure 4.2: Example of a disconnected Voronoi region. The blue Voronoi region is a connected region in (a) continuous space, but is disconnected in (b) discrete space.

4.1.2 Distance Transform

A closely related notion of Voronoi diagram called distance transform is introduced by Rosenfeld and Pfaltz [RP66] for image processing applications. We define the distance transform here using similar notations as in the definition of the Voronoi diagram above.

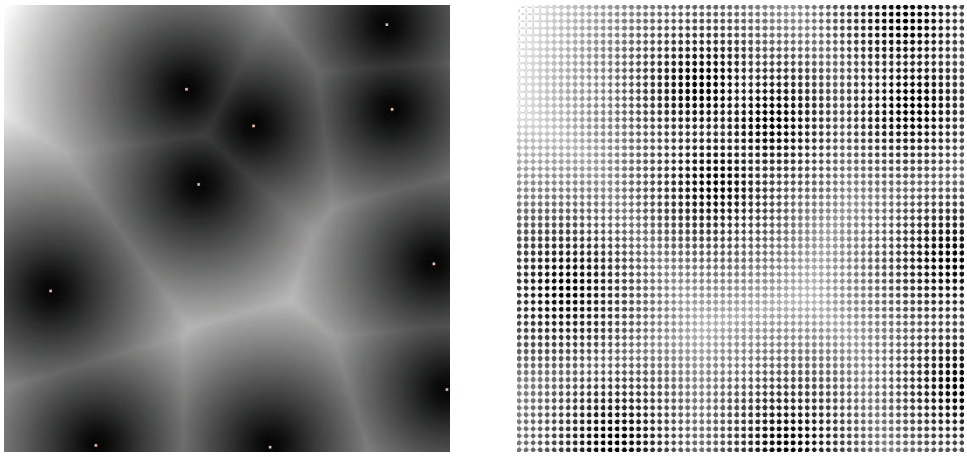
Let Ω be a bounded continuous space in the plane, and let $S = \{s_1, s_2, \ldots, s_n\}$, $2 < n < \infty$, be a set of sites, which can be points, or sets of points such as line segments, curves, or polygons. Distance transform assigns a real value for any point $p \in \Omega$ as follows:

$$dt(p) = \min_{s_i \in S}\{d(p, s_i)\}$$

So, $dt(p)$ is the distance from p to its nearest site. In some applications, both the distance and the nearest site are required. Under such a requirement, the distance transform is defined as a 2-tuple:

$$dt(p) =< \arg\min_{s_i \in S}\{d(p, s_i)\}, \min_{s_i \in S}\{d(p, s_i)\} >$$

where the first element indicates the nearest site of p, while the second element is



(a) (b)

Figure 4.3: (a) Result of distance transform of the ten point sites same as in Figure 4.1(a); (b) Result of the corresponding discrete distance transform (for clarity, a low resolution of 64×64 is used here).

the distance from this site to p. Figure 4.3(a) shows the result of distance transform of the ten point sites same as those in Figure 4.1(a). The colors in Figure 4.3(a) represent the modulated distances from every point to its nearest site. Brighter colors represent further distances, and darker colors nearer.

Similar to the Voronoi diagram, we can define the distance transform in discrete space, where only grid points in the grid $G = \Omega \cap \mathbb{Z}^2$ have values. Figure 4.3(b) shows the result of the discrete distance transform corresponding to the distance transform in Figure 4.3(a).

To date, the distance transform has found many uses in applications beyond image processing, such as computer graphics, pattern recognition, etc. Some example applications are: skeleton computation [Mon68, Dan80], mathematical morphology operation [Rag92a], and displacement mapping [WWT+03, Don05]. For more applications of the distance transform, see a good survey in [Cui99].

Because the Voronoi diagram encodes the information of the nearest sites for

all the points, it is straightforward to compute the result of the distance transform in a constant time after having the Voronoi diagram. Therefore, in this chapter, we focus on using JFA on the computation of the Voronoi diagram only. All the analysis are based on the Voronoi diagram too.

With the definitions of the Voronoi diagram and the distance transform, we next briefly review some related work of them.

4.2 Related Work

As mentioned before, it is straightforward to obtain the distance transform through the Voronoi diagram. In the following, we present the previous researches as they were originally discussed either for the Voronoi diagram or the distance transform. On the former, we discuss in Section 4.2.1 known approaches using the GPU to compute the Voronoi diagram in linear time to the number of sites. These are compared with our algorithm using JFA that runs in constant time once the input sites are stored in a texture. More related work on the Voronoi diagram can be found in [Aur91, OBSC99]. On the latter, we outline in Section 4.2.2 the development of fast-approximate and slow-exact distance transform algorithms, and some conventional approaches adapted to parallelize these sequential algorithms. A comprehensive survey on the distance transform algorithms can be found in [Cui99].

4.2.1 Voronoi Diagram

Hoff et al. [HCK+99] computes a 2D Voronoi diagram by drawing (right-angle) cones with their apexes at the positions of sites. The image of the cones, as viewed

orthogonally from the space of the sites, is a Voronoi diagram of the set of sites. The algorithm thus relies on the GPU to efficiently rasterize the cones in linear time to the total number of triangles used to represent the cones. As pointed out in [Den03], one needs a lot more triangles per cone than suggested in [HCK+99] to compute the correct Voronoi diagram in some extreme cases. This is not favorable for a large number of sites.

Denny [Den03] presents a variant of the above algorithm by drawing a quad with a precomputed depth texture, instead of a cone, at each site. This method is faster and produces more accurate results; see also [ST04] for a similar method. Fischer and Gotsman [FG06] use planes tangent to a paraboloid to replace the cones, and thus avoid the errors caused by the tessellation of the cones.

All these algorithms run in linear time to the size of the input. In other words, their speeds decrease with the increase in the number of sites. Furthermore, they require to know the nature of each site, 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 sites 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 "sites" in the images to construct quads or cones. This limits their uses in many image processing applications.

The standard flooding algorithm has also been used to compute the Voronoi diagram. An example of this idea is presented in Algorithm 4.9.2 (Approximation by a digital image) in [OBSC99].

4.2.2 Distance Transform

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 usually use scan schemes to achieve computational costs linear to the number of pixels (or grid points). Two widely used methods are Chamfer distance transform [Bor84] and sequential Euclidean distance mapping [Dan80]. Both of these use a mask to perform two passes which scan 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. In such a scanning scheme, the order of pixels being processed is crucial, since the later processed pixel needs the values of those pixels which are processed before. Therefore, these algorithms are sequential in nature, and thus are difficult to be parallelized.

There are many kinds of exact DT algorithms, such as storing and sorting the front of the propagation [Rag92b, Egg98], storing several nearest sites instead of just storing the closest one [Mul92], using a big mask [Cui99], etc. Similar to the approximate algorithms above, all these 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 [Yam84, SM92, HM94]. 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 [ER96] 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 the GPU.

Interestingly, a truly massively parallel algorithm on distance transform was mentioned by Danielsson [Dan80]. At that time, no practical hardware was available to execute the presented algorithm. Though JFA is the same in spirit to that proposed by Danielsson, it was discovered independent of the work of Danielsson as we were researching on parallel computation for Voronoi diagram on the GPU. Danielsson stated in his paper that the work was merely a curiosity with no practical use at that time, whereas we have the understanding and feasibility of jump flooding in GPU.

There are also some other researches using the GPU to compute the discrete distance transform. Sigg et al. [SPG03] use the rasterization function of the GPU to compute the distance transform in 3D space; see also Sud et al.'s work [SGGM06]. These algorithms need significant CPU effort to compute the bounding volume of Voronoi regions, and their speeds are still dependent on the size of the input sites. Sud et al. [SOM04] improve the method in [HCK+99] by selecting only the sites having contributions to a certain slice to render. The speed is increased, but is still dependent on the size of the input sites.

An idea similar to JFA extended to 3D space is proposed by Cuntz and Kolb [CK06]. 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 in 3D space.

4.3 JFA on Voronoi Diagram

It is straightforward to apply JFA on the computation of discrete Voronoi diagrams. The basic algorithm with an example is introduced first, some variants are then introduced to improve the algorithm in different aspects.

4.3.1 Basic Algorithm

When JFA is applied on the computation of Voronoi diagrams, the input sites behave as the seeds of JFA. The information of every site is its own coordinate. The criterion to determine the best site for a grid point is the distance between the grid point and the site. The nearest site is then chosen as the best site. In the fragment program in Listing 3.2, *SomeType* is replaced by *float*2, and the variable Information records the coordinate of the nearest site. With these replacements, the fragment program can be used to compute Voronoi diagrams. The first step length of JFA is set to the half of the resolution of the screen, so that we can guarantee the whole screen be filled after the last pass with the step length of 1. Figure 4.4 shows a process of JFA on the computation of the Voronoi diagram of ten sites in a 64×64 screen.

4.3.2 Variants of JFA

As discussed in Chapter 3, the result of JFA may contains some errors. Some variants of JFA in this section can reduce the number of errors. Although the speed of JFA is already very fast and approximately independent to the number of sites, it can be further improved by a variant with the cost of some additional errors.

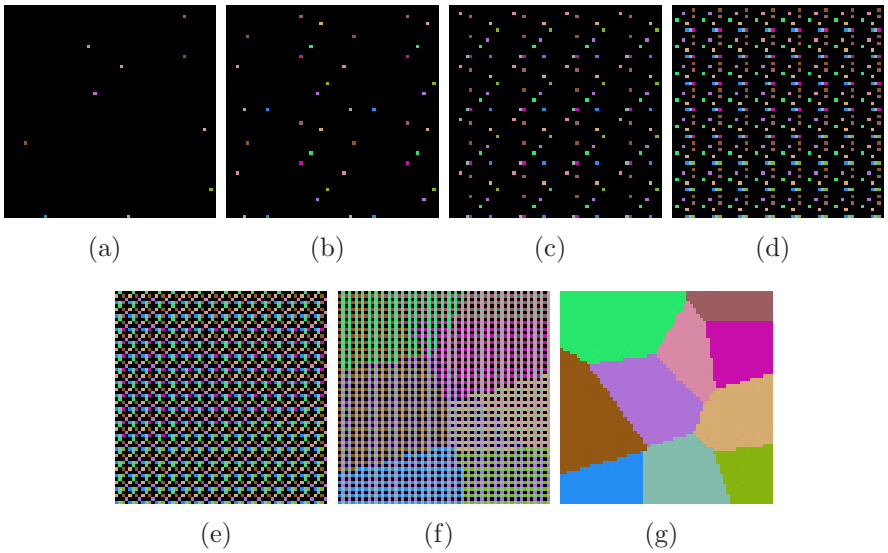


Figure 4.4: The process of JFA on the computation of the Voronoi diagram of ten sites in a 64×64 screen. The first picture shows the ten sites and the other six pictures show the results after the passes with step lengths of $32, 16, 8, 4, 2$ and 1 respectively.

Additional Pass(es) Behind

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 single errors, which means all their neighbors are correct. Such errors are Voronoi vertices themselves (see Property 4.1 later). As such, a simple yet effective way to remove many errors is to execute JFA with one additional pass with the step length of 1, i.e. a total of $\log n + 1$ passes (for a grid with the resolution of $n \times n$) where both of the last two passes are with step lengths 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 passes, with step lengths of 2 and 1 respectively. Likewise, one can do as many additional passes as needed to achieve good accuracy (while

converging to no better than the standard flooding algorithm). We have also used JFA plus $\log n$ additional passes, i.e. $JFA+\log n$ or simply JFA^2. Our experiments as reported in Section 4.5 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 4.2(b). In this case, if the disconnected blue grid point is incorrect (not colored as blue) at the end of JFA, it remains incorrect at the end of JFA^2. This is because the blue Voronoi region does not cover any grid points along the horizontal line, the vertical line and the diagonal lines that this erroneous grid point refers to during the additional $\log n$ passes, i.e. the erroneous grid point thus never receives the information of its best site (the blue site) from any grid points. Note that this is also a case where [Bor84] cannot correctly handle with a 3×3 mask.

Additional Pass Before

Another variant very similar to those above is to have an additional pass with the step length of 1 before the process of JFA. We call this variant $1+JFA$. Although 1+JFA seems similar to JFA+1, it generates less error than JFA+1. The experiment results are given later.

The rational behind this variant is as follows. During the process of JFA, an error occurs when the best site is killed by other sites en route during the flooding process. According to Property 3.4 of JFA, the necessary condition for two sites meeting at a same grid point at the 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 different from that of another site t, these two sites do not meet each other in any grid point in any pass before

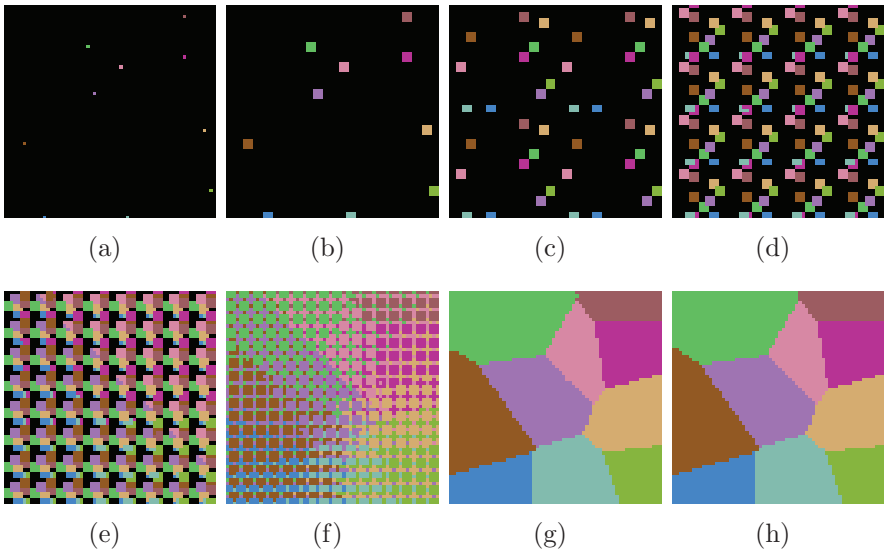


Figure 4.5: The process of 1+JFA on the computation of the Voronoi diagram of ten point sites in a 64×64 screen (same configuration as in Figure 4.4).

the pass with the step length of 1 (i.e. the last pass), and thus one cannot kill the other.

By adding a pass with the step length of 1 before JFA, 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 greatly reduces the chances of all the copies of s being killed by other sites during the flooding process. Figure 4.5 shows the process of 1+JFA on the computation of the Voronoi diagram of the same ten point sites as in Figure 4.4.

Note that this variant cannot totally eliminate all the errors, because the other sites are also flooded by one grid point to their neighbors before the standard JFA and those neighbors together may kill, for example, all the copies of s at different passes of the flooding process. Despite this, in our experiments, we observe very few errors generated by this variant. This is probably because although the path from s to the erroneous grid point may enter the Voronoi regions of other sites,

and thus may be killed by other sites there, the path usually tends to enter the other Voronoi regions by a very short distance, in many cases by only one grid point. In other words, in many cases, even when a path from s enters the other Voronoi regions, not all the nine grid points around s enter the other Voronoi regions together. Some of them still lie within the Voronoi region of s, and thus cannot be killed.

Additional Information Recorded

As noted from Figure 3.3, the problem that the site r does not reach the grid point p is because the grid points p' and p'' do not record r as their nearest sites. In fact, r is the second nearest site of p' and p''. Thus, if we keep both the nearest and the second nearest sites 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 nearest sites. We call this variant *JFA*2*Seed*. Similarly, we can have *JFA*3*Seed*, *JFA*4*Seed*, etc. However, our experiments indicate that this type of variants is inferior to those with additional passes.

Halve Resolution

In doubling the resolution of a grid from $n \times n$ to $2n \times 2n$, the standard JFA needs an additional pass with the step length of n to reach grid points further apart. Besides, each pass now deals with four times the number of grid points, 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 this variant as explained next.

Before starting JFA on a grid with the resolution of $2n \times 2n$, we sub-sample a square of four pixels into a single pixel to result in a grid with the resolution of $n \times n$. In this sub-sampling, if there is one or more sites in the square of the four grid points, we select one of them as the representative site. Next, JFA is applied on the grid with the resolution of $n \times n$ containing the representative sites. After this is completed, we expand the grid back to its original resolution of $2n \times 2n$. In doing this, a grid point (x, y) in the low resolution becomes four grid points in the high resolution, and each of these four grid points derives its nearest site from those sites represented by the representative site at (x, y). With this, the edges between two Voronoi regions generally form a staircase (zigzag) shape. So, we need another pass of flooding with the step length of 1 to smooth the Voronoi edges. Figure 4.6 shows the process of this variant for the same configuration of input sites as in Figure 4.4.

One important note is in order when performing JFA in the low resolution. To compute the distance from a site to a grid point (and to determine 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 4.7.

However, there are cases that such a situation is not avoidable. For example, in Figure 4.7, suppose there is also a site at the position of a', and we select a' as the representative site. After sub-sampling, we have a', b and c only. Many grid points (in the shaded areas in Figure 4.7) belonging to the Voronoi region of a are in the Voronoi region of either b or c after the computation, and these are thus grid points with errors. If the sites are uniformly distributed, this situation occurs

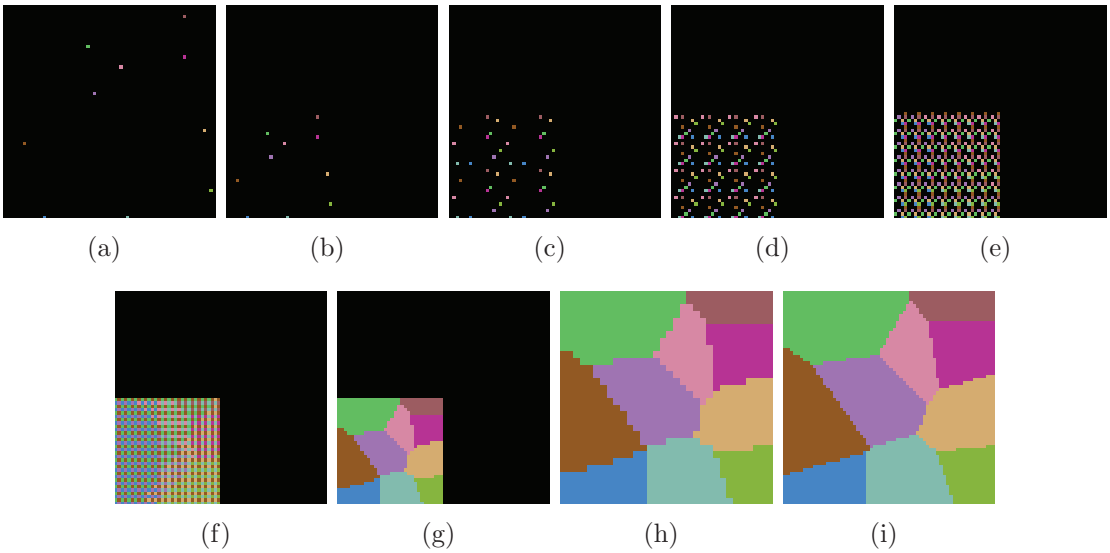


Figure 4.6: The process of the variant halving the resolution on the computation of the Voronoi diagram of ten sites in a 64×64 screen (same configuration as in Figure 4.4). The input sites are as shown in (a). (b) shows the sites after halving the resolution. (c)-(g) show the process of the standard JFA working on the halved resolution, and followed by (h) and (i) on restoring to the original resolution and smoothing Voronoi edges.

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 regions are small in size and some additional passes in the high resolution 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.

4.4 Analysis of Errors

In this section, more properties of JFA when applying on the computation of Voronoi diagrams are given. They focus on the errors of JFA. We first discuss the

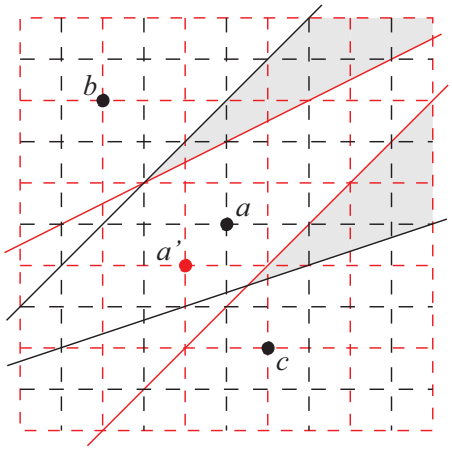
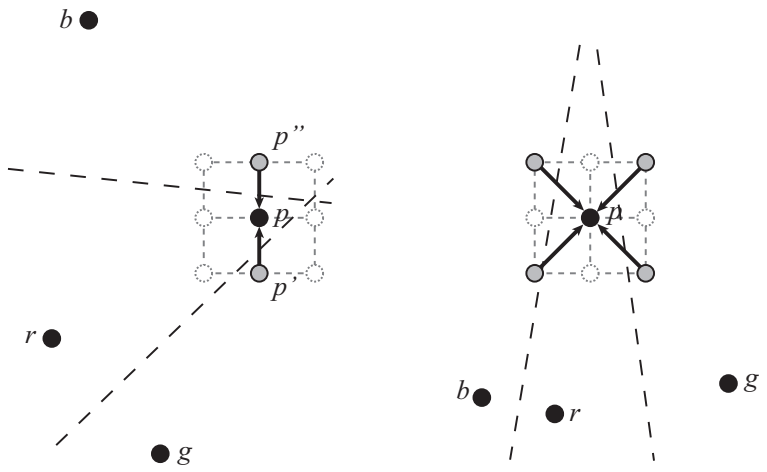


Figure 4.7: Small change of the position of a site can lead to a big change of its Voronoi region. During the sub-sampling, a moves to a' while b and c remain at their original positions. The black lines show the Voronoi region of a before sub-sampling, and the red lines show the Voronoi region after. With our approach of calculating distance using the coordinates in the higher resolution, we still can obtain the correct Voronoi region of a.

possible locations of the error, and then give an estimation of the number of errors.

From our experiments on many runs of JFA with different number of sites, we observe that all errors occur mainly along the boundaries of Voronoi regions. More specifically, for Voronoi regions that lie along the boundary of the grid, grid points with errors can cluster around Voronoi edges; for the other Voronoi regions, all the errors occur at the Voronoi vertices or around them. This is fortunate as all the computed Voronoi diagrams do not have unpleasant looks of "holes" within any Voronoi regions (as shown in Figure 3.4(b)). Additionally, this indicates the importance to analyze errors due to Voronoi vertices. To this end, we have worked out Property 4.1 to Property 4.3 of JFA as follows.

Property 4.1 *If an error occurs at a grid point p, but not at any of its neighboring grid points, p is a Voronoi vertex.*



(a) (b)

Figure 4.8: The grid point p has r as its nearest site but (incorrectly) records instead either g or b as its nearest site.

