gHull: A GPU Algorithm for 3D Convex Hull

MINGCEN GAO, THANH-TUNG CAO, ASHWIN NANJAPPA, and TIOW-SENG TAN,
National University of Singapore
ZHIYONG HUANG, Institute for Infocomm Research Singapore

3

A novel algorithm is presented to compute the convex hull of a point set in $\mathbb{R}^3$ using the graphics processing unit (GPU). By exploiting the relationship between the Voronoi diagram and the convex hull, the algorithm derives the approximation of the convex hull from the former. The other extreme vertices of the convex hull are then found by using a two-round checking in the digital and the continuous space successively. The algorithm does not need explicit locking or any other concurrency control mechanism, thus it can maximize the parallelism available on the modern GPU.

The implementation using the CUDA programming model on NVIDIA GPUs is exact and efficient. The experiments show that it is up to an order of magnitude faster than other sequential convex hull implementations running on the CPU for inputs of millions of points. The works demonstrate that the GPU can be used to solve nontrivial computational geometry problems with significant performance benefit.

Categories and Subject Descriptors: I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modelling; I.3.1 [**Computer Graphics**]: Hardware Architecture

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: GPGPU, Voronoi diagram, star splaying

ACM Reference Format:
Gao, M., Cao T.-T., Nanjappa, A., Tan, T.-S., and Huang, Z. 2013. gHull: A GPU algorithm for 3D convex hull. ACM Trans. Math. Softw. 40, 1, Article 3 (September 2013), 19 pages.
DOI: http://dx.doi.org/10.1145/2513109.2513112

1. INTRODUCTION

The *convex hull* $\mathcal{C}(S)$ of a set S of input points in $\mathbb{R}^3$ is the smallest convex polyhedron enclosing S. Our problem 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*. The convex hull, along with the Delaunay triangulation and the Voronoi diagram (VD) are some of the most basic yet important geometric structures. In particular, the convex hull is useful in many applications and areas of research. In scientific visualization and computer games, convex hull can be a good form of bounding volume that is useful to check for intersection or collision between objects [Liu et al. 2008; Mao and Yang 2006]. In robotics, it is used to approximate robots and obstacles for the purpose of path planning [Okada et al. 2003; Strandberg 2004]. In astronomy, it is a basic structure used to analyze the characteristics of the atmosphere [Fuentes et al. 2001; Amundson

This research is supported by the National University of Singapore under grant R-252-000-337-112.
Project Web site: http://www.comp.nus.edu.sg/∼tants/gHull.html.
Author's address: M. Gao; email: mingcen@comp.nus.edu.sg.

DOI: http://dx.doi.org/10.1145/2513109.2513112

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].

Many CPU-based 3D convex hull algorithms have been developed and implemented over the decades. Among them, QuickHull [Barber et al. 1996] has been the most efficient and popular one in practice. There are also parallel convex hull algorithms [Miller and Stout 1988; Amato and Preparata 1993], but these are theoretical works with no practical implementations.

In recent years, the computational capability of the GPU has surpassed that of the CPU and is being used to solve large-scale problems such as physical and biological simulation. Some convex hull algorithms for the GPU are developed recently, but while in the 2D case, the problem is relatively simple, the 3D or high dimensions case poses a lot more challenges. Existing works in 3D either only accelerate part of the computation on the GPU, or produce output that is close but not always exactly the convex hull. This is because known parallel convex hull algorithms developed for traditional parallel machines do not seem to map well to the current GPU architecture that supports tens of thousands of computational threads. The main difficulty is that such computation generally needs "global" consideration of all input data, and thus does not map well to the massively multithreaded model of the GPU, which requires regularized work on localized data to achieve good performance. QuickHull, using an incremental insertion approach, is very difficult to be implemented efficiently on the GPU for $\mathbb{R}^3$ and higher dimensions, because there are many dependencies during the insertion of points.

In this article, we establish that the GPU is a useful tool to compute the convex hull in 3D with substantial speedup over sequential algorithms. The main idea of our proposed 3D GPU convex hull algorithm is to exploit the relationship between 3D Voronoi diagram and 3D convex hull so as to maximize the parallelism. By computing six slices of the 3D Voronoi diagram in digital space, all together forming a box enclosing the input point set, we derive a good approximation of the convex hull. The other extreme vertices neglected in that computation are found by a two-round checking in the digital and the continuous space successively. Our implementation uses simple data structures and does not require any explicit locking or concurrency control techniques and thus scales well with the number of cores on the GPU.

The article is organized as follows: Section 2 reviews the related work for sequential and parallel convex hull algorithms. Section 3 outlines our 3D convex hull algorithm, while Section 4 details our implementation techniques. Some issues regarding the use of digital space computation are discussed in Section 5. Our experimental results and analysis are presented in Section 6 and finally Section 7 concludes the article.

2. RELATED WORK

In this section we look at a few of the related sequential and parallel algorithms for convex hull. We also briefly discuss the star splaying algorithm [Shewchuk 2005] in $\mathbb{R}^3$ which we adapt to the GPU for our algorithm as explained later.

2.1. Convex Hull Algorithms for the CPU

The incremental insertion algorithm [Clarkson and Shor 1988] constructs the convex hull by inserting points incrementally using the point location technique. QuickHull [Barber et al. 1996] is a variant of such approach. Another technique is divide-and-conquer, which is used in the algorithm of Preparata and Hong [1977]. Both the incremental insertion and the divide-and-conquer approaches have a time complexity of $O(n \log n)$. In 2D and 3D, the optimal output-sensitive convex hull algorithm has a time complexity of $\Theta(n \log h)$ [Chan 1996] where h is the number of extreme vertices. There is no known efficient implementation of this algorithm. Empirically, QuickHull is found to have the same output-sensitive time complexity.

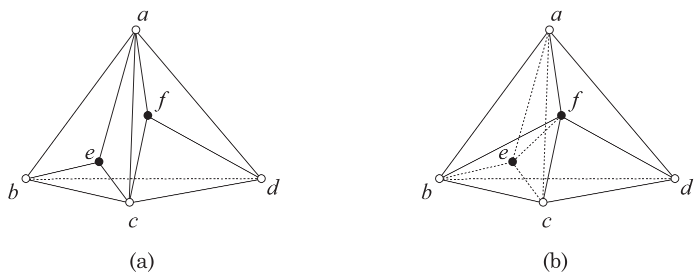


Fig. 1. An example in which CudaQuickHull outputs a wrong result. In (a), after creating the initial tetrahedron $abcd$, CudaQuickHull flags e with triangle abc and f is with triangle acd, and output both as extreme vertices. In the correct result in (b), e is not an extreme vertex since it lies inside tetrahedron $fabc$.

Because of the good time complexity and low overhead in practice, QuickHull has been a popular approach adopted by many applications over the years.

