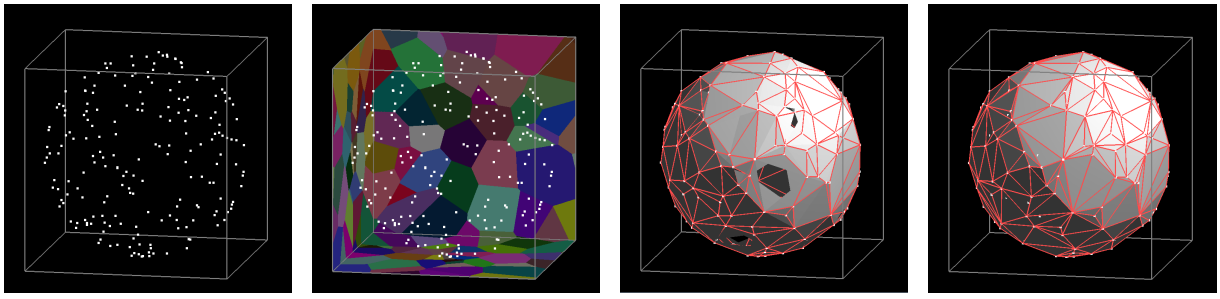
A 3D Convex Hull Algorithm for Graphics Hardware *

Mingcen Gao Thanh-Tung Cao Tiow-Seng Tan
National University of Singapore

Zhiyong Huang
Institute for Infocomm Research Singapore



(a) Input point set (b) Voronoi construction (c) Star identification (d) Hull completion

Figure 1: *Some phases of the gHull algorithm.*

Abstract

This report presents a novel approach, termed *gHull*, to compute the convex hull for a given point set in $\Re^3$ using the graphics processing units (GPUs). While the 2D problem can easily and efficiently be solved in the GPU, there is no known obvious, classical parallel solution that works well in the GPU for the 3D problem. Our novel parallel approach exploits the relationship between the 3D Voronoi diagram and the 3D convex hull so as to maximize the parallelism available in the GPU to compute the answer from the former rather than directly. Our implementation of the approach using the CUDA programming model on nVidia GPUs shows that it is robust and efficient. Our experiment shows that gHull runs up to 10x faster than the fastest CPU convex hull software, QuickHull, on inputs with millions of points.

The work first appeared as a poster presentation in the Symposium on Interactive 3D Graphics and Games (I3D 2011), Marriott Fisherman's Wharf, San Francisco, CA, 18–20 February, 2011.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modelling—Geometric algorithms I.3.1 [Computer Graphics]: Hardware Architecture—Graphics processors

Keywords: GPGPU, computational geometry, Voronoi diagram, Star splaying

1 Introduction

For a set S of points in 3D, the convex hull $\mathcal{C}(S)$ of S is the smallest convex set that encloses all points of S. Our problem here is to compute for a given set S its convex hull represented as a triangular mesh, with vertices that are points of S, bounding the convex hull. Each point of S on the boundary of $\mathcal{C}(S)$ is called an *extreme vertex*. Convex hull is a useful geometric structure in various areas of research and applications. In scientific visualization and video game, convex hull can be a good form of bounding volume to be used in checking for intersection or collision between objects [Liu et al. 2008; Mao and Yang 2006]. In robotics, convex hull is used to approximate robots and obstacles for the purpose of path planning [Okada et al. 2003; Strandberg 2004]. In astronomy, convex hull is a basic structure used to analyze the characteristic of the atmosphere [Fuentes et al. 2001; Amundson et al. 2005]. In general, convex hull is also a useful tool in biology and genetics [Wang et al. 2009] and visual pattern matching [Hahn and Han 2006].

In the past few decades, many CPU-based 3D convex hull algorithms have been developed and implemented. Among them, QuickHull [Barber et al. 1996] is notably the most efficient and popular one that is used in practice. Its algorithm is based on the incremental insertion approach. In 3D, starting from a single tetrahedron, or a volume in general, of 4 extreme vertices, it recursively adds an input point outside the volume to grow its size, till no more input points can be added. During the process, input points found to be within the volume are discarded from the computation. At each step, the farthest point from the faces of the volume, which is an extreme vertex, is chosen to be added. This also potentially maximizes the number of points that can be discarded. The resulting volume is the required convex hull. Though simple, similar to most other incremental insertion algorithms, QuickHull in 3D is not suitable for implementing on the GPU as it is difficult to maximize the parallelism when "growing" the volume due to having many dependencies among the different parts of the growing process. As a side note, the QuickHull in 2D can easily be implemented on the GPU as the mentioned dependencies is minimal in this case.

Over the years, there are also many parallel convex hull algorithms designed; for examples [Miller and Stout 1988; Amato and Preparata 1993]. Those are mainly theoretical work with no practical implementation, and they do not seem to map well to the currently popular GPU architecture that supports tens of thousands of computational threads. This report proposes a novel approach, termed *gHull* algorithm, using the GPU to compute $\mathcal{C}(S)$ for a given S. Our parallel approach exploits the relationship between 3D Voronoi diagram and 3D convex hull so as to maximize the parallelism that can be utilized from the GPU. In particular, instead of computing $\mathcal{C}(S)$ directly, we efficiently compute in the GPU the six slices of the 3D Voronoi diagram of S when points of S are enclosed within a box; see Figure 1(b). Next, we dualise, again in parallel, these six 3D Voronoi diagram slices to obtain an approximation of $\mathcal{C}(S)$; see Figure 1(c). Eventually, we can efficiently transform the approximation to $\mathcal{C}(S)$ in parallel by adapting the Star Splaying algorithm [Shewchuk 2005]; see Figure 1(d).

The main contribution of this report is our gHull algorithm to com-

* {mingcen | caothanh | tants}@comp.nus.edu.sg; zyhuang@i2r.a-star.edu.sg

pute the 3D convex hull using the GPU. Our algorithm is robust because it uses only the 3D orientation check predicate. Our experiment shows that our implementation of gHull using the CUDA programming model [NVIDIA 2010] on nVidia GPUs runs up to 10x faster than the popular QuickHull software. The rest of the report is organized as follows. Section 2 introduces some related work to our gHull algorithm, which is later described in Section 3. Section 4 provides some insights into our approach of using the digital space computation to approximate that in the continuous space. Some important implementation details are discussed in Section 5. The effectiveness of our approach is demonstrated in Section 6, and finally Section 7 concludes the report.

