Technical Report

Computing 2D Alpha Shapes Using GPU

By

Srinivasan Kidambi Sridharan

Ashwin Nanjappa

Assoc. Prof. Tan Tiow Seng

Department of Computer Science

School of Computing

National University of Singapore

2011/2012

Technical Report

Computing 2D Alpha Shapes Using GPU

Department of Computer Science

School of Computing

National University of Singapore

Project ID: H0661030

Project Supervisor: Dr. Tan Tiow Seng

Project Title: 2D Alpha Shapes in GPU

Deliverables:

Report: 1 Volume

Program: 1 Diskette

Abstract

This report presents an approach to compute Alpha Shapes for a 2D un-weighted point set using the graphics processing unit (GPU). The problem of alpha shapes has been well-defined and algorithms have been developed to compute it efficiently in 2D and 3D using CPU. However, the nature of this problem makes it well-suited for solving it in parallel and hence, can gain potential speedup over sequential implementations. The fine-grained parallelism offered by the GPU can be harnessed for this purpose. Our implementation using the CUDA programming model on NVidia GPUs is numerically robust and runs faster than existing CPU algorithms.

Subject Descriptors:

I.3.1 Hardware Architecture - Graphics processors, parallel processing

I.3.5 Computational Geometry and Object Modelling - Geometric Algorithm

Keywords:

CUDA, GPGPU, Alpha Shapes

Implementation Software and Hardware:

Microsoft Windows 7, Microsoft Visual Studio 2008, CUDA 4.0

Intel® Core™ i7 CPU 2600K 3.40 GHz 3.70 GHz 16GB RAM NVIDIA GeForce GTX 580

Table of Contents

List of Figures

Chapter 1

Introduction

Many areas like engineering, molecular biology modelling require accurate shape representation of a set of points in space. The concept of α-shapes gives us a notion of the "shape" of a finite point set in space. Assume we are given a set S of points in 2D and we are interested to have something like "the shape formed by these points". This is quite a vague notion and there are probably many possible interpretations, the α-shape being one of them. Alpha shapes can be used for shape reconstruction from a dense unorganized set of data points. Indeed, an α-shape is demarcated by a frontier, which is a linear approximation of the original shape. Every shape in the α-shape family is a set, uniquely determined by the points, their weights and parameter α, that controls the desired level of detail. It is computed from Delaunay triangulation of the point set.

Conceptually, α-shapes are a generalization of convex hull of a point set S in 2D. They constitute a whole family of shapes, ranging from original point set up to the convex hull, parameterized by a real value α between 0 and∞. As α decreases, the shape shrinks by

gradually developing cavities. For α = 0, the shape consists of just the points. Intuitively, a piece of the Delaunay triangulation disappears (as α is decreasing) when α becomes small enough so that a circle with radius α, or several such circles can occupy its space without enclosing any of the points of S.

Alpha shapes are useful in a lot of areas like modelling molecular structures in biology, reconstructing a surface from scattered point set data, etc. Hence, a lot of work has been done in computing α-shapes in R^2 space and querying them efficiently. Three-dimensional alpha shapes are particularly close to real-life applications. The problem of modelling and studying molecules in computational biology is specifically related to shapes: (weighted) α-shapes are in strict geometric sense dual to the space filling diagrams. These diagrams are union of balls (different atoms are represented by balls of different size) with radii as $(w+\alpha)^2$. α-shapes can be used to compute their topological and metric properties. α-shapes are also used in automatic mesh generation and geometric modelling. All these make the design of robust algorithms and programs for construction of α-shapes important and challenging.

The objective of this project is to compute α-shapes for un-weighted points in R^2 in parallel domain (GPU) to gain a speedup over existing implementations. We refer to our GPU based solution as GAlpha in this report.

1.1 Related Work

α-Shapes have been well studied in R^2, and a good amount of work has been done for α-shapes in R^3. A mathematically rigorous definition of a shape of point set was introduced by Edelsbrunner [9]. For R^3, Edelsbrunner [1] defined alpha shapes for points with no weight as a set of *k*-dimensional simplices that have an empty α-ball. They also present an algorithm in the paper to compute alpha shape simplexes, the intervals of α in which these simplexes are part of the α-shape and the classification of these simplexes as interior, regular and

singular with the above definition. Later in 1995, in [2], alpha shapes is uniformly defined for both weighted and unweighted point sets from its relationship with (weighted) Delaunay complex and weighted Voronoi diagram. When the weights of all points are zero, it corresponds to the unweighted case. An alpha shape is closely related to the Delaunay Triangulation of a point set. Since, our focus is on solving alpha shapes in parallel in GPU, we use a GPU based Delaunay Triangulation GPU-DT [6], from Graphics, Games and Geometry (G^3) lab at National University of Singapore.

1.2 Contributions

We have implemented an algorithm to compute alpha shapes for a 2D un-weighted point set using a combined solution in GPU and CPU. Our implementation is aimed to gain potential speedup over sequential implementations and yet, maintain numeric precision during calculations. Birth time of a simplex is the circumradius of the smallest circumcircle bounding a simplex. The biggest challenge in computing alpha shapes is to order the simplices by their birth times precisely. In CPU, this is done by computing the values as long-integer rationals represented by a long integer numerator and denominator. This is difficult to implement in GPU, as we cannot have arbitrarily growing data structures in GPU. We have to pre-determine the worst case size of the long integer result for this computation before we compute them, so that we can allocate that much memory for this computation in GPU. In addition to this difficulty, it is also not efficient to allocate the same worst case size of memory for every simplex as most of the simplices do not need them (only a small fraction of the simplices need large size to represent their birth times precisely). To solve this problem, we use Shewchuk's method of floating point arithmetic [5] with in our comparison of birth times of the simplices in GPU. The main contribution of this project is a parallel solution to compute 2D Alpha Shapes precisely to leverage the power of GPU for this problem and use CPU minimally for certain worst case scenarios.

Chapter 2

Two Dimensional Alpha Shapes

In this chapter, we will define 2D Alpha Shapes and certain related concepts needed for our implementation. Alpha shapes are a generalization of convex hull of a point set, as mentioned in Mucke's paper [3]. As α decreases from ∞, the shape shrinks by gradually developing cavities. For $\alpha = 0$, the shape consists of just the points.

2.1 Basic Definitions

This section lists some of the basic terminology and definitions used throughout this project.

Point sets

A finite point set S in 2-dimensional Euclidean space R^2 is denoted as $S \in \binom{R^2}{n}$ where $n = |S|$ is the size of the set. A point x is an affine combination of a point set $T = \{p_1, p_2 \ldots p_k\}$ if $x = \sum_{i=1}^{k} \lambda_i p_i$ for suitable $\lambda_i \in R$ with $\sum_{i=1}^{k} \lambda_i = 1$. If none of λ_i is negative, then x is called a convex combination of T. The set of all convex combinations of T, is the convex hull of T, denoted as conv(T). Convex Hull of a point set S, is the minimal convex subset of S

containing all points of S. T is said to be *affinely independent* if no point $p_i \in T$ is an affine combination of $T - \{p_i\}$.

Simplices

If a subset $T \subseteq S$ of size $|T| = k+1$, is affinely independent, its convex hull conv(T), is a k-simplex, denoted by σ_T. The dimension of σ_T is $k = |T|-1$. For convenience, we call *0*-simplex a vertex, *1*-simplex an edge, *2*-simplex a triangle.

Simplical complexes

A simplical complex is a collection C of k-simplices, for $0 \leq k \leq 2$, which satisfies the criteria that if a simplex $\sigma_T \in C$, then $\sigma_{T'} \in C$ for every $T' \subseteq T$. In other words, with every simplex σ_T, C contains all the simplices that are part of σ_T as well.

This means that if a simplical complex contains a triangle, then it will contain all the edges of the triangle and all the points of the triangle too.

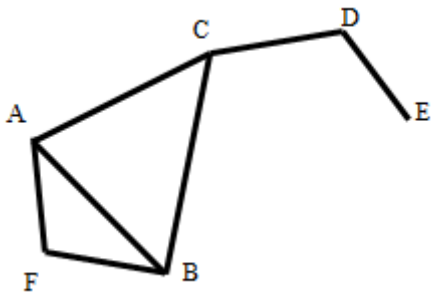


Figure 2.1: A simplical complex

Figure 2.1 gives an example of a simplical complex. In this example, the simplical complex is $C = \{ \Delta ABC, \Delta ABF, AB, BC, CA, BF, FA, CD, DE, A, B, C, D, E, F \}$.

Delaunay Triangulations

A simplical complex *C* is called a triangulation of *S*, if all the vertices of *C* are points of *S*, and the boundary of C is the convex hull of S. If simplices of the triangulation *T* are restricted by the criteria that σ_T is part of the *T* if and only if its circumcircle does not enclose or contain any points of *S*-*T*. It is called the Delaunay triangulation of *S*. We discuss more about Delaunay Triangulation and also define Delaunay complex in section 2.3.

Regular Triangulations

Regular triangulations generalize Delaunay triangulations. It specifies weights for points of *S*. The regular triangulation of weighted point set *S* consists of *k*-simplices σ_T, with *k* ranging from 0 to 3, such that

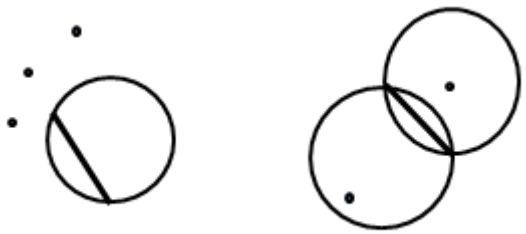
$\exists z \in R^3, w_z \in R$ with $d(z,p)^2 - w_p - w_z = 0$ *for* $p \in T$

and $d(z,p)^2 - w_p - w_z > 0$ *for* $p \in S - T$

where *d(z,p)* denotes the Euclidean distance between points *p* and *z*. Delaunay triangulations are a special case of regular triangulations, just set all the weights to 0.