Parallel algorithms for computational geometry in general and convex hull in particular have been extensively studied in the last few decades. For example, Miller and Stout [1988] and Amato and Preparata [1993] describe $O(\log n)$ parallel algorithms for n points using $O(n)$ processors. These algorithms are only of theoretical interest, and yet to be of practical value as they have no known efficient implementation. One of the reasons is that these algorithms are complex, making them hard to scale on a fine-grained data-parallel massively-multithreaded architecture. For the current multi-core systems with a small number of independent processors, algorithms designed by Dehne et al. [1995] and Gupta and Sen [2003] may be applicable. These algorithms, however, do not have known implementations that demonstrate their usefulness.

2.2. Convex Hull Algorithms for the GPU

There has been a growing interest in using the GPU for computational geometry in recent years. Hoff et al. [1999] and Fischer and Gotsman [2006] compute the digital VD by adapting the traditional graphics pipeline of the GPU. An exact algorithm for such a computation is recently proposed by Cao et al. [2010], using the more flexible computation model available on newer generation GPUs. Rong et al. [2008] make the first serious attempt to compute the 2D Delaunay triangulation using the GPU. Qi et al. [2012] further improve this work to compute the 2D constraint Delaunay triangulation. However, in their works the mesh produced by the dualization is required to be free from topological and geometrical problems. Such a neat result is not known for 3D. Our approach of using the digital VD as an approximation to the continuous result resembles their approach.

Recently there are also some attempt at solving the convex hull problem on the GPU. Srungarapu et al. [2011] and Jurkiewicz and Danilewski [2011] propose two algorithms similar to QuickHull on the GPU that work for only the simple 2D case. Tzeng and Owens [2012] further extend that approach to 3D and higher dimensions. However, their algorithm, CudaQuickHull, only outputs a set of points, and very often the output contains nonextreme vertices; see Figure 1. Stein et al. [2012] propose to compute the convex hull in 3D by iteratively inserting points and flipping. Their claim is that it flips all reflex edges, thus the result is a convex hull. This, however, is not true as the algorithm prohibits flipping of concave edges if that causes self-intersection (as indicated in their algorithm), thus the final result might still contain reflex edges. The only known approach that can produce the correct convex hull in 3D with the help of the GPU is that of Tang et al. [2012]. In this work, the hull is grown on the GPU by iteratively inserting points from an initial tetrahedron. During the process, any

Fig. 2. Star splaying algorithm. (a) A star with its link (in bold). (b) Beneath-beyond insertion.

point found to be inside the hull is removed. Then, those points surviving the process are passed back to the CPU and a CPU-based algorithm (such as CGAL) is used to compute the convex hull. As pointed out by the authors, if most of the input points are extreme vertices, their algorithm is even slower than the CPU-based algorithm due to the time wasted on the (filtering) step on the GPU.

2.3. Star Splaying in $\mathbb{R}^3$

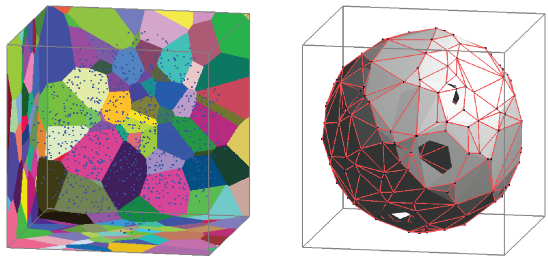
Star splaying [Shewchuk 2005] is a very efficient algorithm to repair convex hulls. Here we briefly outline the algorithm in $\mathbb{R}^3$ that we adapt to the GPU.

In $\mathbb{R}^3$, each vertex s of a polyhedron has a set of edges and triangles incident to it, referred to as the *star* of s. The set of edges opposite to s in the triangles of its star forms a ring, referred to as the *link* of s (Figure 2(a)). By extending the star of s to infinity, we get a *cone*. If the polyhedron is convex, then the cone of each of its vertices is convex too, and it encloses the rest of the vertices of the polyhedron.

The stars of the vertices of a polyhedron are consistent with each other. That is, if the star of s contains the triangle *stu*, then the stars of t and u also contain this triangle. Moreover, a set of consistent stars uniquely defines a surface triangulation. However, an arbitrary collection of stars not coming from a polyhedron may not be consistent with each other.

The star splaying algorithm is based on the idea that if the cones of all the points are made convex and their corresponding stars are made consistent, then these stars uniquely define the convex hull of the point set. Using the set of stars with their cones being convex, the algorithm repeatedly checks for each triangle *stu* in the star of s whether this triangle exists in the star of t (and u). If *stu* does not exist in the star of t, some points (s, u or both) will be inserted into t's star in an attempt to *splay* it to include the triangle. The insertion of a point into the star of t is done using the traditional beneath-beyond method [Kallay 1981] to guarantee that the cone of t is still convex (Figure 2(b)). If these insertions fail, implying that the triangle is enclosed by the cone of t, then some points from the link of t will be inserted into the star of s to splay it further to remove *stu*. In case a star of a point splays too wide that it cannot be contained in a half space, then that point is guaranteed not to be an extreme vertex. Such a star is called a *dead star*.

A nice feature of the star splaying algorithm is that creating stars having convex cones and enforcing their consistencies can both be done independently for each star. This is well suited for the parallel computation model of the GPU since stars can be checked and modified in stages without requiring any locking or concurrency control. We adapt star splaying to efficiently transform our approximation of the convex hull of S into $\mathcal{C}(S)$. In the following section, we describe our convex hull algorithm in more detail.



(a) (b)

Fig. 3. (a) Digital restricted VD. (b) Stars constructed from the digital restricted VD; they might not be consistent.

3. ALGORITHM OVERVIEW

The main idea of our algorithm is to utilize the relationship between the 3D Voronoi diagram and the convex hull computed from the same point set S. In particular, only the Voronoi cells of the extreme vertices of S are unbounded, that is, extend to infinity. 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 $\mathcal{V}(S)$ structure is harder to manage than the convex hull, and is just as expensive to compute. But, on the GPU the Parallel Banding Algorithm (PBA) [Cao et al. 2010] can compute the digital VD very efficiently and it is a good starting point to derive an approximation of $\mathcal{C}(S)$.

In our algorithm, we enclose the input point set S in a large enough box $\mathcal{B}$ that contains integer grid points, each corresponding to one unit cell of $\mathcal{B}$. We use the boundary (six faces) of $\mathcal{B}$ to capture the unbounded Voronoi cells of S, meaning we compute only six slices of the 3D Voronoi diagram. Theoretically, if $\mathcal{B}$ is large enough, dualizing $\mathcal{V}(S)$ restricted to the faces of $\mathcal{B}$, that is, the *restricted VD* (Figure 3(a)), gives us $\mathcal{C}(S)$. However, since the VD we compute is in digital space, and due to the finite size of $\mathcal{B}$, we can only obtain an approximation of the convex hull. We apply the star splaying algorithm [Shewchuk 2005] to transform the approximation into the solution. Our algorithm can be split into five steps.

Step 1: Voronoi Construction. Compute the six 2D slices of the 3D digital VD of S on the boundary of $\mathcal{B}$. Let S' be the set of points whose Voronoi regions appear on it.

Step 2: Star Creation. We first dualize the restricted VD to obtain for each point s in S' a set of neighbors, called the *working set* of s. Then we use that to construct a *convex cone*, represented as a *star*, for the point.