Proof. Suppose r is the nearest site to p which is not recorded by p at the end of JFA. From Property 3.3 of JFA, we consider three scenarios. First, all paths from r to p (when there is no other sites involved in JFA) are of type-v as shown in Figure 4.10(a), reaching the 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 nearest site 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 nearest sites 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 nearest sites. 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 nearest site by the given condition, p is thus a Voronoi vertex by definition.

Similarly, when all paths from r to p are of type-h, we adapt the above argument to show that p is still a Voronoi vertex. Lastly, when all paths from r to p are of type-d as in Figure 4.10(b), we also adapt the above argument with the minor adjustment that r is not the nearest site to the four grid points to the left and to the right (as in the orientation given by Figure 4.10(b)) of p. □

The condition of Property 4.1 is that the error is a single error, i.e. all its neighboring grid points are correct. If this condition is not satisfied, all the grid points between p and p' (or p'') may be errors, and thus p may not be a Voronoi vertex. Figure 4.9 demonstrates an example of this case. Site r is killed by site g and b at the grid points p' and p'' respectively. As a result, there are three connected grid points that are errors, and the grid point p is not a Voronoi vertex. Note that such a case is very rare – one out of tens of thousands runs according to our experiments. Furthermore, our experiments (in Section 4.5) report over 90 percent of errors are single errors, which are at Voronoi vertices according to Property 4.1. This percentage is increasing with the increase in the number of sites.

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

Property 4.2 *Let grid point o be a Voronoi vertex of sites r, g, and b with r as its nearest site. The probability of error at o when all paths from r to o are of*

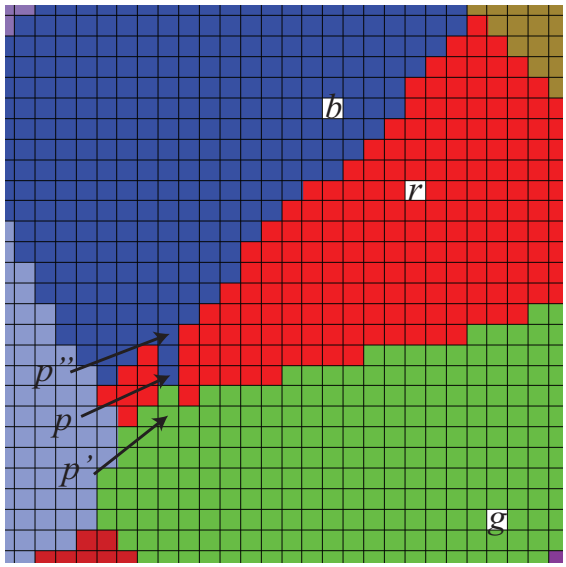


Figure 4.9: There are three connected erroneous grid points around p. Since the condition in Property 4.1 is not satisfied, p is not a Voronoi vertex.

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.

Refer to Figure 4.10(a). By assumption, r, g and b lie on a circle with the 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 *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 3.4 of JFA, the step

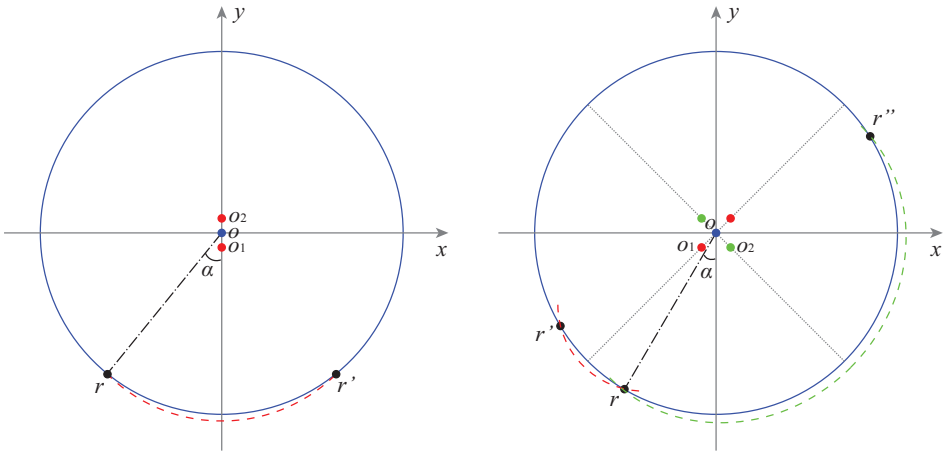


Figure 4.10: (a) Type-v. The red dashed curve indicates the circle with the center at o_1 and passing through point r and r'. (b) Type-d. The red dashed curve indicates the circle with the center at o_1 and passing through point r and r'. The green dashed curve indicates the circle with the center at o_2 and passing through point r and r''.

length from o_1 or o_2 to reach o is 2^k, where $k = \mathsf{order}(r, o)$. We consider each k ranging from $0, 1, 2, \ldots, \log n - 2$ to obtain the summation needed in this property. (Notice that $\mathsf{order}(r, o) = \log n - 1$ 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 $\mathsf{order}(r, o) = k$ is equal to $4^{-k} \times \frac{3}{4} \times \frac{1}{3} = 4^{-(k+1)}$. The first term 4^{-k} is the probability of $\mathsf{order}(r, o) \geq k$. The second term $\frac{3}{4}$ 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 $\mathsf{order}(r, o) = k$.) The third term $\frac{1}{3}$ 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 passes with step lengths $\geq 2^{k+1}$. That is, $\mathsf{order}(r, g) \geq k + 1$ and $\mathsf{order}(r, b) \geq k + 1$, 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 nearer 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. $\frac{2}{2\pi}$) represents the interchanging of g's and b's position over the full circle of 2π, the second term for one site 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 $\frac{\pi}{2}$ then multiplying by 4) gives the required F. □

Property 4.3 *Let grid point o be a Voronoi vertex of sites r, g, and b with r as its nearest site. 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 4.2 appears here. Thus, we only need to indicate the corresponding $f(\alpha)$ and F here for the argument to hold. Refer to Figure 4.10(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

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

Notice we do integration around a circle in the proofs 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 $m > 3$ sites, there are $(2m-h-2)$ Voronoi vertices where h is the number of Voronoi regions 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 regions, then from Property 4.2 and Property 4.3, we have the number of errors at Voronoi vertices estimated at:

$$\begin{aligned} E &= (2m - h - 2)(E_1 + E_2) \\ &\approx 2m(E_1 + E_2) \quad \text{when } m \gg h \end{aligned}$$

$$= \quad 2m\left(\frac{3}{4} - \frac{5}{\pi R} + \frac{6}{\pi^2 R^2}\right)\sum_{k=0}^{\log n-2} 4^{-3(k+1)}$$

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 sites can be killed by the boundary rather than by other sites) 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 passes with step lengths $\geq 2^1$ and then propagate the errors to them in subsequent passes. However, we note that such errors occur very rarely as indicated by the probabilities in Property 4.2 and Property 4.3 with $k \geq 1$. We have also verified experimentally (Section 4.5) 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 percent accuracy.

4.5 Experiment Results

We have implemented JFA and its variants using Visual C++.NET 2005 and Cg 1.5. The hardware platform is Intel Pentium IV 3.0GHz, 1GB DDR2 RAM and NVIDIA GeForce 6800 GT PCI-X with 256MB DDR3 VRAM. For each run of our experiment, we randomly generate input sites for a grid with the resolution of 512×512. The number of sites 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 for the purpose of counting the number of grid points with errors in JFA, we adapt the algorithm of [Den03].

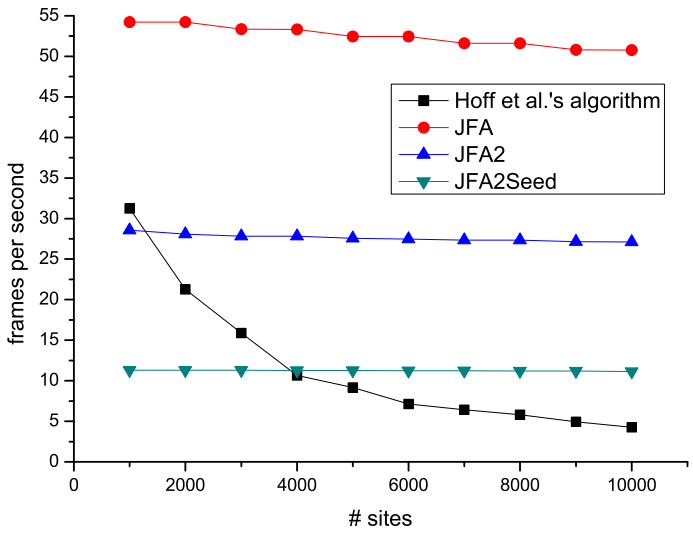


Figure 4.11: The comparisons of speeds of JFA and its variants and that of Hoff et al.'s algorithm.

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

4.5.1 Speed of JFA

The speed of JFA is compared with the popular algorithm of Hoff et al. [HCK+99]. We note that their algorithm, though linear in complexity to the number of sites, decreases rapidly in frame rate as more and more sites are used. On the contrary, the frame rates of JFA and its variants maintain quite consistent for different number of sites. The results are shown in Figure 4.11. The black curve is for Hoff et al.'s algorithm, the red curve is for JFA, the blue curve is for JFA^2, and the green curve is for JFA2Seed. We can see that even JFA^2 is still much faster than Hoff et al.'s algorithm for most number of sites.

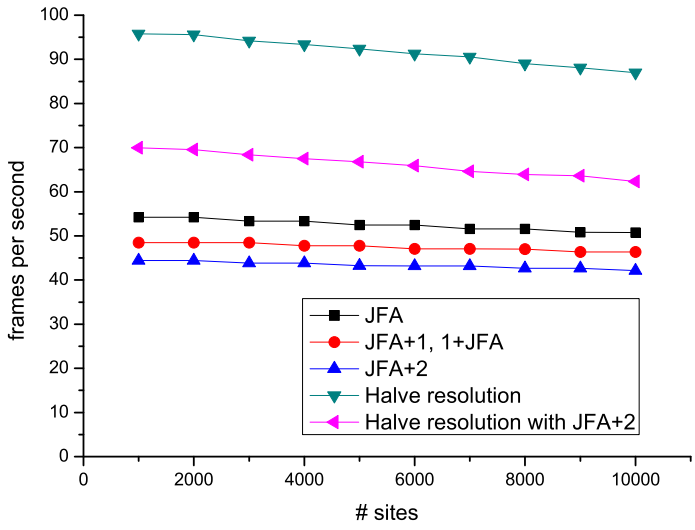


Figure 4.12: The speeds of JFA and its variants.

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 sites into a texture, as in the case of all input sites are already in a texture).

Figure 4.12 compares the speeds of JFA and its variants introduced in Section 4.3.2. As we can see, all of them are approximately independent to the number of sites, and thus maintain quite constant frame rates. The speed of 1+JFA is same to that of JFA+1, because both of them use the same number of passes. The variant that halve the resolution has the highest speed of near 100 fps. As we have mentioned above, this variant can be combined with the other variants (e.g. JFA+2) to reduce the number of errors. In our experiment on NVIDIA GeForce 6800 GPU, we obtain less than 3 errors on average in a grid with the resolution of 512×512 while still enjoy frame rate (of around 70 fps) better than that of the standard JFA (of around 55 fps).

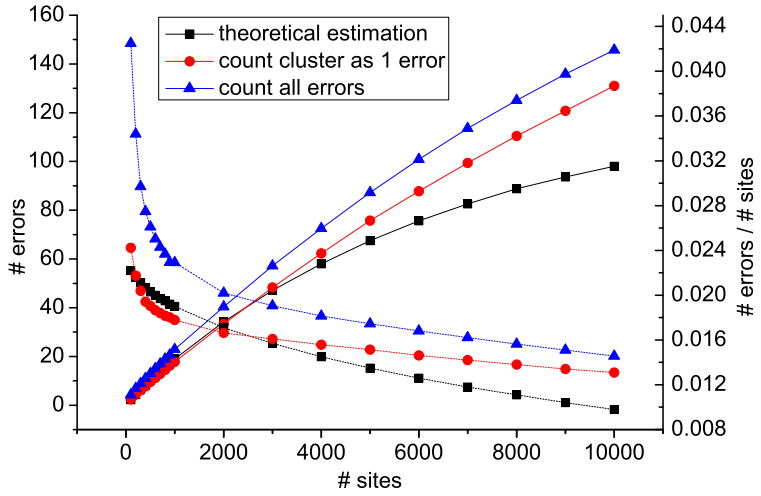


Figure 4.13: The actual and estimated errors of JFA.

4.5.2 Errors of JFA

The solid curves in Figure 4.13 (with y-axis on the left) refer to the average number of errors of JFA for different number of sites. The blue curve counts each error at grid point to get the total number of errors, whereas the red curve counts a cluster of connected grid points with errors as one single error. 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 4.4.

In another view, the dashed curves in Figure 4.13 (with y-axis on the right) show that the ratio of the average number of total errors to the total number of sites decreases with increasing number of sites. This phenomenon is also captured in E of Section 4.4 – when the number of sites increases, the value of R decreases as each Voronoi region decreases in size, and E/m thus decreases too. This underscores

the robustness of JFA in dealing with large number of sites.

Our estimate of errors with E is also plotted as the black curve in Figure 4.13. To do that, we need to estimate R for each case of different number of sites. 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 sites below 3000. The not-so-good estimate for the number of sites above 3000 is because E is rather sensitive to changes in R of small value (of less than 5 grid points) as in this case with a large number of sites.

As observed, 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 4.14. 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 4.2(b) where no number of additional passes to JFA can correct the errors.

It is very interesting to notice that although 1+JFA is very similar to JFA+1, the rate of errors of 1+JFA is smaller 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, as shown in Figure 4.12. So 1+JFA is a better algorithm than JFA+1 because of much smaller rate of errors. In Figure 4.14, we can also see that the rate of errors

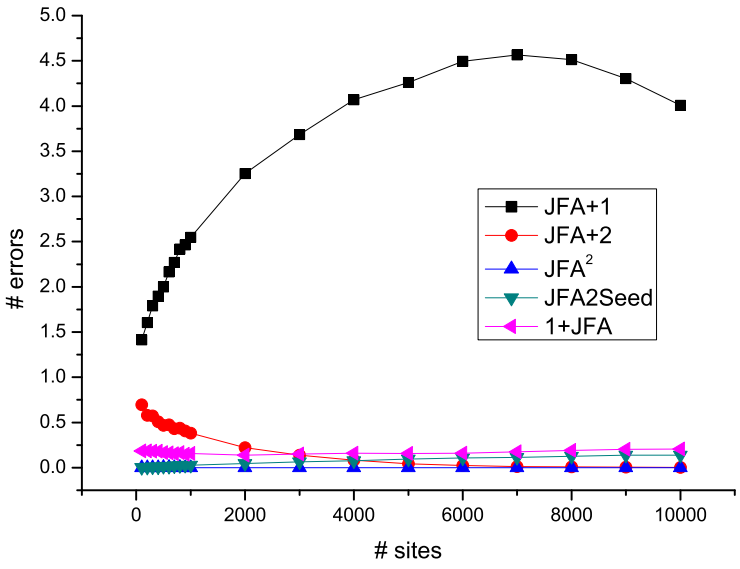


Figure 4.14: Errors of variants of JFA.

of 1+JFA is also smaller than that of the slower variant JFA+2 for small numbers of sites. As the speeds of 1+JFA and JFA+2 are similar, these two variants are the best compromised among all.

4.5.3 Generalized Voronoi Diagram

As we have mentioned in Section 4.1, the sites of the Voronoi diagram can be generalized to line segments or even curves and areas. Such generalized cases can also be computed by JFA; see, for example, Figure 4.15. Though we do not have an analysis of errors for generalized Voronoi diagrams, we expect good results with very few errors. This is because our algorithms treat the generalized sites as collections of point sites and thus expect to inherit the good performance obtained for point sites.

(a) (b)

Figure 4.15: Applying JFA on (a) the input image, we have (b) the generalized Voronoi diagram of the three area seeds.

4.6 Voronoi Diagram in High Dimension

JFA and its variants are applicable to the computation of higher dimensional Voronoi diagrams. However, most of current GPUs, except for the newest NVIDIA GeForce 8800, do not support writing data into 3D textures as required by JFA. Though one can pack a 3D texture into a 2D texture [Har03, CK06], such packing currently works only for 3D textures with small resolutions, and is thus not very useful when we need jump flooding in a large 3D space.

In this section, we first introduce the result using the CPU to simulate JFA sequentially. Then, we introduce a variant of JFA that can compute 3D Voronoi diagrams in a slice-by-slice manner.

4.6.1 CPU Simulation

To assess the effectiveness of JFA for higher dimensions, we perform CPU simulation of JFA and its variants. In our experiment, we randomly generate input

sites for a 3D grid with the resolution of $512 \times 512 \times 512$. The number of sites 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 sites 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 the sequential simulation on the CPU is too slow for large numbers of seeds – even 100 runs take several days to complete.

Figure 4.16 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 sites for JFA is also shown as the black dashed curve. Quantitatively, JFA and its variants perform very well in generating the result which is close to the exact Voronoi diagram. The percentage of erroneous grid points 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 site to each grid point in 3D, and it becomes much harder to kill all paths to a grid point generated by its nearest site. Thus, the probability of a grid point not receiving its nearest site becomes very low.

4.6.2 Slice by Slice

In this section, 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 3.1 of JFA, regardless of where a site is, as long as it is not killed by other sites en route, its information can reach all the grid points in a grid after the pass with the step length of 1. This property suggests that it is not

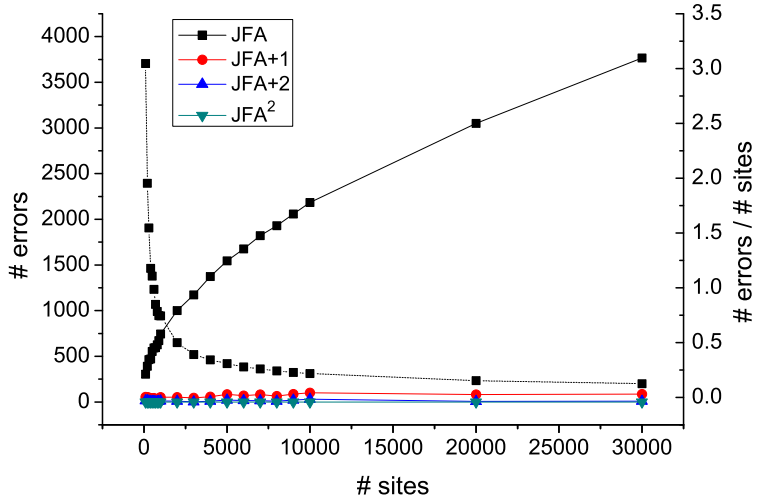


Figure 4.16: Errors of CPU simulation of JFA, JFA+1, JFA+2 and JFA^2 in a 3D grid with the resolution of $512 \times 512 \times 512$.

necessary to put the sites at their original positions to perform JFA. In fact, we can choose to put the sites at any positions that are convenient to an application. (This may lead to different rate of errors in the flooding process.) We note that in some sense, the variant that halve the resolution discussed in Section 4.3.2 is of the same spirit; it can be seen as first shifting all the sites to the positions with both their x- and y-coordinates are even numbers in the grid of the original resolution, and then performing JFA in the original grid for these grid points only.

This understanding can help us to compute the 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 are projected to a same pixel, only the coordinate of the nearest site is recorded, because the Voronoi regions of the other

sites do not intersect this slice and these sites thus need not appear in this slice for flooding. Next, we perform JFA (or its variants) using these projected sites in the slice while utilizing the original 3D coordinates recorded to perform distance computation in order to determine the nearest site for each pixel.

The rate of errors of this variant (black curve), together with that of the CPU simulation (blue curve), is shown in Figure 4.17. Compared with the CPU simulation, the result of this new variant generates more errors. This is expected for the same reason explained in last section. In this variant, JFA is performed in a 2D space, and the number of possible paths for a grid point 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 being killed by other sites is accordingly higher. On the other hand, the red curve in Figure 4.17 shows the rate of errors when this variant is combined with 1+JFA. Once again, the rate of error is far less than that of the standard JFA. An interesting observation is that the red curve is almost coincident with the blue curve. In other words, we 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 on the computation of 3D generalized Voronoi diagrams. We have tested different types of sites including points, line segments, splines, etc. One example is using sphere sites. The Voronoi diagram of spheres have many applications in various areas, such as biochemistry [KKC05]. Our variant is also able to handle this type of sites. Figure 4.18 shows a screenshot of our program using 50 spheres as the sites in a cube with the resolution of $512 \times 512 \times 512$.

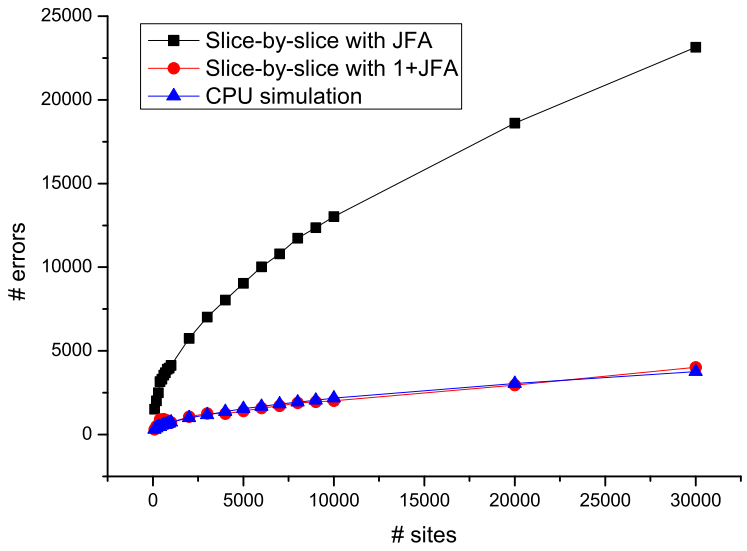


Figure 4.17: Errors of the variant computing the 3D Voronoi diagram slice-by-slice (summing all errors of the 512 slices) in a 3D grid with the resolution of $512 \times 512 \times 512$.