2.2 Alpha Shapes

Let an α-ball be an open ball of radius α, for all $0 \leq \alpha \leq \infty$. For completeness, a *0*-ball is a point, and an ∞-ball is an open half-space. For some two-dimensional point set *S*, an α-ball b is said to be empty if, $b \cap S = \emptyset$. Any simplex (an edge, vertex or a triangle) is said to be α-exposed if there is an empty α-ball with circumscribing the simplex.



(a) (b)

Figure 2.2: (a) α-exposed edge (b) Not α-exposed edge

A fixed α, thus defines sets of simplices (edges and triangle) that are α-exposed. The α-shape of *S*, denoted by S_α, is the polygon whose boundary consists of edges in this set and their vertices.

2.3 Delaunay Complex

A finite two-dimensional point set *S* defines a special triangulation known as Delaunay triangulation of *S*. Assuming general position of the points, i.e., no 3 points lie in a line, this triangulation is unique and decomposes the convex hull of S into triangles. For 0 <= k <=2, let F_k be the set of k simplices σ_T, where σ_T is the convex hull of points in T, $T \subseteq S$ and $|T| = k+1$, for which there are empty balls b with $\partial b \cap S = T$. Thus, F_0 is the just the point set *S*. The Delaunay triangulation of *S*, denoted by *D*, is the simplical complex defined by the triangles in F_2, edge in F_1, and vertices in F_0. By this definition, for each simplex $\sigma_T \in D$, there exists a value $\alpha \geq 0$ so that σ_T is α exposed. This entails the relationship between the boundary of S_α and Delaunay triangulation that F_k is the union of all k-simplices which are part of the boundary of S_α.

Hence, this relationship lets us conveniently represent family of alpha shapes of *S* implicitly by the Delaunay triangulation of *S*.

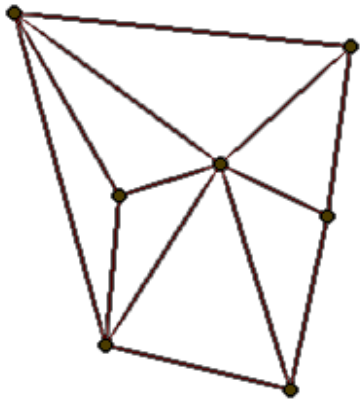


Figure 2.3: Delaunay Triangulation

2.3 Voronoi Diagrams

For every point $p \in S$, let us define $V(p)$, the Voronoi cell of p, as the set of points $x \in R^2$ so that Euclidean distance x and p is less than or equal to the distance between x and any other point of S. Each Voronoi cell is a convex polygon and the collection of Voronoi cells of all point in the point set S is known as the Voronoi diagram of S, denoted by $\mathcal{V}$. Each voronoi cell in 2-dimension, known as a *2-cell* corresponds to a point in Delaunay complex, each intersection of 2 voronoi cells, known as a *1-cell* corresponds to an edge in the Delaunay complex and each intersection of 3 voronoi cells, known as *0-cell* corresponds to a triangle in Delaunay complex. Hence, there is a natural one-to-one correspondence between the k-simplices of S and (2-k)-cells of voronoi diagram. For $T \subseteq S$ and $|T| = k+1$, with $0 \leq k \leq 2$, define $V_T = \cap_{p \in T} V(p)$.

σ_T is a k-simplex of D <=> V_T is a (2-k)-cell of $\mathcal{V}$ for $0 \leq k \leq 2$

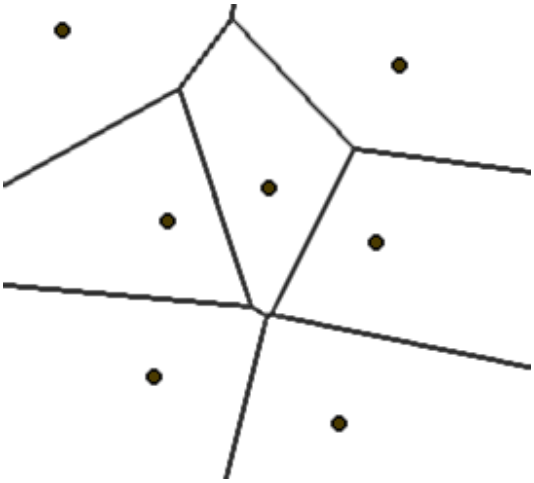


Figure 2.4 Voronoi Diagram

V_T is the set of all points x for which there exist an empty open ball b_x centred at x with $T \subseteq \partial b_x \cap S$. It follows that σ_T is α-exposed if and only if there exists a point x in the relative interior of V_T whose distance to the points in T is α. Since V_T is convex, there exists a single interval so that σ_T is α-exposed if and only if α belongs to this interval. Since all faces of S_α are simplices of D, the interior of S_α are naturally triangulated by the triangles of D. This leads to the concept of α-complexes as defined below.

2.4 Alpha Complexes

Every k-simplex σ_T of D defines an open ball b_T bounded by the smallest circle ∂b_T that contains all points of T. Let ρ_T be the radius of the ball b_T. ∂b_T is the smallest circumcircle of σ_T and ρ_T is the radius of it. For $0 \leq k \leq 2$ and $0 \leq \alpha \leq \infty$, let us define $G_{k,\alpha}$ as the set of k-simplices σ_T belonging to D for which b_T is empty and $\rho_T < \alpha$. $G_{0,\alpha} = S$ for all α. We can now define the α-complex of S, denoted by C_α, as the simplical complex whose k-simplices are either in $G_{k,\alpha}$ or they bound (k+1)-simplices of C_α. By definition,

$C_{\alpha 1}$ is a sub-complex of $C_{\alpha 2}$, if $\alpha_1 < \alpha_2$

The underlying space of C_α, is the union of simplices of C_α. It is a polygon identical to S_α defined in section 2.2.

For all $0 \leq \alpha \leq \infty$, S_α = Union of all simplices in C_α

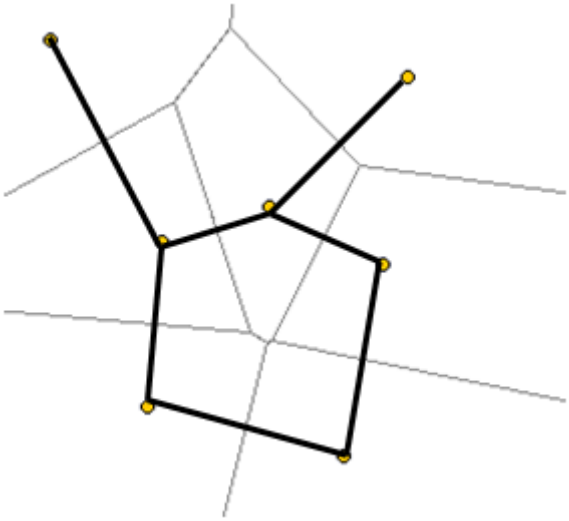


Figure 2.5: Alpha Complex (dark lines) with Voronoi diagram (thin lines behind)

This can be considered as an alternative definition of α-shapes. It makes it easy to specify the intervals of simplices in D during which there exists an empty ball b_T circumscribing the simplex σ_T and hence is '*alive*'. For example, let σ_T be a k-simplex of D. If b_T is empty then σ_T belongs to C_α if and only if $\alpha \in (\rho_T, \infty)$. In case b_T is not empty, and σ_T belongs to C_α if and only if $\alpha \in (\lambda_T, \infty)$ where λ_T is the minimum λ_R over all the $(k+1)$-simplices σ_R of D with $T \subseteq R$.

2.5 Intervals and Face Classification

For each simplex $\sigma_T \in D$ there is a single interval so that σ_T is a face of the α-shape of S_α if and only if α is contained in this interval. It will be convenient to study these intervals for the α-complex C_α rather than the α-shape. We can also break this interval into three (possibly

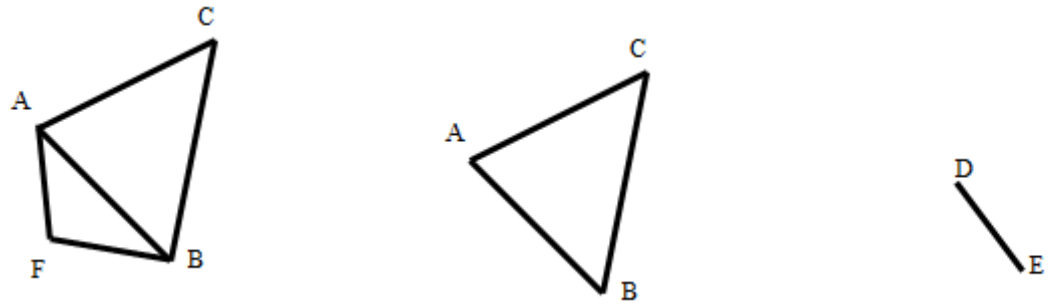
empty) parts that correspond to values of α for which the simplex is an interior, regular or singular simplex of C_α.

