TerraNNI: Natural Neighbor Interpolation on 2D and 3D Grids Using a GPU

PANKAJ K. AGARWAL, Duke University
ALEX BEUTEL, Carnegie Mellon University
THOMAS MØLHAVE, SCALGO USA

7

With modern focus on remote sensing technology, such as LiDAR, the amount of spatial data, in the form of massive point clouds, has increased dramatically. Furthermore, repeated surveys of the same areas are becoming more common. This trend will only increase as topographic changes prompt surveys over already scanned areas, in which case we obtain large spatiotemporal datasets.

An initial step in the analysis of such spatial data is to create a digital elevation model representing the terrain, possibly over time. In the case of spatial (spatiotemporal, respectively) datasets, these models often represent elevation on a 2D (3D, respectively) grid. This involves interpolating the elevation of LiDAR points on these grid points.

In this article, we show how to efficiently perform natural neighbor interpolation over a 2D and 3D grid. Using a graphics processing unit (GPU), we describe different algorithms to attain speed and GPU-memory tradeoffs. Our experimental results demonstrate that our algorithms not only are significantly faster than earlier ones but also scale to much bigger datasets that previous algorithms were unable to handle.

Categories and Subject Descriptors: D.2 [**Software**]: Software Engineering; F.2.2 [**Nonnumerical Algorithms and Problems**]: Geometrical Problems and Computations; H.2.8 [**Database Management**]: Database Applications—*Data mining, Image databases, Spatial databases and GIS*

General Terms: Performance, Algorithms

Additional Key Words and Phrases: LIDAR, massive data, GIS, natural neighbor interpolation, GPU

ACM Reference Format:
Pankaj K. Agarwal, Alex Beutel, and Thomas Mølhave. 2016. TerraNNI: Natural neighbor interpolation on 2d and 3d grids using a GPU. ACM Trans. Spatial Algorithms Syst. 2, 2, Article 7 (June 2016), 31 pages.
DOI: http://dx.doi.org/10.1145/2786757

1. INTRODUCTION

With advances in sensing and mapping technologies, big geospatial datasets are being collected and regularly updated by government agencies at all levels, as well as by private companies, at an unprecedented rate, and the demand for these datasets is increasing. For example, the U.S. Geological Survey (USGS) produces the regularly updated National Elevation Dataset (NED), which includes a national 1/3-arc-second (∼10 meter) digital elevation model (DEM) as well as 1/9-arc-second (∼3 meter)

This work is supported by the NSF under grants CNS-05-40347, IIS-07-13498, CCF-09-40671, and CCF-1012254; by ARO grants W911NF-07-1-0376 and W911NF-08-1-0452; by U.S. Army ERDC-TEC grant W9132V-11-C-0003; by NIH grant 1P50-GM-08183-01; and by a grant from the U.S.-Israel Binational Science Foundation.
Authors' addresses: P. K. Agarwal, Duke University, Box 90129, Durham NC 27708-0129; A. Beutel, 5000 Forbes Avenue, Carnegie Mellon University, Pittsburgh, PA 15232; T. Mølhave, SCALGO, Aabogade 40, 8200 Aarhus N, Denmark.

DOI: http://dx.doi.org/10.1145/2786757

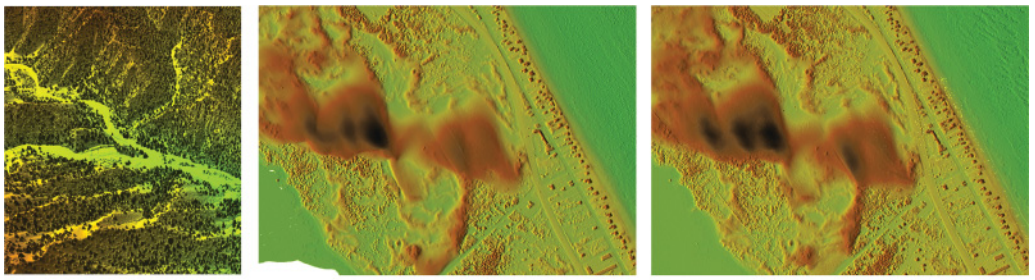


Fig. 1. (a) Grid DEM constructed from LiDAR data over a region in Afghanistan (data source: Army Research Office); trees are clearly visible. (b,c) Jockey's Ridge State Park sand dune in Nags Head, NC.