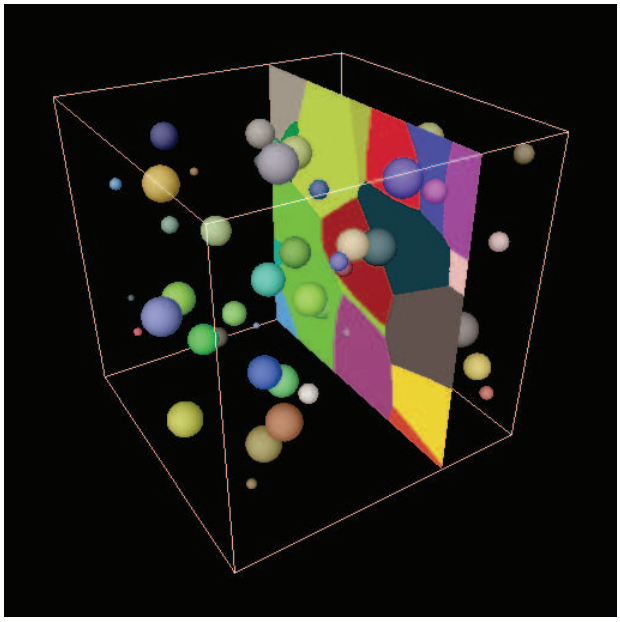


Figure 4.18: Screenshot of the program computing the 3D Voronoi diagram using 50 spheres as sites.

4.7 Summary

In this chapter, JFA is applied on the computation of Voronoi diagrams and distance transforms. We also introduce some variants of JFA. Some of these variants can be combined with each other to build a better variant of JFA with less errors and higher speed. Some new properties of JFA on errors are also proven in this chapter. At the end of this chapter, we show how to use JFA to compute Voronoi diagrams in 3D space, either by using CPU simulation or by using a variant in a slice-by-slice manner.

The idea of putting sites at positions other than their original ones as discussed in Section 4.6 is interesting. This may help to find some special artificial patterns to further reduce the rate of errors or to obtain some special effects.

Chapter 5

Real-Time Soft Shadow

Shadows play an important role in interactive applications. Shadows can greatly improve the reality of the scene and can give the information of the positions and the relationship between objects. In some cases, without the help of shadows, we may misjudge the position of the objects in the image [HLHS03, Wan92].

If the light source is a point or parallel light (in this case, the light source can be seen as a point light source at the infinite position), the boundary of the shadow is sharp. Every point in the scene has an intensity of either 0 or 1 depending on whether the light source is visible to this point. The shadows under this case are called *hard shadows*. However, if the light source has some extent, it may be partially visible to some points. These points have intensities between 0 and 1. The shadows under this case become *soft shadows*. The union of all the points having intensities between 0 and 1 is called *penumbra*, and the union of all the points having intensities of 0 is called *umbra*. Real-time shadow (especially soft shadow) generation usually costs a significant amount of CPU and GPU cycles.

Real-time shadow algorithms can be categorized into object-based algorithms

and image-based algorithms [HLHS03]. Object-based algorithms must deal with all the geometries in the scene. So they do not scale well for complex scenes. In contrast, the speeds of image-based algorithms are less dependent on the complexity of the scene.

This chapter proposes two simple image-based soft shadow algorithms JFA-L (jump flooding in light-space) and JFA-E (jump flooding in eye-space) that are of an order of magnitude faster than existing comparable algorithms. They achieve good frame rates with the current GPU for very complex scenes of over hundreds of thousands of triangles. Before giving the details of these new algorithms, several related researches are first reviewed in next section.

5.1 Related Work

Shadow has attracted the interests of computer graphics researchers for a long time. Since the beginning of computer graphics, researchers have proposed many methods to compute shadows. If we see the scene from the light source, all the visible faces are lit and others are all in the shadow. So any hidden surface removing algorithm can be used to compute shadow.

There are numerous previous work on shadows. Woo et al. [WPF90] gives a good survey of the shadow algorithm. A state of the art survey is given by Hasenfratz et al. [HLHS03]. In this section, we only review some previous work closely related to our new algorithms. Because almost all the soft shadow algorithms are based on the hard shadow algorithms, it is good to briefly review the two dominating hard shadow algorithms first.

5.1.1 Hard Shadow Algorithms

Although there are many hard shadow algorithms as proposed by the researchers in past several decades, *shadow volume* and *shadow mapping* are the two oldest ones, and are most commonly used. Almost all the other hard shadow algorithms are based on them. In hard shadow algorithms, the light source is always assumed as a single point.

The shadow volume algorithm is proposed by Crow in SIGGRAPH'77 [Cro77]. The silhouette edges of the objects seen from the light source are first computed. These silhouette edges are then used to build the new geometries called *shadow volumes*, one for every object. A shadow volume is a closed volume composed by several side faces (one for every silhouette edge of the object) together with one near face and one far face. The side faces of the shadow volumes are defined by the planes passing through these silhouette edges and the light source. The near face is defined by the front faces of the object, and the far face is a face at infinite distance (practically a distance far enough specified by the user). The shadow volumes are clipped according to the view frustum. The remaining faces of the shadow volumes are added to the scene. When the scene is rendered from the eye position, these faces are also rendered. They do not affect the color of the final image, but only affect the stencil buffer. Initially, the values of all the pixels in the stencil buffer are set to 0. When a front face (a face facing to the eye position) of a shadow volume is rendered, the values of the pixels covered by this face in the stencil buffer are increased by one. When a back face (a face facing away from the eye position) is rendered, the corresponding values are decreased by one. The pixels with positive final values in the stencil buffer are within the shadow volumes,

and thus in the hard shadow.

The shadow volume algorithm can generate high quality hard shadows. But since it is an object-based algorithm, it must process all the objects to find all the silhouette edges before generating the final result. So its speed decreases rapidly with the increase of the complexity of the scene. Furthermore, the newly generated shadow volume faces usually cover large area on the screen, and thus consume the fill-rate greatly.

On the contrary, the shadow mapping algorithm is an image-based algorithm. So its speed is much less dependent on the complexity of the scene. The shadow mapping algorithm is proposed by William [Wil78] one year later than the shadow volume algorithm. It is composed of two phases. In the first phase, the scene is rendered from the light source position. The result is stored in a texture called *shadow map*. The second phase is the actual rendering phase where the scene is rendered again, but from the eye position this time. For every pixel in the final image, its corresponding point in the object space is transformed into the light space. The transformed depth value is compared with the corresponding depth value stored in the shadow map. If the value in the shadow map is smaller, the pixel is in the shadow; otherwise it is fully lit.

The shadow mapping algorithm suffers from its own problems. As the scene information from the light is recorded in the shadow map with a finite resolution, there is usually some aliasing in the final result, due to the insufficient sampling density and/or the slant view angle. Many solutions are proposed to alleviate the aliasing. Some famous solutions include Perspective Shadow Map [SD02], Trapezoidal Shadow Map [MT04], Light Space Perspective Shadow Map [WSP04], and Logarithmic Perspective Shadow Maps [Llo07].

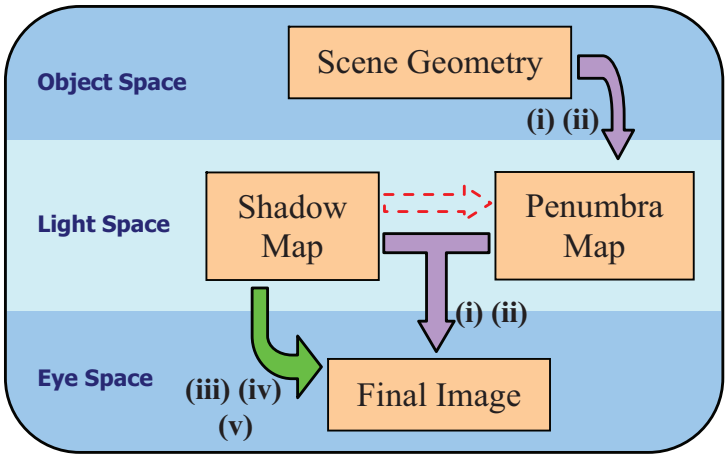


Figure 5.1: The computational mechanisms of the recent soft shadow algorithms based on shadow mapping. The dashed red arrow is our JFA-L's way of generating penumbra map directly from shadow map to assist the computation of soft shadows. Our JFA-E follows the approach indicated by the green arrow (same as iii, iv and v).

Due to the fact that the shadow volume algorithm do not scale well for the complex scene, and because our two new algorithms using JFA are both based on the shadow mapping algorithm, in the next section, we only review some previous soft shadow algorithms that are based on the shadow mapping algorithm.

5.1.2 Soft Shadow Algorithms

In this subsection, we briefly review several soft shadow algorithms which are closely related to our work. They are summarized in Figure 5.1. All of them are based on the shadow mapping algorithm [Wil78], and use shadow maps obtained from a single sample on the light (normally the center of the light). For other soft shadow algorithms, see the survey by Akenine-Möller and Haines [AMH02] and Hasenfratz et al. [HLHS03].

Wyman and Hansen's penumbra map [WH03] ((i) in Figure 5.1) and Chan

and Durand's smoothy [CD03] ((ii) in Figure 5.1) both build some additional geometries in object space and then render them to generate a penumbra map. This penumbra map, together with the shadow map, can be used to generate the final soft shadows. Both algorithms are object-based, require the objects in the scene to be polygonal, and do not scale well for complex scenes.

Arvo et al. [AHT04] use the shadow mapping algorithm to generate an image with hard shadows, and then use the flood filling method to spread out the information of occluders from the boundaries of hard shadows to obtain soft shadows ((iii) in Figure 5.1). Besides some technical problems of the method (discussed in Section 5.4), their approach of flooding is slow for real-time applications.

Uralsky [Ura05] adapts the percentage closer filtering (PCF) method [RSC87] to generate soft shadows directly from a shadow map ((iv) in Figure 5.1). His algorithm uses Monte Carlo sampling and hence generates noisy effects in the results. Uralsky uses back faces (with respect to the light) to build the shadow map. This constrains the objects to be closed manifolds. It can also generate erroneous self shadow for thin objects and wrong brighter regions for concave objects (see also [VdB05] for the discussion of these problems). Furthermore, the optimization in Uralsky's algorithm can result in errors as dark spots in small fully lit areas.

Brabec and Seidel [BS02] also generate soft shadow directly from the shadow map ((v) in Figure 5.1). But the searching for the nearest silhouette pixels in their algorithm is too slow for real-time applications, especially for wide penumbrae. And their use of object identities prohibits self shadows.

Recently, there are two new papers by Guennebaud et al. [GBP06] and Eisemann and Décoret [ED06] that use purely image-based methods to generate real-

time soft shadows, and achieve very good speed.

Both of our new algorithms in this chapter, JFA-L and JFA-E, are based on the shadow mapping algorithm. Both of them propagate the information of the occluders in an image space. JFA-L performs the propagation in the light space to generate a penumbra map directly from the shadow map (as shown by the red dashed arrow in Figure 5.1), while JFA-E performs the propagation in the eye space to generate soft shadows from the result with hard shadows. Next, we introduce the common idea of the both algorithms – the propagation of the information of occluders.

5.2 Propagate Occluder Information

The basic ideas of our soft shadow algorithms JFA-L (Section 5.3) and JFA-E (Section 5.4) are the same. Both of them rely heavily on propagating occluders' (seeds') information stored in pixels on silhouettes or boundaries to other pixels to calculate their intensities. And both algorithms use JFA to fulfill this task. The difference between them is the space where JFA is applied. JFA-L applies JFA in the light space, while JFA-E in the eye space.

We assume the light source is a circle for our following discussion. For a point p receiving information of an occluder point o, it calculates the intensity by shooting a ray from p to o (as in [AHT04]). If the ray does not intersect the light source, p is considered to be either in an umbra region or in a fully lit region based on whether it is in the hard shadow, and does not propagate the information of o to other pixels in subsequent passes. On the other hand, if the ray intersects the light source at a point q (see Figure 5.2), the intensity is estimated based on the

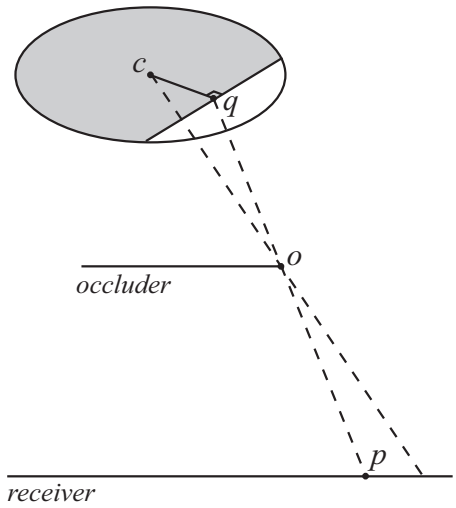


Figure 5.2: Computation of the intensity of a point p receiving the information of an occluder point o.

proportion of the light source visible to p by partitioning the light source with the line orthogonal to cq and passing through q, where c is the center of the light. In the example in Figure 5.2, the unshaded part of the light is visible to the point p. So if the total area of the light is 1, the area of this unshaded area is the intensity of p.

When we use JFA to perform such propagation, the seeds are the pixels corresponding to the occluder points (in the shadow map or in the result with hard shadows). The information of every seed is the 3D coordinate of the corresponding occluder point. The calculated intensity at p is used as the criterion in JFA to determine the best seed of p. This criterion is different for the outer penumbrae (penumbrae that lie outside of the hard shadows) and the inner penumbrae (penumbrae that lie inside of the hard shadows). For an outer (inner, respectively) penumbra point p, the seed that gives the smallest (largest, respectively) intensity is the best seed of p. In every pass of JFA, p keeps the smallest (or largest) in-

tensity so far, and goes on to propagate the information of its best seed to other pixels in subsequent passes.

Such a propagation of the information of occluders to calculate intensities, when implemented as jump flooding to gain good speed, is unclear to work as effectively as the standard flooding algorithm. Unlike the case of Voronoi diagram computation with the empty circle property, we do not know of any property here to provide any guarantee of good soft shadows with such a non-trivial communication pattern. We address in the next two sections the situations presented by JFA-L and JFA-E to generate plausible soft shadows.

5.3 Jump Flooding in Light Space

The novelty of JFA-L is to efficiently generate a penumbra map directly from the shadow map by using JFA in the light space (as shown by the red dashed arrow in Figure 5.1). It uses hard shadows (with respect to the center of the light) to approximate umbra regions. So all the penumbra regions can be seen by the center of the light, and thus can be stored into a texture called *penumbra map*. It only generates *outer penumbrae*, i.e. penumbrae out of hard shadows.

JFA-L algorithm includes four major phases. In the second phase and each pass of jump flooding in the third phase, we use the standard technique of drawing a quad of the same size as the shadow map to trigger a fragment program running on every pixel.

5.3.1 JFA-L Algorithm

The four phases of JFA-L are described in the following.

Building Shadow Map. The first phase of our algorithm is to build a shadow map from the center of the light. This phase is almost identical to that of the standard shadow mapping, except that we also store the light space coordinate of every pixel in addition to its depth value. To increase the resolution of the useful part of the shadow map, we use the trapezoidal shadow map approach [MT04].

Locating Occluders. For every pixel in the shadow map, we test its eight neighboring pixels. If the depth values of one or more neighboring pixels are larger than its own depth (greater than a certain threshold), it is marked as a *silhouette pixel*. These silhouette pixels having information of their light space coordinates behave as occluders for other pixels in penumbra regions, and they are the seeds for JFA in the next phase.

Generating Penumbra Map. To compute a penumbra map, we spread out the coordinate information of occluders from the silhouette pixels to other pixels using the jump flooding algorithm (see Figure 5.3 for our implementation of the jump flooding process with 6 passes). When a pixel receives the coordinate of an occluder, it shoots a ray from itself to the occluder to calculate its own intensity as described in Section 5.2. Among all occluders received, a pixel keeps the one computed with minimum intensity value and then passes it on to other pixels in subsequent passes. We can pre-calculate the intensities for all the possible intersection points of the ray with the light, and store the results in a texture indexed by the intersection points. This greatly increases the speed of the algorithm.

Generating Final Image. In the last phase, we render the scene from the eye position. For every pixel, its corresponding point is transformed to the light space. First, we check whether the point is in hard shadows using the standard shadow mapping algorithm. If so, the intensity of this pixel is set to 0. Otherwise,

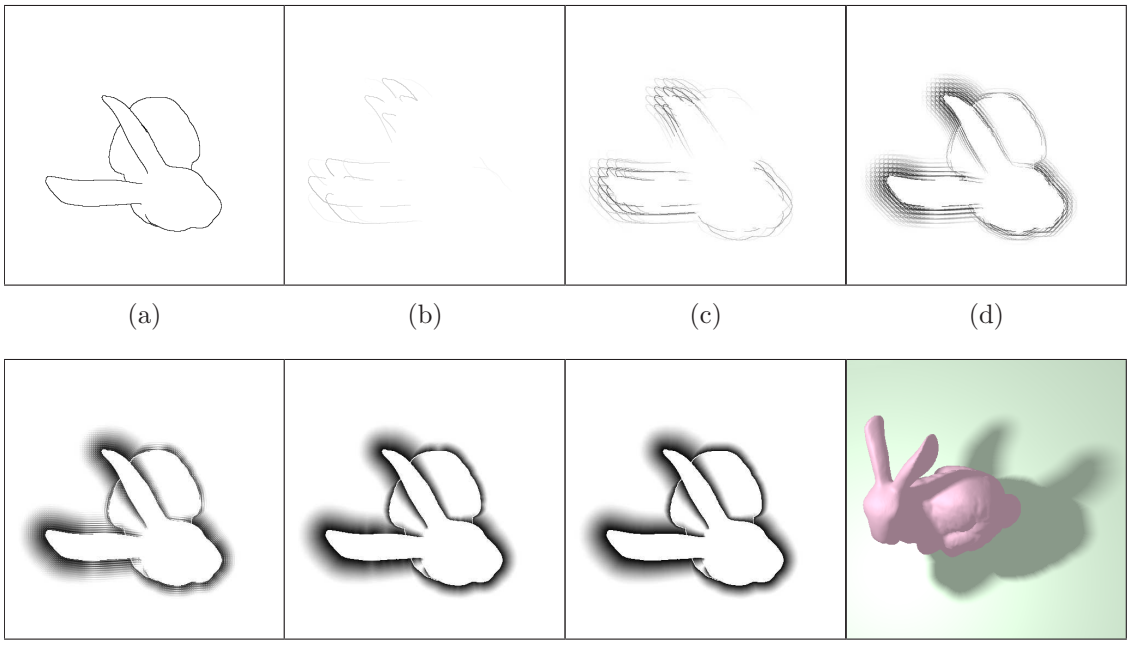


Figure 5.3: The process of JFA-L algorithm. (a) boundaries (silhouette pixels) in shadow map; (b)-(g) passes of the jump flooding process with step lengths of 32, 16, 8, 4, 2 and 1; and (h) final result generated using the penumbra map (g) and the shadow map. The 6 passes of jump flooding in (b)-(g) generate a penumbra region with maximum width of 63 pixels. This number of passes (instead of a complete 9 passes of JFA for this example of 512×512 image) is generally enough for most applications.

we check whether there is a coordinate value stored in the corresponding pixel in the penumbra map. If so, this pixel is in a penumbra region, and its intensity is then calculated by shooting a ray from this pixel to the corresponding occluder stored in the penumbra map. Note that we do not directly use the intensity stored in the penumbra map as that can result in blocky soft shadows when the shadow map does not have a good resolution. If both of the above fail, the pixel is fully lit and thus has an intensity of 1.

5.3.2 Analysis

An immediate limitation of JFA-L is that it does not generate inner penumbrae, i.e. penumbrae inside of hard shadows. We have investigated many attempts (such as the second layer shadow maps [WM94, AW04]), but found no suitable supplement to the JFA-L algorithm to generate inner penumbrae without introducing new problems. Nevertheless, from our experiment and our analysis below, JFA-L remains very useful in generating convincing soft shadows for real-time purposes.

JFA-L does not calculate exact intensities at penumbra regions but only in the form of approximations. This is not an issue for real-time applications, such as games, as long as the algorithm can achieve the following two goals of *parity* and *smoothness* in order to generate convincing soft shadows. On the parity, an algorithm calculates soft shadow intensity for a pixel if, and only if, the pixel is in a penumbra region. On the smoothness, calculated intensities of adjacent penumbra pixels must vary smoothly.

Parity. We have the following simple argument that when a pixel p is calculated to be in a penumbra region by JFA-L, then this is indeed so for p. Let o be the occluder received by p through the jump flooding process for the calculation of the intensity of p. By the fact that the ray from p to o intersects the light source, p cannot be fully lit. Also, p cannot be in an umbra region too, as our algorithm does not assign intensities to pixels in umbra regions. So, p can only be in a penumbra region, as claimed.

On the other hand, if p does not receive intensity value through the jump flooding process, then it is not certain that p is not in any penumbra region. When such p is indeed in the middle of a penumbra region, the undesirable visual effect is

the existence of holes in the calculated penumbra region. In all our experiments, we do not observe such effects. One explanation is that penumbra regions tend to be narrow in width, thus are highly unlikely to accommodate visible holes. Another possible explanation is to view the jump flooding computation of intensity here as in some way related to the jump flooding computation of the distance in Chapter 4. This view is reasonable, but not absolute, as the intensity of a pixel is in some way proportional to the distance from the silhouette pixels in the light space. By the result of Chapter 4 that JFA computes an excellent approximation to the Voronoi diagram, we can expect JFA to propagate the information of occluders to almost all, if not all, pixels (especially those nearby ones) in penumbra regions.

Smoothness. We have the following reasoning to believe that JFA-L computes smooth transition in intensity from pixel to pixel. By the way the intensity is defined, the calculated intensity with a single occluder for a continuous surface is smooth across the pixels (in the penumbra map) representing the surface.

Suppose that all occluders can reach all pixels (which is clearly not true because some occluders are killed along the way during the jump flooding process). Then, for any two neighboring penumbra pixels, regardless of the fact that they record the same or different occluders, the calculated intensities are smooth across the pixels as we always keep in each pixel the occluder that generates the minimum intensity.

By the nature of the jump flooding algorithm with many ways to reach from one pixel to another pixel throughout the passes, many occluders have good chances to communicate with many pixels before being killed. Thus, there are enough occluders to reach enough relevant pixels, i.e. a partial fulfillment of the assumption in the previous paragraph, and these are enough to generate plausible smooth

intensity across pixels. Additionally, as mentioned before, penumbra regions are narrow and near to the hard shadow boundaries, the occluders recorded for two adjacent pixels are either the same or nearby occluders, and thus generate smooth intensity across the adjacent penumbra pixels.

Flooding Experiment. To provide some confidence to the above reasoning, we use a fantasy scene (Figure 5.9) of over a hundred thousands of triangles to generate two series of 462 frames, one using the jump flooding algorithm and the other using the standard flooding algorithm to generate the corresponding penumbra maps. The percentages of pixels (with respect to the total number of 430k to 940k penumbra pixels in each image) with differences in intensities (ΔI) for each frame are plotted as four curves in Figure 5.4(a). The four curves represent the differences in intensities of below 0.05, 0.05 to below 0.1, 0.1 to below 0.2, and 0.2 and above. We note that the intensities of the majority of pixels as generated by the jump flooding algorithm and the standard flooding algorithm are of small differences, if not the same. Figure 5.4(b) and 5.4(c) show the penumbra maps (at the pergola area) of the frame with the largest difference. We do not see significant visible differences for these frames. In the experiment, we also note that the number of pixels with differences in parity between results of the jump flooding algorithm and of the standard flooding algorithm is less than 0.1%, and this thus is not of significance at all.