Step 3: Hull Approximation. Apply the star splaying algorithm to obtain the convex hull $\mathcal{C}(S')$.

Step 4: Point Addition. Collect points of S that lie outside $\mathcal{C}(S')$ and for each of them construct its star using its nearby vertices of $\mathcal{C}(S')$.

Step 5: Hull Completion. Perform star splaying again on our approximation to transform it into $\mathcal{C}(S)$.

3.1. Step 1: Voronoi Construction

Our aim is to approximate $\mathcal{V}(S)$ restricted to the six faces of $\mathcal{B}$. We first translate and then scale the input points such that their bounding box fits inside a 3D grid $\mathcal{B}$ consisting of grid cells. Then, we compute the digital VD of S intersecting each side of the boundary of $\mathcal{B}$ on the GPU. For each side, we project the points onto it, recording one nearest point among those that fall onto the same 2D grid cell. The two coordinates of a point are shifted to the center of the nearest 2D grid cell, while the third coordinate (the distance to the side we are projecting on) is unmodified. We then apply PBA to compute the digital VD.

3.2. Step 2: Star Creation

We dualize the restricted VD obtained in the previous step to get a set of triangles. The corners of grid cells are grid vertices, each of which is incident to a maximum of 4 different Voronoi regions. Each grid vertex incident to 3 or 4 different Voronoi regions is dualized into one or two nonintersecting triangles respectively.

Ideally, dualizing the restricted VD of S on a closed box results in a 3D polyhedron, not necessarily convex, approximating $\mathcal{C}(S')$. However, in the digital restricted VD, a Voronoi cell can be, for example, disconnected, resulting in the dualized polyhedron having holes or duplicated triangles. Instead of constructing a polyhedron, we only record the information on the adjacencies of the Voronoi cells. For each triangle abc thus obtained, we add b and c to the working set of a, and similarly for b and c.

For each s in S' in parallel, we create its star (in the continuous space) from its working set such that its cone is convex (Figure 3(b)). Each GPU thread handling a point s first creates an initial star from 3 points in the working set, and then incrementally inserts the rest using the beneath-beyond algorithm.

3.3. Step 3: Hull Approximation

Star splaying is an iterative approach. Stars are repeatedly checked for inconsistency, and insertions are performed using the beneath-beyond algorithm to splay the stars when needed.

To perform the star splaying algorithm in parallel while achieving regularized work for different GPU threads, we carry out the consistency checking and the insertions of points in two separate steps, alternately performed until all the stars are consistent. For ease of implementation, the consistency checking is performed on each edge of each star in parallel, rather than on triangles. An edge is *consistent* if the stars of its two endpoints have the same two triangles incident to this edge; otherwise, it is *inconsistent*. Any inconsistency can generate up to 4 insertions, two each from the link of this edge on each star. The insertion step is done by first sorting the set of insertions by the indices of the stars they are destined for, and then each parallel thread can perform the insertions into a star independent of others.

It is possible to use a CPU sequential algorithm to compute $\mathcal{C}(S')$ when the set S' derived from the digital restricted VD is small. However, since the stars constructed from the previous step are already very close to $\mathcal{C}(S')$, a parallel implementation of the star splaying algorithm on the GPU gives much better performance.

3.4. Step 4: Point Addition

Due to $\mathcal{C}(S')$ being an approximation, it may not contain all extreme vertices of S. We use $\mathcal{C}(S')$ to check the points in S and remove those that are inside the hull. The rest of the points can potentially be extreme vertices. This is the reason why we perform star splaying in Step 3. We first perform the checking in digital space by rendering the triangles of $\mathcal{C}(S')$ with the view direction orthogonal to each side of $\mathcal{B}$ in turn. Then, we use a depth test to eliminate points that clearly lie inside $\mathcal{C}(S')$. Each GPU thread handling a point s in S projects s onto each side of $\mathcal{B}$ and compares its depth value d_s with the value d on the depth map on that side (with depth value increasing in the viewing direction). If $d_s - d \leq \tau$ where τ denotes the threshold, then s is potentially an extreme vertex. The depth test is done in digital space, so a conservative threshold (which is equal to 1 pixel width) is used to safely remove nonextreme vertices; see Section 5.1.

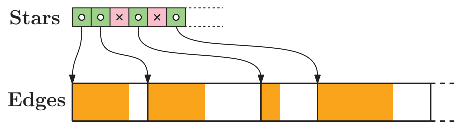


Fig. 4. Data structures for stars and edges. × indicates a *dead* star.

To further eliminate nonextreme vertices, we perform another round of checking in continuous space. For each point s that passes the depth test, we also record a triangle A that covers its projection in one of the viewing directions. Notice that A is close to s. Pick an arbitrary point r on $\mathcal{C}(S')$. The point s is either beyond one of the triangles in the star of r, or the ray $\vec{rs}$ intersects $\mathcal{C}(S')$ at a triangle not in the star of r. Using a technique similar to point location, starting from A we can quickly find such a triangle B, and accurately determine whether s is inside or outside $\mathcal{C}(S')$. If s is outside, we use B to form a star for s, otherwise it is eliminated. All this computation can be done on each point independently in parallel. The new stars together with $\mathcal{C}(S')$ form an approximation of $\mathcal{C}(S)$.

3.5. Step 5: Hull Completion

Our approximation now contains all possible extreme vertices of S and a few more. By performing star splaying again, as detailed in Step 3, we transform our approximation into the convex hull of S.

4. IMPLEMENTATION DETAILS

We implement our algorithm using the CUDA programming model and OpenGL on NVIDIA GPUs. Due to the nature of the GPU, it is more efficient to allocate memory in large chunks rather than dynamically allocate many small blocks. For our implementation, we use two lists to store the description of the stars and their edges, called the *star list* and the *edge list*, as shown in Figure 4. Each star has a contiguous chunk of memory whose size is enough to store all of its current edges plus a certain amount of free space. Each star records the coordinates of its point, its status (whether it is dead or not), the number of edges, the size 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 other endpoint, and a flag for checking of consistency. The edge list of a star represents its link vertices in counterclockwise order.

The difficulty 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 an expansion occurs we also use the opportunity to shrink or expand the storage of all the stars to maintain some free space (say 20%) for each star. This helps to reduce the number of times we need to reallocate the edge list. Note that the size of the free space does not affect the performance much, as we observe in our experiments, because the list expansion is relatively cheap compared to the other processing in the GPU. Also, since we start star splaying with a good approximation of the convex hull, the stars typically do not grow drastically.

4.1. Step 1: Voronoi Construction

Before applying the PBA, we need to project the points on the six sides of the box $\mathcal{B}$. This operation entails a lot of random atomic memory accesses to the global memory that are highly inefficient on the GPU. Instead, we perform all the projections in the GPU shared memory to speed up this step.

$\mathcal{B}$ is partitioned into bricks, each of size $k \times k \times k$. For each point in S, we find the brick that encloses it using a CUDA kernel. We accumulate the points that belong to the same brick into a contiguous chunk using a CUDA radix sort [CUDPP 2012]. We identify the starting offset of each such chunk in the sorted list using another CUDA kernel.

