GPU-assisted Surface Reconstruction on Locally-uniform Samples

Yong Joo Kil and Nina Amenta

University of California, Davis.
kil@cs.ucdavis.edu
amenta@cs.ucdavis.edu

Summary. In point-based graphics, surfaces are represented by point clouds without explicit connectivity. If the distribution of the points can be carefully controlled, surface reconstruction becomes a much easier problem. We present a simple, completely local surface reconstruction algorithm for input point distributions that are locally uniform. The locality of the computation lets us handle large point sets using parallel and out-of-core methods. The algorithm can be implemented robustly with floating-point arithmetic. We demonstrate the simplicity, efficiency, and numerical stability of our algorithm with an out-of-core and parallel implementation using graphics hardware.

1 Introduction

The idea of point-based graphics is that a point sample can be the primary representation of a surface, simplifying computation and saving space by doing without explicit connectivity information. In some situations, for instance a geometric modeling system or the simulation of a moving front, the distribution of the sample points is entirely under the control of the application, which upsamples, downsamples and smooths the distribution as necessary. What kind of a distribution should a point-based application seek to maintain? One criterion, among others, is that the distribution should make later geometry processing operations, such as surface reconstruction, easier. A surface reconstruction algorithm which will be applied to a carefully maintained distribution can be simpler, faster, and easier to implement and to parallelize, and it should scale better.

In this paper we explore how simple and local a surface reconstruction algorithm can be when applied to well-distributed points. Specifically, we consider distributions which are locally uniform, and we give a very easy surface reconstruction algorithm which works well in practice on such distributions. We emphasize that this scenario is very different from the problem of surface reconstruction from acquired data. Rather than designing the algorithm for

robustness to noise and sampling irregularity, we can design for simplicity, speed, and locality of computation.

Outline of algorithm. We construct a octree for the sample points and download it to the GPU. In parallel, we find the k-nearest neighbors of each sample point. Then at each sample point we construct an umbrella of possible surface triangles. Edges that are consistent between adjacent umbrellas are defined to be *consensus edges*. On arbitrary distributions it is possible that the consensus edges will form a very sparse graph, possibly not even connected. But on distributions that are locally uniform, we observe that the consensus edges define a polygonalization of the surface, in which all faces have a small number of edges. We triangulate any non-triangular faces in this graph, using a deterministic rule, completing the triangulation. Our implementation can triangulate a sample consisting of 6 million points on a surface in $\mathbb{R}^3$ in about 35 seconds (not including the CPU octree computation). It is written in CUDA, and uses native GPU floating point arithmetic.

Related work. Surface reconstruction is a huge field and we focus here on only the work most related to ours. The idea of reconciling local umbrellas at each sample point is the central concept in the algorithm of Gopi et al. [1], in which the umbrellas are computed on an advancing front. While the high-level idea of Gopi's algorithm is similar to ours, it is essentially sequential. A recent GPU-assisted algorithm based on Gopi's method is due to appear [2]. This algorithm apparently computes local umbrellas in parallel on the GPU, with both the earlier step of finding nearest-neighbors and a post-processing reconciliation of the umbrellas done sequentially. Also in this same vein, Adamy et al. [3] select umbrellas at each point (using a criterion similar to ours) and reconcile them using simple topological processing followed by a global linear programming step. Our algorithm differs from all of this previous work in that we reconcile the umbrellas locally and in parallel; we do not have to resort to a sequential global process.

Computing and maintaining an umbrella (or "star") at every vertex is also central to Shewchuk's "star splaying" algorithm [4] for maintaining a Delaunay triangulations under perturbations.

Dey et al. [5] and Funke et al. [6] considered surface reconstruction given a locally uniform distribution, and give an almost linear time algorithm. While candidate triangles are found independently at each sample, a final reconciliation stage requires a sequential sweep through the surface. Finally, Dumitriu et al. [7] give a sequential surface reconstruction algorithm which takes a very dense, possibly poorly sampled point cloud as input, and constructs a triangulation of a sparse locally uniform subsample. Their algorithm is interesting in that the only geometric computation required is the construction of a neighborhood graph on the original dense sample. Like us, they find a graph of "good" edges, within which the minimal cycles form a polygonization of the surface; they report cycles of length at most five in practice. In an unpublished manuscript [8], they give a (large) constant upper bound on the size of the cycles produced by their algorithm, which unfortunately cannot be applied

to ours. In contrast, our algorithm requires only the sparse locally uniform input, not the dense input point set as well.

A very recent GPU surface reconstruction algorithm by Zhou et al. [9] implements the Poisson surface reconstruction algorithm of Kazhdan et al. [10]. The paper emphasizes that a major contribution is the introduction of an octree which is not only searched on the GPU, like ours, but also constructed in parallel on the GPU. They report an excellent running time of 0.2 seconds for an input of about 500K points. Poisson reconstruction requires normal information and is a global algorithm; the output size is limited by the resolution of the grid achievable on the GPU. Local algorithms like ours can handle larger inputs by breaking them up into smaller parts.

Locally uniform point samples. We define locally uniformity for a point distribution on a surface using a parameter α.

Definition 1. *Let P be a sample of points from a surface in $\mathbb{R}^3$. For a sample point $v \in P$, let δ_v be the minimum distance from v to any other sample. Let ϵ_v be the maximum distance from v to any* surface *point in the Voronoi cell of v. We say that P is* locally uniform *with parameter α if, for all $v \in P$, $\epsilon_v \leq \alpha\delta_v$.*

This measure of the distribution of a point set is similar to the well-known circumradius to shortest edge criterion for a triangulation. Computing ϵ_v is not possible if we are not given the surface itself. But it is easy to estimate ϵ_v once we have an output triangulation; we use such estimates to illustrate the range of α within which our algorithm is successful in our experiments.