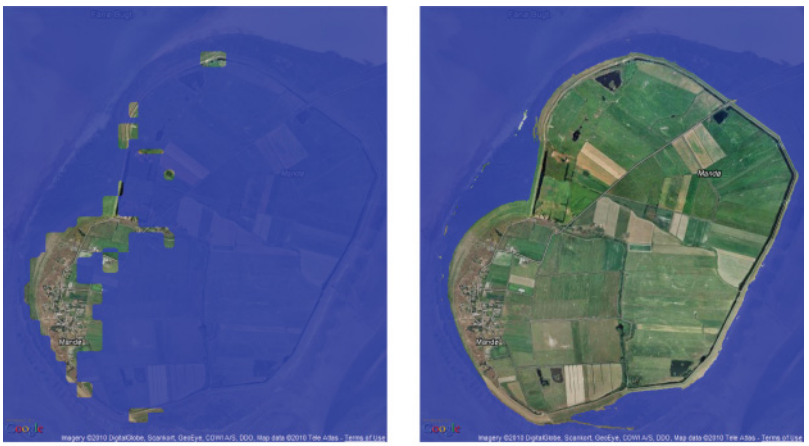


Fig. 2. (b) A flood risk mapping of the island of Mandø in Denmark, using the 90m grid and (c) the same using the 2m grid; both figures are screenshots of a custom map application built on Google Maps.

resolution DEMs for parts of the United States where there is sufficient coverage. Similarly, the coastal region of North Carolina has been mapped every 1 to 2 years since the mid-1990s [NOAA 2014]. Figures 1(b) and 1(c) show a spatiotemporal model of a portion of this region constructed from the dataset. Such datasets provide tremendous opportunities for a wide range of commercial, scientific, and military applications. For instance, unmanned aerial vehicles can survey an area over a certain time period and the resulting spatiotemporal terrain elevation data can be used to detect the location of new buildings, machinery, or vegetation. On a larger scale, they can be used to offer interesting insights into understanding terrain dynamics. For example, there has been much interest on studying the movement of sand dunes and erosion at the North Carolina coast [Mitasova et al. 2005].

It is essential for many applications to exploit the high-resolution datasets since small features may have a large impact on the output. For instance, it is vital that dikes and other features are present in datasets used for hydrological modeling, but these features are often relatively small and unlikely to appear if the resolution is low [Mølhave et al. 2010]. An example of this can be seen in a simple (nontemporal) flood mapping application. Figures 2(b) and 2(c) show the result of the flood risk mapping for the island of Mandø in the Wadden-Sea off the west coast of Denmark. The island has an approximately 5-meter-tall perimeter dike that protects it from the sea. Because of the

small width of the perimeter dike, this feature is not present in low- or midresolution grids. Thus, when flood maps are constructed on coarse grids for a water level of 2 meters, it looks as if most of the island will be underwater. See Figure 2(b) for an example using a 90m grid (the SRTM grid available from NASA [Farr et al. 2007]). The same computation performed on a 2m-resolution grid, shown in Figure 2(c), correctly finds that the dikes, now present in the terrain model, block the water from entering the lower-lying areas inside the perimeter.

Capitalizing on the opportunities presented by high-resolution datasets and transforming the massive amount of spatial and spatiotemporal topographic data into useful information requires that several algorithmic challenges be addressed. To begin with, the scattered point set S generated by LiDAR cannot be used directly by many GIS algorithms. They instead operate on a digital elevation model (DEM). Because of its simplicity and efficiency, the most widely used DEM is a 2D uniform grid in which an elevation value is stored at each cell. A 2D point cloud can be seen as a set S of n (arbitrary) points in $\mathbb{R}^2$ with an associated elevation function $h : S \to \mathbb{R}$. Thus, to construct a grid DEM, h has to be extended via interpolation to a uniform grid $G \subset \mathbb{R}^2$ at the desired resolution.