We use six textures to store the projections on the six sides of $\mathcal{B}$. For each $k \times k$ tile of a texture, we use a block of CUDA threads to process the points enclosed in the bricks that project onto this tile. Algorithm 1 shows the details of projecting these points onto the tile in each block of one kernel. When many points project onto a single pixel, we store the point closest to the tile by using the *atomic minimum* operation. This is applied on a shared memory array of the block and thus is highly efficient compared to using it on global memory. k can typically be chosen to be 32, so that the $k \times k$ tile can fit in shared memory. Though we use floating point for the point-to-tile distance, we can still use the CUDA integer atomic minimum operation. This is because positive floating point numbers can be compared as integers, with the same binary representation, without affecting their order. The result of these projections is coherently written out to global memory to apply PBA and obtain the restricted VD.

ALGORITHM 1: Projecting a chunk of points onto a tile in each block of one kernel

```
declare a k × k real number array A in shared memory corresponding to the tile
for each element of A do in parallel
    initialize the element using maximum real number
synchronize threads
for each point of the chunk do in parallel
    find the pixel and the element of A corresponding to the point
    compute the distance between the point and the pixel
    use atomic minimum operation to store the distance into the element
synchronize threads
for each element of A do in parallel
    output the value into global memory
for each point of the chunk do in parallel
    find the pixel and the element of A corresponding to the point
    compute the distance between the point and the pixel
    if the distance is equal to the value of the element then
        output the point ID into global memory
```

4.2. Step 2: Star Creation

We construct the working set for each point by scanning the resulting VD textures constructed in the previous step. For each triangle identified, we generate 6 pairs of its vertices, each pair (a, b) indicating that b is in the working set of a. First, we let one CUDA kernel count the number of pairs generated by each grid corner. Next, we pre-allocate an array to store the pairs, and use a CUDA parallel prefix sum primitive [CUDPP 2012] to compute for each corner the offset in the array to store its pairs. After that, we call another kernel to scan the textures again, generating the working set pairs. Lastly, we sort the list of pairs using a CUDA radix sort, remove duplicates, and identify the working set for each point as a contiguous chunk of pairs.

Based on the working sets thus constructed, we allocate the storage for the star list and the edge list. A CUDA kernel is used to construct an initial star consisting of 3 link points for every point in S'. Each CUDA thread constructing an initial star takes 3 points from its working set, checks the 3D orientation, and stores these points in the edge list of that star in counterclockwise order.

After that, the rest of the working set of each point is inserted into its star in a single kernel. Each CUDA thread processes the working set of a point, independent of other points. For each insertion of t into s, we go through the star of s, identifying a (continuous) series of beneath triangles, removing their corresponding edges and inserting t into the edge list of s accordingly.

The beneath-beyond insertion relies heavily on the 3D orientation predicate. It is important that the predicate is computed exactly and co-planar cases are handled correctly. More importantly, the predicate should give the same result when checked from different stars for the star splaying algorithm to converge. In order to achieve this, all our predicates are performed with the Simulation of Simplicity (SoS) technique [Edelsbrunner and Mücke 1990] and exact arithmetic [Shewchuk 1997]. The flexibility of the CUDA programming model makes such complex computation possible on the GPU.

4.3. Step 3: Hull Approximation

In this step the star splaying algorithm is adapted for the GPU. The pseudocode of the star splaying implementation on GPU is outlined in Algorithm 2. In Line 3, we do a parallel stream compaction on the edge flags to obtain the list of edges to be checked for consistency. Each inconsistent edge can potentially lead to up to four insertions into different stars (see Section 2.3). We pre-allocate storage for these possible insertions in Line 4. In Line 5, we use a CUDA kernel where each thread processes one edge.

ALGORITHM 2: Star splaying on the GPU

```
1:  flag all edges to be checked for consistency
2:  repeat
3:     collect the edges that are flagged
4:     allocate space for possible insertions
5:     check the flagged edges and generate insertions
6:     sort and compact the list of insertions
7:     if a star needs more space then
8:        expand the edge list
9:     perform the insertions to splay stars
10:    flag edges that need to be checked in the next iteration
11: until there are no more flagged edges
```

The insertions are sorted and compacted in Line 6 and duplicates are removed. Each star then checks if it has enough free space in its edge list and the edge list is expanded if needed (Line 7 and Line 8). This expansion is done by computing the required space for each star using a kernel, allocating a new edge list, and then copying the edges over. The insertions (Line 9) are performed similar to those in Step 2. In Line 10, we flag all newly created edges. Also, during the insertions, if an edge ab in the star of a is deleted, then the edge ba in the star of b, if any, needs to be flagged too.

4.4. Step 4: Point Addition

The first round of checking in this step is carried out in OpenGL, which works seamlessly with other steps done in CUDA. As we keep edges rather than triangles, we first use a CUDA kernel to 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. Similar to other steps, we first count, then use parallel prefix sum to compute the offset before actually generating the triangle list.

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. After the rendering, the depth buffer is processed by a CUDA kernel. Each thread processing a point in $S - S'$ checks the depth value to see whether the point can potentially be outside or not. If outside, this point becomes a candidate for the next round of checking.

In the second round of checking, we use one CUDA thread to check one candidate found in the previous round. Let the candidate be s and the corresponding point recorded at the projection of s in the color buffer be c. Also, let r be an arbitrary point in S' where $r \neq c$. In order to determine the triangle B in $\mathcal{C}(S')$ that is intersected by the ray $\vec{rs}$, we start walking from c. Each vertex t on the link of c together with the line rc forms a plane, and we are interested in the half-plane defined by rc that contains t. The collection of these half-planes partitions the space into several unbounded subspaces around rc; one of these subspaces contains s, which can be identified using 3D orientation checks. This subspace tells us which vertex on the link of c gets us closer to s, until we reach B. Specially, when r is in the star of c, it is possible that none of the subspaces contains s. In this case, s must be beyond one of the two triangles incident to r in the star of c, and we select the first triangle from the star of r as B. After that, using one more 3D orientation check, we can determine accurately if s is outside $\mathcal{C}(S')$, in which case the three vertices of B form the initial star of s. Algorithm 3 shows the pseudocode of the kernel for the second checking.

ALGORITHM 3: Second round of checking in one kernel

for each candidate point s **do in parallel**
  read the corresponding point c from color buffer
  repeat
    $p \leftarrow \emptyset, q \leftarrow \emptyset$
    for each triangle $\triangle cvv'$ in the star of c **do**
      if r is one of v and v' **then** continue
      if s is beyond rcv and beneath rcv' **then**
        $p \leftarrow v, q \leftarrow v'$
        break the loop
    if $p \neq \emptyset$ **and** $q \neq \emptyset$ **then**
      if s is beyond rvv' **then**
        $B \leftarrow \triangle cvv'$
      else $c \leftarrow v$
    else $B \leftarrow$ the first triangle in the star of r
  until the triangle B is found
  if s is beyond B **then**
    construct initial star of s using B
  else label s as nonextreme vertex

5. DIGITAL APPROXIMATION ISSUES

In this section, we discuss some issues of the use of digital space computation to approximate that in the continuous space.