2 Related Work

We feature a few prominent convex hull algorithms in Section 2.1, and discuss briefly the Star Splaying algorithm that we adapt in our proposed algorithm in Section 2.2.

2.1 Convex Hull Algorithms

A series of well known algorithms has been designed to compute the convex hull $\mathcal{C}(S)$ for a set S of n points in 3D. The incremental insertion algorithm [Clarkson and Shor 1988] constructs the convex hull by inserting points one by one with the point location technique. As mentioned, QuickHull [Barber et al. 1996] is a variant of the incremental insertion approach, where the outermost point is inserted in each round. The algorithm of Preparata and Hong [1977] constructs the convex hull recursively using the divide-and-conquer technique. Both the incremental insertion and the divide-and-conquer approach have the time complexity of $\mathrm{O}(n \log n)$. In 2D and 3D, the optimal output-sensitive convex hull algorithm has the time complexity of $\Theta(n \log h)$ [Chan 1996] where h is the number of extreme vertices. Empirically, QuickHull has the same output-sensitive time complexity. Because of the good time complexity and low overhead in practice, QuickHull has been a popular approach adopted by many applications over the years.

Besides the above mentioned sequential algorithms, there exist also many parallel convex hull algorithms. For example, Miller and Stout [1988] and Amato and Preparata [1993] describe $\mathrm{O}(\log n)$ parallel algorithm for n points using $\mathrm{O}(n)$ processors. These algorithms are only of theoretical interest and not yet of practical value as there are no known efficient implementations of them. One of the reasons is that these algorithms are complex, making them hard to scale on a fine-grain data-parallel massively-multithreaded architecture likes the GPU. For the current commercial systems of a small number of processors (multi-core), algorithms designed by Dehne et al. [1995] and Gupta and Sen [2003] may be applicable, but these algorithms have no known implementation that demonstrate their uses.

2.2 Star Splaying Algorithm

As mentioned in the introduction, our algorithm uses an approach of approximating the convex hull and then repairing it to obtain the required convex hull. Star Splaying [Shewchuk 2005] is an algorithm to repair a good approximation of a convex hull in time close to linear to the number of vertices in practice. For star splaying in 3D, each input point s maintains a set of neighbours which are incident to some same edges as s to be its *star*. Two points neighboring to each other are *consistent* if their stars agree in forming two triangles incident to them. The goal is that given the precondition mentioned below, if all the stars are convex and all pair of neighbouring points are consistent, then the set of stars represent the convex hull of the point set. The Star Splaying algorithm first repairs all the stars to make them convex, then a splaying process is repeatedly applied to mend those stars that are not consistent. During the process, an input point is *killed* if its star is no longer contained within a half-space, i.e. it is known not to be an extreme vertex. To ensure the correctness of the algorithm, a required precondition is that each point s, except for the lexicographically smallest point (in terms of the coordinates), must contain within its star at least one point that lexicographically precedes s.

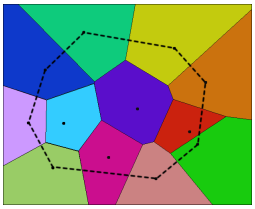


Figure 2: *An illustration in 2D where points having unbounded Voronoi cells are extreme vertices. They are connected in dotted lines to form the convex hull.*

A nice feature of the Star Splaying algorithm is that making the stars convex as well as enforcing their consistencies is done independently in each star. This is well suited to doing parallel computation. As such, we adopt the Star Splaying algorithm in our gHull algorithm to efficiently transform our approximation of the convex hull of S into $\mathcal{C}(S)$.

3 The gHull Algorithm

The main idea of our algorithm is to utilize the relationship of the Voronoi diagram and the convex hull of the same point set S. In particular, only the Voronoi cells of the extreme vertices of S are unbounded (i.e. extended to infinity); see Figure 2. Thus, one can first identify these Voronoi cells to derive the extreme vertices of S. Traditionally, this observation is not computationally useful as the Voronoi diagram $\mathcal{V}(S)$ is a harder to manage structure than the convex hull, and is as expensive to compute. On the other hand, there are recent development of very efficient GPU algorithms that compute digital Voronoi diagrams [Cao et al. 2010], $\mathcal{V}_D(S)$, and these become the good starting point to derive first an approximation, then the solution $\mathcal{C}(S)$.

In our proposed algorithm, we enclose the input point set S in a large enough cube $\mathcal{Q}$ which contains integer grid points, each corresponds to one unit cell of $\mathcal{Q}$. We use the six faces ($i = 1, 2, ..., 6$) of the cube $\mathcal{Q}$ to capture the unbounded Voronoi cells of S. Theoretically, if the cube is large enough, dualising the six slices of the 3D Voronoi diagram $\mathcal{V}(S)$ on the faces of the cube $\mathcal{Q}$ gives us $\mathcal{C}(S)$. However, since the Voronoi diagram we compute is in the digital space, and due to the size limitation of $\mathcal{Q}$, we can only obtain an approximation of the convex hull, as illustrated in Figure 1. The gHull algorithm consists of 5 phases:

Phase 1. Voronoi Construction. Construct the intersection of $\mathcal{V}_D(S)$ with each face i of $\mathcal{Q}$, denoted as $\mathcal{V}_i(S)$. We use $S' \subseteq S$ to denote the set of points with non-empty Voronoi cells in some sides.

Phase 2. Star Identification. Compute a star for each $s \in S'$ from the neighbourhood information obtained from each $\mathcal{V}_i(S)$.

Phase 3. Hull Approximation. Splay stars of points in S' to arrive at $\mathcal{C}(S')$.

Phase 4. Point Addition. To $\mathcal{C}(S')$, add a point $s \in S - S'$ if s is outside $\mathcal{C}(S')$. An initial star of s is also constructed. We use S'' to denote the set of such points.

Phase 5. Hull Completion. Apply star splaying once more to compute $\mathcal{C}(S' \cup S'')$, which is also $\mathcal{C}(S)$.