For spatiotemporal data, the notion of a 2D uniform grid translates into a 3D grid with time representing the third dimension. One has to extend the elevation measured by LiDAR at the points of S via interpolation to a uniform grid $\mathbb{Q} \subset \mathbb{R}^3$ of the desired resolution; the interpolation is performed in both time and space. We note that the need to interpolate a spatiotemporal dataset on a uniform grid arises in a variety of applications. For example, one may want to interpolate data generated by a sensor network deployed over a wide area [Ghosh et al. 2012].

There is extensive work on statistical modeling of spatial and spatiotemporal data in many disciplines, including geostatistics, environmental sciences, GIS, atmospheric science, and biomedical engineering. It is beyond the scope of this article to discuss these methods here. We refer to Kyriakidis and Journel [1999], Mateu et al. [2003] and Wikle et al. [1998] for reviews of many such results. In the context of GIS, interpolation methods based on kriging, inverse distance weighting, shape functions, random Markov fields, and splines have been proposed; see Mitasova et al. [1995], Shekhar and Xiong [2008], Li and Revesz [2002], and Miller [1997] and references therein. Although sophisticated spline-based methods (e.g., regularized splines with tension (RST)) and kriging produce high-quality output, especially when data is sparse, they are computationally expensive and not scalable because of their usage of nontrivial polynomials. On the other hand, simple methods such as constructing triangulation on input points in $\mathbb{R}^3$ [Li and Revesz 2002] and linearly interpolating the elevation on grid points don't produce a smooth surface, especially in the areas where the data is relatively sparse. The resulting DEM can appear jagged both when viewed directly and in derived products, such as contour maps or river networks.

In this article, we use the well-known *natural neighbor interpolation* (NNI) strategy [Sibson 1981]. Given a finite set S of points in $\mathbb{R}^k$, a height function $h : S \to \mathbb{R}$ can be extended to the entire $\mathbb{R}^k$ using natural neighbor interpolation. In particular, for a point $q \in \mathbb{R}^k$,

$$h(q) = \sum_{p \in S} w_p(q) h(p), \tag{1}$$

where $w_p(q) \in [0, 1]$ is the fractional volume of $\mathrm{Vor}_{S \cup \{q\}}(q)$ that belongs to $\mathrm{Vor}_S(p)$ (see Figure 3 for a 2D example), that is,

$$w_p(q) = \frac{\mathrm{Vol}(\mathrm{Vor}_S(p) \cap \mathrm{Vor}_{S \cup \{q\}}(q))}{\mathrm{Vol}(\mathrm{Vor}_{S \cup \{q\}}(q))}, \tag{2}$$