5.4 Jump Flooding in Eye Space

The novelty of JFA-E is to efficiently generate a final image with soft shadow from an image with hard shadows. This algorithm seems to follow immediately from

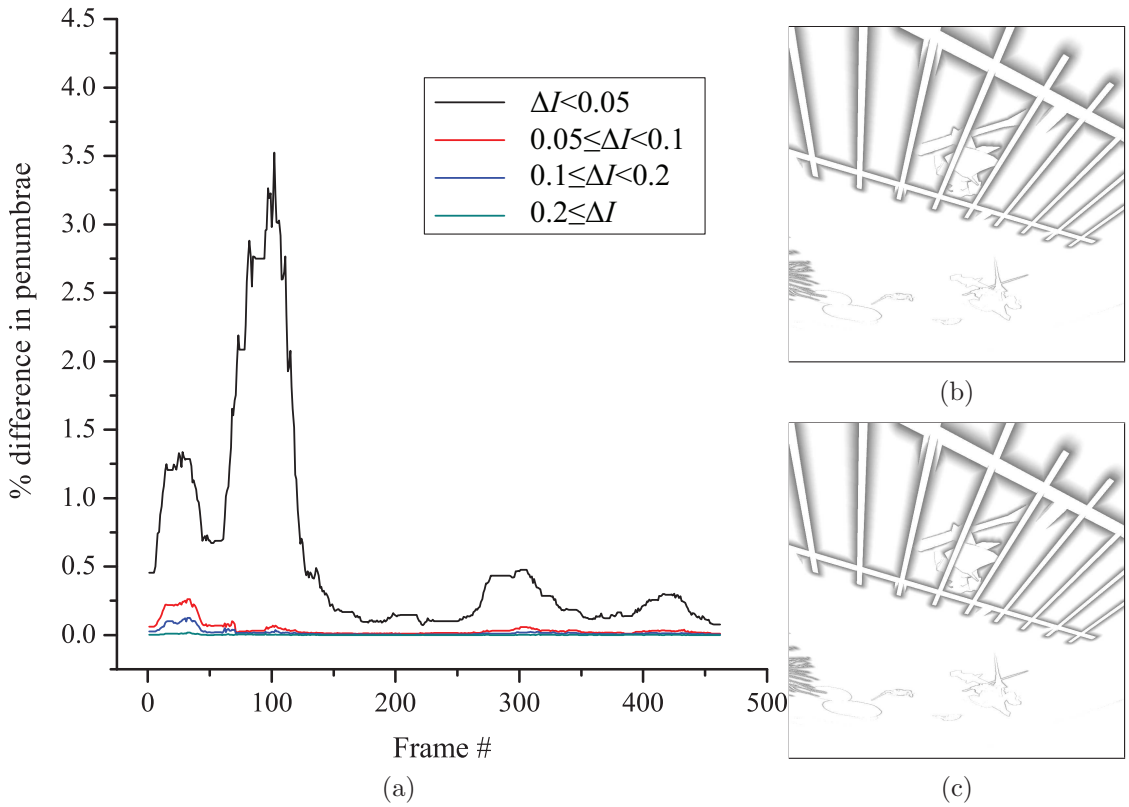
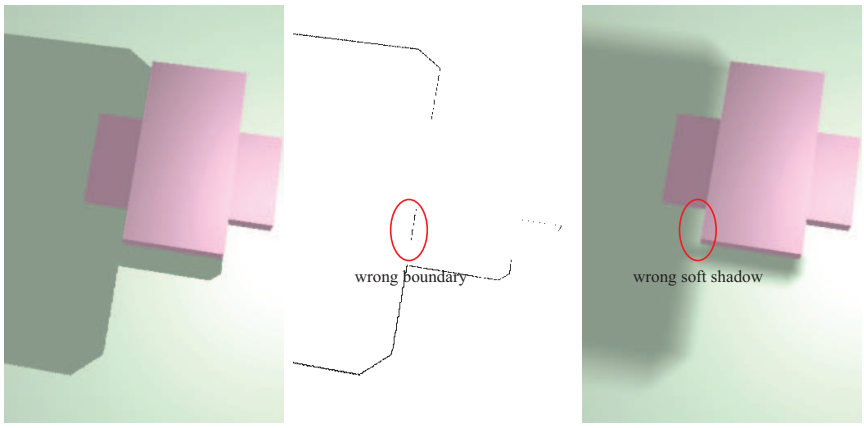


Figure 5.4: (a) Statistics of pixels with differences in intensities; (b) the penumbra map generated by the jump flooding algorithm for the frame with the largest difference; and (c) the penumbra map generated by standard flooding algorithm for the frame with the largest difference.



(a) (b) (c)

Figure 5.5: Wrong soft shadow generated by Arvo et al.'s algorithm. (a) The hard shadow due to two rectangular blocks forming a cross in space; (b) The boundaries found using the two conditions in Arvo et al.'s paper; and (c) The wrong soft shadow.

Arvo et al.'s algorithm [AHT04] by replacing the standard flooding algorithm with JFA. However, we need to deal with two problems. Firstly, Arvo et al.'s algorithm sometimes falsely recognizes pixels on hard shadow boundaries when using the conditions that such a pixel (1) must lie between lit and shadow regions, and (2) must be occluded by a silhouette pixel in the shadow map. This is because condition (2) is not foolproof: some pixels occluded by a silhouette pixel in shadow map may also be occluded by other objects in between them, and thus is not on hard shadow boundaries. These incorrectly identified pixels lead to holes in umbra regions. Figure 5.5 demonstrates this phenomenon. Secondly, we need to address an undesirable "jump-too-far" effect of the jump flooding process that also results in holes in umbra regions (in Section 5.4.2. and Figure 5.7). The next subsection describes the five phases of the JFA-E algorithm.

5.4.1 JFA-E Algorithm

There are five phases in JFA-E algorithm. They are described in detail in the following.

Building Shadow Map. This phase is the same as that of JFA-L.

Generating Hard Shadows. In this phase, we generate hard shadows using the standard shadow mapping algorithm. For every pixel, in addition to the binary information that indicates whether it is in the hard shadow, we also store the 3D coordinate of the point corresponding to it. Each coordinate is stored in RGB channels, and the flag indicating whether the pixel is in hard shadows in the alpha channel.

Locating Occluders. To extract the boundaries of hard shadows, we check every pixel that is in hard shadows for its eight neighboring pixels using a fragment program. If at least one of the neighboring pixels is lit, and if the distance (in 3D space) between the corresponding points of this lit pixel and the pixel in hard shadows is small enough (less than a threshold), we consider this pixel as a *boundary pixel*. Once a boundary pixel is ascertained, the coordinate of the corresponding occluder is also recorded for the pixel into a texture, which becomes the input of the following jump flooding process.

Generating Penumbra Regions. In this phase, we run a fragment program with a required number of passes of jump flooding to calculate intensities for pixels in penumbrae. The computation of the intensity of every pixel is same to that described in Section 5.2. As in the JFA-L algorithm, we use a look-up table indexed by the intersection point to obtain pre-calculated intensity values. For each pixel in a penumbra region, we keep the highest or lowest intensity value, respectively,

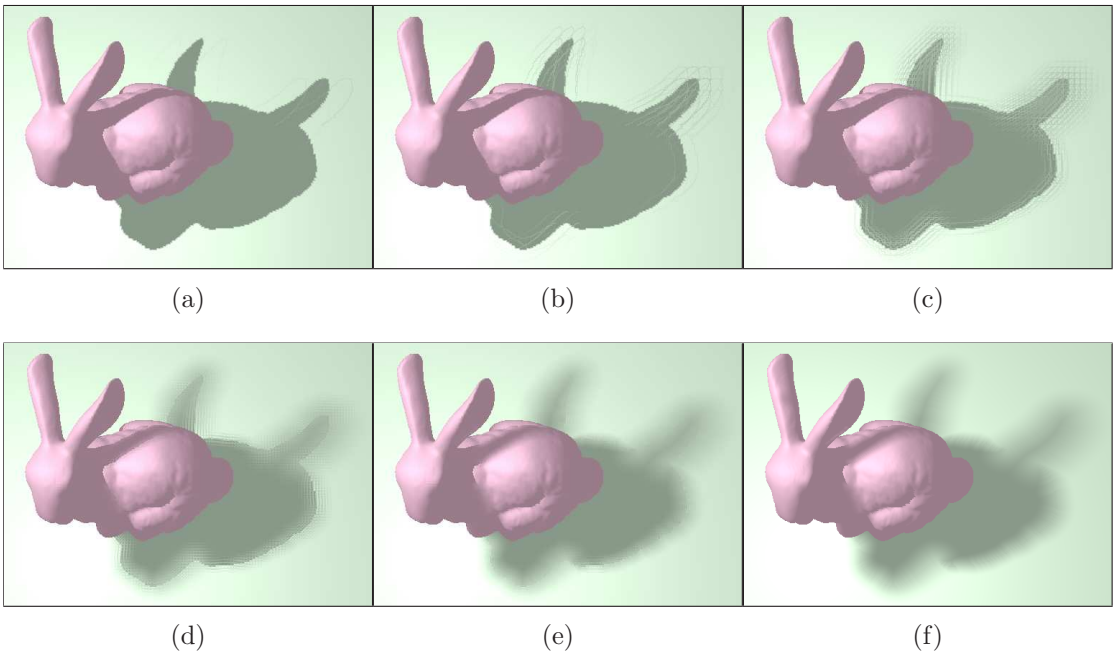


Figure 5.6: The process of JFA-E algorithm. (a)-(f) are steps of the jump flooding process with step lengths of 32, 16, 8, 4, 2 and 1, and (f) is the final result.

during the flooding process according to whether it is inside or outside, respectively, hard shadows. Figure 5.6 shows the jump flooding process with 6 passes.

Generating Final Image. For every pixel, the result of jump flooding in the previous phase can determine whether it is in a penumbra region. If so, its intensity in the final image is read from the output in the previous phase. Otherwise, we use the hard shadow result generated in the second phase to determine whether it is in hard shadows. If so, its intensity in the final image is set to 0. If both of the above fail, the pixel is fully lit and thus has a normal intensity of 1.

5.4.2 Analysis

Jump-Too-Far Problem. The advantage of JFA-E as compared to JFA-L is that it attempts to generate both the inner and outer penumbrae. However, this

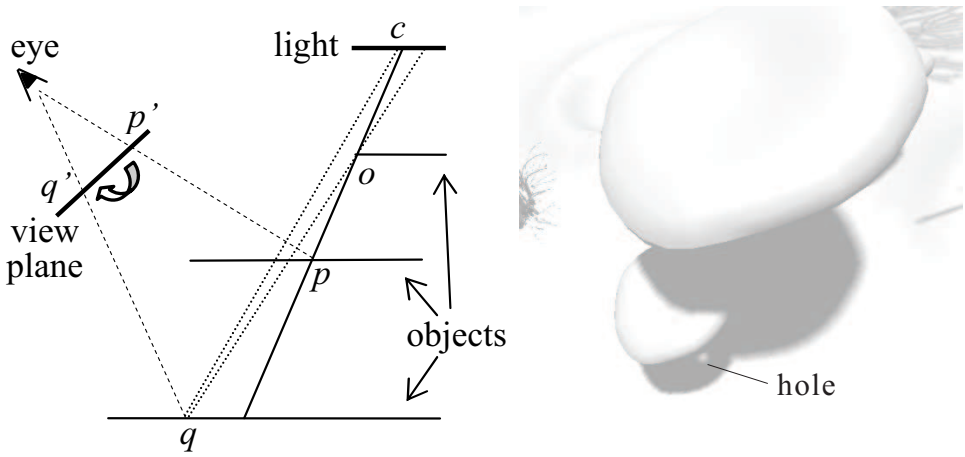


Figure 5.7: (a) 2D illustration of the jump-too-far problem; and (b) a corresponding example of an undesirable hole in the hard shadow under the mushrooms in our fantasy scene.

comes with a side effect of some parts of umbra regions to be mistaken as inner penumbrae that appeared visually as holes inside umbra regions. See Figure 5.7(a) for an illustration where we have the situation as seen from the eye position. Pixel p' is on hard shadow boundaries, and it may jump to q' in a pass with a large step length. The ray passing through point q and the occluder point o intersects with the light. So q' is treated as in a penumbra region, while it should be in an umbra region. q' may further flood its occluder to other nearby pixels. This results in a hole around point q' in the final image. Figure 5.7(b) gives an example in our fantasy scene with such an artifact.

We have two simple ways to reduce the occurrences of such problems. The first way is to record object identities of occluders, and to allow the jump flooding of an occluder only to the pixels with the same object identity to this occluder. (Note that this is not the same use of object identities as in [BS02] where self-shadowing is lost.) This effectively eliminates the jump-too-far problem of wrong

inner penumbrae. However, this method still fails for some concave objects. The second way is a simple form of continuity check: a midpoint between the boundary of the occluder and the pixel under consideration must have penumbra intensity before the pixel under consideration can have penumbra intensity. Again, one can create an arrangement to show that this does not fully remove the jump-too-far problem. Nevertheless, our experiments with both of them are very positive – both manage to remove all the hole artifacts.

Parity and Smoothness. We note that the relationship of distance and intensity used in our argument for JFA-L is generally not true here for flooding in the eye space. This is because the image seen by the eye is distorted by the projection, with respect to the center of the light. Two points may be very near to each other when seen from the center of the light, but are actually very far apart when seen from the eye. So we cannot ascertain that the intensity is in any way related to the distance in the eye space, and thus cannot use the results on distance to argue for the good quality of JFA-E. Nevertheless, we do not detect significant problems, as testified in the experiment discussed in the next paragraph, when using the jump flooding algorithm in place of the standard flooding algorithm.

Flooding Experiment. Similarly to JFA-L, we generate from the fantasy scene (Figure 5.9) two series of 462 frames; one using the jump flooding algorithm and the other using the standard flooding algorithm. Figure 5.8(a) shows the corresponding curves in the case of JFA-E (with respect to the total number of 10k to 99k penumbra pixels in each image) similar to that of JFA-L in Figure 5.4(a). The results of the frame with the largest difference are shown in Figure 5.8(b) and 5.8(c). Once again there are no significant and noticeable visual differences between each pair of frames. In the experiment, we also note that the number of

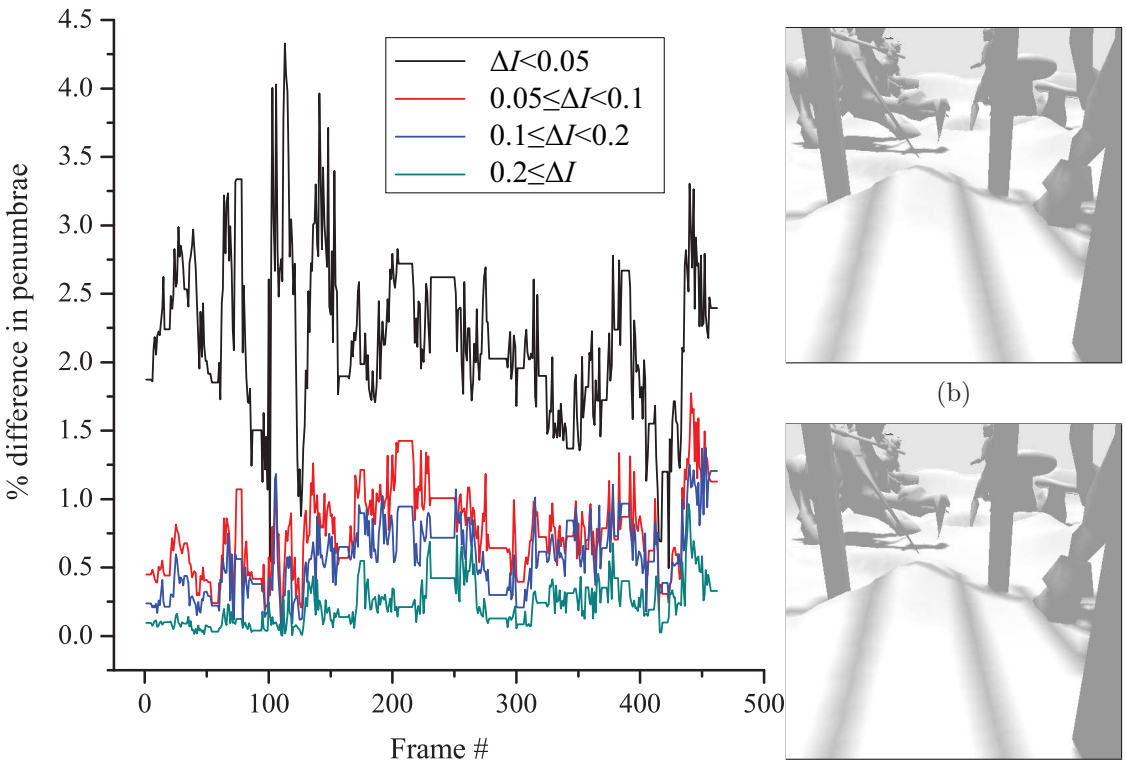


Figure 5.8: (a) Statistics of pixels with differences in intensities; (b) the result generated by the jump flooding algorithm for the frame with the largest difference; and (c) the result generated by the standard flooding algorithm for the frame with the largest difference.

pixels with differences in parity between the results of the jump flooding algorithm and of the standard flooding algorithm is less than 6.8%.

5.5 Experimental Results

We have implemented JFA-L and JFA-E with trapezoidal shadow maps [MT04] using Visual C++.NET 2003 and Cg 1.4. The hardware platform is Intel Pentium IV 3.0GHz, 1GB DDR2 RAM and NVIDIA GeForce 6800 GT PCI-X with 256MB DDR3 VRAM.

Model	Knight	Teapot	Bunny	Fantasy	Fantasy2
#Triangles	634	6,320	69,451	101,861	216,440

Table 5.1: Numbers of triangles in the testing scenes.

We use five different scenes with increasing magnitude of number of triangles to test both JFA-L and JFA-E. The numbers of triangles of these scenes are listed in Table 5.1. The fantasy scene is shown in Figure 5.9. This scene includes a tree, a pergola, three mushrooms, a rock, a flower and four animated characters in a terrain. The Fantasy2 scene is similar to the Fantasy scene with some duplicated objects to increase the number of triangles.

JFA-L Results. We perform JFA-L on the five scenes. For every scene, we use the shadow maps of three different resolutions, 256×256, 512×512 and 1024×1024, while fixing the resolution of the screen at 512×512. The result of the fantasy scene using JFA-L is shown in Figure 5.9(a). Figure 5.10(a) shows the average time taken per frame by JFA-L for different scenes. Each time bar highlights the part taken by JFA in a light color. It is clear that the time taken by JFA is mostly dependent on the resolution of the shadow map. For a same model, when we double the resolution, the number of the total pixels increases by four times and so the time taken by JFA increase accordingly. Among different models, the time taken by JFA differs slightly as different number of pixels are active (with more complex silhouette having more) in executing the same fragment program.

JFA-E Results.

We perform JFA-E on the five scenes too. For every scene, we use the screens of three different resolutions, 256×256, 512×512 and 1024×1024, while fixing the resolution of the shadow map at 1024×1024. The result of the fantasy scene using

(a) (b)

Figure 5.9: Results of the fantasy scene using (a) JFA-L, and (b) JFA-E.

JFA-E is shown in Figure 5.9(b). Figure 5.10(b) shows the average time taken per frame by JFA-E for different scenes. We note that JFA-E runs with 6 passes of the jump flooding process on a screen with resolution of 512×512 can achieve 23 frames per second for the knight model, as compared to Arvo et al.'s approach of about only 2 frames per second with 63 passes of standard flooding process (to achieve similar quality as ours). When using only 20 passes as suggested in their paper, Arvo et al.'s algorithm can only achieve 6 to 7 frames per second, while generating results of lower quality. Figure 5.11 shows the comparison of the result of JFA-E and the results of Arvo et al.'s algorithm with two different numbers of passes.

JFA-L and JFA-E. By the way the intensity is calculated, both algorithms have the problem of shrinking umbra regions as that of Arvo et al.'s algorithm. This problem is observed at the shadow of any two objects crossing with each other;

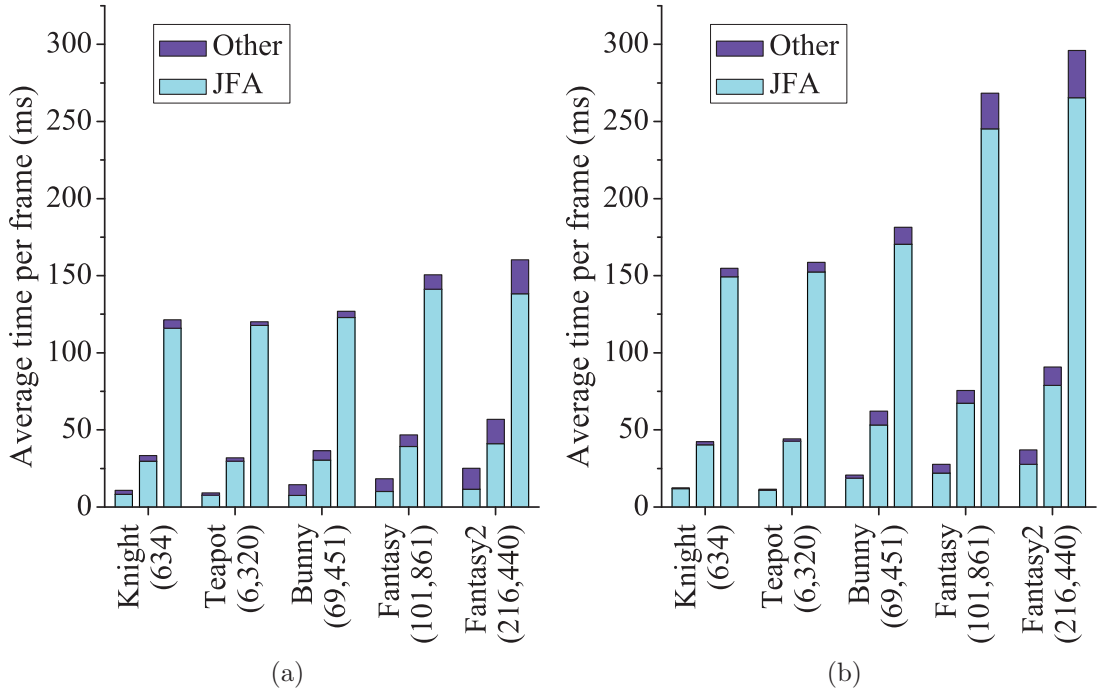


Figure 5.10: Comparison of the time of JFA and other parts in (a) JFA-L and (b)JFA-E. The three time bars for every model (from left to right) represent resolutions of 256×256, 512×512 and 1024×1024 of the shadow map (for JFA-L) or the screen (for JFA-E). The numbers in the parenthesis are the numbers of triangles of the models.