The details of each phase is as follows; see the accompanying video for an animation of the proposed algorithm.

Phase 1. Voronoi Construction

The aim here is to quickly identify potential extreme vertices through the relationship of the Voronoi diagram and the convex hull. This phase computes six slices of the 3D digital Voronoi diagram of S intersecting with the boundary of the cube $\mathcal{Q}$. Each slice is a weighted Voronoi diagram of the projection of S on the corresponding face. By dualising these slices, we can obtain a good approximation of $\mathcal{C}(S)$. From the Voronoi diagram slices, we can quickly filter out points in S that are less likely to be extreme vertices (they do not have their Voronoi cells appearing on any slice).

To compute a slice of $\mathcal{V}_D(S)$, we adopt the Parallel Banding Algorithm (PBA) of Cao et al. [2010]. Points in S are first shifted to the nearest grid points in $\mathcal{Q}$, and then projected onto the corresponding side of the cube. We keep only the nearest point in case when multiple points are projected onto the same pixel. After that, PBA can efficiently compute the corresponding slice of $\mathcal{V}_D(S)$.

A few notes are in order. First, before the above computation, we use parallel reduction to identify the lexicographically smallest point $s_\ell \in S$. This point always overrides, for the purpose of the projections onto the cube faces, other points of S when they fall onto the same grid point. This ensures that s_ℓ appears in our subsequent phases to guarantee the correctness of our algorithm. Second, due to the digital nature of our computation, some extreme vertices do not manage to have their Voronoi cells on any $\mathcal{V}_i(S)$. This is because such a point is projected to a grid point that is occupied by another point in S, or it is no longer an extreme vertex after being shifted. All such extreme vertices will be recovered in Phase 4. Third, those points with Voronoi cells on some side of $\mathcal{Q}$ do not necessarily mean they are extreme vertices, due to the size limitation of $\mathcal{Q}$ as well as the shifting from continuous to digital space.

Phase 2. Star Identification

The aim here is to quickly derive a rough "structure" of the approximate convex hull from the above digital Voronoi diagram slices $\mathcal{V}_i(S)$. Ideally, dualising the restricted Voronoi diagram of S on a closed cube results in a 3D polyhedron, not necessarily convex, approximating the convex hull of S. However, in the digital Voronoi diagram slices, a Voronoi cell can be, for example, disconnected, resulting in the dualised polyhedron having holes or duplicated triangles. Instead of constructing a polyhedron, we only construct the stars of the points (with non-empty Voronoi cells) using the adjacencies of the Voronoi cells.

For each $\mathcal{V}_i(S)$, we scan for corners (which are shared by 4 pixels) incident by 3 or more Voronoi cells. Such a corner is termed a *digital Voronoi vertex*; see Figure 3. For a digital Voronoi vertex incident by 3 Voronoi cells of the vertices v_1, v_2, v_3 in clockwise order (Figure 3(a)), we insert edge v_1v_2 into the star of v_2; v_2v_3 into the star of v_3; and v_3v_1 into the star of v_1. For a digital Voronoi vertex incident by 4 Voronoi cells of the vertices v_1, v_2, v_3, v_4 in clockwise order (Figure 3(b)), edges v_1v_2, v_4v_2 are inserted into the star of v_2; v_2v_3, v_1v_3 into the star of v_3; v_3v_4, v_2v_4 into the star of v_4; and v_4v_1, v_3v_1 into the star of v_1. On the whole, this phase is also efficiently carried out in parallel where each thread is assigned to check a corner.

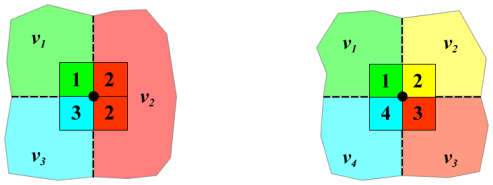


Figure 3: *A corner incident by (a) 3 Voronoi cells or (b) 4 Voronoi cells is used to derive edges to be inserted into the stars.*

A few notes are in order. First, when inserting an edge to a star, we use the beneath-beyond method to maintain the convexity of that star. Second, for the star of each point s, if it contains no point that lexicographically precedes s, we augment it with s_ℓ. This is to ensure the correctness of our algorithm. Third, in Figure 3(a), we do not need to insert the edge v_2v_1 into the star of v_1, as such an edge is, most of the time, also seen and inserted from another digital Voronoi vertex. In certain cases, some edges might not be inserted, but that does not affect the correctness of the algorithm.

Phase 3. Hull Approximation

The aim is to obtain the first approximation of $\mathcal{C}(S)$. This approximation is the convex hull $\mathcal{C}(S')$ of S'.

To achieve this, we adapt the Star Splaying algorithm of Shewchuk [2005] to work in the GPU. The adaptation is to do multiple consistency checking and enforcement of edges in parallel. We divide the Star Splaying algorithm into two stages: *checking stage* and *inserting stage*, alternately performed until all the stars are consistent (i.e. no more edges to check). In the checking stage, we gather (using stream compaction) all edges that need to be checked and assign each to a thread for consistency checking (i.e. the consistency of the stars of points incident to the edge). If an edge fails the consistency check, we create up to four edges (depending on the situations as discussed in [Shewchuk 2005]) to be inserted in the next stage. In the inserting stage, we gather (with sorting) each set of edges to be inserted into the same star and assign it to a thread to perform the insertions. If a new edge is inserted into a star, it is marked for checking in the next checking stage. If an edge ab in the star of a is removed either because it lies inside the new star of a after splaying, or because a is killed (i.e. a was confirmed not to be an extreme vertex), then the edge ba in the star of b, if exists, will be marked for checking.

We note that the amount of work here depends on how close the initial stars are to those in $\mathcal{C}(S')$. For our case, as shown in our experiment, this phase is relatively fast as the first two phases have arrived at a very good approximation to $\mathcal{C}(S')$. Also, due to the checking of edges can be independently performed, and the insertion of edges can be concurrently done in different stars, the whole process works very well on the GPU.