5.1. Digital Depth Test

In Step 4, we use the six sides of the boundary of $\mathcal{B}$ as the 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

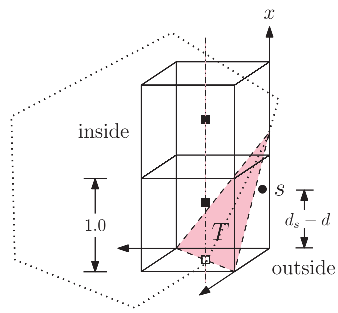


Fig. 5. The digital depth test of a point s against a triangle T on the boundary of $\mathcal{C}(S')$ when s is outside $\mathcal{C}(S')$.

the projection of s is actually the depth value of the center of the cell containing this projection. Depending on the triangle covering that projection, $(d_s - d)$ can be arbitrarily large; see Figure 5. The following claim shows that as long as we keep every point s that has $(d_s - d) < 1$ in one of the projections, we do not miss any point outside $\mathcal{C}(S')$. This tight bound allows us to throw away most of the points that are inside $\mathcal{C}(S')$.

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 $\mathcal{B}$, we have $(d_s - d) \leq \tau$ where $\tau = 1$ pixel width.*

PROOF. The point s is inside a unit cell of $\mathcal{B}$ whose center is the grid point $(\bar{x}, \bar{y}, \bar{z})$. The coordinates 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 cells containing the projections of s in different viewing directions, and the plane equation of T be $ax + by + cz + K = 0$. Without loss of generality we assume that $a \geq b \geq c$.

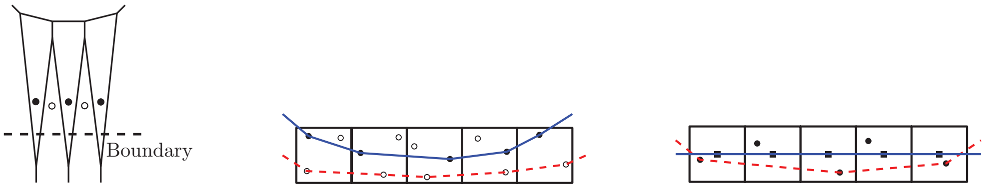
Since T appears in the depth buffer, and $\mathcal{C}(S')$ is convex, T must be visible from three different viewing directions. This forms a coordinates system in which the plane equation of T has $a, b, c \geq 0$. In the viewing direction along the positive x-axis, $d_s = \bar{x} + \delta_x$ and d 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:

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

It is possible that the depth values of s used in 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 covers 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 $(d_s - d') \leq 1$, as required. □

5.2. Convex Hull Approximation

Due to the nature of digital space and that of our approach, there are three issues that can affect the performance of our algorithm: *slicing problem*, *underapproximation problem*, and *overapproximation problem*; see Figure 6. In the experiment, we will



(a) Slicing (b) Underapproximation (c) Overapproximation

Fig. 6. Three problems associated with the computation in digital space.

show that these problems affect the efficiency of the algorithm in different cases and manners.

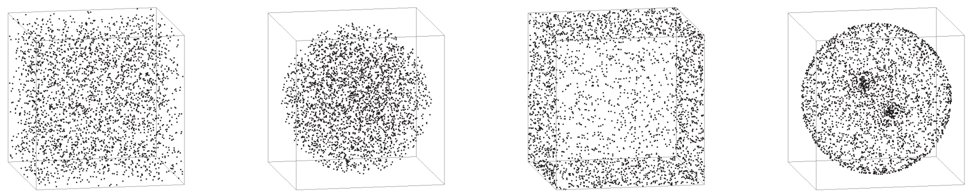
Slicing problem. This problem is the result of using a bounded box $\mathcal{B}$ to find the Voronoi cells that are unbounded. As some of the bounded cells can extend beyond $\mathcal{B}$, they are captured although they do not correspond to extreme vertices. Figure 6(a) shows a 2D example where among the five cells being captured, only those of the round white points are unbounded. To reduce the number of wrongly captured Voronoi cells, we scale the point set to a slightly smaller volume inside $\mathcal{B}$ when performing Step 1.

Underapproximation problem. When we have multiple points projected to the same pixel, we can only record one point, and thus there are potentially many more points outside $\mathcal{C}(S')$. See Figure 6(b) for a 2D illustration where the round black points are kept, the solid line denotes part of $\mathcal{C}(S')$ and the dashed line denotes part of $\mathcal{C}(S)$. The round white points are missing points, many of which are outside $\mathcal{C}(S')$. By using a very efficient depth test in Step 4 of our implementation and accurate location of a nearby triangle for every point outside $\mathcal{C}(S')$, we are able to construct a very good star for that point. This reduces the amount of splaying needed in Step 5.

Overapproximation problem. This problem is caused by the shifting of points in Step 1. In certain cases, for example when points are distributed near the surface of a cube axis-aligned with $\mathcal{B}$, many points that are extreme vertices are shifted inward, while many nonextreme vertices are shifted outward and are legitimately captured in Step 1. This possibly leads to a lot more points captured in Step 1 and need to be removed in Step 3. See Figure 6(c) for a 2D illustration, where after Step 1 all the round black points, after shifted to the square black grid points, are captured. In our implementation, for each side of $\mathcal{B}$ we only shift 2 coordinates of the points while keeping the third one untouched. This produces a much better approximation of the restricted VD and thus reduces the effect of this problem.

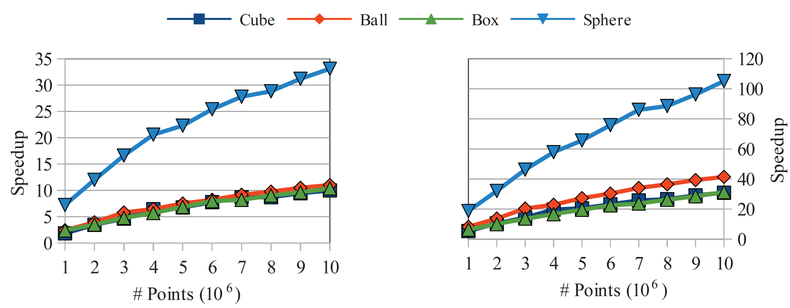
6. EXPERIMENTAL RESULTS

Our algorithm is implemented using the CUDA programming model by NVIDIA, and can easily be ported to OpenCL. All the experiments are conducted on a PC with an Intel i7 2600K 3.4GHz CPU, 16GB of DDR3 RAM and an NVIDIA GTX 580 Fermi graphics card with 3GB of video memory, unless otherwise stated. Visual Studio 2008 and CUDA 4.0 Toolkit are used to compile all the programs, with all optimizations enabled. We compare the performance of our implementation, called gHull, with the two fastest sequential implementations of the Quickhull algorithm: Qhull [2012] and CGAL [2012]. Qhull handles roundoff errors from floating point arithmetic by generating a convex hull with “thick" facets: any exact convex hull must lie between the inner and outer plane of the output facets. On the other hand, CGAL uses exact arithmetic, which is similar to our implementation. We adopt the exact arithmetic [Shewchuk 1997] on the GPU and apply the Simulation of Simplicity (SoS) technique [Edelsbrunner and Mücke



(a) Cube (b) Ball (c) Box with thickness of 0.01 (d) Sphere with thickness of 0.01

Fig. 7. Four distributions of tested data.