Local uniformity of a point sample does not guarantee that it is dense enough to allow for a correct reconstruction of the surface. Besides being locally uniform, our algorithm also requires the input point distribution to be everywhere sufficiently dense; as noted by Amenta et al. [11], the minimum required density varies over the surface linearly with distance to the medial axis. Any distribution which everywhere exceeds this minimum density and is locally uniform is an appropriate input to our surface reconstruction algorithm.

Producing and maintaining locally uniform samples. It is indeed possible to produce locally uniform point samples on surfaces, and to maintain them as the surface changes. A perfect triangular grid of points in the plane achieves the best possible $\alpha = 1/\sqrt{3} \approx 0.577$ everywhere, but we cannot expect to do so well on an arbitrary surface. A simple greedy sampling strategy [12], however, does achieve $\alpha \leq 1$ on an arbitrary surface: choose a small enough δ, and then while there is some point of the surface farther than δ from any sample, place a new sample at any such point. This procedure can also be adapted to handle varying sampling density. The distribution is improved if the new point is chosen uniformly at random (the Poisson disk sampling method; see Dunbar et al. [13]). Another strategy is to insert the point furthest from any previously-placed sample (eg. Boissonnat and Oudot [14]).

The greedy algorithm creates locally uniform point sets. Among heuristics which can be used to maintain local uniformity after a perturbation (e.g. during a modeling operation or simulation step), Lloyd's algorithm works well in practice. It moves every sample to the center of mass of its Voronoi cell, iteratively, until convergence. Surazhsky et al. [15] uses local parameterizations and Llyod's relaxation to produce locally uniform sampling on arbitrary genus surfaces. Recent work on optimal Delaunay triangulations [16] uses a similar idea to produce even nicer-looking distributions on the plane. On implicit surfaces, a classic paper by Witkin and Heckber [17] uses repulsion and attraction forces to produce and maintain locally uniform point distributions; the distribution in Figure 9 was produced using our implementation of their algorithm. More recently, Meyer et al. [18] gave an improved force-based algorithm along similar lines.

2 Geometric intuition and groundwork

Let's begin by considering the uniform unit triangular grid in a plane embedded in R^3, the vertices of which form a distribution P with the minimum $\alpha = 0.577$. An *empty circumsphere* of a triangle has the vertices of the triangle on its boundary and has no other point of P in its interior. The minimal empty circumspheres of the triangles of the grid have radius α, while any other triangle with vertices on the grid has no empty circumsphere of non-infinite radius. This motivates our umbrella selection criterion: we will choose an umbrella in which each of the triangles has a small empty circumsphere.

In more realistic situations, the surface is curved and the distribution is not so nice. We are however always guaranteed that every sample point in P has at least one umbrella consisting of triangles with small circumspheres. The *restricted Delaunay triangulation* [19] is the set of triangles with empty circumspheres centered at points on the surface; these centers are the points at which the edges of the 3D Voronoi diagram of P pierce the surface.

Observation 1 *A triangle t of restricted Delaunay triangulation has circumspheres of radius at most ϵ_v, for any vertex v of t.*

Since there is an umbrella of triangles at each vertex, each of which has a small circumsphere, we are always able to select such an umbrella. Not all triangles with small circumspheres will be restricted Delaunay triangles, however, even on distributions that are everywhere dense and locally uniform. The difficulty has to do with *slivers*: very flat tetrahedra, the vertices of which are nearly co-circular. It is possible (although not necessary) for all of the triangular faces of a sliver tetrahedron in the 3D Delaunay tetrahedarlization of P to have small circumspheres. Also, even when P is locally uniform, five, six or more points can be nearly co-circular and define many triangles with small circumspheres; the number of points that can be nearly co-circular increases with α. If v and several of its neighbors are nearly co-circular, they might

choose umbrellas that are inconsistent with each other. On the other hand, any subset of these triangles chosen for the umbrella at v will include the two edges connecting v to its neighbors u, w along the "circle". Similarly, these edges will also appear in any subsets of the triangles chosen for the umbrellas at u and w. This intuition leads us to the definition of consensus edges.

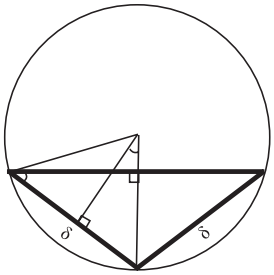
Definition 2. *We define the Delaunay edge uv to be a* consensus edge *if it appears in every umbrella which contains the two vertices u, v. Note that this includes the umbrellas of points other than u and v.*

We conclude with some observations about the triangles with small circumspheres (of radius at most $\alpha\delta_v$). These observations will be useful in selecting the triangles, and they also clarify the connection between α and the quality of the output triangles.

Since every triangle in P is enclosed in a circumsphere of radius at most $\epsilon_v \leq \alpha\delta_v$, we know that each triangle has a circumcircle of radius at most ϵ_v. Since every edge must have length at least δ_v, the smallest possible circumcircle (and hence the smallest possible circumsphere), is the one surrounding an eqilateral triangle with side length δ_v.

Observation 2 *The smallest triangle circumcircle has radius $1/\sqrt{3}\delta_v \approx 0.577\delta_v$.*

Consider the figure below.