Phase 4. Point Addition

The aim of this phase is to recover extreme vertices of S lost in Phase 1. Basically, we want to find all the points in S that are outside $\mathcal{C}(S')$. We utilize the graphics rendering pipeline to achieve good performance without losing the accuracy and the robustness of the algorithm.

The idea is to perform the checking in the digital space first to handle most of the trivial cases, before using a more accurate yet costly

checking in the continuous space. In the first round of checking, we render the triangle faces of $\mathcal{C}(S')$ with the view direction orthogonal to each side of $\mathcal{Q}$ in turn. For each rendering, we use the color and depth buffer to record the index and the depth value, respectively, of the triangle covering each pixel. Then, we project each point $s \in S - S'$ in the corresponding viewing direction and compare the depth value d_s of s with the depth value d in the depth buffer at the corresponding position. If $d_s - d \leq \tau$ where τ is a constant threshold (which is equal to 1 pixel width), then s is potentially outside $\mathcal{C}(S')$, and subjects to the second round of checking.

After the first round, most of the points that are clearly inside $\mathcal{C}(S')$ would have been removed. For each given point s that is potentially outside, we also record a triangle that covers its projection in one of the viewing direction. This triangle is close to s. Pick an arbitrary point r in the boundary of $\mathcal{C}(S')$. The point s would either be below one of the triangle in the star of r, or the ray $r\vec{s}$ would intersect $\mathcal{C}(S')$ at a triangle T. Using a technique similar to point location, starting from the recorded triangle of s, we can quickly find T, and accurately determine whether s is inside or outside $\mathcal{C}(S')$. If s is outside, we include s in S'' and use T to form the initial star of s. This second round of checking can be done in parallel for each point that is potentially outside, using one thread each.

A few notes are in order. First, during the first round of checking, $d_s - d > \tau$ in one of the viewing direction does not necessary means s is inside $\mathcal{C}(S')$. In fact, $d_s - d$ can be arbitrarily large yet s is still outside. However, in Section 4, we prove that if $d_s - d > \tau$ in all six viewing directions, then s is guaranteed to be inside $\mathcal{C}(S')$, and $\tau = 1$ pixel width is indeed the optimal choice. Second, we again need to augment each newly constructed star s with s_ℓ if s does not contain a point that lexicographically precedes s.

Phase 5. Hull Completion

The aim here is to take $\mathcal{C}(S')$ together with the stars of points in S'' to apply once again the above parallel version of the Star Splaying algorithm to obtain $\mathcal{C}(S' \cup S'')$, which is $\mathcal{C}(S)$.

The Correctness of gHull

It is clear from the 5 phases of the algorithm that gHull computes the convex hull $\mathcal{C}(S)$ of the given point set S. Phases 1 to 3 give a good approximation of the convex hull, while Phase 4 makes sure that we do not lose any extreme vertices in S (the correctness of the choice of τ is discussed in the next section). Since we fulfill the required precondition of the Star Splaying algorithm, Phase 5 successfully constructs the desired convex hull.

4 Digital versus Continuous Space

Our algorithm makes use of the fast computation in the digital space to approximate most of the computation, before actually transforms the result to the original continuous space. In this section, we analyze this approach in terms of both accuracy and efficiency.

4.1 Digital Depth Test

In Phase 4 of the algorithm, we use the six faces of $\mathcal{Q}$ as six viewing planes. We compare the depth d_s of each point s with the minimum depth value of $\mathcal{C}(S')$ at the corresponding projection of s to quickly exclude points that are inside $\mathcal{C}(S')$. However, since the depth buffer we obtain when rendering $\mathcal{C}(S')$ is of finite resolution, the depth value d of the projection of s is actually the depth value of the center of the pixel containing this projection. Depending on the triangle covering that projection, $d_s - d$ can be arbitrarily large. The following claim shows that using a constant $\tau = 1$ pixel width in the checking in all the different viewing directions is accurate and optimal in deciding whether s is inside $\mathcal{C}(S')$, i.e. we do not miss any point that is actually outside.

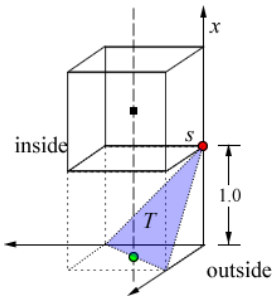


Figure 4: *The digital depth test of a point s against a triangle T on the boundary of $\mathcal{C}(S')$ when s is outside $\mathcal{C}(S')$. Recall $S' \subseteq S$ denotes the set of points with non-empty Voronoi cells in Phase 1.*

Claim 1. Let $s \in S - S'$ be a point outside $\mathcal{C}(S')$. In (at least) one of the six renderings of $\mathcal{C}(S')$ orthogonal to a side of the cube $\mathcal{Q}$, we have $d_s - d \leq \tau$ where d is the depth value of the center of the pixel containing the projection of s, d_s is the depth of s, and $\tau = 1$ pixel width.

Proof. The point s is inside a unit cell $Q \in \mathcal{Q}$ whose center is the grid point $(\bar{x}, \bar{y}, \bar{z})$. The coordinate of s is $(\bar{x} + \delta_x, \bar{y} + \delta_y, \bar{z} + \delta_z)$ where $\delta_x, \delta_y, \delta_z \in [-0.5, 0.5]$. Let T be the triangle covering the pixels containing the projections of s in different viewing directions, and the plane equation of T be $ax + by + cz + K = 0$.

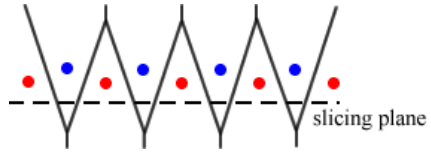
Since T appears in the depth buffer, and $\mathcal{C}(S')$ is convex, T must be visible from three different viewing directions. This forms a coordinate system in which the plane equation of T has $a, b, c \geq 0$. There exists one viewing direction that $a \geq b \geq c$, and without loss of generality we assume that this direction is along the positive x-axis; see Figure 4 for an illustration where the outside region is below the triangle T and the inside region is above.