A simplex $\sigma_T \in C_\alpha$ is said to be

Interior, if $\sigma_T \notin \partial S_\alpha$

Regular, if $\sigma_T \in \partial S_\alpha$ and it bounds some higher dimensional simplex in C_α, and

Singular, if $\sigma_T \in \partial S_\alpha$ and it does not bound any higher dimensional simplex in C_α



(a) (b) (c)

Figure 2.6: (a) Interior Edge *AB* (b) Regular Edge *AB* bounding triangle ABC (c) Singular edge *DE*

There can be Delaunay edges that can never be singular because their smallest circumcircle encloses other points of *S*. Hence, we can call a simplex $\sigma_T \in D$,

Attached, if $|T|=2$ and $b_T \cap S \neq \phi$, and

Unattached, otherwise

ρ_T is the radius of the smallest circumcircle of the simplex σ_T. In order to break up the interval during which $\sigma_T \in S_\alpha$, let us introduce $\underline{\mu}_T$ and $\overline{\mu}_T$ as the values of α at which the simplex changes from singular to regular and regular to interior respectively. Also, let up(σ_T) be the set of all simplices in *D* that contain the simplex $\sigma_T \in D$, with $|T| < 2$, as a proper face.

$$up(\sigma_T) = \{ \sigma_{T'} \in D \mid T \subset T' \}$$

In 2D, if σ_T is a triangle, then $\underline{\mu}_T$ and $\overline{\mu}_T$ are the same as ρ_T. However, if σ_T is not a triangle, then

$\underline{\mu}_T = \min\{ \rho_{T'} \mid \sigma_{T'} \in \text{up}(\sigma_T), \text{unattached} \}$ and

$\bar{\mu}_T = \max\{ \rho_{T'} \mid \sigma_{T'} \in \text{up}(\sigma_T) \}$

However, it is sufficient to consider only the set

$$\text{up}_1(\sigma_T) = \{ \sigma_{T'} \in \text{up}(\sigma_T) \mid |T'| = |T| + 1 \}$$

that is, all faces incident to σ_T whose dimension is one higher than that of σ_T, in order to derive the values $\underline{\mu}_T$ and $\overline{\mu}_T$:

$\underline{\mu}_T = \min(\{ \rho_{T'} \mid \sigma_{T'} \in \text{up1}(\sigma_T), \text{unattached} \} \cup \{ \underline{\mu}_{T'} \mid \sigma_{T'} \in \text{up1}(\sigma_T), \text{attached} \})$ and

$\bar{\mu}_T = \max\{ \rho_{T'} \mid \sigma_{T'} \in \text{up}(\sigma_T) \}$

Specifying Intervals

The intervals of α for which σ_T is an interior, regular or singular simplex of C_α are specified in the table below. It is also necessary to distinguish simplices that bound the convex hull of S from others.

Consider an edge $\sigma_T \in D$, $T = \{ p_i, p_j \}$, that does not bound the convex hull of S (denoted by $\sigma_T \notin \partial\text{conv}(S)$). Let $\sigma_{T'}$ and $\sigma_{T''}$ be two incident triangles in D and let $T' = T \cup \{p_u\}$ and $T'' = T \cup \{p_v\}$. If $0 < \rho_{T'} < \rho_{T''} < \alpha < \infty$, then $\underline{\mu}_T$ of the edge is $\rho_{T'}$ and $\overline{\mu}_T$ of the edge is $\rho_{T''}$. This edge is not α-exposed as both the triangles are in C_α. It belongs to the interior part of S_α. However, if $\rho_{T'} < \alpha < \rho_{T''}$, then the triangle is α-exposed and $\sigma_{T'}$ is in C_α but $\sigma_{T''}$ is not. This implies that σ_T is a regular edge of C_α. If $\alpha < \rho_{T'} < \rho_{T''}$, then both the triangles are not part of C_α, but still the edge can be part of C_α as a singular edge iff $\rho_T < \alpha$ and neither of p_u or p_v are

inside the ball b_T. If one of the two points is inside b_T, then σ_T is attached and it can never be a singular triangle of C_α, no matter what value of α is selected.

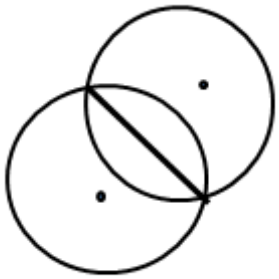


Figure 2.7: Attached edge, can never be singular

The α-complex consists of all interior, regular and singular simplices for a given α value. The interior of the α-shape is triangulated by the interior simplices. The boundary of the interior is formed by the set of regular edges and their vertices.

Simplex σ_T	**Singular**	**Regular**	**Interior**
Triangle			(ρ_T, ∞)
Edge • $\notin \partial$conv(S), Unattached • $\notin \partial$conv(S), Attached • $\in \partial$conv(S), Unattached • $\in \partial$conv(S), Attached	$(\rho_T, \underline{\mu}_T)$ $(\rho_T, \underline{\mu}_T)$	$(\underline{\mu}_T, \overline{\mu}_T)$ $(\underline{\mu}_T, \overline{\mu}_T)$ $(\underline{\mu}_T, \infty)$ $(\underline{\mu}_T, \infty)$	$(\overline{\mu}_T, \infty)$ $(\overline{\mu}_T, \infty)$
Vertex • $\notin \partial$conv(S) • $\in \partial$conv(S)	$(0, \underline{\mu}_T)$ $(0, \underline{\mu}_T)$	$(\underline{\mu}_T, \overline{\mu}_T)$ $(\underline{\mu}_T, \infty)$	$(\overline{\mu}_T, \infty)$

Table 2.1: Intervals of α for which $\sigma_T \in C_\alpha$

The endpoints of the intervals in the table above are referred to as α-thresholds. This does not include 0 and ∞. Since all $\underline{\mu}_T$ and $\overline{\mu}_T$ are ρ values of other simplices, each α-threshold is the radius of a simplex in D. More specifically, the set of α-thresholds is exactly the set of radii of all unattached *k*-simplices for $1 \leq k \leq 2$. α-Spectrum is defined as the sorted sequence of α-thresholds.

Chapter 3

Data Structures

Alpha Shape of a point set requires the connectivity information of all its simplices to represent the shape of point set for a particular α. To meet this requirement, we need a data structure to store the topology of the simplices in the alpha complex. Next, we would like to query the alpha shape by passing a parameter α. We need a convenient way to store the whole family of alpha shapes, so that the alpha shapes can be queried efficiently. In this chapter, we discuss how we can store these information efficiently to meet the above mentioned needs.

The whole family of α-shapes of a given data set *S* can be stored using two data structures. The first data structure represents the connectivity and order among the simplices of the two-dimensional Delaunay triangulation *D* of *S*. The second data structure is used to implement the face classification of the simplices. This is implemented as a collection of intervals in a few linear arrays. Since this report focuses on alpha shapes, we discuss about the triangle-edge data structure very briefly and then move on to discuss about the alpha shape data structure.

Triangle-edge data structure is designed to represent the topology and order of a two-dimensional triangulation. This data structure is triangle based. It stores the indices of the 3 vertices belonging to the triangle and indices of the triangles opposite (or neighbouring) to each of the vertices. In this way, we have information about all the triangles and the connectivity between them. The data structure's atomic unit is the *triangle-edge* pair $a = <f, v>$, $0 \leq v \leq 2$. Sometimes, it will be convenient to represent edges as triangle-edge pairs.

3.1 Alpha Shape Data Structure

The whole family of α-shapes can be represented by simplices of the Delaunay triangulation augmented by a set of intervals. Since it is often convenient to have the interior of a shape triangulated, we represent α-complexes instead of α-shapes. As already discussed, there is a single interval for each simplex $\sigma_T \in D$, such that σ_T belongs to the α-complex C_α if and only if α is contained in this interval.

Face Indexing

The data structure which is used to store *D* is triangle based, hence edges are only represented implicitly. However, some sort of indexing for faces of all dimensions is needed in order to conveniently store the additional information like collection of intervals of each face in linear arrays. In CPU, hashing is used to index edges and tetrahedral. However, it is not possible to use hashing in GPU currently. Hence, we define the following rules to index edges.

An edge '*belongs*' to only one of the triangles that contains it. This rule is enforced to have a unique way of addressing the edge by the triangle index and the index of the vertex opposite to it. The edge *belongs* to that triangle in which the indices of its points occur in increasing order. An edge also belongs to a triangle if it is a convex hull edge, as irrespective of the

order of the indices of its vertices in that triangle, it does not have another triangle to *own* it. In this way, an edge will always belong to only one of the triangles which contain it.

From the above definition, a triangle may contain 1-3 edges, depending on the order of the indices of its vertices. Notice that a triangle will contain 3 edges only if it is a convex hull triangle, otherwise it cannot contain 3 edges (if, say the indices of its points are *a,b,c* such that *a*<*b*<*c*, then the triangle will contain the edge *ab* and *bc*, but cannot contain *ca*).

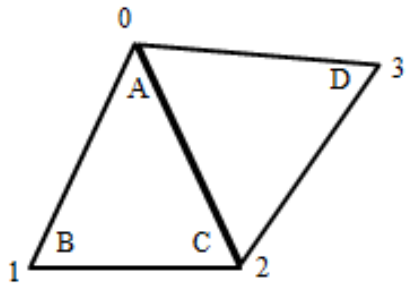


Figure 3.1: Ownership of Edges in a Delaunay Triangulation

In figure 3.1 above, each triangle consists of the indices all its vertices which are pointers to entries in a global *Points* array containing the coordinates of the input points. Side *AC* belongs to triangle *ABC* as the order of indices of its vertices in *ACD* is *0-2* and in triangle *ABC*, the order is *2-0* (decreasing).

We need another linear array *tri_edge_indices* to store the starting index of edges owned by a triangle in the edges list. Now that the rules of ownership have been established, we define w to get the index of an edge from its triangle and vice-versa. The vertices of a triangle are locally numbered 0, 1 and 2. The first edge which is from vertex 0 to vertex 1 is numbered 0, second edge from vertex 1 to vertex 2 is numbered 1 and the third edge from 2 to 0 is numbered as 2.