The triangle with the largest possible angle has to look like the bold triangle; the radius of the circumcircle is as large as possible, that is, ϵ_v. To get as large an angle as possible at v, we spread the two adjacent edges as far apart as possible, until they are as short as possible, that is, δ_v. The two marked angles are the same, and must be $\arcsin((\delta_v/2)/\epsilon_v) = \arcsin(1/2\alpha)$. So the opposite angle in each of the right triangles is $\arccos(1/2\alpha)$, and the large angle at the bottom is $2\arccos(1/2\alpha)$. Also, notice that to make as small an angle as possible, we also need to use as large a circumcircle as possible, and the smallest angle will be opposite the smallest edge of the triangle. So any angle of a triangle with circumradius ϵ_v, opposite an edge of length δ_v, has the smallest possible angle. Hence:

Observation 3 *The largest angle of any triangle is* $2\arccos((1/2\alpha)$ *and the smallest angle of any triangle is* $\arcsin(1/2\alpha)$. *For instance, with* $\alpha = .9$, *we have largest angle of* 112.5 *and a smallest of* 33.75 *degrees.*

3 Algorithm

We now describe the parallel GPU part of our algorithm. Each step in the following description is run in parallel over each vertex $v \in P$.

Algorithm 1 Parallel surface reconstruction for each $v \in P$

1: Retrieve the k nearest neighbors N_v of v.
2: Compute Delaunay faces F_v connected to v using N_v
3: Compute umbrella U_v by choosing a subset from F_v.
4: Determine if out-going edges of U_v are consensus edges C_v by searching through all umbrellas in N_v.
5: Find cycles for every edge in C_v and output polygons.

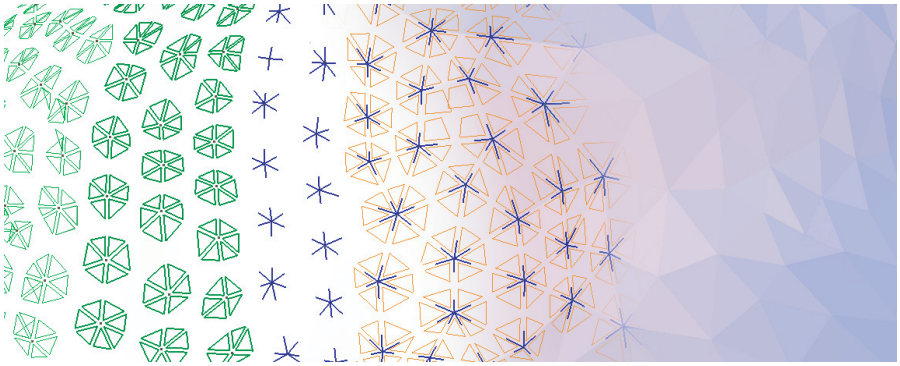


Fig. 1. A visualization of the steps of our algorithm. We draw the simplicies by scaling them towards their respective vertices. Starting from the left, we compute Delaunay and nearly-Delaunay triangles (light green), umbrellas (dark green), consensus edges (dark blue), and cycles (orange). We then render the resulting triangles.

Let us discuss these steps in detail. The first step retrieves the k nearest neighbors using a search structure; we use an octree which is pre-computed on the CPU. The second step computes Delaunay faces connected to v by considering samples in N_v, which is sufficient as long as the k nearest neighbors contain all points within $2\epsilon_v$ distance from v. The choice of k depends on the uniformity of the point-set.

We defer discussion of the second parallel step, which produces a set of triangles at each vertex v, until Section 4. From this set, we extract an umbrella of triangles, each with small circumball radius in the third step, using the following algorithm. An *outgoing* edge of v is an edge connected to v. Triangles in F_v are *adjacent* if they share an outgoing edge. Outgoing edges of an umbrella U_v are *manifold edges*, meaning that edge is connected to exactly two triangles. To form an umbrella from F_v and to ensure that the resulting umbrella if formed by triangles with small circumball radii, we look for triangles with large circumball radii that can be removed. A triangle is a *flake* if one of its outgoing edges does not meet another triangle at a large dihedral angle. A flake is never a restricted Delaunay triangle and can always be removed immediately. Second, a triangle is a *pocket* triangle if it is not a flake and if the dihedral angle between it and any of its adjacent triangles are small; adjacent triangles with manifold edges are also pockets. Figure 2 shows an example of flakes and pockets and Algorithm 2 describes the umbrella filtering.

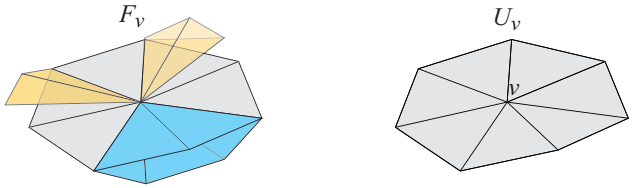


Fig. 2. The left figure is a F_v with flake triangles highlighted in yellow and pocket triangles highlighted in blue. The right figure is a resulting umbrella U_v. Note that removable of a pocket triangle can change pocket triangles into flake triangles.

Algorithm 2 UMBRELLA(v, F_v)

$U_v \leftarrow F_v$
while U_v is not an umbrella **do**
  Remove all flakes in U_v.
  Remove the pocket triangle with the largest circumball radius in U_v.
return U_v.

We may remove a restricted Delaunay triangle which is a pocket triangle, but if so the remaining umbrella will consist only of triangles with even smaller minimal circumspheres.