(a) Qhull (b) CGAL

Fig. 8. Speedup of gHull over (a) Qhull and (b) CGAL.

1990] for gHull to guarantee its exactness. As such, for input with no degeneracy, the output of gHull is the same as that of CGAL. When degeneracy occurs, the output of gHull is one of several possible correct result, which can be different from that of CGAL since CGAL uses a different perturbation method. In our experiment, we found that CGAL always runs slower than Qhull due to its use of exact arithmetic.

All the results below of gHull are based on the same set of parameters: grid size 1024^3 (i.e., each slice is of size 1024^2), while point set is scaled to 80% of the volume of $\mathcal{B}$. The rendering buffer in Step 4 is fixed at 512^2. Using a larger grid size gives a better approximation at the cost of slower VD computation, so it gives little running time improvement. A larger buffer for the depth test is also not desirable, since it incurs extra rendering cost.

Representative data. We generate points randomly with coordinates between [0.0, 1.0]. Points are distributed uniformly in four distributions: a cube, a ball of radius 0.5, a box with thickness of 0.01, and a sphere with thickness of 0.01; see Figure 7. The cube distribution has very few points on the convex hull, while many points inside can easily be removed by the Quickhull algorithm. The ball distribution is similar, but with a bit more points on the convex hull. The box distribution also has very few extreme vertices, but points are distributed close to the convex hull, so it is harder to eliminate them. The sphere is the extreme case where many points are on the convex hull, while the rest of them are also close to it. These synthetic test cases are highly representative for testing and stressing our algorithm.

The speedup of gHull over CGAL and Qhull are presented in Figure 8. In general, gHull is $2\times$ to $10\times$ faster than Qhull, and is $2\times$ to $40\times$ faster than CGAL for the cube, ball and box distributions. Notably, for the sphere distribution where not only there are many extreme vertices but there are also many points close to the convex hull, gHull

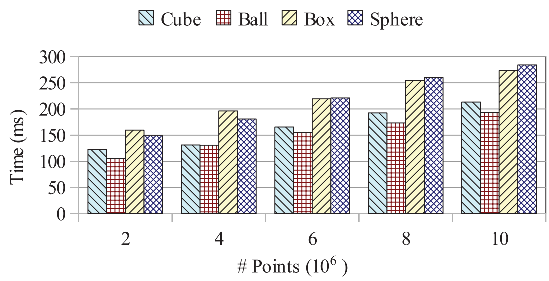


Fig. 9. Running time of gHull on different test cases.

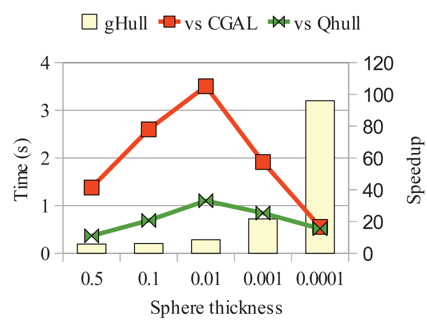


Fig. 10. Running time of gHull on spheres of different thickness.

is up to 100× faster than CGAL and up to 30× faster than Qhull, even with all the computation being exact. This is mainly because our digital restricted VD gives a very good approximation, which our star splaying implementation can quickly transform to a convex hull, and Step 4 of our algorithm is also very fast in eliminating nonextreme vertices.

In comparison, the approach of Tang et al. [2012] achieves 22× speed-up for the ball distribution with 10^7 points, while our speed-up is 40x. Note also that their experiment is done on a similar GPU as ours but with a slower CPU, so CGAL runs slower in their system. According to the authors, their speed-up drops when more points are extreme vertices, because the GPU process becomes less effective. On the other hand, our algorithm achieves an even higher speed-up when running on the sphere distribution with many points on or close to the convex hull. Their implementation is not available for us to do a more thorough comparison.

As an example, for 10^7 points in the sphere distribution, Step 2 returns around 54,000 points, of which Step 3 keeps 32,000. Step 4 then adds 57,000 points, and the final convex hull after Step 5 has 55,000 points. These are very small numbers compared to the number of input points. Similar performance can be observed for other point distributions, as Figure 9 shows similar running time of gHull for the same number of points.

Sensitivity Analysis. The use of the digital approximation is affected by how close the points are to each other and to the convex hull. We investigate this in Figure 10. Here we show the running time and speedup of gHull on the sphere distribution with thickness varying from 0.5 (a ball) to 0.0001 for 10^7 points. The running time increases as the sphere gets thinner, especially at thickness 0.0001 which is only 0.1 pixel width given that we use a 1024^3 grid size. The speedup over the CPU implementations initially

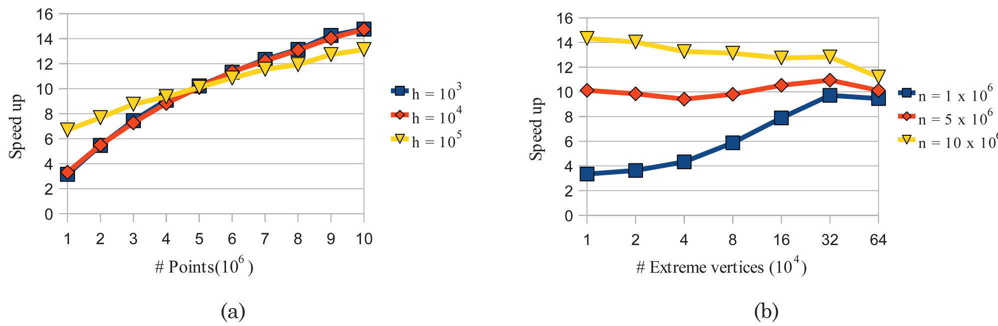


Fig. 11. The speed-up of gHull over Qhull while (a) fixing the number of extreme vertices h, and varying the total number of points n, and (b) fixing the total number of points and varying the number of extreme vertices.

increases as the Quickhull algorithm becomes less effective in eliminating nonextreme vertices, then decreases but is still 20× faster.

Scalability on the number of extreme and nonextreme vertices. In order to investigate the effect of the number of extreme vertices and nonextreme vertices on the performance of the algorithm, we use a different ball distribution, called controlled ball, 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 11(a) shows the speedup of gHull over Qhull when we fix h and vary n in the range of 10^6 to 10^7. As n is larger, the speedup increases from 4× to 15×. Note that the speedup with $h = 10^3$ and $h = 10^4$ is very close, while is slightly lower when $h = 10^5$. This is the consequence of the underapproximating problem when the texture cannot capture all the extreme vertices due to its limited size.