Given a triangle t_1 and a vertex k, to find the index of the edge opposite to vertex k in triangle t_1, the first step is to find the order of indices of the vertices opposite to vertex k. Let they be i and j respectively. If $i < j$, then this edge belongs to triangle t_1, else it belongs to the triangle opposite to vertex k in triangle t_1, say t_2. If the former is true, find the edge's number in the triangle, call it e (it could be the 1st edge belonging to the triangle, or 2nd or 3rd) and add it to the starting index of edges of this triangle taken from *tri_edge_indices* array. However, if the latter is true, then find the edge's number in triangle t_2 and add it to the starting index of edges in that triangle.

In figure 3.1, triangle ABC *owns* edges *AB* and *CA*, but not *BC*. So the edge number of edge *AB* is 1 and *CA* is 2. We add this to the starting index of edges of ABC from *tri_edge_indices* to get the index to the edges list. Now, we'll look at how to compute *tri_edge_indices*.

Before computing the intervals for all the edges, we pre-compute the number of edges belonging to each triangle by the rules mentioned above. This will give us a count of edges for each triangle. Then performing an exclusive scan on this array will give us the starting indices of edges of each triangle.

Triangle Indices	:	0 1
Number of edges owned by triangle:		2 3
tri_edge_indices	:	0 2

Figure 3.2: *tri_edge_indices* computation for Figure 3.1

Figure 3.2 presents an example on how to compute the *tri_edge_indices* list for each triangle in the Delaunay Triangulation *D*. Now, given a triangle t_1 and vertex k, we can get the edge index as follows:

```
function getEdgeIndex : int t1, int vOpp

    // Get the indices of vertices opposite to vOpp = edge vertices
    i := (vOpp + 1) mod 3
    j := (vOpp + 2) mod 3

    if i < j
        edge_number_t1 := edge number of current edge owned by triangle t1
        Check edges that satisfy i<j || opposite triangle of
        vOpp is -1 (Convex Hull edge) and count till current edge

        edge_index := edge_number_t1 + tri_edge_indices[ t1 ]
    else
        edge_number_t2 := edge number of current edge owned by triangle t2
        Compute edge_number_t2 as described above for edge_number_t1
        edge_index := edge_number_t2 + tri_edge_indices[ t2 ]

    return edge_index

end
```

3.2 Alpha ranks

The set of α-thresholds consists of all end points of the intervals in table 2.1 for all *k*-simplices where $0 \leq k \leq 2$. The sorted sequence of α-thresholds α_1, α_2... α_n forms the *α-spectrum*. For convenience, we always assume $\alpha_1 = 0$ and $\alpha_n = \infty$. If 0 and ∞ do not occur in the set if α-thresholds, these values are added to the spectrum. The threshold α_r is said to have rank r. Each rank stands for an endpoint of a certain value. These ranks are used to store the intervals of the faces of D in three *rank-tables*, one for each dimension. Instead of storing the actual α-threshold, which will be a floating point value (with double precision), just the rank of the α-threshold in the spectrum is stored. For triangles, only the rank of ρ is stored (as a triangle is already interior when it is *born* at ρ), for edges the ranks of ρ, $\underline{\mu}$ and $\overline{\mu}$ are stored and for points, the ranks of $\underline{\mu}$ and $\overline{\mu}$ are stored (as ρ value of un-weighted points are always 0). Certain intervals can be empty though as illustrated in table 2.1. A “void” rank *0* is used to encode these.

- Attached edges have ρ-rank = 0
- Faces (edges) bounding the convex hull have $\mu \neq 0$ and $\overline{\mu} = 0$ as they can never become interior

3.3 Master List

The endpoints of α-intervals of many simplices may map to the same α-threshold in the spectrum. This is because endpoints $\underline{\mu}$ and $\overline{\mu}$ of intervals of points are nothing but the ρ values (radius of smallest circumcircle bounding edges: *birth time*) of edges containing them and the endpoints $\underline{\mu}$ and $\overline{\mu}$ of intervals of edges are ρ values of the triangles containing them. In case of attached edges, the endpoint $\underline{\mu}$ of the intervals of points of these edges will contain ρ value of the triangle containing the edge. Hence, endpoints of intervals of many simplices will be mapped to the same α-threshold in the spectrum.

For many applications of alpha shapes like area and other metric features of α-diagrams or space filling diagrams [10], it is desirable to be able to retrieve, given a rank *r*, all faces whose intervals have α_r as an end point. Hence, we need to store, for each α-threshold α_r, the list of simplices whose intervals have this threshold as an end point. For convenience, these lists are merged in the order of increasing rank to a list of sublists, called the *master list*. Each entry stores the corresponding face index, the face type (edge, triangle or vertex) and the rank type (ρ, $\underline{\mu}$ or $\overline{\mu}$). The simplices in each sublist are sorted in order of non-decreasing dimension (vertex, then edge, then triangle).

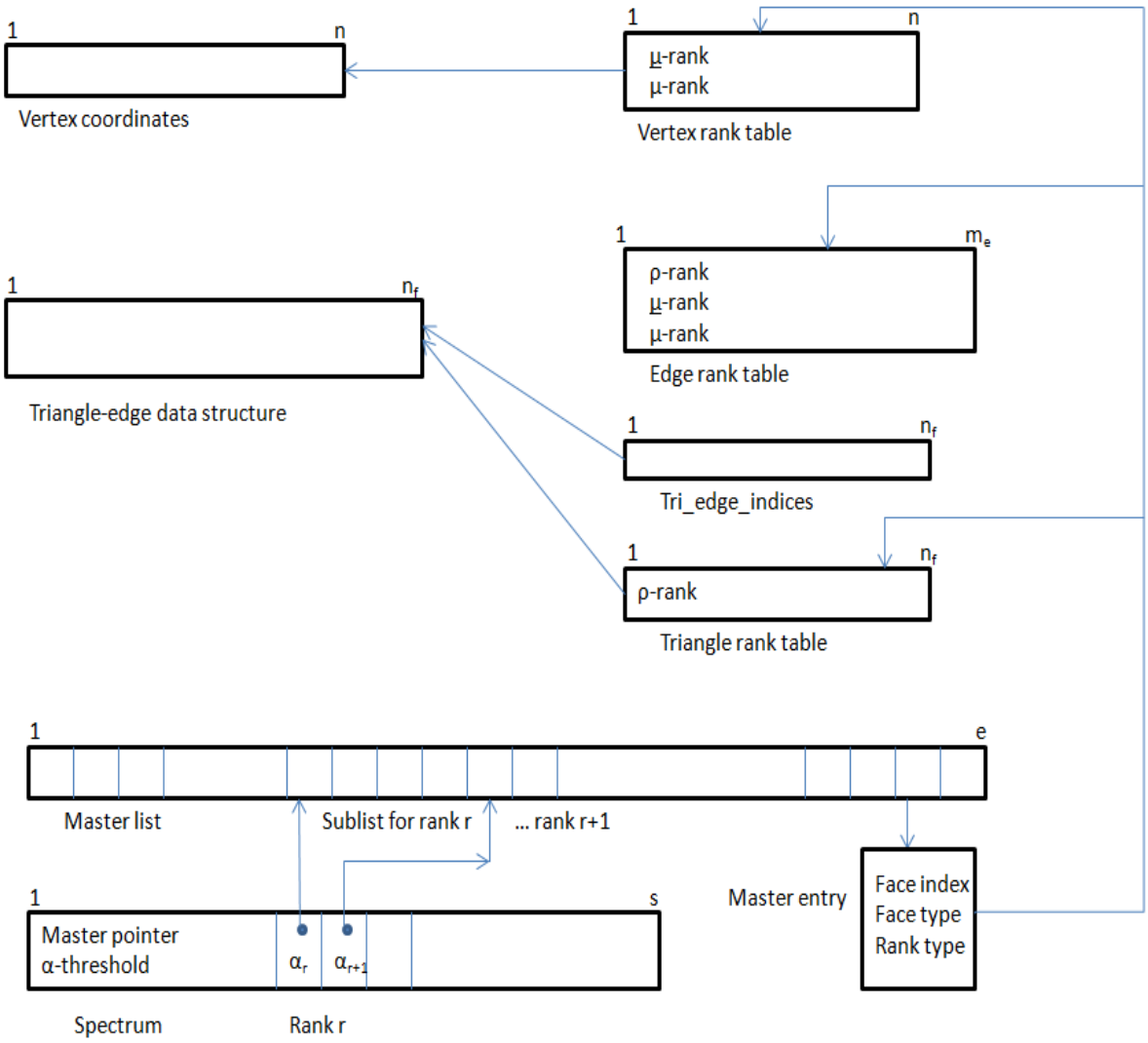


Figure 3.3: Alpha Shape Data Structure

The master list can be implemented as a linear array along with another array containing pointers to the sub lists containing endpoints of all simplices that map to the same α-threshold. We compute the α-thresholds using Shewchuk's floating point computation method, but since the computation may lead to millions of floats, we store only one double precision value for each α-threshold. The sorting of the α-thresholds is performed in 3 different phases using both GPU and CPU. The exact threshold values computed using Shewchuk's floating point computation are used to sort the spectrum consistently. After this, they are discarded and mapped to double precision values. Master list, spectrum and the rank tables represent the family of α-shapes for a point set $S \in R^2$.