(a) (b) (c)

Figure 5.11: Soft shadows of Knight generated using (a) JFA-E with 6 passes, (b) Arvo et al's algorithm with 63 passes, and (c) Arvo et al's algorithm with 20 passes.

see, for example, the shadows cast by the crossing of the ear and the buttock of the bunny in Figure 5.3(h) and Figure 5.6(f).

Both algorithms rely on the quality of the shadow map. As such, when there is an artifact in the shadow map (for JFA-L) or hard shadows (for JFA-E), the corresponding soft shadow results also have artifacts. In particular, when some features of hard shadows appear and disappear, they are further exaggerated due to the flooding and appear as undesirable blinking of soft shadows.

JFA-E has a problem with inner penumbrae when a small hole inside the hard shadows is changing in shape or disappearing due to the insufficient resolution of the shadow map. When this happens, we observe undesirable blinking of soft shadows – such effect does not appear in JFA-L as it has no inner penumbrae. JFA-E thus is not appropriate for scenes with small holes in hard shadows. Such blinking effects also appear when a part of the hard shadow boundaries is occluded by other objects in one frame but become visible in the next. Without this artifact, during walkthrough or flythrough, JFA-E produces more realistic soft shadows, because it can generate both inner and outer penumbrae. On the other hand, we like JFA-L for it is robust in producing plausible soft shadows for scenes with complex hard shadow boundaries, and it runs faster than JFA-E.

To compare soft shadows derived from the shadow volume algorithm and from the shadow mapping algorithm, we experiment with some codes available from the web on the algorithm based on the shadow volume algorithm [AAM03]. This algorithm is object-based and handles only objects that are polygonal and manifolds. It does not seem to scale well for complex scenes as it performs with low frame rate for scenes containing just a few thousands of triangles.

5.6 Concluding Remarks

This chapter presents two novel and simple real-time soft shadow algorithms based on the shadow mapping algorithm. In JFA-L, we use JFA to generate a penumbra map directly from the shadow map, and then use it to generate soft shadows. In JFA-E, we improve upon Arvo's algorithm in speed and in quality of soft shadows. Both algorithms are purely image-based. They can achieve interactive speed with plausible soft shadows even for very complex scenes. One possible future work is to record more information of occluders, such as silhouette edges rather than just silhouette points, to compute more accurate intensity.

In another view, these algorithms extend our understanding in utilizing the GPU for non-trivial communication among processing units that is seldom found in the present uses of the GPU. Possible future work is to continue searching for other uses of JFA or other interesting communication patterns utilizing the GPU.

Chapter 6

Delaunay Triangulation

As reviewed in Chapter 4, there are many researches using the GPU to compute Voronoi diagrams and distance transforms. However, the dual graph of the Voronoi diagram – the Delaunay triangulation has been barely touched in the GPGPU area. No work has been reported to use the GPU to compute (exact) Delaunay triangulations. Previous work does mention the possibility of computing from a discrete Voronoi diagram its "dual" graph to obtain the corresponding Delaunay triangulation, while our work realizes such a task is not a simple feast.

Since JFA has already been successfully used on the computation of the Voronoi diagram in discrete space, in this chapter, we discuss how to apply it on the computation of the Delaunay triangulation in continuous space. Next, the definition of Delaunay triangulation and its some basic properties are given first, so that we can discuss specifically the challenges of this task.

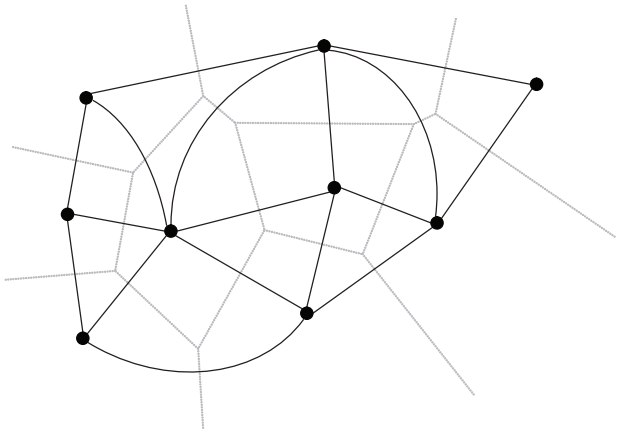


Figure 6.1: The dual graph of Voronoi diagram.

6.1 Definition

There are various ways to define the Delaunay triangulation. Here, we adopt the one using the duality between the Voronoi diagram and the Delaunay triangulation. Following our definition of Voronoi diagram in Section 4.1, we create a *dual graph* $\mathcal{G}$ of the Voronoi diagram $\mathcal{V}(S)$ of a set of sites S. The nodes of this graph $\mathcal{G}$ are the sites of the Voronoi diagram. So there is one node in $\mathcal{G}$ for every Voronoi region in $\mathcal{V}(S)$. There is an edge of $\mathcal{G}$ between two sites if, and only if, the Voronoi regions of these two sites sharing a Voronoi edge. So the edges in $\mathcal{G}$ one-to-one correspond to the Voronoi edges in $\mathcal{V}(S)$. As the result, the faces of $\mathcal{G}$ also one-to-one correspond to the Voronoi vertices of $\mathcal{V}(S)$. An example of such a dual graph (in black), together with the Voronoi diagram (in gray), is shown in Figure 6.1.

Now we embed $\mathcal{G}$ into plane using straight lines, where all the edges of $\mathcal{G}$ become line segments, and thus all the faces of $\mathcal{G}$ become polygons. Such embedding of $\mathcal{G}$ is called the *Delaunay graph* of S. This is named after the Russia mathematician Boris Nikolaevich Delone (Russian: Борис Николаевич Делоне; Delaunay is the French spelling of his surname). Figure 6.2 shows the Delaunay graph (in black)

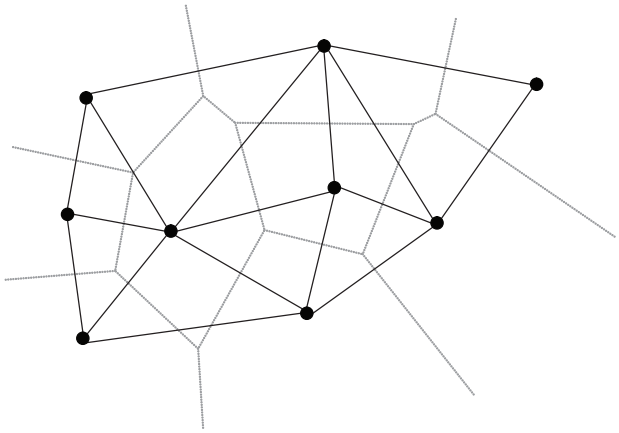


Figure 6.2: The Delaunay graph superimposed on the Voronoi diagram.

superimposed on the corresponding Voronoi diagram (in gray).

The number of edges of a face in the Delaunay graph is same to the number of the Voronoi regions incident to the Voronoi vertex corresponding to this face. So, if the sites in S are non-degenerated, where every Voronoi vertex having only three Voronoi regions around it, all the faces in in the Delaunay graph are triangles, and the Delaunay graph is thus a triangulation. Such triangulation is called *Delaunay triangulation*, denoted by $\mathcal{T}(S)$. In the following discussion in this chapter, unless otherwise specified, we only consider non-degenerated cases, and thus do not distinguish between the Delaunay graph and the Delaunay triangulation.

The Delaunay triangulation has many nice properties. We list some of them without proofs as follows: (for more details about the proofs, see [PS85, OBSC99, dBvKOS00])

- The Delaunay graph of a planar point set is a plane graph.
- Two sites a and b form an edge in the Delaunay triangulation if and only if there is a circle passing through a and b and does not contains any sites of

S in its interior.

- Three sites a, b and c form an triangle in the Delaunay triangulation if and only if the circle passing through a, b and c contains no sites of S in its interior.

- A triangulation $\mathcal{T}$ of a set of sites S is a Delaunay triangulation if and only if the circumcircle of any triangle of $\mathcal{T}$ dose not contain a site of S in its interior.

- Delaunay triangulation of a set of sites S maximizes the minimum angle over all triangulations of S.

6.2 Related Work

Similar to the Voronoi diagram, the Delaunay triangulation is also a very important concept in computational geometry. It has been thoroughly studied during many decades, and there are numerous papers on it. Good surveys of the CPU algorithms on Delaunay triangulation can be found in [For92, SD95]. Among all the algorithms computing the Delaunay triangulation, three algorithms are most commonly used. They are: divide-and-conquer, plane-sweep and random incremental.

The divide-and-conquer algorithm recursively divides the set of sites into two smaller sets, until every set is small enough (say containing only 3 sites) to be Delaunay triangulated in a constant time. Then, it gradually merges two small Delaunay triangulations into a bigger one, until all the parts are merged into one triangle mesh, which is the final result.

The plane-sweep algorithm uses a sweeping line to sweep the whole plane from left to right, and gradually builds the triangle mesh. Every time the line touches a site, or a circle passing through three sites, the triangle mesh is updated accordingly (new triangles are inserted or existing triangles are removed). This algorithm is first proposed to compute the Voronoi diagram. As the Delaunay triangulation is the dual graph of the Voronoi diagram, it can also be used to compute the Delaunay triangulation.

The random incremental algorithm starts from a triangle bounding all the sites, and then inserts the sites in S one by one, in a random order. Every time a new site is inserted into the triangle mesh, a triangle containing this site is split into three triangles (or, if the site happens to lie on an edge, the two triangles sharing this edge are both split into two triangles).

All these three algorithms have the same expected time complexity of $O(n \log n)$. However, this does not mean all run with the same efficiency in practice. Shewchuk [She96] implements all three algorithms in his *Triangle* package. His experience is that the divide-and-conquer algorithm is the fastest, and the random incremental algorithm slowest. To our best knowledge, *Triangle* is the fastest program to-date to compute 2D Delaunay triangulations. It has been optimized for more than ten years to have efficient memory management and robust geometry computations.

Although there are numerous algorithms using the CPU to compute Delaunay triangulations, as we mentioned before, there is no known exact algorithm that capitalizes on the parallel capability of the GPU to compute Delaunay triangulations. On a quick glance, due to the duality between the Voronoi diagram and the Delaunay triangulation, it is easy to compute the Delaunay triangulation from those algorithms that compute discrete Voronoi diagrams with the GPU. In fact,

Hoff et al. mention a little bit on constructing the Delaunay triangulation from the Voronoi diagram in [HCK+99]. They first construct a Voronoi diagram using the algorithm proposed, and then find the Voronoi edges in it. Due to the duality between Voronoi edges and Delaunay edges, they can construct a "Delaunay triangulation" of the sites.

However, there are several reasons making the simple approaches, such as the one in [HCK+99], not working well. We name here just two reasons. First, The adjacency relation between Voronoi regions in a Voronoi diagram in continuous space may not be the same as that in the corresponding discrete Voronoi diagram. For example, in Figure 6.3, the red and blue Voronoi regions should be adjacent to each other in the Voronoi diagram in continuous space. However, they are separated by the yellow and green Voronoi regions here in the discrete Voronoi diagram. Second, as illustrated by Figure 4.2, Voronoi regions in a discrete Voronoi diagram may consist of two or more disconnected components. For both reasons, the topology changes in the discrete Voronoi diagram, and the dual graph of a discrete Voronoi diagram is not the required Delaunay triangulation any more. Thus, special processing needs to be incorporated in order to still derive efficiently the Delaunay triangulation from a discrete Voronoi diagram. Besides, there are also challenges due to limited texture size and computational precision of the discrete Voronoi diagram to consider. We discuss all these in later sections.

Another approach in attempting to build Delaunay triangulation using graphics hardware is proposed by Yamamoto [Yam04]. The main purpose of his work is to compute the 3D convex hull. Because Delaunay triangulation in n dimension can be computed by lifting the sites to $n+1$ dimension and computing the convex hull in $n+1$ dimension [Ede87], the proposed method can also be used to compute

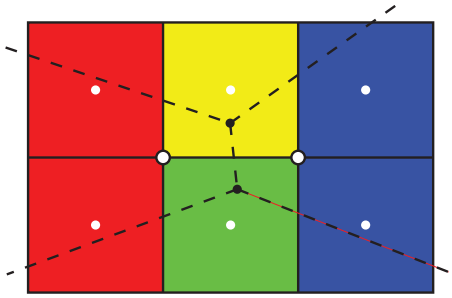


Figure 6.3: Adjacency differs in the discrete Voronoi diagram. The red and blue Voronoi regions should be adjacent to each other in continuous space, but are separated by the yellow and green regions in discrete space. Dashed lines are the Voronoi edges in continuous space.

the 2D Delaunay triangulation. However, as pointed by the author, the Delaunay triangulation generated by this method is not exact and may contain some errors (topology error and numerical error).

Our algorithm in this chapter uncovers numerous challenges to first overcome in order to transform discrete Voronoi diagrams to Delaunay triangulations, and, very importantly, to out-perform the best implementation of 2D Delaunay triangulations such as *Triangle*. These challenges are next explained in detail together with the steps in our algorithm.

6.3 Algorithm

Because the texture used in GPU programming is discrete in nature, the GPU is often used to solve problems in discrete space, where the coordinates of all the points are integers. In this chapter, we propose an algorithm, based on JFA, to compute the Delaunay triangulation in continuous space, where the coordinates of all the points are real numbers.

6.3.1 Algorithm Overview

The idea of our algorithm is straightforward: we map the set of sites S to a texture, compute the discrete Voronoi diagram $\mathcal{D}(S')$ where $|S'| \leq |S|$ and S' is on the left-bottom corners of the pixels, use it to derive a triangulation $\mathcal{T}'(S')$ that is close to the real Delaunay triangulation $\mathcal{T}(S)$, and then perform edge flip and repair on $\mathcal{T}'(S')$ to obtain $\mathcal{T}(S)$. There are two major decisions made in designing the above process as we discuss in the following two paragraphs. In the discussion, the color of every pixel is decided by the color of the Voronoi region containing its center.

In the generation of a triangulation from a Voronoi diagram, we could either generate *edges* or *triangles* for the triangulation. From $\mathcal{D}(S')$, an edge of $\mathcal{T}'(S')$ can be derived from each pixel which has neighboring pixels belonging to exactly one different Voronoi region. As a result, an edge of $\mathcal{T}'(S')$ may be derived from many pixels, and the algorithm would need to remove duplicate edges. It is time consuming to check duplicates and thus the alternative of computing triangles for $\mathcal{T}'(S')$ is adopted: a corner (other than those on the boundary of the texture) is shared by four pixels; a triangle is derived from a corner surrounded by pixels belonging to exactly three Voronoi regions, and, similarly, two triangles by pixels belonging to exactly four Voronoi regions. Note that these triangles are not necessarily triangles in $\mathcal{T}(S)$ due to the discrete nature of $\mathcal{D}(S')$.

The point set S is mapped via a function f to S'. The measure of distance between a' and b' in S' (in the computation of $\mathcal{D}(S')$) could be defined as the Euclidean distance between a and b, where $a' = f(a)$, $b' = f(b)$. According to our experiments, this approach may generate triangles with orientations not consistent to the Voronoi vertices, and may generate crossing edges. Both of them

needs additional computation. In our approach, we define the distance between a' and b' as the Euclidean distance between themselves. On the whole, our approach derives $\mathcal{T}'(S')$ such that there are no duplicate and overlapping triangles, and the orientation of triangles are consistently counter-clockwise. In summary, $\mathcal{D}(S')$ and $\mathcal{T}'(S')$ are both treated as in digital geometry.

The algorithm contains following ten steps, six on the GPU and the other four on the CPU (the prefix G means a GPU step, and C means a CPU step):

Step G1. Write input sites S (in continuous space) into a texture (discrete space) with surviving sites denoted as S';

Step G2. Compute the discrete Voronoi diagram $\mathcal{D}(S')$ using JFA;

Step G3. Re-assign, if needed, each island (explained later) to a Voronoi region that is connected to a site;

Step G4. Find Voronoi vertices that are corners surrounded by three or four Voronoi regions;

Step G5. Chain up Voronoi vertices within each row in the texture;

Step G6. Construct one or two triangles for $\mathcal{T}'(S')$ from each Voronoi vertex;

Step C1. Complete the construction of $\mathcal{T}'(S')$ with triangles around the convex hull of S';

Step C2. Shift each site $s' \in S'$ in $\mathcal{T}'(S')$ to its respective position in S to result in a triangulation $\mathcal{T}'(S'')$ where $S'' \subseteq S$;

Step C3. Insert each point in $S - S''$ into $\mathcal{T}'(S'')$ resulting in a triangulation $\mathcal{T}'(S)$; and finally

Step C4. Flip edges in $\mathcal{T}'(S)$ to result in $\mathcal{T}(S)$ as required.

Next, we explain every step in detail one by one.

6.3.2 GPU Steps

The purpose of the GPU steps is to compute a mesh that will be passed on to the CPU to complete as $\mathcal{T}'(S')$. G1 to G5 are carried out using shading programs in Cg, and G6 using CUDA that enables parallel construction of triangles for the mesh.

Step G1 – Write Sites into Texture

We scale the continuous space such that all sites can fit into our chosen texture (discrete space), and then render these sites into the texture. Note first that due to the resolution of the texture and the distribution of the sites, there may be two or more sites mapped to the same pixel. Only one of these sites is written at that pixel, and the rest thus become *missing sites*, and will be inserted to be a part of the triangulation in Step C3. Note second that we use half-float format for the texture to represent precisely integers between -2048 and 2047. This matches the largest texture size of 4096×4096 in currently available graphics cards. We should switch to texture of integer format when Cg 2.0 becomes available.

Step G2 – Compute Voronoi Diagram

With the texture of size $m \times m$ storing some sites, there are various ways to generate the discrete Voronoi diagram [HCK+99, Den03, RT06a, FG06]. The naïve approach of *standard flooding* can propagate information of a site step-by-step radically but it takes time proportional to $\sqrt{2}m$. We choose the better propagation approach – 1+JFA to compute the discrete Voronoi diagram. The details of this step are already explained in Chapter 4. We note that 1+JFA does not compute the exact

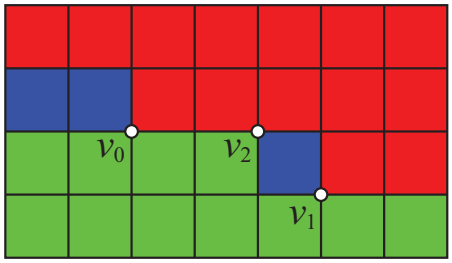


Figure 6.4: Islands generate duplicate triangles and the triangle with inconsistent orientation.

discrete Voronoi diagram. The possible errors (on a few pixels) have little chance to alter the topology of the Voronoi diagram. This thus does not have adverse effects on the correctness of the algorithm.

Step G3 – Remove Islands

As shown in Figure 4.2, a Voronoi region of a site in a discrete Voronoi diagram may contain more than one connected region due to the discrete nature of the texture. We call all those connected components not containing the site *islands*. Their presence can complicate the identification of Voronoi vertices (in next step). Careless identification of the islands can result in duplicate triangles and crossing edges in $\mathcal{T}'(S')$.

Figure 6.4 demonstrates the problem of duplication triangles. In this figure, v_0 and v_1 are both Voronoi vertices. Both of them are incident to the same three Voronoi regions, i.e. red, blue and green regions (in counterclockwise order). So two same triangles with red, blue and green sites as their vertices are generated by v_0 and v_1. Moreover, the Voronoi vertex v_2 generates a triangle with the same three sites, but different orientation. Such triangle with inconsistent orientation is also not desirable.

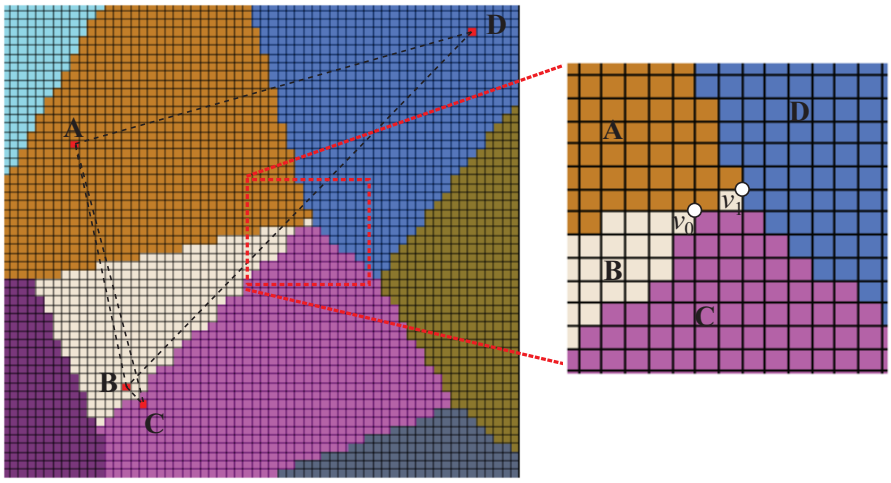


Figure 6.5: Islands generate crossing edges.

Figure 6.5 shows a case of the problem of crossing edges. In this figure, A, B, C and D are four sites, and v_0 and v_1 are two Voronoi vertices. v_0 generates the triangle ABC, and v_1 generates the triangle ABD. The edge AC crosses with the edge BD. Thus, the result is no longer a triangulation.

For our computational purposes, we discover a pixel as a part of an island as follows. For a pixel p belonging to the Voronoi region of a site s, we build a coordinate system with the origin at p and aligning with the texture. If s is in the first quadrant, we check the three neighbors of p at $(1, 0)$, $(1, 1)$ and $(0, 1)$. If all these three neighbors have colors different from that of p, we say p is in an *island*. We define similarly when s is in the other three quadrants. If s is on one of the axes, we only check the neighbor next to p and between p and s. For example, if s is on the positive x-axis, we only check the neighbor at $(1, 0)$. If this neighbor has a color different from that of p, we say p is in an island. For a pixel in an island, we determine the nearest site among those of its three (or one) neighbor(s), which we use to identify it as an island. The color of the island pixel is then replaced by

the color of the closest sites determined.

The removal of one pixel that is in an island can expose other pixels to be in islands. The operation of removing these pixels is then repeated until no new ones are found. The OpenGL extension `GL_ARB_occlusion_query` is useful in detecting if any pixels in islands are removed and the process must be continued. In practice, a small number (less than 5) of iterations are sufficient to remove all the islands.