The fourth step determines if the outgoing edges of U_v are consensus edges. For a vertex v, let us define L_v as the *link* of U_v, which contains all edges in U_v not touching v. Let O_v be outgoing edges of U_v. In order for an edge $vw \in O_v$ to be consensus, we check two conditions. First, we check if $wv \in O_w$. Second, we check for all $x \in N_v$ such that if vertices $\{v, w\} \in U_x$, then $vw \in L_x$. If either of these conditions fail, then vw is not a consensus edge.

We can easily extend the consensus definition to handle boundaries. A point on a boundary cannot have an umbrella, so the UMBRELLA algorithm will produce the empty set (see Figure 3). In order to produce triangles that partially cover a boundary sample, we simply ignore consensus tests that involve empty U_w or U_x. This works quite well when the boundary is nicely sampled, as in Figure 3.

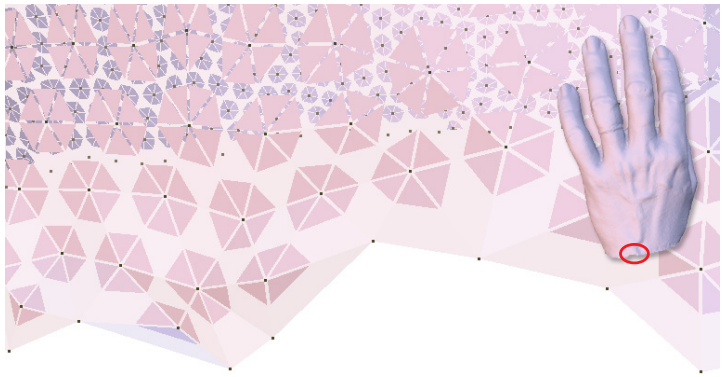


Fig. 3. A closeup of the boundary of the hand model. Our algorithm handles boundaries gracefully.

Finally, in the last step, we extract the polygonal faces by detecting cycles in the graph of the consensus edges. This is complicated by the fact that we do not have vertex normals or a consistent orientation. Following a consensus edge from v to w, we orient w consistently with v as follows. We consider the triangle t_w that is on the same side as t_v of the plane that is perpendicular to t_v and passing through vw (t_w may or may not be t_v). If t_w has a consensus edge wz opposite of vw, we choose wz as the next consensus edge in the cycle. If t_w does not have a consensus edge, we traverse the adjacent triangles in the umbrella until we do find a consensus edge; if we get back to vw before finding another consensus edge we quit and do not output a cycle. For a consensus edge touching a boundary sample b, we do something similar to define a face: we start a path from the edge containing b and terminate the search once the path reaches another boundary sample (which might be b).

Once a cycle is found, we can return the cycle as a polygon or triangulate it if necessary. To make sure all vertices compute the same triangulations, we deterministically break each cycle into a triangle fan with the center located at the vertex with the smallest memory address. By not relying on any orientation information, we can extract non-orientable surfaces, such as a Mobius strip as shown in Figure 4.

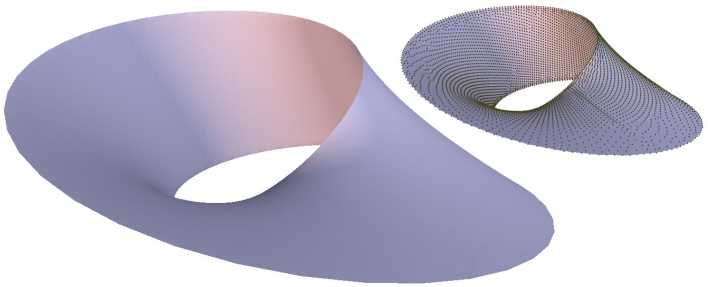


Fig. 4. Our algorithm works on non-orientable surfaces. Here, the input point samples are shown at the top right.

4 Nearly-Delaunay computation with floating-point arithmetic

In this section, we discuss the computation of the set of Delaunay and nearly-Delaunay triangles from which the umbrellas are selected. This computation is done with the GPU's floating-point arithmetic, which is generally single-precision and does not conform to the IEEE floating-point standard. Our approach is to be tolerant of numerical error.

The umbrella selection algorithm requires at each vertex v a set of triangles, each labeled with an estimate of the radius of its smallest empty circumsphere. In the interests of maximizing the number of consensus edges, we would like this set to be as small as possible. In this section we consider the geometric computation we use to find that set. We first select a list of possible candidate triangles, and then we check each candidate. We can eliminate a candidate if we determine that it is definitely not Delaunay, or if we determine confidently that its smallest empty circumsphere is much larger than the distance from v to its nearest neighbor.

We begin with the candidate set of all well-shaped triangles T_v in N_v connected to v. To estimate the radius of the smallest circumsphere for a triangle touching $t \in T_v$ we consider the Voronoi edge dual to t. We parameterize the dual edge as $e(\lambda) = c + \lambda n$, where c is the circumcircle center of t, n is the unit normal perpendicular to the plane of t and $\lambda \in \mathbb{R}$. Let B_λ be the circumsphere centered at $e(\lambda)$, touching the vertices of t. For each point $q \in \{N_v \backslash t\}$, we find the interval Λ_q that contains values of λ for which B_λ is empty of q. This idea is elaborated in Algorithm 3.