On the other hand, Figure 11(b) shows the speedup of gHull over Qhull when we fix n and vary h multiplicatively from $2^0 \times 10^4$ to $2^6 \times 10^4$. For $n = 10^6$ the speedup increases as h becomes larger, because gHull can quickly capture extreme vertices and remove nonextreme vertices compared with Qhull. Also, speedup is higher for larger n, as reflected earlier in Figure 11(a). On the other hand, the speedup slightly decreases for $n = 10 \times 10^6$ as h becomes larger because of the underapproximating problem. The explanation here is that when n is larger, there are more points near the convex hull boundary, with multiple points falling on the same pixel. As such, many extreme vertices are not captured.

A similar behavior can be observed when compared with CGAL, with the speedup being 3× to 4× better.

Scalability on the number of cores. Since our algorithm is designed for regularized computation on localized data, it is expected that it scales well with the number of cores on the GPU. This is confirmed in Figure 12. Here we run gHull on 10^7 points on different graphics cards: a GTS 450 with 192 cores, a GTX 460 with 336 cores and a GTX 580 with 512 cores. Clearly our implementation scales well with the number of cores, achieving close to 3× speedup when the number of cores increases by 3×, in all the different distributions. When we normalize the running time on different cards by multiplying it with the number of cores and their clock-speed (1.7GHz, 1.4GHz, and 1.6GHz respectively), we get roughly the same results, with newer generation GPUs being slightly faster than the older ones. The observation is true for other test cases with different number of points that we tested, as long as the number is not too small.

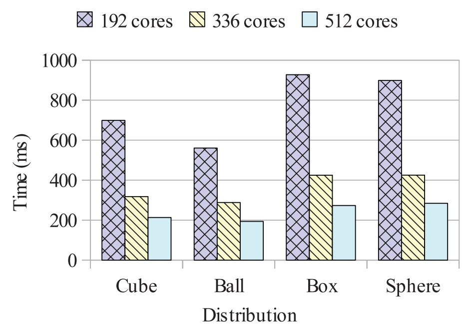


Fig. 12. Running time of gHull on the GPUs with different number of cores.

Table I. Running Time on Different 3D Models

Model	# Points (millions)	Running time (ms)		
		Qhull	CGAL	gHull
Asian dragon	3.6	540	1181	150
Thai statue	5.0	692	1538	168
Lucy	13.9	1884	4488	266

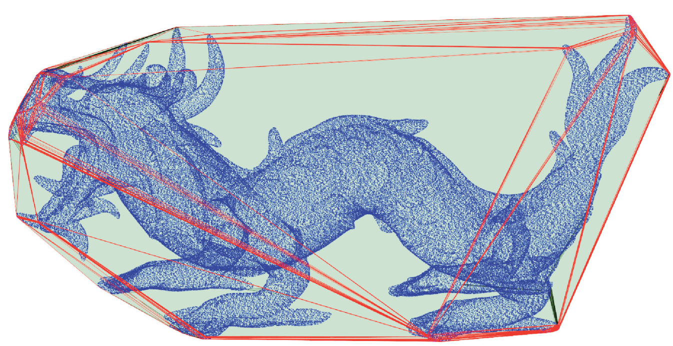


Fig. 13. Convex hull of the Asian dragon model.

Popular 3D Models. See Table I. We use models of over a million points from the Stanford 3D scanning repository [Stanford 2012]. Figure 13 shows one sample data set and its convex hull. Points of these models are densely distributed on the surface, while their convex hulls have very few points (only around 1,000). Nevertheless, gHull still manages to out-perform Qhull 4× to 7× and CGAL by 8× to 17×.

Process analysis. Figure 14 shows the running time of each step of our algorithm on different point distribution with 10^7 points. As expected, the behavior differs on different distributions. While the running time of Step 1 and Step 4 remains the same since it is not affected by how the points are distributed, the running time of other steps varies significantly. Step 3 takes more time on the box distribution due to the overapproximation problem, while Step 5 takes more time on the sphere distribution due to the underapproximation problem. Step 2 only takes a small portion of running time for all distributions.

To test the scalability of each step of our algorithm, we run gHull on the controlled ball distribution as described earlier. We also accumulate the time spend in different GPU primitives such as prefix sum, compaction and radix sorting. In Figure 15(a), we fix the number of the extreme vertices at 10^4 and vary the total number of points from 10^6 to 10^7. Here we observe that Step 1 and Step 4 dominate the running time

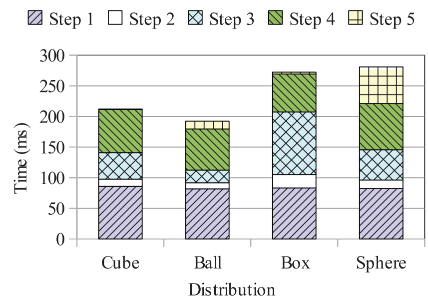


Fig. 14. Running time of gHull on different steps.

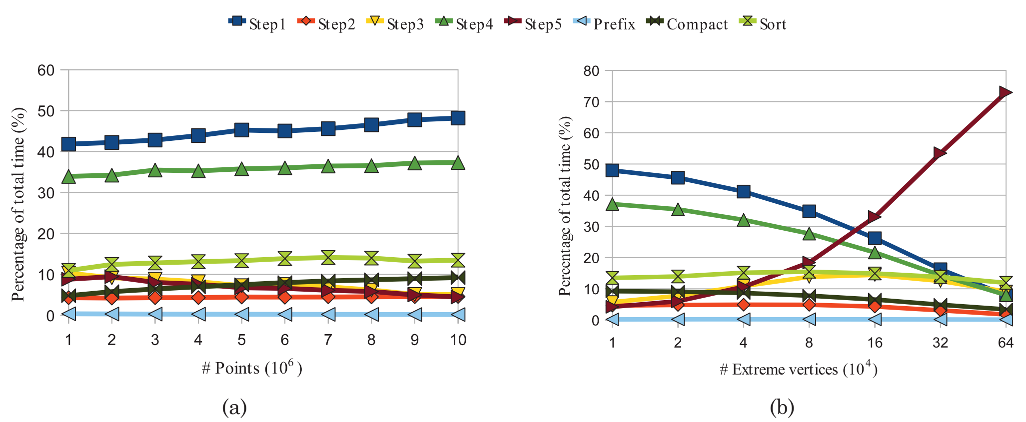


Fig. 15. The running time percentage of different steps as well as of some of the important GPU primitives in gHull. (a) We keep the number of extreme vertices at 10^4 while varying the number of points. (b) We keep the number of points at 10^7 while varying the number of extreme vertices.

and their percentages slightly increase, since the work in these two steps are directly proportional to the number of input points. The running time percentages of Step 3 and Step 5 actually decrease since the actual running time is almost only depends on the number of extreme vertices. Note that the running time of Step 1 is dominated by the cost of projecting the points onto the boundary, while the VD computation is almost constant. The time spent in the primitives is quite insignificant, with only the sorting taking more than 10% of the total time, and its percentage also slightly increases when the number of points increases.