Step G4 – Locate Voronoi Vertices

With all islands removed, we next identify and record Voronoi vertices at pixels. For this purpose, we check all the corners of the pixels, not the pixels themselves, to find the Voronoi vertices. The advantage of using corners is that there are only four pixels around a corner. So we need to do only four texture queries for every corner. On the contrary, there are eight neighbors around a pixel, so if we check for pixels, we must do eight texture queries for every pixel. Furthermore, four neighbors around does not only reduce the number of texture queries, but also reduce the possible combinations of them. With only four neighbors around every corner, there are only three possible cases:

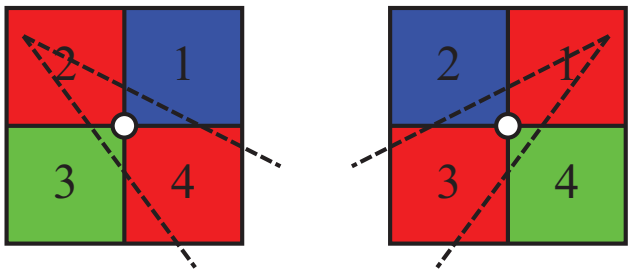
- Case 1: there are only one or two colors, then this corner is not a Voronoi vertex;
- Case 2: there are three colors in total, then this corner is a Voronoi vertex and generates only one triangle (with some exceptions, see explanation below);
- Case 3: there are four colors in total, then this corner is a Voronoi vertex and generates two triangles.

Case 1 is the simplest case, and can be ignored in Step G6. For case 2 and 3, the GPU inserts one or two triangles into the triangle mesh accordingly in Step G6. However, there are two special cases of case 2 that we need to pay more attention to. Figure 6.6 shows these two exceptions of case 2. In these cases, although there are three colors around a corner, the two red pixels are not adjacent by an edge of a pixel. This means the red Voronoi region is a slim one, and separates the green Voronoi region and the blue Voronoi region. So such a corner should not be identified as a Voronoi vertex.

So case 2 has only four possible valid subcases (using the numbering in Figure 6.6):

- Subcase (a): pixels 1 and 2 have the same color and pixels 3 and 4 have the other two colors;
- Subcase (b): pixels 2 and 3 have the same color and pixels 4 and 1 have the other two colors;
- Subcase (c): pixels 3 and 4 have the same color and pixels 1 and 2 have the other two colors; and
- Subcase (d): pixels 4 and 1 have the same color and pixels 2 and 3 have the other two colors.

These four subcases together with case 1 and case 3 are all given a unique case number. To facilitate the computation, the case number is recorded at the lower left pixel incident to the corner. Notice that case numbers of corners along the bottom and left boarders of the texture are not stored at any pixels. This does



(a) (b)

Figure 6.6: Two cases which are not identified as Voronoi vertices.

not pose any problem as these corners have at most two colors around them and are never Voronoi vertices by our definition.

Step G5 – Chain up Voronoi Vertices

The Voronoi vertices identified in a texture are used to generate corresponding triangles in Step G6. If we would use the texture directly, the GPU must go through all the pixels to process only the Voronoi vertices. To accelerate this process, we introduce another step on the GPU to perform a standard parallel prefix computation [Ble90] to record for each pixel the nearest Voronoi vertex to its right in the same row. This step is in fact a JFA process in 1 dimension.

As discussed in Chapter 4, JFA can be applied on the computation of the Voronoi diagram. If JFA is used in 1D (in our practice, we perform JFA in every row of the texture), we can get the Voronoi diagram in 1D. That means every pixel knows its nearest site in the same row. In this step, the Voronoi vertices are used as the sites of the 1D-JFA. The 1D-JFA in this step is slightly different from the standard JFA. When performing the flooding process, the information is only propagated to the left side of every site. In other words, every pixel only knows the nearest site to its right. This change reduces the number of texture queries in

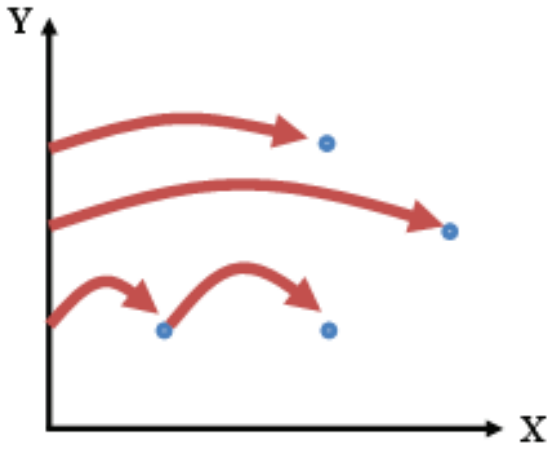


Figure 6.7: Illustration of 1D-JFA.

every pass of JFA from two to one, and thus is faster.

After the 1D-JFA process, every pixel at the left most column of every row records the position of the first Voronoi vertex in the row as its nearest site to the right. So we can "jump" over all the pixels which are not Voronoi vertices. Similarly, the right neighbor of the first Voronoi vertex records the position of the second Voronoi vertex in this row. Continuing this process, we can go through all the Voronoi vertices in one row. Figure 6.7 illustrates the idea of the usage of 1D-JFA. The idea of 1D-JFA is similar to that in [Hor05]. This step is particularly useful for the texture with a large resolution.

Step G6 – Construct a Triangle Mesh

The result of Step G5 stored in an OpenGL texture is read into a pixel buffer object to allow CUDA to construct triangles in parallel. Reading a big texture takes time. One can use the OpenGL extension `GL_ARB_pixel_buffer_object` to achieve the asynchronous read-back: one can issue the command *glReadPixels* to

start the reading and immediate move on to do other processing without being blocked by the reading (until the command *glMapBuffer* is called). In practice, we use this read-back time to derive a list of vertex coordinates in integer format from the input in double format. This list is used in Step C2 for shifting sites. Note that this Step G6 could not be done well with Cg as is the case of Step G1 to Step G5, since we need scatter operations that are only available in CUDA. Also, Step G1 to Step G5 do not perform well when implemented with CUDA since they map naturally to the parallel computation using shader programs.

To construct the mesh $\mathcal{T}'(\mathcal{S})$, we use many parallel processes in CUDA, each of which handles one row of data in our texture. With the information on the number of triangles to be generated from Step G5, we can pre-allocate memory to enable processes to concurrently insert triangles into the mesh. For each pixel recording a Voronoi vertex due to three colors, a triangle is formed with the three sites represented by these colors. Similarly, for a Voronoi vertex due to four colors, two triangles are formed by the corresponding four sites. Note that CUDA running on the NVIDIA GeForce 8800 GTX does not support atomic operations. As such, updating the list of triangles incident to each site (which is used to link up adjacent triangles) cannot be performed in parallel. This results in a slight penalty on the computation time.

6.3.3 CPU Steps

After all the six GPU steps, the triangle mesh $\mathcal{T}'(S')$ is "almost" a Delaunay triangulation with some issues to fix. First, not all triangles incident to the boundary of the convex hull are necessarily included in $\mathcal{T}'(S')$, and $\mathcal{T}'(S)$ may contain more

than one connected component. Second, the sites in $\mathcal{T}'(S')$ are still in discrete space, and they need to be shifted back to their corresponding coordinates in continuous space. Third, some sites are not included in $\mathcal{T}'(S')$ and need to be inserted into $\mathcal{T}'(S')$ in order to get a triangle mesh $\mathcal{T}'(\mathcal{S})$. Last, some triangles in $\mathcal{T}'(\mathcal{S})$ are not in the Delaunay triangulation $\mathcal{T}(S)$, and we need to perform the edge flip operation on them. All these are addressed in the CPU steps.

Data stored in the texture are no longer needed, except for pixels on the boundary of the texture. These are read in parallel by CUDA into an array and copied to the main CPU memory for Step C1.

Step C1 – Fix Convex Hull

Because some Voronoi vertices may lie outside the texture, their corresponding triangles may miss from the triangle mesh generated in Step G6. In the worse case, such missing triangles may separate the triangle mesh into several disconnected parts. Figure 6.8 illustrates this phenomenon.

To fix this, we traverse along the boundary of the texture in counterclockwise order starting from the lowest site (with the idea similar to the Graham's scan algorithm [Gra72, And79]), to discover the sites appearing along the boundary. We push the first two sites found into a stack. From the third site on, we check every consecutive three sites. If they are in clockwise order, it means there is a missing triangle there. A new triangle is generated and inserted into $\mathcal{T}'(S')$. Otherwise, the new site is pushed into the stack, and we go on to discover the next site. After we finish the whole round (returning to the lowest site), all the missing triangles are inserted into the mesh. The sites left in the stack are those on the convex hull.

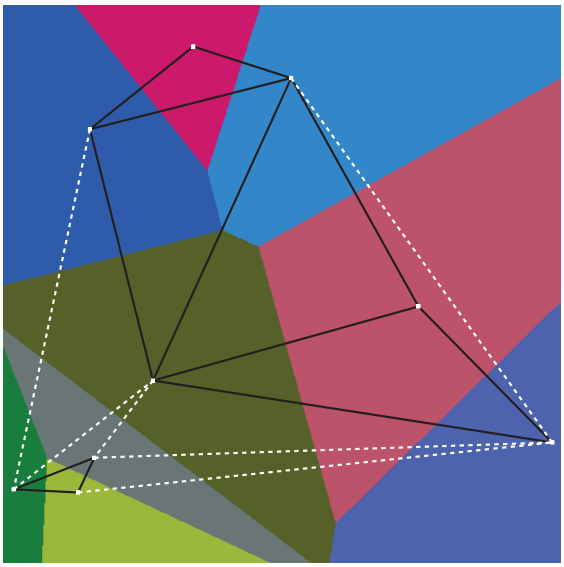


Figure 6.8: Missing triangles due to Voronoi vertices lying outside the texture. The five triangles with one or more white dashed edges are missing. The triangle mesh is separated into two disconnected parts.

After this step, we have a triangle mesh in discrete space $\mathcal{T}'(S')$. The next step shifts it back into continuous space.

Step C2 – Shift Sites

This step shifts each site at s' in discrete space back to their original position at s in continuous space to compute $\mathcal{T}'(S'')$ where $S'' \subseteq S$. For every site s', we first categorize it into one of the four cases according to its original position s and the triangle fan consisting of all the triangles around s':

- Case 1: s is in one of the triangles in the triangle fan (Figure 6.9(a)-(c));
- Case 2: s is in a triangle that is not in the triangle fan (Figure 6.9(d)-(e));
- Case 3: s is out of the triangle mesh, but lies in the open region defined by an angle of the triangle fan, (Figure 6.9(f));

- Case 4: otherwise (Figure 6.9(g)-(h)).

The four cases are processed differently.

For case 1, there are three subcases: case 1a (Figure 6.9(a)), case 1b (Figure 6.9(b)) and case 1c (Figure 6.9(c)). For case 1a, directly moving s' to s does not create any intersection. So we just change the coordinates of s' to that of s, and we are done. This is the simplest case, and we call it "good case". In practice, more than 90 percent of sites are of the good case (case 1a). Accordingly, all the other cases are called "bad cases".

When we cannot directly move s', we first insert s into $\mathcal{T}'(S')$ and split the triangle containing it into three small triangles (or if s lies on an edge, we split the two triangles sharing this edge into four small triangles). Then we delete all the triangles around s'. If s' is not on the convex hull (case 1b), we get a polygonal hole. This hole is triangulated and the new triangles are inserted into $\mathcal{T}'(S')$. When s' lies on the convex hull (case 1c), we get a polygonal line after deleting all the triangles around s', and locally fix the convex hull for it.

Case 2 has two subcases, case 2a (Figure 6.9(d)) and case 2b (Figure 6.9(e)). The processing of case 2 is similar to that of bad subcases of case 1. We first insert s into $\mathcal{T}'(\mathcal{S}')$, and then delete the triangle fan around s'. After that, we patch the polygon hole or locally fix the convex hull for the polygonal line depending on whether s' is in the middle of the triangle mesh (case 2a) or on the convex hull (case 2b).

Case 3 only happens when s' is not on the convex hull. So we can delete the triangle fan around s' and then patch the polygon hole. When we categorize the case of s', we move from s' to s. For case 3, we finally move out of the convex hull. So we connect to s the end points of the last edge we passed to create a new

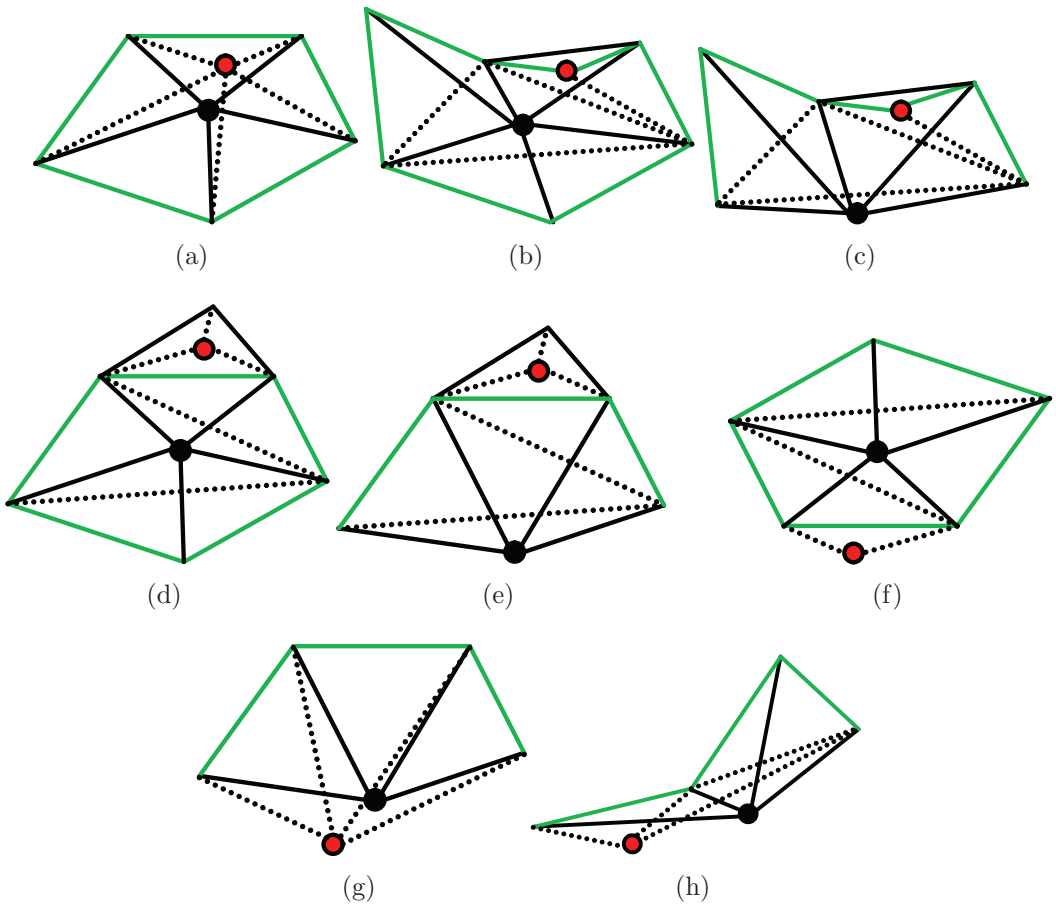


Figure 6.9: Cases for shifting sites. s' is represented by the black point and s by the red point. The solid lines show the triangles before shifting while the dotted lines show the triangles after shifting (the result of locally fixing of the convex hull is not shown here). The green lines show the hole or the polygonal line after deleting the triangle fan around s'.

triangle, and locally fix the convex hull around s.

For case 4, if directly moving s' to s does not generate any intersection, we just change the coordinates of s' to that of s, and locally fix the convex hull around it. This is case 4*a* (Figure 6.9(g)). Otherwise, when it is case 4*b* (Figure 6.9(h)), we delete the triangle fan around s' to get a polygonal line, and locally fix the convex hull. After that, we find a newly inserted edge on the convex hull which can be connected to s to create a new triangle. Lastly we locally fix the convex hull again around s.

Note that counterclockwise tests are often used in the process. Step G6 while generating triangles can pre-compute and record the outcomes of some of these tests. This is an optimization that improves the running time significantly. Not all such tests can be recorded as there are triangles generated in Step C1 that are not by Step G6, and there are triangles (after shifting some of their vertices) that CUDA (which currently supports only float format as compared to the input data that is in double format) cannot perform reasonably robust tests due to insufficient precision.

During this shifting process, we always maintain a triangle mesh with no crossing edge and with the convex hull fixed. After this step, the whole triangle mesh is already in original continuous space.

Step C3 – Insert Missing Sites

In Step G1, when we map all the sites from continuous space to discrete space (the texture), some sites may be mapped to a same pixel. Only one of them are recorded, and the others become *missing sites*. These missing sites must be inserted back into the triangle mesh for the completeness.

The basic idea of this step is similar to the random incremental algorithm [GKS92]. We can see the triangle mesh got from above steps as an intermediate result of the random incremental algorithm, and insert the missing sites one by one using this algorithm. The only problem need to note is how to find the triangle containing the new site. In the random incremental algorithm, there is a special data structure to help to quickly find such a triangle. In our case, we have a better way to do that.

Assume the missing site is t. t is missing because there is another site s maps to a same pixel of t, and s is already in the mesh achieved from last step. Since these two sites map to a same pixel, they cannot be very far away. We can just find the triangle contains t around s. The finding process is similar to the category process in Step C2 – just imagine s is the site to be shifted and t is the destination. So we can continue to use the four cases discussed in Step C2.

We do not need to delete any triangles in this step. For case 1 and case 2, we can directly insert the t into the triangle which contains it. For case 3, we connect t and the end points of the last edge we passed to it, to create a new triangle, and then locally fix the convex hull. For case 4, the triangle fan around s is a partial triangle fan. We find the first edge and the last edge, and find one of them (or both) that can directly connect to t without causing any crossing. We connect t to the edge, and then locally fix the convex hull.

After all these steps, we get a triangle mesh in continuous space $\mathcal{T}'(S)$. The final step performs the usual edge flip process to guarantee the triangle mesh is a Delaunay triangulation.

Step C4 – Flip Edges

There are lots of reasons that the triangle mesh $\mathcal{T}'(S)$ may not be a Delaunay triangulation, and thus we need to perform this edge flip step. First, the low resolution of the texture may cause the topology change (as shown in Figure 6.3). Second, in Step G6, when there are four colors around a Voronoi vertex, we generate the two triangles arbitrarily, and such triangles may not be Delaunay. Third, when we shift the Voronoi sites from discrete space to continuous space, the Delaunay property of some triangles may be violated. And more, for cases other than case $1a$ and case $4a$, we need to triangulate a polygonal hole or a polygonal line, our triangulation also does not maintain Delaunay properties. Fourth, our fixing of convex hull (in Step C1, C2 and C3) does not guarantee to generate Delaunay triangles. Fifth, when we insert missing sites, we split the triangles containing them and generate new triangles which may not be Delaunay triangles.

In this step, we traverse all the triangles in the mesh one by one. For an edge ab of $\triangle abc$, it passes the empty circle property if ab is a convex hull edge, or if the other triangle $\triangle adb$ incident to ab is such that d lies outside the circumcircle of $\triangle abc$. If ab fails the empty circle property, an edge-flip occurs to replace $\triangle abc$ and $\triangle adb$ by triangles $\triangle adc$ and $\triangle cdb$, and new tests apply to edges of these new triangles. This step is performed until all edges pass the empty circle property. The resulting mesh is the required Delaunay triangulation $\mathcal{T}(S)$.

Next, we prove that our algorithm always generate the exact result of the Delaunay triangulation, despite the fact that JFA can generate some errors in the result of the discrete Voronoi diagram.

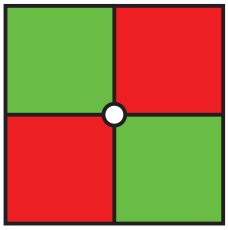


Figure 6.10: Two colors alternate around a corner. This case is impossible in the result of the standard flooding algorithm.

6.4 Correctness

This section proves that our algorithm computes correctly the Delaunay triangulation, $\mathcal{T}(S)$ of the input point set S. The main challenge is to show that we can derive from the discrete Voronoi diagram $\mathcal{V}(S')$ a triangulation (without performing extra effort in validity checking), or more accurately, a subgraph of some 2-manifold defined on S' (in Step G6) that we can augment with more triangles into a triangulation (in Step C1).

Consider using the standard flooding algorithm to generate a Voronoi diagram. A corner, incident by four pixels, cannot have two colors alternate around the corner (see Figure 6.10); otherwise, one pixel lies on the wrong side of the perpendicular bisector of these colors. In the following property, we adopt 8-connectedness where two pixels that are 8-connected at a corner is interpreted as connected through a tiny neck at the corner. For example, the pixels 2 and 4 in Figure 6.6(a) and pixels 1 and 3 in Figure 6.6(b) are all considered connected to each other.

Property 6.1 *Each Voronoi region generated by the standard flooding is simply-connected.*

Proof. Each Voronoi region generated by the standard flooding is connected by induction: the site whose information first reaches a pixel is the nearest site to

that pixel, so a pixel never changes its color (i.e. nearest site) once assigned; each pixel is connected to a neighbor having the same color, which is in turn inductively connected to the nearest site to the pixel. If a region $D(S'; s_i)$ is not simple, then at least one other region, say $D(S'; s_j)$, appears as a hole in $D(S'; s_i)$. Consider the line l passing through s_i and s_j, it is clear that the part of l closer to s_j than s_i and intersects $D(S'; s_i)$ cannot be part of $D(S'; s_i)$. So, there is no such hole $D(S'; s_j)$ inside $D(S'; s_i)$. □

Instead of the standard flooding algorithm, our algorithm uses the more efficient jump flooding algorithm. This is at the expense of two potential problems. First, a Voronoi region generated by JFA is not necessarily simply-connected. One reason is that there can be one $D(S'; s_i)$ that is disconnected into one containing the s_i plus the other islands. Islands are removed by Step G3, and thus no longer an issue. Another possibility is the existence of a $D(S'; s_j)$ appearing as a hole in $D(S'; s_i)$. Such a scenario has not been detected in our extensive experimentations. Nevertheless, we can perform some additional passes of flooding with the step length of 1, utilizing `GL_ARB_occlusion_query`, to remove holes. Second, JFA may not generate correct $D(S'; s_i)$ for all s_i. However, according to the results in Chapter 4, there are only a few pixels with errors, and these errors are at Voronoi vertices in most cases. All these do not change the topology of the discrete Voronoi diagram, and are thus not issues to our algorithm. We can thus assume Property 6.1 and Figure 6.10 hold for our algorithm.

In the following properties, *triangle* refers to one that was generated during Step G6. Let the triangle mesh generated at the completion of Step G6 be $\mathcal{T}'$.