In this viewing direction, $d_s = \bar{x} + \delta_x$ and d is the depth of T at $(\bar{y}, \bar{z})$. As s is outside $\mathcal{C}(S')$ and thus is in front of the plane of T, $a(\bar{x} + \delta_x) + b(\bar{y} + \delta_y) + c(\bar{z} + \delta_z) + K \leq 0$, and we thus have:

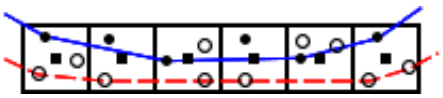
$$\begin{aligned} d_s - d &= (\bar{x} + \delta_x) - \left(-\frac{b\bar{y} + c\bar{z} + K}{a}\right) \\ &\leq \frac{a(\bar{x} + \delta_x) + b(\bar{y} + \delta_y) + c(\bar{z} + \delta_z) + K}{a} - \frac{b\delta_y}{a} - \frac{c\delta_z}{a} \\ &\leq -\frac{b\delta_y}{a} - \frac{c\delta_z}{a} \\ &\leq \frac{b}{2a} + \frac{c}{2a} \\ &\leq 1 \end{aligned}$$

There is a technical issue not covered in the above discussion. That is when the depth values d used in the checking in the six viewing directions belong to different triangles. Suppose that the depth value of triangle T is used in one of the directions, then from the above argument, there is one direction in which the depth d of the plane containing T fulfills the inequality $d_s - d \leq 1$. Suppose T' is the other triangle that actually covers the projection of s in that direction, then due to the convexity of $\mathcal{C}(S')$, the depth d' of T' must be no smaller than d, and thus $d_s - d' \leq d_s - d \leq 1$, as required. □

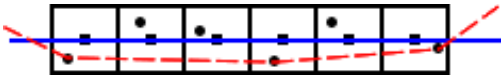
We have two notes here. First, the above threshold of $\tau = 1.0$ is the best possible as Figure 4 illustrates an extreme case where the depth difference is 1 pixel width in all three viewing directions. Second, the above proof and result can be generalized to any dimension D where the threshold τ is then equal to $\frac{D-1}{2}$.



(a) Slicing problem



(b) Under-approximating problem



(c) Over-approximating problem

Figure 5: *Three problems associated to the computation in the digital space.*

4.2 Digital Convex Hull Approximation

Our algorithm first computes an approximation of the digital convex hull of S, where points in S are shifted to the grid points of $\mathcal{Q}$. Then we transform this into a convex hull of a subset S' of S by adapting the Star Splaying algorithm, which is used as an approximation of the convex hull of S. Due to the nature of the digital space and of our approach, there are three problems that can possibly hinder the performance of gHull. They are: *slicing problem*, *under-approximating problem* and *over-approximating problem*; see Figure 5.

Slicing problem: This problem is the result of using a finite cube to find the Voronoi cells that are unbounded. As some of the bounded Voronoi cells can extend beyond the cube $\mathcal{Q}$, they are captured in our six Voronoi diagram slices, although they are not corresponding to extreme vertices; see Figure 5(a). This results in more work for Phase 2 and Phase 3. To reduce this problem, we can use a bigger cube $\mathcal{Q}$, while scaling the point set to only a small volume in the center of $\mathcal{Q}$, effectively pushing the slicing planes away from the point set. This reduces the number of bounded Voronoi cells wrongly captured in Phase 1.

Under-approximating problem: This problem occurs when we have multiple points shifted to the same grid point. Since we can only record one point per grid point, we potentially miss many points outside $\mathcal{C}(S')$, causing more work for Phase 5. See Figure 5(b) for a 2D illustration where the square black points are grid points, the round black points are kept after the shifting, and the solid line denotes the computed convex hull after Phase 3. The round white points are points not mapped in Phase 1, many of which are outside $\mathcal{C}(S')$. By efficiently locating a nearby visible face for each point outside $\mathcal{C}(S')$ during Phase 4, we construct a very good star for that point, thus reducing the splaying needed in Phase 5. As such, this problem does not severely affect the performance of our algorithm.

Over-approximating problem: This problem is also caused by us shifting the points in Phase 1. In certain cases, for example when points are distributed near the surface of a cube axis-aligned with $\mathcal{Q}$, many points that are not extreme vertices are shifted outside and are legitimately captured in Phase 1. See Figure 5(c) for a 2D illustration, where after Phase 1 all the round black points, after shifted to the square black grid points, are captured. Many of them need to be removed during Phase 3. This causes a lot of wasteful work for Phase 2 and Phase 3. This is a disadvantage of gHull, and is only slightly resolved by increasing the size of $\mathcal{Q}$.

5 Some Implementation Details

We implement the gHull algorithm using the CUDA programming model and OpenGL on nVidia GPUs. By the nature of the GPU, it is better to allocate memory in big chunks rather than dynamically allocate many small blocks, for efficient manipulation. For our implementation, we use two major lists to store the stars and the edges (of the stars), as shown in Figure 6. Each star has a continuous chunk of memory whose size is enough to store all of its current edges plus a certain amount of free space. Each star records its position (the coordinates of its point), its status of alive or dead (when the corresponding point of the star is killed), the number of edges, the actual size of the chunk of memory allocated for it, and the starting location of its storage in the global edge list. Each edge of a star records the index of the star's corresponding neighbour, and a flag to be marked true when checking for consistency is needed. The challenge here is that the edge list has a dynamic size as stars are shrinking as well as expanding during the star splaying process. Any time a star uses up its chunk of allocated storage, we have to expand the edge list. When such occasion happens, we also take this opportunity to shrink or expand the storage of all the stars to maintain a "healthy" ratio of available storage for each star. This helps to reduce the number of times we need to reallocate the edge list. Also, since we start star splaying on a good approximation of the convex hull, the stars usually do not grow drastically.

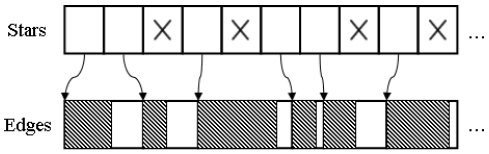


Figure 6: *The data structures to keep stars and edges for parallel star splaying. $\times$ means that the corresponding point was filtered out during Phase 1 or Phase 4.*

Phase 1. Voronoi Construction

We need to perform a series of projections of the point set before applying the Parallel Banding Algorithm to compute the Voronoi diagram slices. Since a huge amount of random memory access is inefficient in the GPU, we use a simple bucket sort technique to speed up the process. For each side of the cube, represented as a texture, we divide it into grid cells of small size. Using the GPU radix sort, we can quickly re-arrange the point set such that points projected onto the same cell are grouped together. The size of a grid cell is small enough such that we can fit it in the on-chip shared memory. We let one block of threads to project all the points of the same grid cell onto a shared memory array first, using the *AtomicMin* operation to keep the point with the smallest distance to the plane when multiple points are projected on the same grid point. After that, we coherently write out the result to the global memory.