In Figure 15(b), we vary the number of the extreme vertices from $2^0 \times 10^4$ to $2^6 \times 10^4$ with the total number of points being 10^7. Here we see the same behavior as the above case. Step 1 and Step 4 drop in percentages since their timing is almost stable when the total number of points are stable, while the timing of Step 3 and Step 5 increases along with the increases of the number of extreme vertices. The running time of Step 5 increases the most because when the extreme vertices are too many compared to the size of the grid, the underapproximation problem becomes serious.

Correctness of degenerate cases. We also test gHull using two extreme cases: the first case is when all input points are almost coplanar, and the second case is when all input points are almost colinear. For both cases, gHull correctly produces one of the multiple possible convex hull of the input.

Limitation. While the digital space allows us to perform most processing in parallel with regularized work and localized data, its limitation to approximate the computation in the continuous space lies in its uniformity. It is possible to design a test case where points are badly distributed (e.g., points arranged on a thin line convoluted in the space), resulting in a bad digital approximation, and thus lower overall performance. Such a case, however, is not common in practice. The next issue with our approach is with the use of SoS during the star splaying process. Since our digital approximation cannot take into account the perturbation, the resulting approximation might be different from the final result with SoS, especially when there are many perfectly coplanar points on the convex hull. In this case, the algorithm performs a large amount of exact computation for no good purpose. Lastly, our current implementation requires at least 3 times more memory compared to CGAL and Qhull. Part of the reason is because in order to achieve a very high level of parallelism, our algorithm needs to use some large textures and maintain several auxiliary arrays for the parallel primitives. Also, the data structure used during the star splaying process is more costly than the standard triangulation data structure used in other approaches.

7. CONCLUDING REMARKS

This article introduces an algorithm to compute the convex hull in 3D using the GPU. By first computing an approximation of the solution, our algorithm converts the given problem into one that is easier to process concurrently using a massive number of threads performing regularized work on localized data. With a careful design, our implementation is efficient yet remains exact and scalable to the number of GPU cores. To our knowledge, our algorithm is the first for the 3D convex hull problem that is fully accelerated on the GPU while producing exact result. Our experiment on different test cases shows that our implementation on CUDA is more than an order of magnitude faster than the best sequential CPU implementations. Further, our approach may be used for the convex hull and its related problems in higher dimensions such as to construct the 3D Delaunay triangulation.

REFERENCES

AMATO, N. M. AND PREPARATA, F. P. 1993. An NC parallel 3D convex hull algorithm. In *Proceedings of the 9th Symposium on Computational Geometry (SoCG '93)*. ACM, New York, 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. Math. Softw. 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 *Proceedings of the ACM Symposium on Interactive 3D Graphics and Games (I3D'10)*. ACM, New York, 83–90.

CGAL. 2012. CGAL, computational geometry algorithms library. http://www.cgal.org.

CHAN, T. M. 1996. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete Comput. 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 *Proceedings of the 4th Symposium on Computational Geometry (SoCG'88)*. ACM, New York, 12–17.

CUDPP. 2012. CUDPP, CUDA data parallel primitives library. http://gpgpu.org/developer/cudpp.

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 *Proceedings of the 7th ACM Symposium on Parallel Algorithms and Architectures (SPAA '95)*. ACM, New York, 27–33.

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

FISCHER, I. AND GOTSMAN, C. 2006. Fast approximation of high-order Voronoi diagrams and distance transforms on the GPU. *J. Graph. Tools 11*, 4, 39–60.

FUENTES, O. AND GULATI, R. K. 2001. Prediction of stellar atmospheric parameters from spectra, spectral indices and spectral lines using machine learning. *Exp. Astron. 12*, 1. 21–31.

GUPTA, N. AND SEN, S. 2003. Faster output-sensitive parallel algorithms for 3D convex hulls and vector maxima. *J. Parallel Distrib. Comput. 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*, Lecture Notes in Computer Science, vol. 4292, 126–135.

HOFF III, K. E., KEYSER, J., LIN, M., MANOCHA, D., AND CULVER, T. 1999. Fast computation of generalized Voronoi diagrams using graphics hardware. In *Proceedings of the ACM SIGGRAPH International Conference on Computer Graphics and Interactive Techniques*. ACM, New York, 277–286.

JURKIEWICZ, T. AND DANILEWSKI, P. 2011. Efficient quicksort and 2D convex hull for CUDA, and MSIMD as a realistic model of massively parallel computations. http://www.mpi-inf.mpg.de/∼tojot/papers/chull.pdf.

KALLAY, M. 1981. Convex hull algorithms in higher dimensions. Unpublished manuscript, Department of Mathematics, University of Oklahoma, Norman, OK.

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

MAO, H. AND YANG, Y.-H. 2006. Particle-based immiscible fluid-fluid collision. In *Proceedings of Graphics Interface (GI'06)*. Canadian Information Processing Society, Toronto, ON, Canada, 49–55.

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

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 *Proceedings of the International Conference on Intelligent Robots and Systems (IROS'03)*. IEEE, 2155–2160.

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

QHULL. 2012. Qhull, computational geometry algorithms library. http://www.qhull.org.

QI, M., CAO, T.-T., AND TAN, T.-S. 2012. Computing 2d constrained Delaunay triangulation using the GPU. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D'12)*. ACM, New York, 39–46.

RONG, G., TAN, T.-S., CAO, T.-T., AND STEPHANUS. 2008. Computing two-dimensional Delaunay triangulation using graphics hardware. In *Proceedings of the Symposium on Interactive 3D Graphics and Games (I3D'08)*. ACM, New York, 89–97.

SHEWCHUK, J. R. 1997. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete Comput. Geometry 18*, 3, 305–363.

SHEWCHUK, J. R. 2005. Star splaying: An algorithm for repairing Delaunay triangulations and convex hulls. In *Proceedings of the 21st Symposium on Computational Geometry (SoCG'05)*. ACM, New York, 237–246.

SRUNGARAPU, S., REDDY, D. P., KOTHAPALLI, K., AND NARAYANAN, P. J. 2011. Fast two dimensional convex hull on the GPU. In *Proceedings of the IEEE Workshops of International Conference on Advanced Information Networking and Applications (WAINA'11)*. IEEE, 7–12.

STANFORD. 2012. The stanford 3d scanning repository. http://graphics.stanford.edu/data/3Dscanrep/.

STEIN, A., GEVA, E., AND EL-SANA, J. 2012. CudaHull: Fast parallel 3D convex hull on the GPU. *Comput. Graph. 36*, 4, 265–271.

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

TANG, M., YI ZHAO, J., TONG, R., AND MANOCHA, D. 2012. GPU accelerated convex hull computation. *Comput. Graph. 36*, 5, 498–506.

TZENG, S. AND OWENS, J. D. 2012. Finding convex hulls using Quickhull on the GPU. CoRR abs/1201.2936.

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. Comput. Biol. Bioinf. 6*, 4, 583–593.

Received April 2012; revised November 2012, March 2013; accepted April 2013