The fundamental operation in the algorithm is computing the interval Λ_q, which is equivalent to finding the center of the sphere touching the three vertices of triangle t and the fourth point q. This circumcenter operation is notoriously numerically unstable: if the four points are nearly co-circular, then round-off error in fixed-precision floating-point computation can lead to wildly incorrect results. We use our assumption that the input P is locally uniform

Algorithm 3 DELAUNAY_FACES(v, N_v)

$F_v \leftarrow \{\}$
Let T_v be the set of all possible triangles from N_v, touching v, and without very large or very small angles.
for all $t \in T_v$ **do**
  $\Lambda_t \leftarrow (-\infty, +\infty)$
  for all $q \in \{N_v \backslash t\}$ **do**
    Compute the valid interval Λ_q
    $\Lambda_t \leftarrow \Lambda_t \cap \Lambda_q$
  if Λ_t is not empty **then**
    Find the smallest $|\lambda|$ in Λ_t to determine the smallest circumball radius r_t.
    Add $\{t, r_t\}$ in F_v.
return F_v.

in several ways to deal with the problems caused by numerical instability. Our goal is to avoid eliminating restricted Delaunay triangles, while filtering out triangles that cannot possibly be restricted Delaunay triangles, thus producing better umbrellas and more consensus edges.

First, we eliminate all triangles that have any angle that is too large or too small, which is justified by Observation 3. In our experiments, we simply ignore triangles with angles smaller than one degree.

Second, we observe that because triangle t is well-shaped, the computation of its normal n and the center of its circumcircle c are, while not exact, at least numerically stable. Instability in these computations occur when the three triangle vertices are nearly co-linear, and these situations have been eliminated. So we compute c and n once for each triangle and use them in the computation of each Λ_q. In computing n and c, we improve the stability by taking v as the origin, and then in the computation of Λ_q we use c as the origin. Also, we ensure that the vertices of a triangle are always used in the same (memory address) order when computing n and c, so that n and c will be identical when t is processed at each of its vertices.

To solve for the endpoint in the interval Λ_q, we solve for λ such that $|e(\lambda) - p| = |e(\lambda) - q|$, where p is a vertex of t. If c is at the origin, then

$$\lambda = \frac{(p-q)^T(p+q)}{2(p-q)^T n} \tag{1}$$

This computation will *not* be stable if q and the vertices of t are nearly co-circular or nearly co-planar. We do some special processing to identify these cases. We take a conservative approach and set Λ_q to $(-\infty, \infty)$ if the four points are nearly co-circular, passing the triangle (as far as q is concerned) as nearly-Delaunay. At the same time, we eliminate triangles with large minimal empty circumspheres.

First, let us consider an idea for avoiding unnecessary evaluations of λ. Recall that Observation 2 tells us that the radius of the circumcircle of each

well-shaped triangle is no smaller than $\sqrt{3}\delta_v$, while Observation 1 tells us that any restricted Delaunay triangle has a smallest circumsphere of radius at most $\epsilon_v = \alpha\delta_v$. We consider two balls, B_a and B_b, passing through the vertices of t with the center of B_a above and the center of B_b below the plane of t (see Figure 5), each of radius $\sqrt{3}\alpha$ multiplied by the circumcircle radius of t. Any sample q_{out} outside $B_a \cup B_b$ cannot improve our lower bound on the size of the smallest empty circumsphere, so immediately conclude that t is acceptable with respect to q_{out} and continue on to the next q. Any sample q_{in} inside $B_a \cap B_b$ forces the minimum empty circumsphere to have radius radius larger than ϵ, so we immediately reject t for F_v.

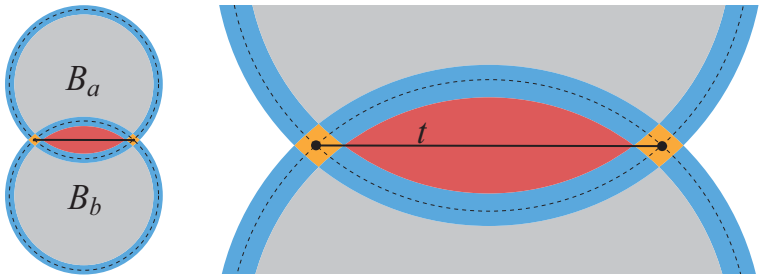


Fig. 5. This figures shows the regions considered in our robust floating-point computation. The triangle t and its points are shown in black, the two balls B_a and B_b are shown in dotted lines, and the shells are shown in blue. The right figure is a closeup of the area near the triangle. Points in the orange region are considered to be co-circular with the vertices of t. Points in the blue region are considered to be co-spherical with either B_a or B_b. Points in the red region are considered to have circumball radius larger than ϵ. Points in the gray region are numerically safe to use Equation 1.

Now we consider the unstable case. Notice that in Equation 1, if the vector $(p - q)$ is very small or is almost perpendicular to n, then the denominator will be close to zero. The dot product between these two vectors may produce a catastrophic cancelation that results in a large relative error. To avoid doing the computation in the unstable case, we consider the *shells* S_a and S_b of B_a and B_b, respectively. A shell of a ball B is formed by the points between the two balls sharing the same center as B with radii $r(1-\mu)$ and $r(1+\mu)$, where r is the radius of B and $0 < \mu \ll 1$. If $q \in S_a \cap S_b$, then we assume q is co-circular with the vertices of t and we treat Λ_q as unbounded. If $q \in (S_a \cup S_b) - (S_a \cap S_b)$, we handle it as described above for B_a and B_b. Only if $q \notin (S_a \cup S_b)$ and $q \in (B_a \cup B_b) - (B_a \cap B_b)$ do we compute λ using the formula above, and in that case the computation is stable. To make sure the resulting Λ_t has an valid interval when its interval is very small, we also scale each interval Λ_q by $(1 + \mu)$. In our experiments, we set μ as low as 10^{-5} without noticing any problems with the 32-bit floating point arithmetic on our graphics hardware.