Chapter 4

Algorithm

Computing two-dimensional α-shapes involves construction of Delaunay triangulation of the input point set S. This project uses *GPU-DT software* code from [6] to compute Delaunay triangulation of a 2D point set data S, developed by *Graphics, Games and Geometry lab* (G^3 lab) at National University of Singapore (NUS). Hence, we discuss only the algorithm used to compute the additional information which is needed for face classification defined in section 2.5. Then, a list of all geometric predicates required for the alpha shapes computation is discussed. Finally, the methods of precision used to sort the spectrum consistently are presented. We use both CPU and GPU for this sorting as it is difficult to completely satisfy the memory requirements of the computation in GPU for all the CUDA threads.

4.1 Computing Alpha Intervals

The goal is to build the interval structure of figure 3.3, which augments the triangle-edge data structure of Delaunay Triangulation *D*. Together, these two data structures provide a compact and complete representation of the whole family of alpha shapes for a given point set *S*. The difficult steps in constructing the intervals are the computation of the radii of the faces in *D*

and test whether a face is attached or not. Another aspect that has to be considered is the robust implementation of these formulas. Once these problems are solved, then it becomes straight forward to construct the α-shape file and data structure using O(m) space and time, m denoting the number of simplices in the Delaunay triangulation D.

A simplex σ_T can be classified as attached or unattached by checking if any point from any simplex in its $\text{up}_1(\sigma_T)$ is inside the open ball bounded by the smallest circumcircle of σ_T. The time it takes to classify all the simplices is proportional to the number of simplices in D. In other words, a simplex can be classified in constant amortized time.

For any simplex, ρ_T can be computed in constant time. By computing triangles before edges we can get $\underline{\mu}_T$ and $\overline{\mu}_T$ as the minimum and maximum of the values $\rho_{T'}$, $\underline{\mu}_{T'}$ and $\overline{\mu}_{T'}$ for $\sigma_{T'} \in \text{up}_1(\sigma_T)$. This also takes constant amortized time per simplex.

4.2 Algorithm

We describe the high level algorithm of our GPU based parallel solution for 2D alpha shapes in this section.

- Compute Delaunay triangulation of the point set using GPU-DT
- For each triangle and edge, compute the circumradius ρ_T of the smallest circle enclosing it in parallel. This step can be performed very quickly in GPU as the individual computations do not have any dependencies between them. The ρ values computed for the triangles and edges are stored in a linear array, the spectrum as described in section 3.2.
- Sort the spectrum by the increasing order of ρ values. This is a computationally intensive task as we need robust arithmetic methods for a correct ordering of the ρ values, which otherwise would affect the topology of the alpha shape queried.

- Compute the intervals of face classification according to the formulae given in section 2.5 by propagating the ρ values.
- Compute the master list by adding an entry for all the interval thresholds of every simplex in the Delaunay triangulation (edge, triangle and vertex).
- Sort the master list by spectrum rank of each α-threshold.

4.3 Implementation Details

In this section, we discuss the important implementation details of our algorithm. The previous section does not discuss about the problems of implementing certain steps in GPU. We outline them here and discuss how they are solved. The first challenge is representing edges without a hash table based solution, since it is not suited for GPU.

4.3.1 Representing Edges

Implement linear array for edges with indexing of edges done by *tri_edge_indices*. The number of edges *owned* by a triangle can be computed in constant time per triangle, as we just need to check how many of its 3 edges satisfy the either of the following criteria as described in section 3.1.1:

i. Order of vertices of the edges should be in increasing order (or)
ii. Edge should be a convex hull edge

This can be computed in *parallel* for each triangle in an array *tri_edge_count*. Then a parallel prefix scan of *tri_edge_count* would give us *tri_edge_indices*.

4.3.2 Computing ρ

Process each unattached simplex $\sigma_T \in D$ and compute its ρ value *parallely* using the formulas described later in section 4.3. For the current step, we compute these values using inexact

computation as we do not need an exact representation yet and for performance reasons. The computed values are placed in the spectrum in consecutive locations for each edge and then the triangles. The list of these ρ values is not yet sorted. Each simplex (an edge or a triangle) points to its corresponding ρ entry in the spectrum. Now, the most important concern is how do we update the edges and triangles once the spectrum is sorted the ρ values are moved around in the spectrum. To solve this problem, we store a back referencing value for each entry in the spectrum and move it along whenever a ρ value is moved.

4.3.3 Sorting the Spectrum

The aim of this step is to order the ρ values in the spectrum in ascending order so that we know all the simplices in the alpha shape for a given α as simplices before α in the spectrum. In order to maintain the correct topology of the alpha shapes, the sorting needs to be precise and hence is performed in 3 phases as described below:

1.) Phase 1

Sort the inexact values in the spectrum computed in previous step parallely using efficient *CUDA thrust library*. This phase is very fast as it is based on the pre-computed ρ values.

2.) Phase 2

Use Shewchuk's fast robust arithmetic computation method [4] to check if any consecutive pair of ρ values in the spectrum is out of order. This step computes the ρ values using multiple floats to avoid loss of precision during intermediate computations. It requires more memory to construct the ρ values precisely and it is not possible to pre-allocate the required memory for every thread in GPU. Hence, we compute the smallest circumradius precisely only if it requires less than a pre-defined fixed number of floats in GPU, else push it for later checking in CPU in phase 3.

Since the array is roughly sorted after phase 1, we just need to confirm correct local ordering of entries with their neighbours (left and right) to ensure correct global order. To do this efficiently in GPU, we use the Odd-Even Transition sort which is a parallel version of bubble sort. We use this algorithm as it checks consecutive items for correct order to finally reach an overall sorted order. Hence, it suits our need better than other algorithms in GPU.

Odd-Even Transition sort: It compares two entries and switches them if the first is greater than the second, to achieve a left to right ascending ordering. We need two passes in each iteration, wherein the first pass checks odd-even pairs (consecutive pairs of elements with the first element located on an odd index and the second element on an even index) for flipping and the second pass checks even-odd pairs. We need two passes, otherwise two consecutive flips may overlap while performing them parallely which will lead to incorrect result.

Analysis: Odd-Even transition sort requires a maximum of $n/2$ iterations of dual phase sort as shown above. But in our scenario, all similar items will be together as the array has already been roughly sorted (ambiguities only exist between very close elements), hence we will mostly need only a few iterations to get a completely sorted order and since this algorithm can detect if can array is sorted like bubble sort, we will quite right then.

3.) Phase 3

After phase 2, we have all the out-of-order or possibly equal simplices pushed for checking in CPU. We perform complete robust arithmetic in CPU using Shewchuk's floating point arithmetic to insert every one of the these values in their appropriate position using insertion sort, as it is best suited for almost sorted arrays. In CPU we can satisfy the worst case memory requirements of this computation and the overall

worst case time complexity of this phase is O(*km*) where *k* is the number of out-of-order simplices and *m* is the number of entries in spectrum. But, from our experiments we find that we rarely need the worst case memory requirements for constructing the ρ values precisely for simplices in 2D.

4.3.4 Computing Alpha Intervals

Compute $\underline{\mu}$ and $\overline{\mu}$ ranks for edges and points based on the ρ ranks. It takes amortized constant time to compute them for each simplex as the overall time required for updating all the simplices is proportional to the number of simplices in D. This step can be highly parallelized too by computing the edge intervals for each edge parallely as they are not dependent on each other and then each vertex parallely. For each edge, check the two triangles that contain the edge to compute $\underline{\mu}$ and $\overline{\mu}$ ranks as the minimum and maximum of the ρ values of the triangles.

For vertices, we pre-compute a point to triangle map by updating all the vertices belonging to a triangle in parallel. Then we walk along the fan of edges containing the vertex to find the minimum and maximum of the ρ values of the edges containing this vertex. This operation can also be performed in parallel for each vertex.

4.3.5 Computing Master List

The Master list, which consists of the $\underline{\mu}$ and $\overline{\mu}$ ranks of points, ρ, $\underline{\mu}$ and $\overline{\mu}$ ranks of edges and ρ ranks of triangles in the Delaunay triangulation is computed. Then the master list entries are sorted with respect to the ρ,$\underline{\mu}$, $\overline{\mu}$ ranks which are integers. This sorting is also performed parallely using the fast and efficient *CUDA thrust library*.

4.4 Geometric Primitives

The set of primitives needed to compute α-shapes can be separated into two groups: one for constructing the Delaunay triangulations, and one for computing intervals for face classifications. Constructing two-dimensional Delaunay triangulations requires 2D orientation tests which are not discussed here as the focus of this project is building Alpha shapes on top of GPU based solution for 2D Delaunay triangulation from G^3 lab at NUS. This section discusses the geometric predicates and arithmetic tests required by 2D Alpha Shapes.

4.4.1 Radius of smallest circumsphere

We are interested in the radius ρ_T of the ball b_T, the smallest circumcircle of the simplex σ_T, for all *k*-simplices $\sigma_T \in D$, with $1 \leq k \leq 2$. However, since ρ_T is a square root term, it cannot be computed precisely, hence we compute ρ_T^2 for triangles and edges using the formulae given below:

$$T = \{p_i, p_j\}: \rho_T^2 = \frac{(x_i - x_j)^2 + (y_i - y_j)^2}{4}$$

$$T = \{p_i, p_j, p_k\}: \rho_T^2 = \frac{|P_iP_j|.|P_jP_k|.|P_kP_i|}{4|P_iP_jP_k|}$$

$$\text{Where } |P_iP_jP_k|^2 = \frac{1}{4}.\begin{vmatrix} x_i & y_i & 1 \\ x_j & y_j & 1 \\ x_k & y_k & 1 \end{vmatrix}^2$$

In the formulas, the following notation is used. Let *a*, *b*, *c* be points in R^2. The notation |*ab*| denotes the length of the edge *ab*, and |*abc*| denotes the area of the triangle *a*, *b*, *c*. It is straightforward to see that the length of the edge is $|ab|^2 = (x_a - x_b)^2 + (y_a - y_b)^2$ and the radius of the smallest circumcircle is ½ the length of the edge.