Phase 2. Star Identification

For each point s, we use the beneath-beyond algorithm, inserting its neighbors one by one to construct its *convex star* [Shewchuk 2005]. Through scanning the textures, we first create records of the insertions to be done to different stars. Then, we sort the records and count the number of insertions per star, for the purpose of allocating memory for the edge list. Finally, we assign one thread to work on one star, processing all of its insertions independently without any need to coordinate with other threads.

Phase 3. Hull Approximation and Phase 5. Hull Completion

We adapt the Star Splaying algorithm to work in parallel by breaking it into the checking and the inserting stage. In the checking stage, we do a parallel compaction on the flags of the edges to obtain the list of edges to be checked for consistency. Each checking can potentially lead to up to four insertions of edges into different stars. All these insertions are sorted and compacted, as what we have done in Phase 2, for the actual insertions into the stars to be carried out in parallel in the next stage. After this checking stage, we also need to perform an expansion of the edge list if needed. In the inserting stage, when performing the actual insertions, an edge ab in the star of a might be deleted, and the corresponding edge ba in the star of b, if any, need to be marked for consistency checking. Since the edges of the stars are being modified in parallel, such marking needs to be recorded to be performed later, when all the insertions have finished.

In certain cases when some of the stars do not have a good approximation, many splaying rounds may be required. Thus, once it reaches the situation where only a small number of stars still require many rounds of splaying, the computation is moved to the CPU. Although transferring data between the devices takes time, it is worthwhile since CPU is more efficient here and only a small amount of data is transferred.

For the robustness of the computation in which we only need orientation checks, we use Shewchuk's fast robust predicates [Shewchuk 1997]. To handle degeneracy in the input point set, the simulation of simplicity technique of Edelsbrunner and Mücke [1990] is employed.

Phase 4. Point Addition

The first round of checking in this phase is carried out in OpenGL, which works seamlessly with other phases being done in CUDA. As we keep stars rather than triangles, we first generate a list of triangles in $\mathcal{C}(S')$ from the stars. To avoid generating duplicate triangles, each triangle abc is created only by the star of a where a has the smallest index among the three. We first count the number of triangles generated by each star, and use parallel prefix to determine the indices for each set of triangles in the triangle list, before actually generating the triangles for rendering. When a triangle is rendered, we record in the color buffer the index of one of the three vertices so that we can use it as the starting point for our point location in the second round of checking.

6 Experimental Results

Note: The experimental results mentioned below are as they first appeared in the I3D 2011 poster (February 2011). Our algorithm has since been improved to have much better performance. The new manuscript is currently under preparation.

The experiment is done on a PC with an Intel i7 930 2.8Ghz, 6GB DDR3 memory and a single NVIDIA GTX 460 Fermi graphics card with 2GB of video memory. Our implementation is compiled using CUDA 3.1 toolkit, while QuickHull is compiled using Microsoft Visual Studio 2008 with all optimizations enabled. All the point coordinates are in the range of $[0, 1)$.

Different point sets

To compare the efficiency of our approach with QuickHull, we generate randomly uniformly distributed points in: a *cube*, a *ball*, a *thin sphere* with thickness 0.01, and a *thin box* with thickness 0.01. For the ball and the thin sphere test cases, we use a texture of size 512x512 when computing the Voronoi diagram slices. For the cube and the thin box test cases, we use a larger texture of size 1024x1024 and 2048x2048 respectively, but scale the point set to only the 512x512 center region, to reduce the effect of the slicing problem. Note that using larger texture does not necessarily improve the performance of the algorithm, due to the extra cost in computing the digital Voronoi diagram slices.

Figure 7 (left) shows the running time of gHull when the number of points ranges from 10^6 to 10^7. Our gHull algorithm scales almost linearly for all four different test cases. It performs best in the ball test case, while slightly slower in the cube test case due to the effect of the over-approximating problem. On the other hand, also being affected by the over-approximating problem, gHull is significantly slower while processing the thin box test case. This is because in the digital space, most of the points appear as extreme vertices on the boundary of the box. On the other hand, in the continuous space, the actual number of extreme vertices are very small, only a few hundreds among millions of input points.

Figure 7 (right) compares the speed up of our algorithm over QuickHull for different test cases, with gHull up to 10x faster. Having a better method to quickly remove non-extreme vertices, gHull easily outperforms QuickHull, even on a difficult thin sphere test case. Being executed in parallel, gHull scales much better than QuickHull when the number of points increases, resulting in a large increase in speed up. Due to a fixed overhead of computing the digital Voronoi diagram slices (which is input-independent and only depends on the texture size), gHull is only a little faster than QuickHull when the number of points is small.

Varying the number of non-extreme vertices

In order to investigate the effect of the number of extreme vertices and non-extreme vertices on the performance of the algorithm, we use a controlled ball test case, in which we first generate h points randomly on a sphere, and then generate $n - h$ points randomly inside a ball of slightly smaller radius. This gives us a point set with n points, out of which h points are extreme vertices.

Figure 8 shows the running time of gHull (on the left) and the speed up over QuickHull (on the right) when we fix h and vary n in the range of 10^6 to 10^7. Excluding the fixed overhead of computing the Voronoi diagram slices, the running time of gHull scales linearly at a very slow rate when n increases. Note that the running time with $h = 10^3$ and $h = 10^4$ are very close, while when $h = 10^5$ gHull is substantially slower. This is the consequence of the under-approximating problem when the texture cannot capture all the extreme vertices due to its limited size. Regardless of that, gHull is still 2x to 6x faster than QuickHull, with a large increase in the speed up when n increases.