Figure 6 shows an example of a difficult input, containing many truly cocircular quadruples of points, with Λ_q computed using a naive floating-point implementation and also with our more careful conservative strategy. Notice that we retain triangles which are possibly not Delaunay. The later umbrella computation and the choice of consensus edges tolerate this behavior.

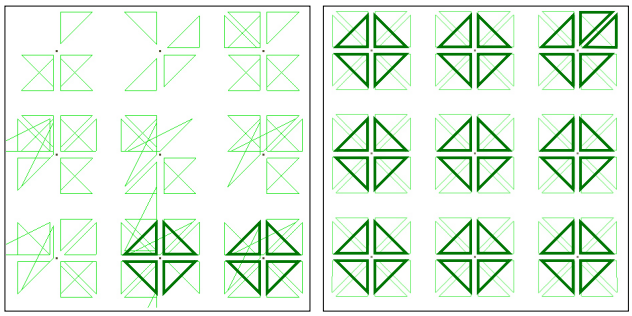


Fig. 6. The computed Delaunay faces for each vertex are highlighted in green and the darker green triangles are the resulting umbrellas. The left figure shows triangles computed using without using the careful numerical approach described in Section 4, and the right figure shows the triangles using the approach.

5 Parallel and out-of-core implementation

We wrote our parallel algorithm using NVIDIA's CUDA GPU Computing environment [20], using OpenGL for rendering. Each of the steps in Algorithm 1 runs in parallel as a CUDA kernel. In order to communicate between kernel stages, we create workspace buffers to hold intermediate data, such as the pointers to the k neighbors of each vertex in P. If these workspace buffers collectively take up too much memory, we partition P into smaller groups, which are then computed serially. The groups are bounded by chains of input samples, which are included in both groups. To make sure the samples in these boundary groups produce consistent triangles, the neighboring samples on either side should also be included in both groups. In our experiments so far, however, we treat the group boundaries as described in Section 3, and we have not noticed holes in the resulting outputs.

For the Delaunay computation of F_v, each vertex v iterates over all possible candidate triangles formed by v and two other samples in N_v. To speed this up, we create a thread for each of v's neighbors. We allow these threads to communicate by caching vertex positions into the on-chip memory (shared memory) that is shared amongst v's neighbors.

Reducing the number of nearest-neighbors k of course speeds up the computation of the nearly-Delaunay triangles. But small values of k can lead to

trouble when we have varying sampling density, as in Figure 7. In this situation it might happen that some of v's neighbors find triangles including v which are not found by the computation at v itself. To remedy this, we share near-Delaunay faces between neighbors after they are computed. We do this by searching through each v's k^+-nearest neighbors. In our experiments, we chose $k^+ = 2k$. This allows us to reduce the size of k

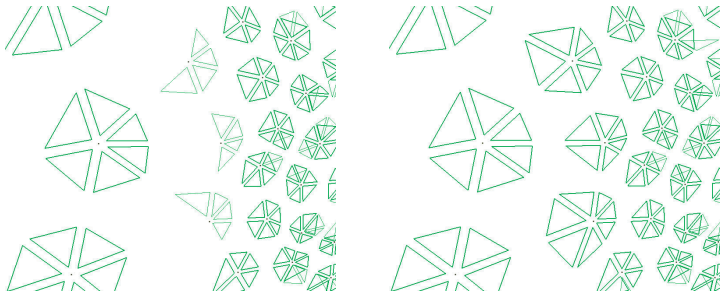


Fig. 7. The left figure shows the Delaunay faces each v finds by considering just its k nearest neighbors. Notice that with the quickly varying sampling density, some triangles at v are missed. The right figure shows the results after each v searches for faces connected to v from its k^+ neighbors.

6 Results

We used an NVIDIA GeForce 8800 GTX GPU with 768MB connected via PCI Express 16x to an Intel Core2 Quad 2.4GHz CPU. Our applications are built on top of Windows XP and the CUDA Toolkit 1.1.

To see how running time varies as function of the input size, we produced inputs of varying size using subdivision. The timing results are shown in Figure 8. To focus on our algorithm, we do not include rendering time nor the time required to compute the octree on the CPU, which takes less than a minute for 6 million points. In most cases, rendering adds an additional 20% to the overall time. The first three stages of Algorithm 1 take the majority of time: getting the neighbors takes about 20%, making nearly-Delaunay triangles takes 40%, and selecting umbrellas takes 20%. We optimized our code for the Delaunay computation by caching vertex positions into the shared memory space, which reduced the timing by a factor of four. We have not applied similar optimizations to the other stages yet, which we expect would yield similar speedups.

A method to generate our sampling requirement is shown in Figure 9. We begin with a uniform random distribution, and use our implementation of the Witkin-Heckbert technique [17] to improve the distribution. Any vertex

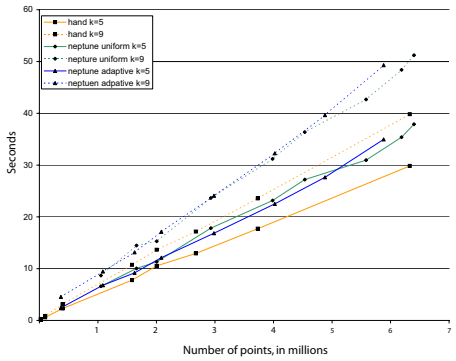


Fig. 8. Timing results (GPU only) for varying input sizes and for different choices of k. As expected, we see that that reconstruction time scales linearly with the input size.

that does not have an umbrella or has less than three consensus edges is considered to have failed the reconstruction. The percent of vertices for which the reconstruction fails appears in the figure. The α value is estimated at each v by finding δ_v, the distance to the closest other sample, and then estimating ϵ_v by the largest circumcircle radius in v's umbrella. Histograms of these estimated alpha values can be seen in Figures 12-11.