The formula for the circumradius of triangle mentioned above is a standard formula. Each of the 3 terms in the numerator is length of the 3 sides and denominator is 4 times the area of the triangle.

4.4.2 Attached and Unattached edges

We need to solve the problem of deciding whether an edge $\sigma_T \in D$ is attached or not. By definition, a simplex σ_T is attached if there is a higher dimensional simplex $\sigma_R \in \text{up}_1(\sigma_T)$ such that the point in $R - T$ is inside the ball b_T. If σ_T is an edge, say $T = \{p_i, p_j\}$ belonging to a triangle $R = \{p_i, p_j, p_k\}$ and $R - T = \{ p_k \}$, then this can be checked by comparing ρ_T with distance between p_k and $\frac{pi+pj}{2}$. Straight forward algebraic manipulations lead to the following inequality:

$$\left[(x_i - x_j)^2 + (y_i - y_j)^2\right] - \left[(x_i + x_j - 2x_k)^2 + (y_i + y_k - 2y_k)^2\right] > 0$$

4.5 Optimizations

We employed certain optimization techniques in GPU to gain potential speedup. The first technique we applied was to set preference to L1 cache over shared memory since we are not using shared memory. Also, branching instructions inside a CUDA kernel which lead different workloads for different threads in a warp slow down the kernel. To avoid this, we align all threads performing similar comparisons like all edge-edge comparisons together, all edge-triangle together and so on. These two optimizations provide together a speedup of 20% of the whole program time. Other optimizations that we performed include loop unrolling of our exact construction of the circumradii and performing the parallel odd-even transition sort for only that portion of the spectrum where there are flips.

Chapter 5

Precision in GPU

Software libraries for arbitrary precision floating-point arithmetic can be used to accurately perform many error-prone computations that would be infeasible using only hardware-supported approximate arithmetic. Alternatively, an exact arithmetic library can be used and will yield correct result, but it is very slow. Also, GPU cannot support arbitrarily growing data structures like in exact arithmetic libraries. In this project, we use Shewchuk's method of fast extended precision calculations for geometric predicates and arithmetic tests.

5.1 Background

Floating point numbers are stored in the form $\pm significand \times 2^{exponent}$. The significand is a p-bit number of the form $b.bbb...$, where each b is a single bit. One additional bit represents the sign. Floating-point values are generally normalized, which means that if a value is not zero, then its most significant bit is set to 1 and the exponent is adjusted accordingly. Exact arithmetic often produces values that require more than p bits to store. For algorithms used in this project, each arbitrary precision value is expressed as an expansion $x = x_n + ... + x_2 + x_1$,

where each x_i is called a component of *x* and is represented by a floating point value with a *p*-bit significand. To impose some structure on expansions, they are required to be *nonoverlapping* and ordered by magnitude (x_n largest and x_1 smallest).

Multiple term algorithms are faster than multiple digit algorithms because the latter require expensive normalization of results to fixed digit positions, whereas multiple term algorithms can allow the boundaries between terms to wander freely. Boundaries are still enforced, but can fall at any bit position. It takes time to convert an ordinary floating-point number to an internal format of a multiple digit library, however a single float is just an expansion of length 1. Conversion overhead can account for significant part of cost of small extended precision computations. The main difference between multiple digit algorithms and multiple term algorithms is that the former perform exact arithmetic to avoid round-off error, whereas the latter allow round-off to occur, and then account for it later.

When two expansions are added, the resulting sum can be captured precisely in a maximum of the sum of number of components in each of the operands. For example, let *e* and *f* be two expansions with n_e and n_f components each, then the expansion

$h = e + f$

will have $n_e + n_f$ components in it in the worst case, i.e., if the last component in *e* and the first component in *f* are nonoverlapping. When an expansion *e* is multiplied with a float *b*, then the resulting expansion *h* may have upto $2*n_e$ components in it.

This project represents ρ values as expansions internally and uses Shewchuk's library of multiple term algorithms for addition and multiplication of expansions.

5.2 Precise Comparison

In this section, we describe how we perform exact comparison mentioned in phase 2 and phase 3 of sorting in section 4.1. After phase 1 of sorting, we have a roughly sorted list of spectrum with possible out-of-order entries of ρ_T values. In phase 2, we perform extended precision comparison of consecutive entries of the spectrum in GPU to check the correctness of their order. The comparison could be of 3 types as described below:

1. **Triangle-Triangle comparison**

   Let the two triangles be $T_1 = \{ p_1, p_2, p_3 \}$ and $T_2 = \{ p_4, p_5, p_6 \}$ and the lengths of the sides $|p_1p_2|$, $|p_2p_3|$ and $|p_3p_4|$ in triangle T_1 be a, b and c respectively. Similarly, let d, e and f be the lengths of the sides of Triangle T_2. From section 4.3.1, the smallest circumradius of the triangles is given the formulae:

$$T_1 = \{p_1, p_2, p_3\}: \rho_{T1}^2 = \frac{|P_1P_2|.|P_2P_3|.|P_3P_1|}{4|P_1P_2P_3|}$$

$$\text{Where } |P_1P_2P_3|^2 = \frac{1}{4}.\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix}^2 = \frac{1}{4}{\Delta_1}^2$$

$$T_2 = \{p_4, p_5, p_6\}: \rho_{T2}^2 = \frac{|P_4P_5|.|P_5P_6|.|P_6P_4|}{4|P_4P_5P_6|}$$

$$\text{Where } |P_4P_5P_6|^2 = \frac{1}{4}.\begin{vmatrix} x_4 & y_4 & 1 \\ x_5 & y_5 & 1 \\ x_6 & y_6 & 1 \end{vmatrix}^2 = \frac{1}{4}{\Delta_2}^2$$

$\rho_{T1}^2 - \rho_{T2}^2 > 0$

If the above inequality is true, then the simplices are out of order and we need to insert T_2 in its correct position in the spectrum.

Straightforward algebraic simplifications lead to the following inequality (cross multiply the area determinants to avoid division):

$$a^2b^2c^2\Delta_2^2 - d^2e^2f^2\Delta_1^2 > 0$$

$$[\,(x_1-x_2)^2+(y_1-y_2)^2\,)\;(\,(x_2-x_3)^2+(y_2-y_3)^2\,)\,(\,(x_3-x_1)^2+(y_3-y_1)^2\,)\;(\,x_5y_6 - x_6y_5 - x_4y_6 + x_6y_4 + x_4y_5 - x_5y_4\,)^2\,] -$$

$$[\,(x_4-x_5)^2+(y_4-y_5)^2\,)\;(\,(x_5-x_6)^2+(y_5-y_6)^2\,)\,(\,(x_6-x_4)^2+(y_6-y_4)^2\,)\;(\,x_2y_3 - x_3y_2 - x_1y_3 + x_3y_1 + x_1y_2 - x_2y_1\,)^2] > 0$$

To compute this expression precisely, we may need upto 1,633,536 floats by the facts stated earlier in section 5.1 about the number of components in the resulting expansion for multiplication and addition. Since we cannot allocate this many floats for every comparison of two consecutive triangles executing in parallel in GPU, we compute the expression using multiple term algorithms until there is an overflow. If we detect an overflow, then we flag it to check in CPU.

2. Triangle-Edge Comparison

Let the triangle be $T_1 = \{ p_1, p_2, p_3 \}$ and edge be $E = \{ p_4, p_5 \}$ and the lengths of the sides $|p_1p_2|$, $|p_2p_3|$ and $|p_3p_4|$ in triangle T_1 be a, b and c respectively. Similarly, let d be the length of the edge E. From section 4.3.1, the smallest circumradii of triangle and edge is given the formulae:

$$T_1 = \{p_1, p_2, p_3\}: \rho_{T1}^2 = \frac{|P_1P_2|.|P_2P_3|.|P_3P_1|}{4|P_1P_2P_3|}$$

$$\text{Where } |P_1P_2P_3|^2 = \frac{1}{4}.\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix}^2 = \frac{1}{4}\Delta_1^2$$

$$E = \{p_4, p_5\}: \rho_E^2 = \frac{(x_4-x_5)^2+(y_4-y_5)^2}{4}$$

$\rho_T^2 - \rho_E^2 > 0$

If the above inequality is true, then the simplices are out of order. We need to insert *E* in its correct position in the spectrum. Again, algebraic manipulations lead to the following inequality:

$$a^2b^2c^2 - d^2\Delta_1^2 > 0$$

$$[\ (x_1-x_2)^2+(y_1-y_2)^2\ )\ (\ (x_2-x_3)^2 + (y_2-y_3)^2\ )\ (\ (x_3-x_1)^2 + (y_3-y_1)^2\ )\ ] -$$

$$[\ (x_4-x_5)^2+(y_4-y_5)^2\ )\ (\ x_2y_3 - x_3y_2 - x_1y_3 + x_3y_1 + x_1y_2 - x_2y_1\ )^2] > 0$$