Varying the number of extreme vertices

On the other hand, Figure 9 shows the running time of gHull (on the left) and the speed up over QuickHull (on the right) when we fix n and vary h multiplicatively from $2^0 \times 10^4$ to $2^6 \times 10^4$. In the first few steps, gHull scales almost linearly to $log\ h$. However, gHull is much slower when h is over 10^5, due to the under-approximating problem. As such, the speed up over QuickHull decreases, although it is still in the range of 3x to 5x.

The running time of different phases

We can look at the running time break down of different phases to further confirm the consequence of the three problems mentioned earlier. Figure 10 shows the running time of gHull on different test cases, with 10^7 points in the input. The use of larger textures in the cube and the thin box test cases results in a slightly larger running time for Phase 1. Phase 2 takes a very small amount of time, except for the thin box test case where there is an excessive number of vertices mistakenly recorded as extreme vertices. This is also the

same reason for the huge amount of time wasted in Phase 3 for this test case. On the other hand, Phase 4 takes about the same amount of time for all the cases, since the number of triangles rendered is small while the total area rendered is roughly the same. Lastly, Phase 5 takes more time on the thin sphere and the thin box test cases due to having much more points very near the actual convex hull.

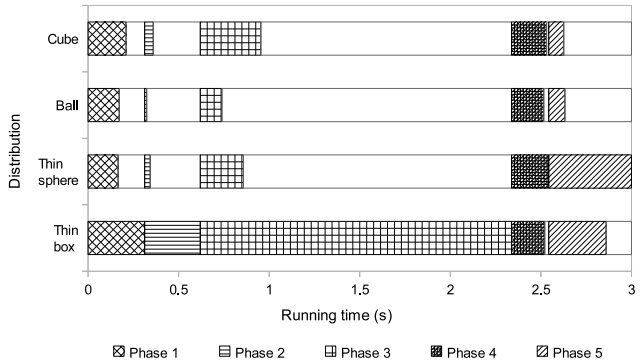


Figure 10: *The running time of different phases of gHull on different test cases.*

7 Concluding Remarks

We present gHull, a novel algorithm to compute the convex hull of a given point set in $\Re^3$ using the graphics processing units. Our algorithm exploits the relationship between the Voronoi diagram and the convex hull and utilizes the fast computation in the digital space to approximate that in the continuous space to maximize the parallelism available in the GPUs. With a careful design, gHull can be efficient yet remains accurate and robust. Our experiment on different test cases shows that gHull implemented on the CUDA programming model is an order of magnitude faster than QuickHull, the fastest convex hull software running in the CPU. With the faster growth in speed of the GPUs compared to that of the CPUs, and the larger amount of data to be processed, gHull offers an advantage over other CPU convex hull algorithms.

We have also discussed three main problems of our approach, and our solutions (partially) to these problems, to improve the performance of gHull. To further enhance the algorithm, instead of using star splaying to transform the approximate polyhedron to the desired convex hull, one can possibly explore the Star Flipping algorithm proposed by Shewchuk in the same paper [Shewchuk 2005]. However, obtaining a topologically correct polyhedron from the six slices of the digital Voronoi diagram is still a challenge.

References

AMATO, N. M., AND PREPARATA, F. P. 1993. An NC parallel 3D convex hull algorithm. In *SCG '93: Proc. 9th Symp. Computational geometry*, ACM, New York, NY, USA, 289–297.

AMUNDSON, N. R., CABOUSSAT, A., HE, J., AND SEINFELD, J. H. 2005. An optimization problem related to the modeling of atmospheric organic aerosols. *Comptes Rendus Mathematique 340*, 10, 765 – 768.

BARBER, C. B., DOBKIN, D. P., AND HUHDANPAA, H. 1996. The Quickhull algorithm for convex hulls. *ACM Trans. Mathematical Software 22*, 4, 469–483.

CAO, T.-T., TANG, K., MOHAMED, A., AND TAN, T.-S. 2010. Parallel banding algorithm to compute exact distance transform with the GPU. In *SI3D: Proc. Symp. Interactive 3D Graphics and Games*, 83–90.

CHAN, T. M. 1996. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete and Computational Geometry 16*, 361–368.

CLARKSON, K. L., AND SHOR, P. W. 1988. Algorithms for diametral pairs and convex hulls that are optimal, randomized, and incremental. In *SCG '88: Proc. 4th Symp. Computational geometry*, ACM, New York, NY, USA, 12–17.

DEHNE, F., DENG, X., DYMOND, P., FABRI, A., AND KHOKHAR, A. A. 1995. A randomized parallel 3D convex hull algorithm for coarse grained multicomputers. In *SPAA '95: Proc. 7th ACM Symp. Parallel algorithms and architectures*, ACM, New York, NY, USA, 27–33.

EDELSBRUNNER, H., AND MÜCKE, E. P. 1990. Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms. *ACM Trans. Graphics 9*, 66–104.

FUENTES, O., GULATI, R. K., AND ELECTRONICA, O. Y. 2001. Prediction of stellar atmospheric parameters from spectra, spectral indices and spectral lines using machine learning. In *Experimental Astronomy 12:1*, 21–31.

GUPTA, N., AND SEN, S. 2003. Faster output-sensitive parallel algorithms for 3D convex hulls and vector maxima. *Journal of Parallel and Distributed Computing 63*, 4, 488 – 500.

HAHN, H., AND HAN, Y. 2006. Recognition of 3D object using attributed relation graph of silhouette's extended convex hull. In *Advances in Visual Computing*, vol. 4292 of *Lecture Notes in Computer Science*. 126–135.

LIU, R., ZHANG, H., AND BUSBY, J. 2008. Convex hull covering of polygonal scenes for accurate collision detection in games. In *GI '08: Proc. Graphics Interface*, Canadian Information Processing Society, Toronto, Ont., Canada, Canada, 203–210.

MAO, H., AND YANG, Y.-H. 2006. Particle-based immiscible fluid-fluid collision. In *GI '06: Proc. Graphics Interface*, Canadian Information Processing Society, Toronto, Ont., Canada, Canada, 49–55.

MILLER, R., AND STOUT, Q. 1988. Efficient parallel convex hull algorithms. *IEEE Trans. Computer 37*, 12, 1605–1618.