Property 6.2 *All triangles in $\mathcal{T}'$ are consistent in orientation.*

Proof. We first consider three sites, a, b and c whose Voronoi regions are incident, in counterclockwise order, at a corner v. Then, we have $\triangle abc \in \mathcal{T}'$. Suppose we remove all sites other than a, b, c, and then generate a discrete Voronoi diagram $\mathcal{D}' = \mathcal{D}(\{a, b, c\})$. Note that those grid points colored by the colors of a, b and c, respectively, in $\mathcal{D}(S')$ remain the same in $\mathcal{D}'$; in particular, corner v is still incident by the same three colors in the same order. By joining a ray each from v to a, b and c, we read around the boundary of $\mathcal{D}'$ (making $\mathcal{D}'$ large enough) to note the order of the intersection points a', b', c'. Notice that all the three angles at v formed by the three rays are never larger than π; thus, a, b, c are in the same order as a', b', c'. Suppose on the contrary that this order is clockwise, then $\mathcal{D}'$ has a corner $v' \neq v$ that is incident by alternate colors of a and b, or b and c, or c and a so that each of the three Voronoi regions can remain simply connected (as shown in Property 6.1) while able to reach v in counterclockwise order. Such a v' is impossible as explained in Figure 6.10. So, a, b, c are in counterclockwise order, as needed.

Next, we consider four sites, a, b, c and d whose Voronoi regions are incident at a corner v in $\mathcal{D}(S')$. Viewing from v, we see that a, b, c and d are in the same order as their Voronoi regions incident at v; otherwise, for any three of them, we can use the argument in the previous paragraph to derive a contradiction. □

Note that in the case of a corner incident by four colors, one of the two triangles generated by the algorithm may have zero area. This does not cause any problem in the final result, as such triangles are removed (flipped away) in Step C4. We thus ignore, for simplicity, the existence of such triangles in our argument.

Property 6.3 *No two edges of triangles in $\mathcal{T}'$ intersect. Additionally, no endpoint*

of any edge lies on another edge.

Proof. Suppose on the contrary that a triangle with edge ac crosses another triangle with edge bd. Then, sites a, b, c, d form a convex polygon. Consider the discrete Voronoi diagram $\mathcal{D}' = \mathcal{D}(\{a, b, c, d\})$, which (as in the proof of Property 6.2) maintains grid points colored by the colors of a, b, c and d. The dual graph $\overline{\mathcal{D}}'$ of $\mathcal{D}'$ is a complete graph K_4 of four vertices as each site is a neighbor to the other three. We note that the only plane embedding of K_4 is with three vertices incident to edges that bound a region that encloses the fourth vertex (K_4 is the smallest non-outerplanar graph). But, our $\overline{\mathcal{D}}'$ has a 4-gon as its outer region, and thus cannot be a plane graph. This contradicts the fact that $\overline{\mathcal{D}}'$ is a plane graph as each Voronoi region of $\mathcal{D}'$ is simply connected (Property 6.1).

The above argument, with minor adjustment, is also applicable to show that no endpoint of any edge lies on another edge. □

Property 6.4 *No two faces of triangles in $\mathcal{T}'$ overlap in area.*

Proof. Because of Property 6.3, there are only two ways for triangles to overlap in area: first, when the two triangles have the same set of three vertices, and second, when one triangle is completely enclosed by the other. The former means that there are duplicate triangles, say $\triangle abc$. This means that in Step G6, there are two Voronoi vertices having colors of sites a, b and c around them. This leads to two of these three colors occurring alternatively around a corner, which is impossible as explained in Figure 6.10.

Next, we consider the latter way of overlapping. Let triangle $\triangle xyc$ be enclosed in triangle $\triangle abd$ where x may be a, and y may be b. Among x, y, c, we can assume the extremal property that c is closest to d when $x \neq a$ or $y \neq b$. So $\mathcal{D}(S')$ is

such that $D(S';c)$ appears as a hole in the union of $D(S';a)$ and $D(S';b)$. Now, consider the Voronoi diagram $\mathcal{D}' = \mathcal{D}'(\{a,b,c\})$. Grid points colored by the colors of the three sites in $\mathcal{D}(S')$ remain having the same colors in $\mathcal{D}'$. As a result, $\mathcal{D}'$ has the Voronoi region $D(\{a,b,c\};c)$ that is not connected or is enclosed by the other two regions. This is not possible as each Voronoi region for three sites must be connected and must appear on the boundary of a large enough $\mathcal{D}'$. □

Property 6.5 *Each edge in $\mathcal{T}'$ is shared by at most two triangles.*

Proof. This is immediate from the previous property as three or more triangles sharing an edge would result in triangles with overlapping areas. □

Property 6.6 *The union of all triangles in $\mathcal{T}'$ does not enclose a bounded region that is not part of $\mathcal{T}'$.*

Proof. Suppose on the contrary that there exists such a bounded region, which is a polygon. Note that any polygon of three or more vertices has an ear. Suppose we have consecutive vertices a, b, c in counterclockwise order forming an ear. Let l_{ab} and l_{bc} be the perpendicular bisectors of a, b and b, c, respectively, and they intersect at v. We consider two cases where v is or is not a Voronoi vertex; refer to Figure 6.11.

First, if v is a Voronoi vertex, then v must be outside $\mathcal{D}(S')$; otherwise, we are done as $\triangle abc$ exists in $\mathcal{T}'$. Figure 6.11 shows the bounded region where d is possibly c, and e is possibly a. Consider the Voronoi diagram $\mathcal{D}' = \mathcal{D}(\{a,b,c,d,e\})$. By the familiar argument as in the proofs of the above properties, it is clear that there cannot be an edge in $\mathcal{T}'$ that crosses from the half-space containing e defined by l_{ab} to that containing d defined by l_{bc}. In other words, the assumption on the existence of a bounded region is incorrect.

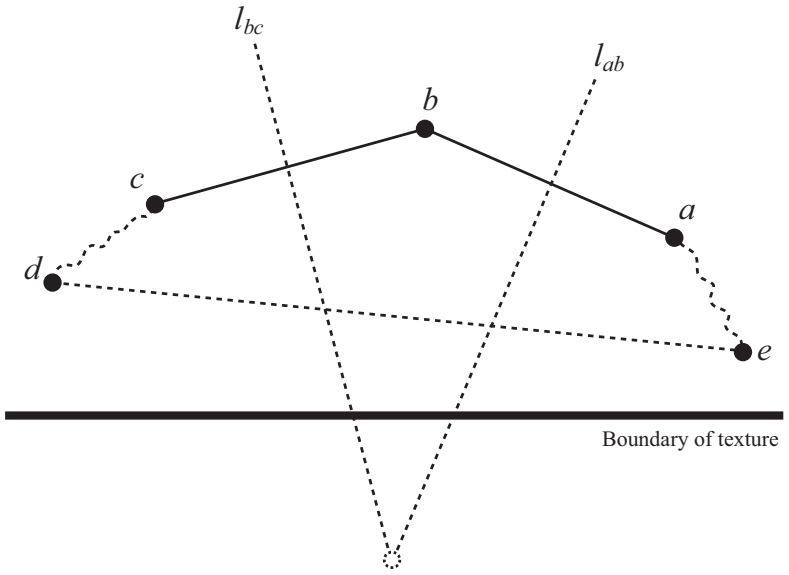


Figure 6.11: Proof of no holes in the triangle mesh.

Second, if v is not a Voronoi vertex, then some points in $\triangle vbc \cap l_{bc}$ (or, in a similar case, $\triangle vab \cap l_{ab}$) belong to Voronoi regions other than that of b and c (or, in a similar case, a and b). One such point v' is on a perpendicular bisector of two vertices of the polygon and is closer to these two vertices than v is to b (and c). We can thus employ an extremal property to find a v that is an intersection of two perpendicular bisectors of vertices of the polygon and v is indeed a Voronoi vertex. With these, we can adapt the argument in the previous paragraph to arrive at a contradiction. □

Property 6.7 *At the end of Step C*1*,* $\mathcal{T}'(S')$ *is a triangulation of* S'*.*

Proof. At the end of Step G6, $\mathcal{T}'$ is a mesh in most cases, with the exception that it may have more than one component (where an isolated site can be a component), or there may be pinch points at some vertices. If $\mathcal{T}'$ is one single component, the algorithm clearly completes $\mathcal{T}'$ to a triangulation. So, suppose it is otherwise.

Then, there are two or more Voronoi regions that spread from one boundary to another boundary as shown in Figure 6.8. In such case, the algorithm going around the boundary of $\mathcal{D}(S')$ can connect in consecutive order one vertex of a component to a vertex of the other component. As such, the algorithm can link up all vertices of S' into a single component in $\mathcal{T}'(S')$. □

Theorem 6.1 *The proposed algorithm computes the Delaunay triangulation of S.*

Proof. From the above properties, we know that up to Step C1, we have a triangulation of S'. Step C2 shifts the set of sites so that it is now a subset of S while maintaining the mesh as a triangulation. Then, Step C3 includes all points of S into the triangulation constructed so far. Finally, Step C4 converts our triangulation to a Delaunay triangulation through edge-flip, and we are done. □

6.5 Experimental Results

We have tested our algorithm on an Intel Core2 Duo 1.86GHz PC with 2GB DDR2 RAM and an NVIDIA GeForce 8800 GTX PCI-X with 768MB DDR3 VRAM. Our program is developed using Microsoft Visual C++.NET 2005, and compiled with all optimization options enabled. The GPU programs are written and compiled with NVIDIA Cg 1.5 and NVIDIA CUDA 1.0. For the purpose of comparison, our program outputs Delaunay triangulations in the same data structure as that by the *Triangle* program, and uses the same robustness routine as *Triangle* [She96]. We, however, have not implemented other useful features, such as incorporating constrained edges or augmenting the given sites with new vertices, available in *Triangle*.

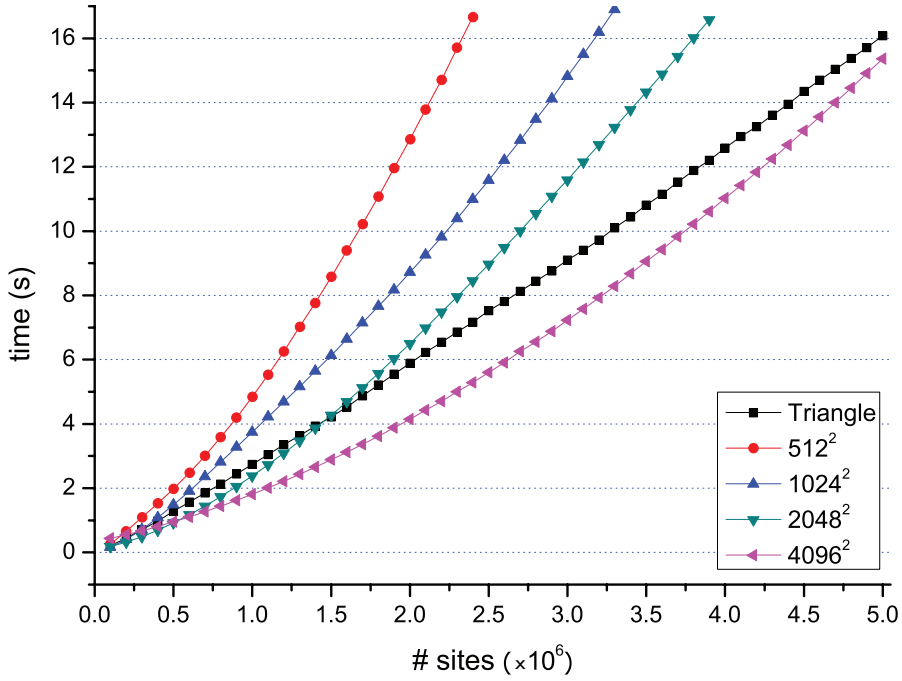


Figure 6.12: Comparison of running time of our algorithm and *Triangle*.

Running Time. On the whole, our algorithm (with still more optimization possible) running on uniformly distributed sites can achieve up to 53% improvement over *Triangle*. Figure 6.12 shows the comparison of the running time of *Triangle* and our algorithm for different texture resolutions. The ratio of the difference between the running time of *Triangle* and our program to the smaller running time of the two gives the percentage of improvement. Positive improvement means ours is faster than *Triangle*; see Figure 6.13.

For a small number of sites, *Triangle* is faster than our program, as our program has certain overhead due to the use of the GPU computation. For a fixed texture resolution (such as 4096×4096 in Figure 6.14), the computational time of the GPU, excluding Step G6, is relatively constant since these steps are almost independent of the total number of Voronoi sites. The timings of Step G6 and the CPU part of

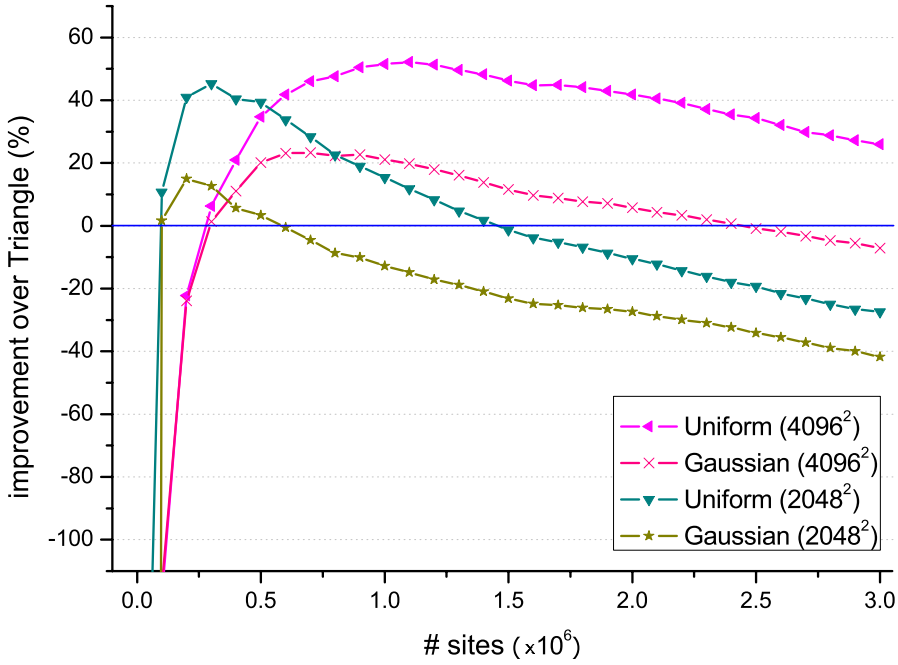


Figure 6.13: Speed improvements over *Triangle*.

our algorithm increase with the increase in the number of Voronoi sites. Figure 6.14 when applied to other texture sizes also looks similar. Currently, those CPU steps, especially C2 to C4, still dominate the computational time. New features in the GPU such as atomic operations may help to eventually realize some of them in the GPU, and thus further improve computational time.

On the other hand, our algorithm does not perform as fast, though remains robust as *Triangle*, when running on an input with non-uniformly distributed sites such as the Gaussian, nearly co-circular, or nearly collinear cases. In the case of the Gaussian distribution, many sites are concentrated near the centroid of these sites. Thus, the algorithm has more missing sites to handle and cannot perform as efficiently as it can in the uniformly distributed case. Figure 6.15 shows that the algorithm performs up to 22% faster for Gaussian distribution of

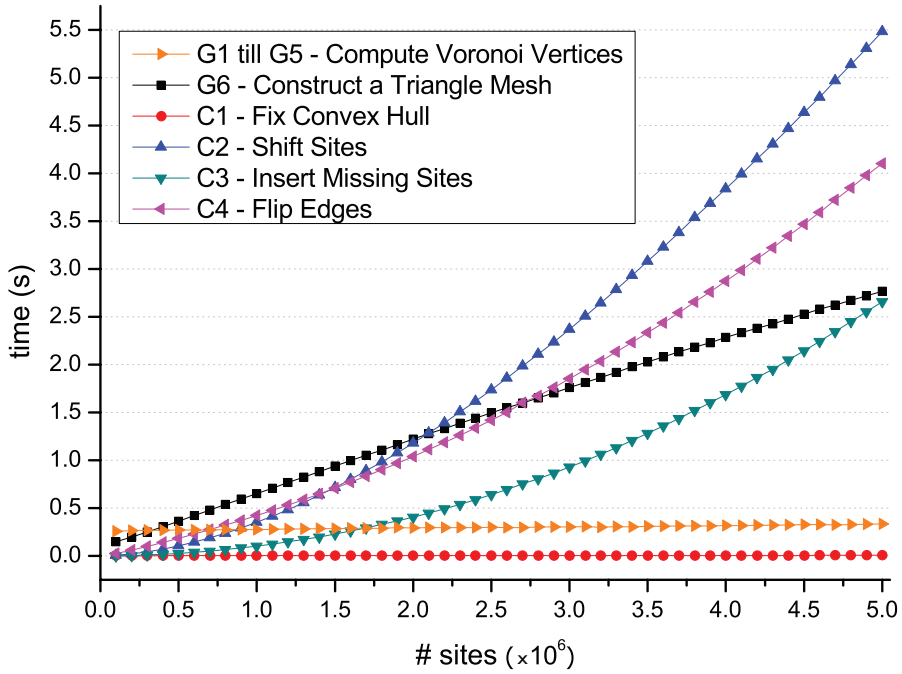


Figure 6.14: Running time of different steps in our algorithm.

sites (as compared to 53% for uniformly distributed cases), using a texture of 4096×4096, but only slightly faster when using a texture of 2048×2048. As for co-circular or nearly collinear cases, there are even more missing sites to handle as the mapping of Step G1 results in most parts of the texture remain empty. As such, our algorithm behaves like the randomized incremental algorithm (but without any form of optimization) in constructing a Delaunay triangulation, thus it runs very slowly when compared to *Triangle* which uses a divide-and-conquer approach.

Texture Resolution. The resolution of the texture is a parameter in our algorithm; it affects the running time of the CPU steps as shown in Figure 6.16. For a larger texture, Step C1 spends slightly more time to traverse along the boundary, while Step C3 has fewer missing sites to handle and Step C4 has fewer edges to flip.

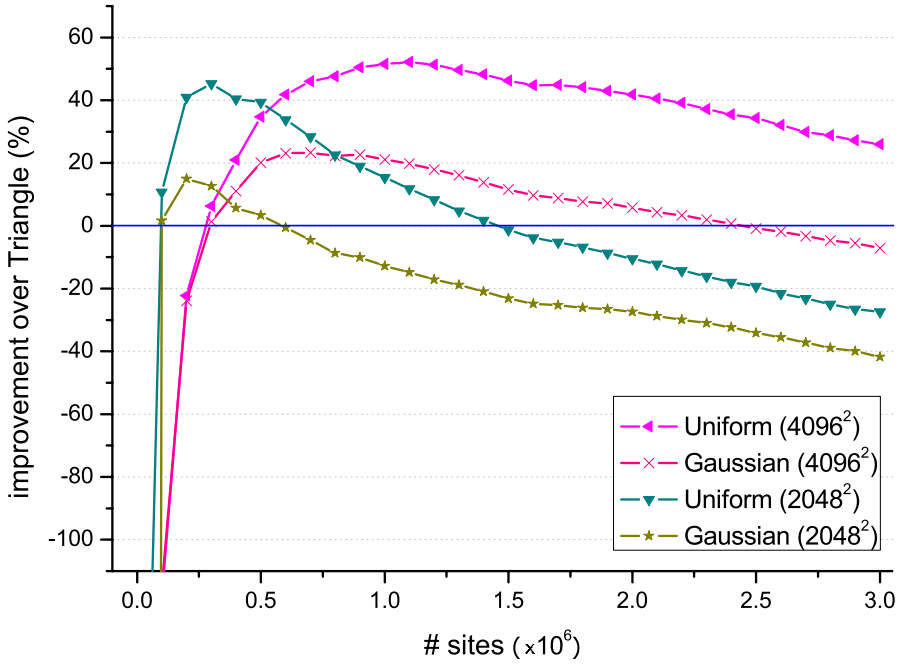


Figure 6.15: Comparison of speed improvements for uniform distribution and Gaussian distribution.

Increasing the texture resolution moves the problem closer to continuous space, and Step G6 thus computes a mesh closer to the required output and Steps C3 and C4 have less to do. On the other hand, Step C2 does not behave in a monotonic manner. With a larger texture, more sites are shifted straightforwardly as shown in Figure 6.9(a), but there are also more sites to be handled due to less missing sites. As such, the time needed for Step C2 can increase with the increase in the texture resolution.

Nevertheless, the total CPU time decreases with an increase in texture resolution. For our hardware configuration and as shown in Figure 6.13, we should use resolution of 4096×4096 for more than 600K sites, but smaller resolution otherwise. Examining carefully Figure 6.13 and also Figure 6.15, we notice the crossing pattern among the four curves of different texture sizes: as the number of sites

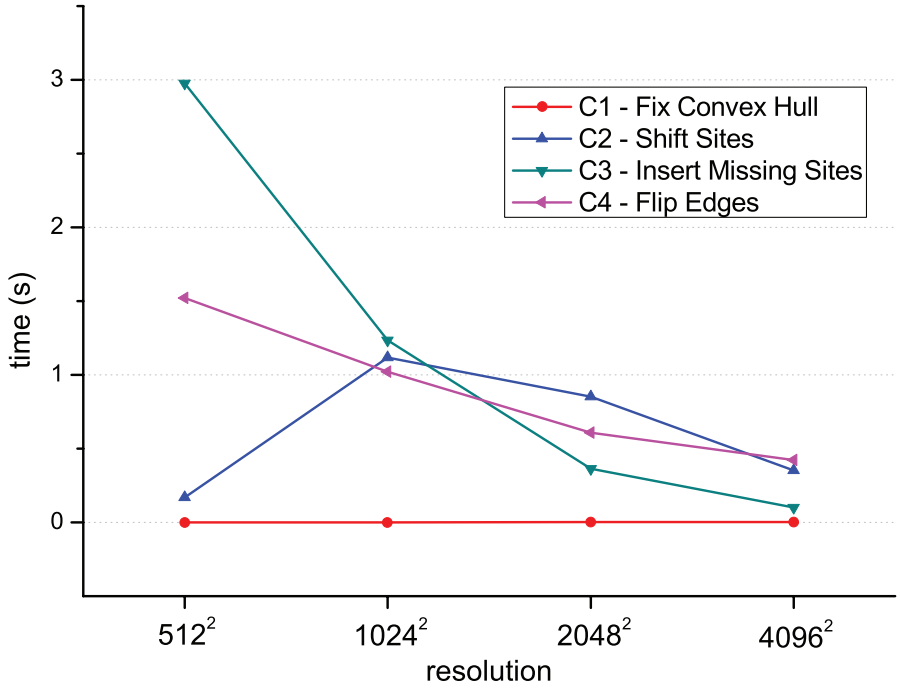


Figure 6.16: Timings on one million sites using different texture resolutions.

increases, the algorithm should gradually switch to a larger texture size to remain efficient when compared to a pure CPU algorithm. This means that, assuming there is no limit on the texture size (and memory), our algorithm can possibly run well for very large data sets. More experiments should be conducted to ascertain this when larger texture sizes become available in the future.