3. Edge-Edge comparison

Let the edges be $E_1 = \{\ p_1, p_2\ \}$ and $E_2 = \{\ p_3, p_4\ \}$ and the lengths of the sides $|p_1p_2|$ and $|p_3p_4|$ be *a*, *b* respectively. From section 4.3.1, the smallest circumradii of edges are given the formulae:

$$E_1 = \{p_1, p_2\}: \rho_{E_1}^2 = \frac{(x_1-x_2)^2 + (y_1-y_2)^2}{4}$$

$$E_2 = \{p_4, p_5\}: \rho_{E_2}^2 = \frac{(x_4-x_5)^2 + (y_4-y_5)^2}{4}$$

$\rho_{E1} - \rho_{E2} > 0$

If the above inequality is true, then the simplices are out of order and we need to insert E_2 in its correct position in the spectrum. Again, algebraic manipulations lead to the following inequality:

$$[\ (x_1-x_2)^2+(y_1-y_2)^2\ )] - [\ (x_3-x_4)^2+(y_3-y_4)^2\ )\ ] > 0$$

5.3 Error Bounds

In the above comparisons, we only need the sign the inequalities and not their actual values. In most of the comparisons, the hardware supported floating point result is correct. Hence, we do not need to resort to any exact arithmetic technique unless the inexact result falls below a certain error bound, where the sign of the inequality is in question. The derivation of error for these values is a bit tricky, hence an example is shown here. The easiest way to apply forward error analysis to an expression whose value is calculated in floating-point arithmetic is to express the exact value of each subexpression in terms of the computed values plus an unknown error term whose magnitude is bounded. The error incurred by the computation $x = a \oplus b$ is no larger than $\epsilon|x|$ and is no smaller than $\epsilon|a+b|$. We are interested mainly in the upper bound of the error in the computation. Each of these bounds is different under different circumstances. Let t be the true value of $a+b$, an abbreviated way of expressing these notations is to write $t = x \pm \epsilon|x|$ and $t = x \pm \epsilon|t|$. From now on, this notation will be used as shorthand for the relation $t = x + \lambda$ for some λ that satisfies $|\lambda| \leq \epsilon|x|$ and $|\lambda| \leq \epsilon|t|$.

Example: To compute error bound for expression $u = (x_1-x_2)^2+(y_1-y_2)^2$

Let $a = (x_1-x_2)$ $\quad t_a = a \pm \epsilon|a|$

$b = (x_1-x_2)^2 = a * a$ $\quad t_b = a.a \pm (2\epsilon + \epsilon^2)|a.a|$

$= b \pm \epsilon|b| \pm (2\epsilon + \epsilon^2)(|b| \pm \epsilon|b|)$

$= b \pm (3\epsilon + 3\epsilon^2 + \epsilon^3)|b|$

$c = (y_1-y_2)^2$ $\quad t_c = c \pm (3\epsilon + 3\epsilon^2 + \epsilon^3)|c|$

$u = b + c$ $\quad t_u = b + c \pm (3\epsilon + 3\epsilon^2 + \epsilon^3)||b| + |c||$

$= u \pm \epsilon|u| \pm (3\epsilon + 3\epsilon^2 + \epsilon^3)\,|\,|u| \pm \epsilon|u|\,|$

$$= u \pm (4\epsilon + 6\epsilon^2 + 4\epsilon^3 + \epsilon^4)\,|u|$$

In this way, error bound is computed for all the comparisons discussed in section 5.2 using a program and if the ρ values fall below these bounds, then we use exact comparison by Shewchuk's method.

Chapter 6

Performance

This chapter discusses the empirical results based on the actual implementations of the algorithms and methods used in this project. We discuss about the speedup achieved and the penalty of exact arithmetic used.

6.1 Experiment Setup

The experiments were run on a Windows 7 workstation which has in Intel i7 CPU with 3.40 GHz with a memory of 16 GB RAM and a NVidia Geforce GTX 580 graphic card. We used CUDA version 4.0 with SM_20 architecture for our parallel solution for 2D alpha shapes in GPU.

6.2 Experiment Results

In order to analyse the improvement provided by our parallel implementation of alpha shapes. we compare our results with CGAL library for computational geometry, which has the best known CPU implementation of 2D Alpha Shapes. The codes were run several times on each

data set in the same hardware (*mentioned above*) and the mean $\bar{Y}$ of the corresponding measurements was taken.

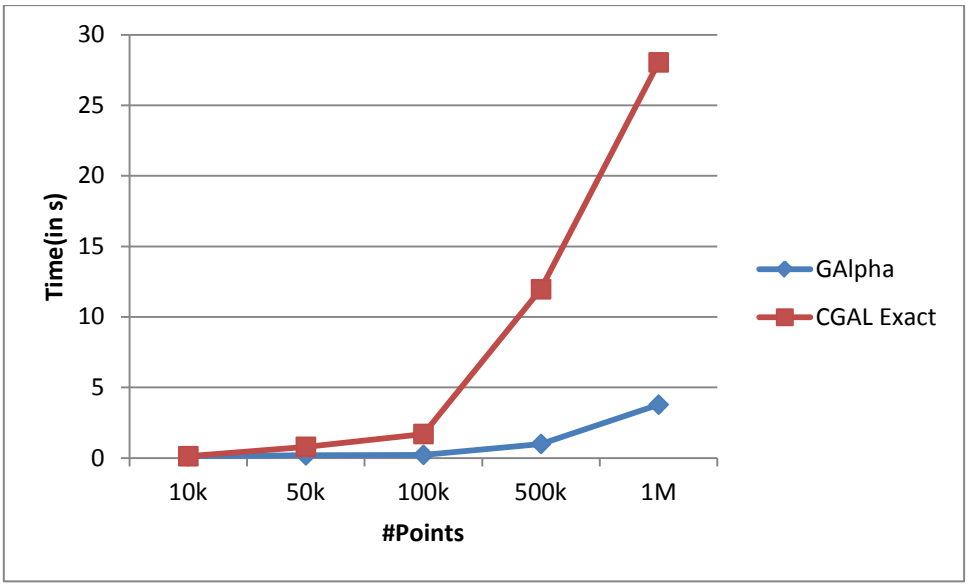


Figure 6.1: Alpha Shapes Time Comparison data between CGAL with exact constructions and GAlpha

Figure 6.1 shows the time taken by CGAL 2D Alpha Shapes with exact constructions and our alpha shapes in seconds implementation for various data sets of uniform distribution. The measurements show an evident speedup of up to 3 times for 500K data sets and 7 times for 1M data sets.

It may be noted that exact construction is needed only for alpha shapes and not for Delaunay Triangulation. When we compared our results with CGAL, this setting (exact construction only for alpha shapes) was not stable for large input sets in 2D. Hence, we compared our performance with CGAL's alpha shapes with the setting, exact construction for Delaunay Triangulation and Alpha Shapes.

We also compared our performance with CGAL 2D Alpha Shapes with inexact constructions and the results are presented in figure 6.2.

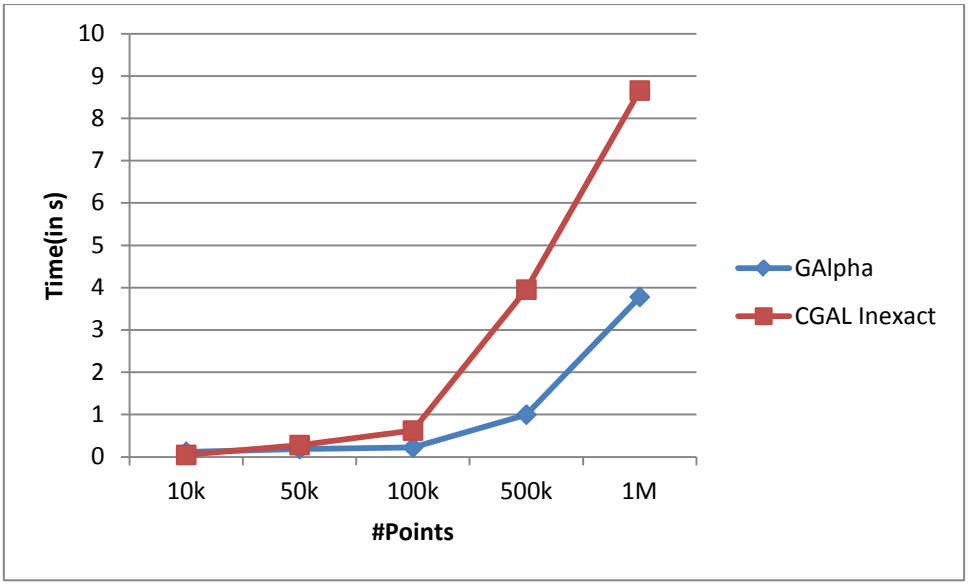


Figure 6.2: Alpha Shapes Time Comparison data between CGAL with Inexact constructions and GAlpha

The speedup compared with inexact constructions is about 3 times for 1M points compared to 7 times in the previous scenario. However, the inexact constructions yields erroneous spectrum results and hence, is not reliable.

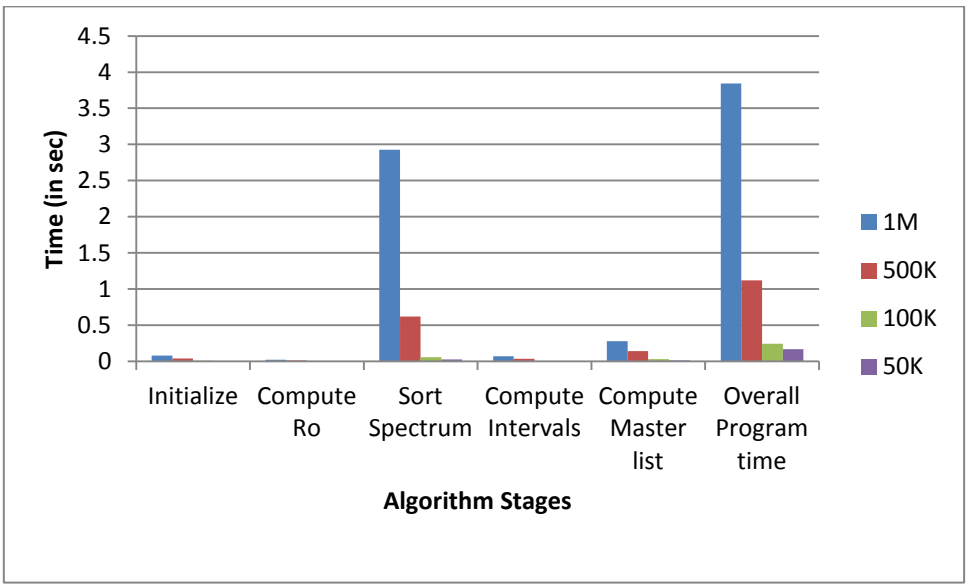


Figure 6.3: Alpha Shapes Time Comparison data between various stages of the algorithm

We present an analysis of the time taken by different stages of our algorithm. Computing 2D Delaunay triangulation in GPU is very efficient and takes roughly 1s for 1 million input point set. The largest chunk of time taken in our GPU implementation of alpha shapes is spent for the robust implementation of sorting of the spectrum described in section 4.1, which concludes that precision is the real overhead of computing alpha shapes. This is evident because we are constructing a non-boolean value and cannot afford to lose precision. Computing ρ is efficient and take less time. The next largest time is used by computing master list as the size of the master list is very high.