NVIDIA. 2010. CUDA programming guide.

OKADA, K., INABA, M., AND INOUE, H. 2003. Walking navigation system of humanoid robot using stereo vision based floor recognition and path planning with multi-layered body image. In *IROS '03 Int'l Conf. Intelligent Robots and Systems*, IEEE, 2155 – 2160 vol.3.

PREPARATA, F. P., AND HONG, S. J. 1977. Convex hulls of finite sets of points in two and three dimensions. *Communication of ACM 20*, 2, 87–93.

SHEWCHUK, J. R. 1997. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. *Discrete & Computational Geometry 18*, 3 (Oct.), 305–363.

SHEWCHUK, J. R. 2005. Splaying: an algorithm for repairing delaunay triangulations and convex hulls. In *SCG '05: Proc. 21st ACM Symp. Computational Geometry*, Press, 237–246.

STRANDBERG, M. 2004. Robot path planning: An object-oriented approach. *PhD Thesis, KTH Royal Institute of Technology*.

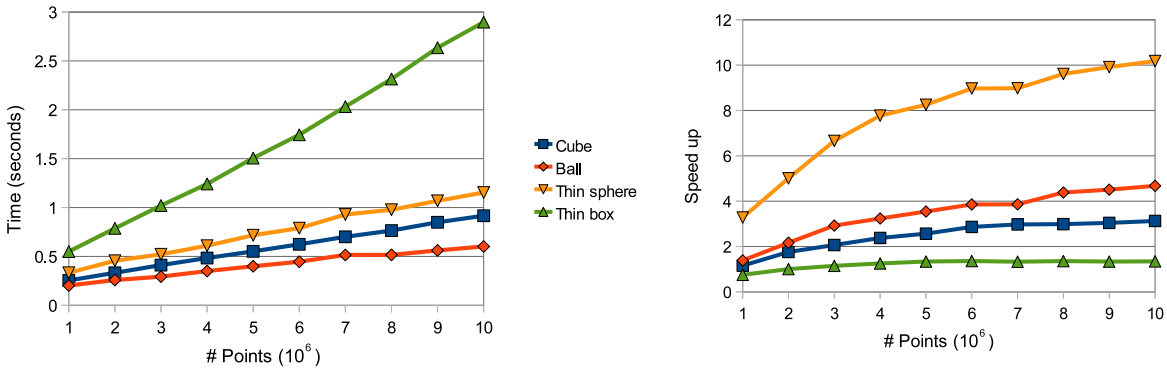


Figure 7: *Running time of gHull (left) and its speed up over QuickHull (right) on different test cases.*

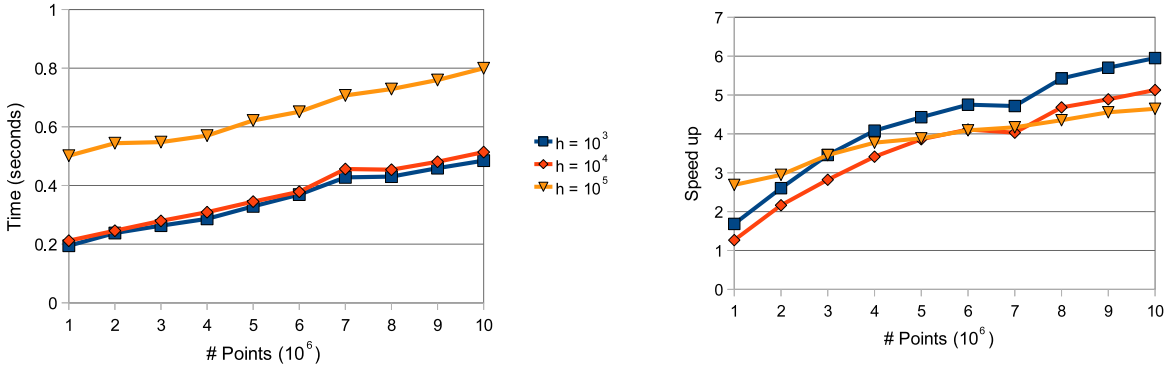


Figure 8: *Running time of gHull (left) and its speed up over QuickHull (right) while fixing the number of extreme vertices, h, and varying the total number of points, n.*

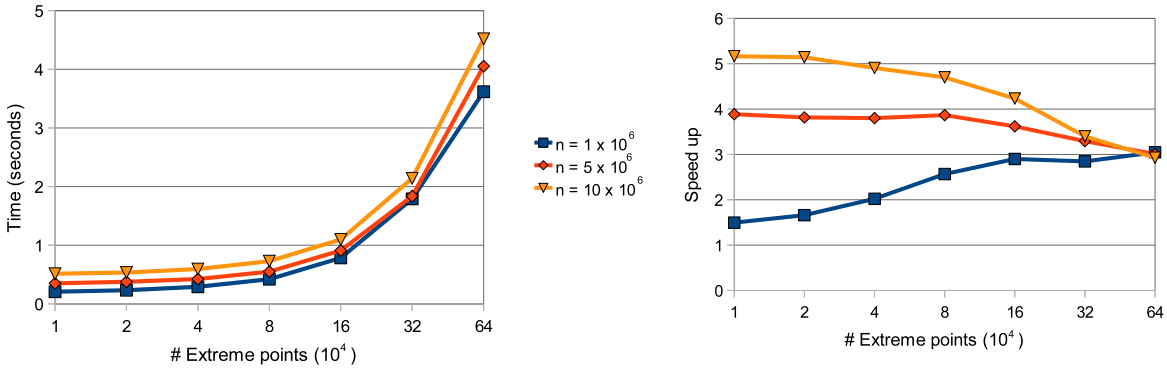


Figure 9: *Running time of gHull (left) and its speed up over QuickHull (right) while fixing the total number of points, n, and varying the number of extreme vertices, h.*

WANG, Y., LING-YUN, W., ZHANG, J.-H., ZHAN, Z.-W., XIANG-SUN, Z., AND LUONAN, C. 2009. Evaluating protein similarity from coarse structures. *IEEE/ACM Trans. Computational Biology and Bioinformatics 6*, 4, 583–593.