We considered the overall reconstruction successful if a very large percentage of the vertices produce at least three consensus edges. The number of nearest neighbors required to get a successful reconstruction varies with α. The uniformly sampled Neptune only requires $k = 9$, while the non-uniformly sampled Neptune model requires $k = 15$. For the very uniform hand model, $k = 7$ produces perfect reconstruction. We also tried our algorithm on a very non-uniform sampling with the dragon model shown in Figure 11. The dragon model works on 95% of the vertices with $k = 15$.

7 Future work

This work is one example of a geometry processing problem which is easier when applied to locally uniform data. We emphasize that our algorithm is not meant to reconstruct noisy data from laser scanners, but instead meant to represent surfaces purely from point clouds that can be carefully controlled. If some vertices do not meet the sampling condition, one can explicitly maintain their umbrellas. However, we hope that research in this direction will inspire more work on producing and especially maintaining nice sampling distributions, and showing that these distributions have useful properties.

We are currently working on a proof that an exact version of this algorithm would always produce cycles with small constant maximum length on inputs that are locally uniform and sufficiently dense. On the practical side, we plan

to incorporate this surface reconstruction algorithm into an application that maintains a locally uniform distribution on a moving surface.

Our algorithm fails to be completely parallel since we begin by computing an octree on the CPU. While this is not the bottleneck, developing a good octree implementation on the GPU is an important issue, recently addressed in the algorithm of Zhou et al. [9]. We plan on incorporating similar data structure in order to make our algorithm run completely in the GPU.

8 Acknowledgments

We would like to thank Shubho Sengupta and Brian Budge for their suggestions with CUDA. The Oliver hand and Nepture models are provided in courtesy of INRIA in the AIM@SHAPE Shape Repository. The Asian Dragon Model is provided in courtesy of XYZ RGB Inc. This work was supported by NSF grants CCF–0331736 and CCF–0635250.

References

1. M. Gopi and S. Krishnan. A fast and efficient projection based approach for surface reconstruction. *International Journal of High Performance Computer Graphics*, 2002.
2. Carlos Buchart, Diego Borro, and Aiert Amundarain. Gpu local triangulation: an interpolating surface reconstruction algorithm. *Computer Graphics Forum*, 27(3), 2008.
3. Udo Adamy, Joachim Giesen, and Matthias John. Surface reconstruction using umbrella filters. *Comput. Geom. Theory Appl.*, 21(1):63–86, 2002.
4. Richard Shewchuk. Star splaying: an algorithm for repairing delaunay triangulations and convex hulls. In *SCG '05: Proceedings of the twenty-first annual symposium on Computational geometry*, pages 237–246, New York, NY, USA, 2005. ACM Press.
5. T. Dey, S. Funke, and E. Ramos. Surface reconstruction in almost linear time under locally uniform sampling, 2001.
6. S. Funke and E. Ramos. Smooth-surface reconstruction in near-linear time. In *Proc. 15th Annu. ACM-SIAM Sympos Discrete Algorithms*, 2002.
7. Daniel Dumitriu, Stefan Funke, Martin Kutz, and Nikola Milosavljevic. How much geometry it takes to reconstruct a 2-manifold in r3. In *Workshop on Algorithm Engineering and Experiments*, 2008.
8. D. Dumitriu, S. Funke, M. Kutz, and N. Milosavljevic. On the locality of reconstructing a 2-manifold in r3, unpublished manuscript. http://www.mpi-inf.mpg.de/ funke, 2008.
9. Kun Zhou, Minmin Gong, Xin Huang, and Baining Guo. Highly parallel surface reconstruction. Technical report, April 2008.
10. Michael Kazhdan, Matthew Bolitho, and Hugues Hoppe. Poisson surface reconstruction. In *SGP '06: Proceedings of the fourth Eurographics symposium on Geometry processing*, pages 61–70, Aire-la-Ville, Switzerland, Switzerland, 2006. Eurographics Association.

11. Nina Amenta and Marshall Bern. Surface reconstruction by voronoi filtering. *Discrete and Computational Geometry*, 22:481–504, 1999.
12. T. F. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 38(2–3):293–306, 1985.
13. Daniel Dunbar and Greg Humphreys. A spatial data structure for fast poisson-disk sample generation. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pages 503–508, New York, NY, USA, 2006. ACM Press.
14. Jean-Daniel Boissonnat and Steve Oudot. Provably good sampling and meshing of surfaces. *Graph. Models*, 67(5):405–451, 2005.
15. Vitaly Surazhsky, Pierre Alliez, and Craig Gotsman. Isotropic remeshing of surfaces: a local parameterization approach. In *Proceedings of 12th International Meshing Roundtable*, 2003.
16. Long Chen. Mesh smoothing schemes based on optimal delaunay triangulations. In *In Proceedings of 13th International Meshing Roundtable*, pages 109–120, 2004.
17. Andrew P. Witkin and Paul S. Heckbert. Using particles to sample and control implicit surfaces. *Computer Graphics*, 28(Annual Conference Series):269–277, 1994.
18. M. Meyer, P. Georgel, and R.T. Whitaker. Robust particle systems for curvature dependent sampling of implicit surfaces. In *In Proceedings of the International Conference on Shape Modeling and Applications (SMI)*, pages 124–133, June 2005.
19. Herbert Edelsbrunner and Nimish R. Shah. Triangulating topological spaces. In *SCG '94: Proceedings of the tenth annual symposium on Computational geometry*, pages 285–292, New York, NY, USA, 1994. ACM.
20. NVIDIA Corporation. NVIDIA CUDA compute unified device architecture programming guide. http://developer.nvidia.com/cuda, January 2007.