Phase 2 of our spectrum sorting mentioned in section 4.3.3 was performed with a fixed number of 200 floats per CUDA thread. From our experiments in computing 2D alpha shapes for uniform and Gaussian distributions of size up to 1 million, we found that none of the comparisons actually required to be pushed to CPU. Hence, our previous claim in section 4.3.3 that the array is almost sorted by the time it goes to CPU is justified and hence, the final phase 3 in CPU would be very fast in certain worse cases when the exact comparisons of the simplices require more than 200 floats.

Next, we compare our performance with CGAL in exact constructions setting for a Gaussian distribution. For this distribution, there is not much speed difference for 1 million points because of the number of exact comparisons that need to be made. In this case, our optimizations do not apply as the number of exact computations is very high. If circumradii of many simplices in a Delaunay triangulation are comparable and too close, then our algorithm is quite slow as it performs exact check on those simplices. This is a limitation of our algorithm.

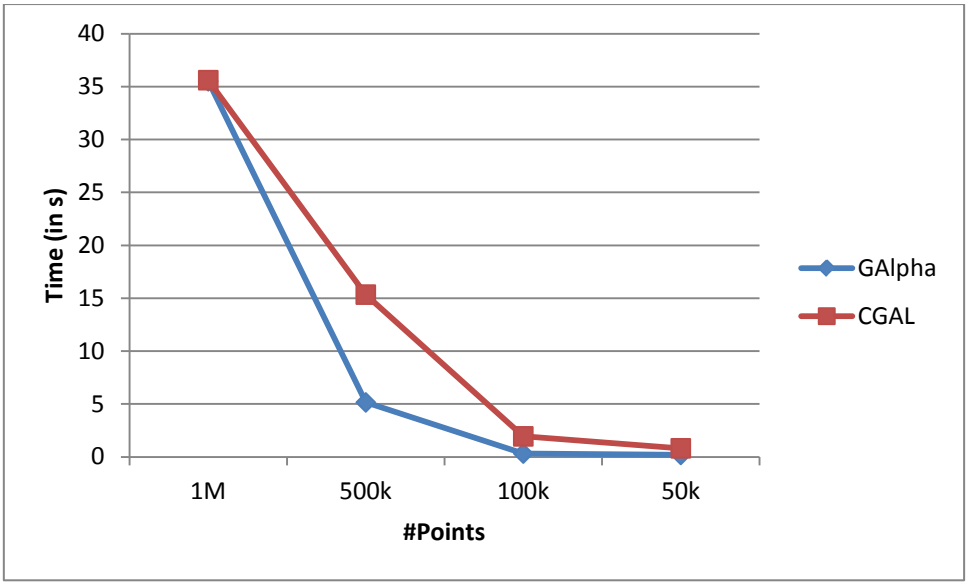


Figure 6.4: Alpha Shapes Time Comparison data between CGAL and GAlpha for a Gaussian distribution

Next, we present a snapshot of a running example of alpha shapes from our software code:

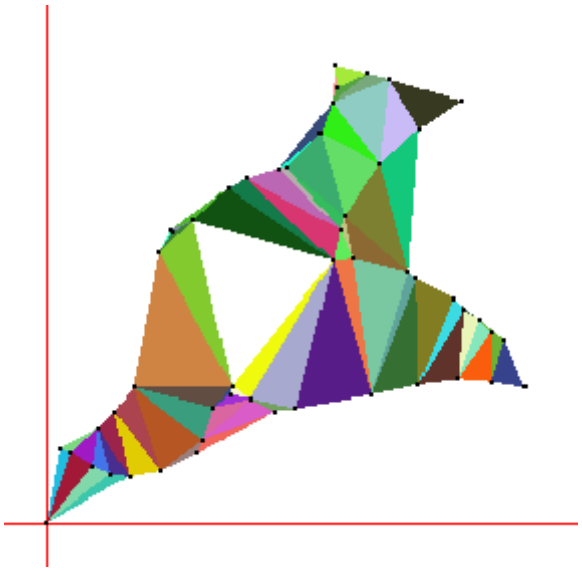


Figure 6.5 Alpha Complex for a particular α

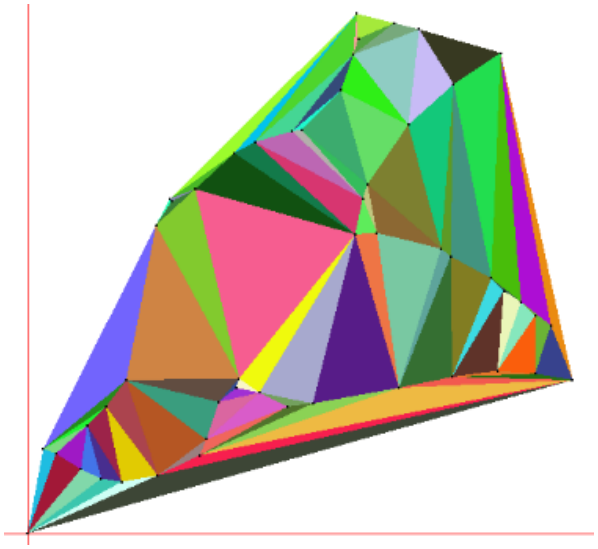


Figure 6.6 Alpha Complex

With these snapshots, we conclude this chapter having analysed the performance metrics of our algorithm and compared them with CGAL.

Chapter 7

Conclusion and Future Works

Alpha shapes formalize the intuitive notion of “shape” for spatial point sets of data which occurs frequently in computational sciences. Given a finite point set S and a real parameter α, the α-shape of S is a polytope which is not necessarily convex or connected. The set of all real values α leads to a family of α-shapes capturing the intuitive notion of crude versus fine shapes of a point set. The parameter α controls the maximum curvature of any cavity of the polytope.

A major contribution of this project is the parallel implementation of discussed theoretical concepts. This includes a program for constructing 2D Alpha shapes which is based on an underlying GPUDT program that constructs 2D Delaunay triangulation using parallel methods. Robustness is achieved via Shewchuk’s fast floating-point arithmetic as discussed in chapter 5. Overall, we have been able to speed up Alpha Shapes several times and it is up to 7x faster than CGAL’s 2D Alpha Shapes with exact constructions setting for a million points. There are some limitations in our project, like it slows down by an order of magnitude if many simplices have too close circumradii values. We are looking at optimizing the program to overcome this limitation. Ultimately, our goal is to be able to perform exact comparisons completely in GPU in the place of the current GPU and CPU combined solution.

References

[1] Edelsbrunner, H. and Mucke, E.-P. 1994. Three-dimensional Alpha Shapes. ACM Transactions Graphics 13, 43-72.

[2] Akkiraju, N., Edelsbrunner, H., Facello, M., Fu, P., Mücke, E.-P. AND Varela, C. 1995. Alpha Shapes: Definition and Software. Available at http://www.cs.duke.edu/~edels/Papers/1995-P-06-AlphaShapesSoftware.pdf.

[3] Mücke, E.P. 1993. Shapes and Implementations in Three-dimensional Geometry. Available at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.17.2624.

[4] Cheng Ho-lun, Alan. 2002. Algorithms for smooth and deformable surfaces in 3D. Available at http://www.comp.nus.edu.sg/~hcheng/academic/thesis/PhD.pdf.

[5] Shewchuk, J.R. 1996. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates, CMU-CS-96-140. Available at http://www.cs.cmu.edu/~quake/robust.html.

[6] Meng, Q., Cao, T.-T., AND Tan, T.-S. 2011. Computing 2D Constrained Delaunay Triangulation Using Graphics Hardware. Available at http://www.comp.nus.edu.sg/~tants/cdt_files/TRB3-11-report.pdf.

[7] Mark De Berg, Otfried Cheong, Marc Van Creveld, Mark Overmars. Computational Geometry: Algorithms and Applications, Second Edition. ISBN 978-3-540-77973-5.

[8] Goldberg, D. 1991. What every Computer Scientist should know about floating-point arithmetic. ACM Computing Surveys 23(1):5-48.

[9] Wilkinson, J.H. 1963. Rounding Errors in Algebraic Processes. Prentice-Hall, Englewood Cliffs, New Jersey.

[10] Edelsbrunner, H., Kirkpatrick, D.G., AND Seidel, R. 1983. On the Shape of a Set of Points in the Plane. IEEE Transactions Volume 29, Issue 4, 551-559.

[11] Shewchuk, J. R. (1996a). Robust Adaptive Floating-Point Geometric Predicates. Proceedings of the Twelfth Annual Symposium on Computational Geometry (pp. 141–150), May, 1996: Association for Computing Machinery.