Memory Usage. The CPU memory needed for *Triangle* is about $80n$ bytes where n is the number of input sites. For our CPU computation, we need, besides the above amount, about $48n$ bytes more to keep track of vertices for shifting and a list of references to triangles incident to each site. Such additional CPU memory is used in linking up adjacent triangles sharing a site. Also, we need 384MB each in the CPU and GPU for the 4096×4096 textures and pixel buffer objects. Specific to just the GPU, we need about $72n$ bytes more to keep information on the triangles

generated. With all these on our GPU with 768MB of memory, our program can run on slightly more than five million sites. It is possible to reduce the amount of memory usage, and we are continuing to optimize our current implementation. On a side note, Step G6 requires the copying of an OpenGL texture to a pixel buffer object for CUDA to use. This results in a bigger memory footprint. Should it be possible to read OpenGL textures directly with CUDA, our program would run with a smaller memory footprint.

6.6 Concluding Remarks

In this chapter, a new algorithm is proposed to compute the Delaunay triangulation in continuous space using the GPU and the CPU together. Although previous researches use the GPU to compute discrete Voronoi diagrams, it is not a trivial task to derive a Delaunay triangulation in continuous space from a discrete Voronoi diagram. This chapter addresses all the problems and successfully implements an algorithm that can run faster than the best Delaunay triangulation program for a large number of uniformly distributed sites. We have also proven the correctness of our algorithm. This algorithm demonstrates a new direction in using the GPU (with the help of the CPU) to solve geometry problems in continuous space, and it serves to challenge investigations of such an approach on other geometry problems.

Although the speedup of the proposed algorithm is yet to be as exciting as that of the traditional parallel computation, where there can even be superlinear speedup [KKv05, SAAL94], our approach here remains fundamentally sequential in that it runs on a single (inexpensive) machine. Thus, it may not be appropriate to compare our algorithm to the traditional parallel ones. On the other hand,

there remains possible work to further improve our implementation. For example, one can consider the possibility of utilizing atomic operations in CUDA to better realize some parts that are currently performed by the CPU.

Chapter 7

Conclusion

This thesis proposed the jump flooding algorithm (JFA) as a new paradigm for the general-purpose computation on the GPU. JFA can propagate the information of certain initial pixels to the others very fast. By starting with a big step length and halving the step length in every subsequent pass, JFA achieves a speed exponentially faster than that of the standard flooding algorithm. Furthermore, the speed of JFA is approximately independent to the input size. It is mainly dependent on the resolution of the texture it works on. Although in many cases, JFA can only generate an approximation of the exact result, according to our analysis and experiments, the error rate is usually very low, and thus fits the requirements of most of the applications.

We have applied JFA on the computation of Voronoi diagrams and distance transforms in discrete space. The experimental results show that JFA runs much faster than the previous algorithms of Voronoi diagrams and distance transforms, especially for large input sizes. Its speed is independent to the input size. This is expected because JFA runs on the GPU and does communication on all the

fragments in parallel. Because it does not process every input site separately, its speed is only dependent on the resolution of the texture used in JFA. With the increase of the input size, the speed of JFA remains almost constant. On the other hand, the speed of Hoff et al.'s algorithm [HCK+99] is linear to the input size. So the speed of their algorithm decreases rapidly with the increase of the input size.

Although the error rate of JFA is very low, we have provided some variants of it to further decrease the error rate. Such variants includes JFA+1, JFA+2, JFA^2, JFA2Seeds and 1+JFA. Among them, 1+JFA is the optimal choice because it generates less errors than JFA+1 and run much faster than all the others (same speed as that of JFA+1). We have also provided a variant which halves the resolution of the texture. This variant can greatly increase the speed of JFA. To compute the Voronoi diagram in 3D space, we have used the CPU to simulate JFA in 3D space, and have provided a variant to compute the Voronoi diagram in 3D in a slice-by-slice manner. Our experiments shows the combination of this variant with 1+JFA gives almost the same error rate as that of real 3D JFA simulated by the CPU. So such a combination is a good way to approximate JFA in 3D space without the requirement of the feature of writing into 3D texture, which most of current GPUs does not support.

JFA is also applied on the computation of real-time soft shadows. We introduced two novel algorithms based on JFA to compute soft shadows. Both of them are purely image-based algorithms. JFA-L performs the JFA in the light space to propagate the information of occluders to generate a penumbra map directly from the shadow map, and then uses this penumbra map, together with the shadow map, to generate the final result of soft shadows. JFA-E performs the JFA in the eye space to propagate the information of occluders, and generates the soft shadows

from the result with hard shadows. The speeds of both algorithms are approximately independent to the complexity of the scene, and thus are very suitable to real-time applications.

Based on the discrete Voronoi diagram generated by JFA, we have proposed a new algorithm to compute the Delaunay triangulation in continuous space, using the GPU and the CPU together. We use the GPU to generate a discrete Voronoi diagram, remove the islands in it, find the Voronoi vertices in parallel, chain them up, and build a triangle mesh using CUDA. Then we use the CPU to fix the convex hull, shift the sites back to continuous space, insert the missing sites, and finally flip the edges to guarantee that our result is the Delaunay triangulation. This algorithm is the first attempt to use the GPU to solve a geometry problem in continuous space. The correctness of the algorithm is proven. Although the implementation of the new algorithm is rather rudimental, the speed of the algorithm has already exceeded the fastest 2D Delaunay triangulation program – *Triangle*.

Since JFA has already been successfully applied on different applications, one natural direction of the future work is to find more applications for JFA. Because the JFA mainly deal with 2D textures, it is most possible to find such applications in fields like image processing, visualization, computer vision, etc. For example, the level set method [Set99] which is widely used in visualization area is very similar to the idea of JFA. So we may apply JFA to the problems in this area, such as fluid dynamic. It is possible that we can find many applications for JFA. Most of them may be done only on the CPU today, but, with the help of JFA, we may be able to perform them on the GPU in parallel and thus achieve the real-time speed.

JFA reveals a new pattern of using the communication among pixels. One interesting and important future work is to find and understand more patterns, and

apply them on different applications. As stated in Chapter 4, the information is not necessary to be stored at the original positions. This is a very important property of JFA. However, we have not fully understood the meaning of this property. Besides the analysis of the error rate of JFA in Chapter 4, analysis of the error rates of the variants of JFA have not been investigated. Such analysis may help us to further understand the variants of JFA.

NVIDIA GeForce 8800 has introduced many new features including geometry program, integer texture, etc. Finding a way to use these new features is another interesting future work. The new uniform structure in NVIDIA GeForce 8800 and the newly released CUDA language may help us to further improve JFA and to apply it on more general-purpose applications.

Bibliography

[AAM03] Ulf Assarsson and Tomas Akenine-Möller. A geometry-based soft shadow volume algorithm using graphics hardware. *ACM Transactions on Graphics*, 22(3):511–520, 2003.

[AHT04] Jukka Arvo, Mika Hirvikorpi, and Joonas Tyystjärvi. Approximate soft shadows using image-space flood-fill algorithm. *Computer Graphics Forum*, 23(3):271–280, 2004. (Proceedings of Eurographics 2004).

[AMH02] Tomas Akenine-Möller and Eric Haines. *Real-Time Rendering*. AK Peters, Ltd., second edition, 2002.

[And79] A. M. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, 9(5):216–219, 1979.

[Aur91] Franz Aurenhammer. Voronoi diagrams – a survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3):345–405, 1991.

[AW04] Jukka Arvo and Jan Westerholm. Hardware accelerated soft shadows using penumbra quads. *Journal of WSCG*, 12(1):11–18, 2004.

[BFH+04] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Transactions on Graphics*, 23(3):777–786, 2004.

[Ble90] Guy E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990.

[Bor84] Gunilla Borgefors. Distance transformations in arbitrary dimensions. *Computer Vision, Graphics, and Image Processing*, 27:321–345, 1984.

[BP03] Avi Bleiweiss and Arcot Preetham. Ashli – advanced shading language interface. ACM SIGGRAPH Course Notes, 2003. Available at `http://ati.amd.com/developer/techreports/2003/Bleiweiss-AshliNotes.pdf`.

[BP04] Ian Buck and Tim Purcell. A toolkit for computation on GPUs. In Randima Fernando, editor, *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*, chapter 37, pages 621–636. Addison-Wesley, 2004.

[BS02] Stefan Brabec and Hans-Peter Seidel. Single sample soft shadows using depth maps. In *Proceedings of Graphics Interface*, pages 219–228, 2002.

[CD03] Eric Chan and Frédo Durand. Rendering fake soft shadows with smoothies. In *Proceedings of Eurographics Symposium on Rendering*, pages 208–218, 2003.

[CHH02] Nathan A. Carr, Jesse D. Hall, and John C. Hart. The ray engine. In *Proceedings of the ACM SIGGRAPH/Eurographics conference on Graphics Hardware*, pages 37–46. Eurographics Association, 2002.

[CK06] Nicolas Cuntz and Andreas Kolb. Fast hierarchical 3D distance transforms on the GPU. Technical report, Institute for Vision and Graphics, Universität Siegen, 2006.

[Cro77] Franklin C. Crow. Shadow algorithms for computer graphics. *Computer Graphics (Proceedings of ACM SIGGRAPH 77)*, 11(3):242–248, 1977.

[Cui99] Olivier Cuisenaire. *Distance Transformations: Fast Algorithms and Applications to Medical Image Processing*. PhD thesis, Université catholique de Louvain, 1999.

[Dan80] Per-Erik Danielsson. Euclidean distance mapping. *Computer Graphics and Image Processing*, 14:227–248, 1980.

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

[Déc05] Xavier Décoret. N-buffers for efficient depth map query. *Computer Graphics Forum*, 24(3):393–400, 2005. (Proceedings of Eurographics 2005).

[Den03] Markus Oswald Denny. *Algorithmic geometry via graphics hardware.* PhD thesis, Universität des Saarlandes, 2003.

[Don05] William Donnelly. Per-pixel displacement mapping with distance functions. In Matt Pharr and Randima Fernando, editors, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter 8, pages 123–136. Addison-Wesley, 2005.

[ED06] Elmar Eisemann and Xavier Décoret. Plausible image based soft shadows using occlusion textures. In *Proceedings of the 19th Brazilian Symposium on Computer Graphics and Image Processing, (SIBGRAPI'06)*, Conference Series, pages 155–162, 2006.

[Ede87] Herbert Edelsbrunner. *Algorithms in Combinatorial Geometry.* Springer-Verlag, 1987.

[Egg98] Hinnik Eggers. Two fast Euclidean distance transformations in Z^2 based on sufficient propagation. *Computer Vision and Image Understanding*, 69(1):106–116, 1998.

[ER96] Hugo Enbrechts and Dirk Roose. A parallel Euclidean distance transformation algorithm. *Computer Vision and Image Understanding*, 63(1):15–26, 1996.

[FG06] Ian Fischer and Craig Gotsman. Fast approximation of high order Voronoi diagrams and distance transforms on the GPU. *Journal of Graphics Tools*, 11(4):39–60, 2006.

[FK03] Randima Fernando and Mark J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley, 2003.

[For92] Steven Fortune. Voronoi diagrams and Delaunay triangulations. In *Computing in Euclidean Geometry, Edited by Ding-Zhu Du and Frank Hwang, World Scientific, Lecture Notes Series on Computing – Vol. 1*, pages 163–172. World Scientific, Singapore, 1992.

[GBP06] Gaël Guennebaud, Loïc Barthe, and Mathias Paulin. Real-time soft shadow mapping by backprojection. In *Proceedings of Eurographics Symposium on Rendering*, pages 227–234, 2006.

[GKS92] Leonidas J. Guibas, Donald E. Knuth, and Micha Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica*, 7(4):381–413, 1992.

[GPG07] GPGPU, 2007. `http://www.gpgpu.org`.

[Gra72] Ronald L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 1(4):132–133, 1972.

[Har03] Mark J. Harris. *Real-Time Cloud Simulation and Rendering*. PhD thesis, University of North Carolina at Chapel Hill, 2003.

[HCK+99] Kenneth E. Hoff, Tim Culver, John Keyser, Ming Lin, and Dinesh Manocha. Fast computation of generalized Voronoi diagrams using graphics hardware. In *Proceedings of ACM SIGGRAPH* 99, pages 277–286, New York, 1999. ACM Press / ACM SIGGRAPH. Computer Graphics Proceedings, Annual Conference Series, ACM.

[HLHS03] Jean-Marc Hasenfratz, Marc Lapierre, Nicolas Holzschuch, and François Sillion. A survey of real-time soft shadows algorithms. In *Eurographics*, pages 753–774, 2003. (State of the Art Reports).

[HM94] C. Tony Huang and Owen Robert Mitchell. A Euclidean distance transform using grayscale morphology decomposition. *IEEE Transaction on Pattern Analysis and Machine Intelligence*, 16(4):443–448, 1994.

[Hor05] Daniel Horn. Stream reduction operations for GPGPU applications. In Matt Pharr and Randima Fernando, editors, *GPU Gems* 2*: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter 36, pages 573–589. Addison-Wesley, 2005.

[Int07] Intel. Moore's law. `http://www.intel.com/technology/mooreslaw/index.htm`, 2007.

[Kes06] John Kessenich. The OpenGL shading language (version 1.20.8). `http://www.opengl.org/documentation/glsl/`, September 2006.

[KKC05] Deok-Soo Kim, Donguk Kim, and Youngsong Cho. Euclidean Voronoi diagrams of 3D spheres: Their construction and related

problems from biochemistry. In *IMA Conference on the Mathematics of Surfaces*, volume 3604 of *Lecture Notes in Computer Science*, pages 255–271. Springer-Verlag, 2005.

[KKv05] Josef Kohout, Ivana Kolingerová, and Jiří Žára. Parallel Delaunay triangulation in E^2 and E^3 for computers with shared memory. *Parallel Computing*, 31(5):491–522, 2005.

[KW05] Peter Kipfer and Rüdiger Westermann. Improved GPU sorting. In Matt Pharr and Randima Fernando, editors, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter 46, pages 733–746. Addison-Wesley, 2005.

[LFW06] Philipp Lucas, Nicolas Fritz, and Reinhard Wilhelm. The CGiS compiler – a tool demonstration. In Alan Mycroft and Andreas Zeller, editors, *Proceedings of the 15th International Conference on Compiler Construction*, volume 3923 of *Lecture Notes in Computer Science*, pages 105–108. Springer-Verlag, 2006.

[Llo07] Brandon Lloyd. *Logarithmic Perspective Shadow Maps*. PhD thesis, University of North Carolina at Chapel Hill, 2007.

[MGAK03] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: a system for programming graphics hardware in a C-like language. *ACM Transactions on Graphics*, 22(3):896–907, 2003.

[MIA+04] Patrick S. McCormick, Jeff Inman, James P. Ahrens, Charles Hansen, and Greg Roth. Scout: A hardware-accelerated system for quantita-

tively driven visualization and analysis. In *IEEE Visualization* '04, pages 171–178. IEEE Computer Society, 2004.

[Mic05] Microsoft. HLSL shader reference. `http://msdn.microsoft.com/archive/default.asp?url=/archive/en-us/directx9\_c\_Dec\_2005/dx9\_graphics\_reference\_hlsl.asp`, December 2005.

[Mon68] Ugo Montanari. A method for obtaining skeletons using a quasi-Euclidean distance. *Journal of the ACM*, 15(4):600–624, 1968.

[MT04] Tobias Martin and Tiow-Seng Tan. Anti-aliasing and continuity with trapezoidal shadow maps. In *Proceedings of Eurographics Symposium on Rendering*, pages 153–160, 2004.

[MTP+04] Michael McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan, and Kevin Moule. Shader algebra. *ACM Transactions on Graphics*, 23(3):787–795, 2004.

[Mul92] James C. Mullikin. The vector distance transform in two and three dimensions. *Graphical Models and Image Processing*, 54(6):526–535, 1992.

[NVI07] NVIDIA. NVIDIA CUDA – compute unified device architecture programming guide, January 2007. `http://developer.nvidia.com/cuda`.

[OBSC99] Atsuyui Okabe, Barry Boots, Kokichi Sugihara, and Sung Nok Chiu. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*. John Wiley & Sons, second edition, 1999.

[OLG+07] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[PBMH02] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics*, 21(3):703–712, 2002.

[PDC+03] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 41–50. Eurographics Association, 2003.

[PS85] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

[Rag92a] Ingemar Ragnemalm. Fast erosion and dilation by contour processing and thresholding of distance maps. *Pattern Recognition Letters*, 13(3):161–166, 1992.

[Rag92b] Ingemar Ragnemalm. Neighborhoods for distance transformations using ordered propagation. *CVGIP: Image Understanding*, 56(3):399–409, 1992.

[RP66] Azriel Rosenfeld and John Pfaltz. Sequential operations in digital picture processing. *Journal of the ACM*, 13(4):471–494, 1966.

[RSC87] William T. Reeves, David H. Salesin, and Robert L. Cook. Rendering antialiased shadows with depth maps. *Computer Graphics (Proceedings of ACM SIGGRAPH 87)*, 21(4):283–291, 1987.

[RT06a] Guodong Rong and Tiow-Seng Tan. Jump flooding in GPU with applications to Voronoi diagram and distance transform. In *Proceedings of the Symposium on Interactive 3D Graphics and Games*, pages 109–116. ACM Press, 2006.

[RT06b] Guodong Rong and Tiow-Seng Tan. Utilizing jump flooding in image-based soft shadows. In *ACM Symposium on Virtual Reality Software and Technology*, pages 173–180, 2006.

[RT07] Guodong Rong and Tiow-Seng Tan. Variants of jump flooding algorithm for computing discrete Voronoi diagrams. In *Proceedings of the 4th International Symposium on Voronoi Diagrams in Science and Engineering (ISVD'07)*, pages 176–181, 2007.

[RTCS08] Guodong Rong, Tiow-Seng Tan, Thanh-Tung Cao, and Stephanus. Computing two-dimensional Delaunay triangulation using graphics hardware. In *Proceedings of the Symposium on Interactive 3D Graphics and Games*, 2008. (to appear).

[SAAL94] James Sienicki, Prathima Agrawal, Vishwani D. Agrawal, and Michael L.Bushnell. Superlinear speedup in multiprocessing environment. In *Proceedings of the First International Workshop on Parallel Processing*, pages 261–265, 1994.

[SD95] Peter Su and Robert L. Scot Drysdale. A comparison of sequential Delaunay triangulation algorithms. In *Symposium on Computational Geometry*, pages 61–70, 1995.

[SD02] Marc Stamminger and George Drettakis. Perspective shadow maps. *ACM Transactions on Graphics*, 21(3):557–562, 2002.

[Set99] James A. Sethian. *Level Set Methods and Fast Marching Methods: Evolving Interfaces in Computational Geometry, Fluid Mechanics, Computer Vision, and Materials Science.* Number 3 in Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, second edition, 1999.

[SGGM06] Avneesh Sud, Naga Govindaraju, Russell Gayle, and Dinesh Manocha. Interactive 3D distance field computation using linear factorization. In *Proceedings of ACM Symposium on Interactive 3D Graphics and Games*, pages 117–124, 2006.

[She96] Jonathan Richard Shewchuk. Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator. In Ming C. Lin and Dinesh Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag, 1996.

[SM92] Frank Yeong-Chyang Shih and Owen Robert Mitchell. A mathematical morphology approach to Euclidean distance transformation. *IEEE Transaction on Image Processing*, 1(2):197–204, 1992.

[SOM04] Avneesh Sud, Miguel A. Otaduy, and Dinesh Manocha. DiFi: Fast 3D distance field computation using graphics hardware. *Computer Graphics Forum*, 23(3):557–566, 2004. (Proceedings of Eurographics 2004).

[SPG03] Christian Sigg, Ronald Peikert, and Markus Gross. Signed distance transform using graphics hardware. In *Proceedings of IEEE Visualization*, pages 83–90, 2003.

[ST04] Robert Strzodka and Alexandru Telea. Generalized distance transforms and skeletons in graphics hardware. In *Proceedings of EG/IEEE TCVG Symposium on Visualization (VisSym '04)*, pages 221–230, 2004.

[TPO06] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: using data parallelism to program GPUs for general-purpose uses. In *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 325–335. ACM Press, 2006.

[Ura05] Yury Uralsky. Efficient soft-edged shadows using pixel shader branching. In Matt Pharr and Randima Fernando, editors, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter 17, pages 269–282. Addison-Wesley, 2005.

[VdB05] Michal Valient and Willem H. de Boer. Fractional-disk soft shadows. In Wolfgang Engel, editor, *ShaderX^3: Advanced Rendering with Di-*

rectX and OpenGL, chapter 5.2, pages 411–424. Charles River Media, Inc., 2005.

[Wan92] Leonard Wanger. The effect of shadow quality on the perception of spatial relationships in computer generated imagery. In *Proceedings of the Symposium on Interactive* 3*D Graphics*, pages 39–42. ACM Press, 1992.

[WH03] Chris Wyman and Charles Hansen. Penumbra maps: approximate soft shadows in real-time. In *Proceedings of Eurographics Symposium on Rendering*, pages 202–207, 2003.

[Wil78] Lance Williams. Casting curved shadows on curved surfaces. *Computer Graphics (Proceedings of ACM SIGGRAPH* 78*)*, 12(3):270–274, 1978.

[WM94] Yulan Wang and Steven Molnar. Second-depth shadow mapping. Technical Report TR94-019, University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, 1994.

[WPF90] Andrew Woo, Pierre Poulin, and Alain Fournier. A survey of shadow algorithms. *IEEE Computer Graphics and Applications*, 10(6):13–32, 1990.

[WSP04] Michael Wimmer, Daniel Scherzer, and Werner Purgathofer. Light space perspective shadow maps. In *Proceedings of Eurographics Symposium on Rendering*, pages 143–151, 2004.

[WWT+03] Lifeng Wang, Xi Wang, Xin Tong, Stephen Lin, Shimin Hu, Baining Guo, and Heung-Yeung Shum. View-dependent displacement mapping. *ACM Transactions on Graphics*, 22(3):334–339, 2003.

[Yam84] Hiromitsu Yamada. Complete Euclidean distance transformation by parallel operation. In *Proceedings of 7th International Conference on Pattern Recognition (ICPR)*, pages 69–71, 1984.

[Yam04] Osami Yamamoto. Fast computation of 3-dimensional convex hulls using graphics hardware. In *Proceedings of International Symposium on Voronoi Diagrams in Science and Engineering*, pages 179–190, 2004.