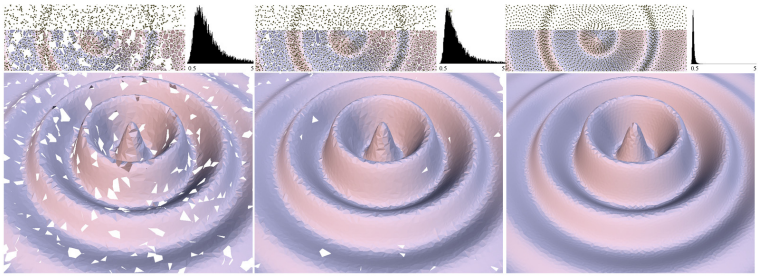


Fig. 9. The left figure shows the initial uniform-random distribution on a sinusoidal surface. Using $k = 9$ the reconstruction works on 92% of the vertices. The middle figure shows the distribution after one iteration of the Witkin and Heckber particle smoothing algorithm with 96% correct reconstruction. The right figure shows the perfect reconstruction after the particles converge to a stable configuration, which takes about ten iterations.

Fig. 10. We consider the Oliver hand model. We see from the left figure that the hand has very uniform sampling. The two hands in the middle show the reconstruction using using $k = 3$ with 10% correct (left) and $k = 7$ with 100% correct (right). Notice from the most right figure that the regions between the fingers with high curvature are reconstructed properly.

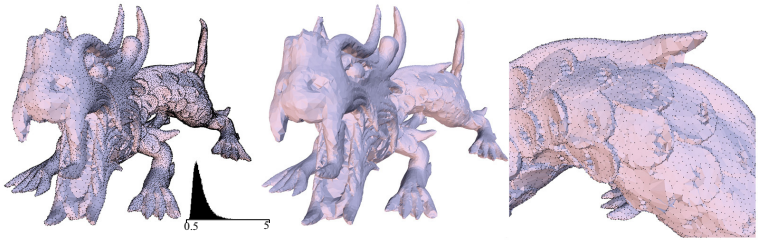


Fig. 11. To consider a very non-uniform sampling, we decimated the dragon model from 3.6 million points to 100k points. With $k = 16$ the reconstruction works on 95% of the vertices. Notice from the right figure that the scales are reconstructed reasonably well. The spike at the top of this figure is sampled too sparsely to have a meaningful reconstruction.

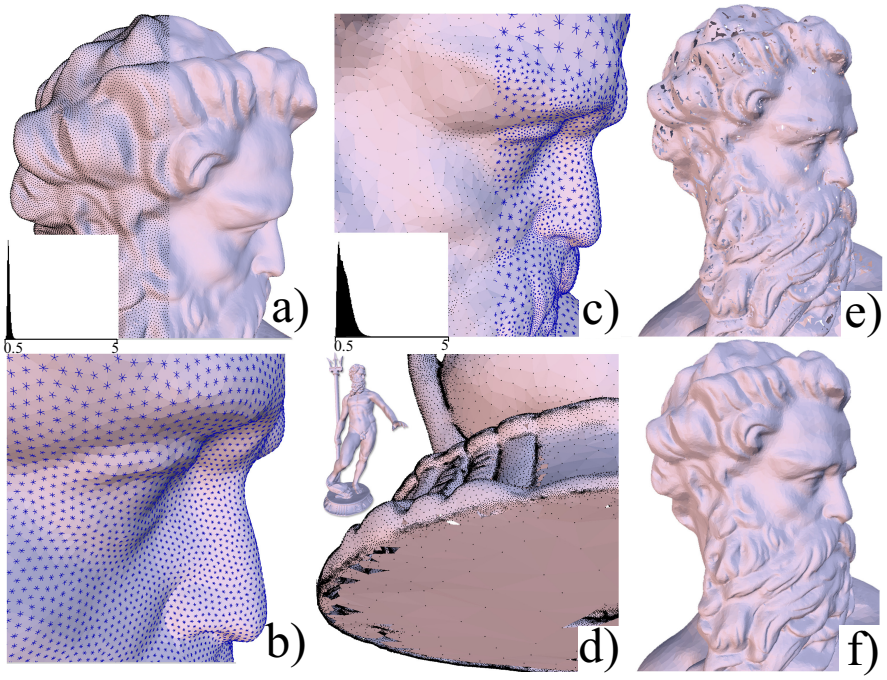


Fig. 12. These figures show the uniform and non-uniform Neptune models. Histogram of their estimated alpha values are also shown. (a) The uniformly sampled Neptune model. (b) A close up of the uniform model with consensus edges. Using $k = 9$, the reconstruction fails on 13 of the 1.6 million vertices. (c) The non-uniform Neptune model. (d) A close up of the base. Notice that regions near extremely varying sampling density are not reconstructed properly. (e) Reconstruction of the non-uniform model using $k = 9$ with 97% correct (f) and $k = 15$ with 98% correct.