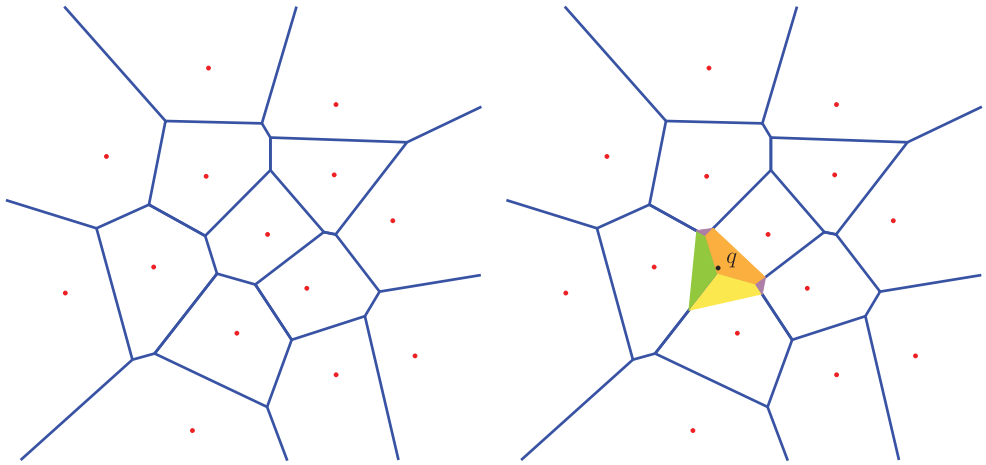


Fig. 3. (a) Voronoi diagram $\mathrm{Vor}(S)$ of a set S of points. (b) Natural neighbor interpolation for a point set S and query point q in $\mathbb{R}^2$. The shaded cell is $\mathrm{Vor}_{S\cup\{q\}}(q)$, and each color denotes the area stolen from each cell of $\mathrm{Vor}(S)$.

where $\mathrm{Vor}_A(z)$ denotes the Voronoi cell of the point $z \in A$. NNI is known to have several nice properties. For example, it is local, using only sample points that surround a query point; it does not introduce new critical points; and the interpolated surface passes through input points and is smooth everywhere except at input points [Sibson 1981; Watson 1992]. Because of these desirable properties, NNI is widely used in many fields [Boissonnat and Cazals 2000; Owen 1992; Sukumar et al. 1998].

Although NNI is more efficient than RST and other similar interpolation methods, its traditional implementations (both 2D and spatiotemporal versions) are significantly slower than linear interpolation and thus not widely used for very large datasets.

There are several reasons that NNI is computationally challenging. First, the size of the Voronoi diagram in 3D can be quadratic in the worst case even for the Euclidean metric [de Berg et al. 1997]. As mentioned later, for spatiotemporal data, we are interested in computing the Voronoi diagram under more general metrics. Even if the size of the Voronoi diagram is near linear for realistic datasets [Attali and Boissonnat 2004], computing it is still expensive, especially on large datasets that do not fit in main memory. Since S cannot be assumed to be in the general position, robust implementations must take great care to handle cases such as point duplicates and groups of multiple points on the same plane/sphere; the existing implementations of Voronoi diagram construction such as Qhull [Barber et al. 1996] are slow on degenerate inputs. Furthermore, performing NNI involves computing the intersection of two polyhedra and the volume of this intersection. Hemsley [2009] has implemented NNI in 3D under the Euclidean metric, but the implementation is not scalable; see Section 6 for more details. We are unaware of any robust implementation of NNI for points in $\mathbb{R}^3$ that can handle large datasets.

To attain significant speedup in the NNI computation, we exploit the graphics processing units (GPUs) available on modern PCs. Although originally designed for quickly rendering 3D geometric scenes on an image plane (screen) and extensively used in video games, they can be regarded as massively parallel vector processors suitable for general-purpose computing. Known as general-purpose GPUs (GPGPUs), their tremendous computational power and memory bandwidth make them attractive for applications far beyond the original goal of rendering complex 3D scenes. As GPUs

have become more flexible and programmable (e.g., NVIDIA's CUDA [NVIDIA 2010] library), they have been used for a wide range of applications, for example, geometric computing, robotic collision detection, database systems, fluid dynamics, and solving sparse linear systems. See Owens et al. [2007] for a recent survey. In the context of grid DEM construction, Fan et al. [2005] and Park et al. [2006] have described a GPU-based algorithm for NNI; more recently, You and Zhang [2012] described a GPU-based NNI algorithm using CUDA. Danner et al. [2012] described an algorithm using MPI and GPU for parallel grid construction across a cluster of machines, focusing on RST interpolation.

Our contributions. In this article, we present *TerraNNI*, a simple yet very fast GPU-based algorithm that, given an input point set S in $\mathbb{R}^k$ for $k = 2, 3$ and a uniform grid $\mathbb{Q}$ in $\mathbb{R}^k$, computes the elevation of all points of $\mathbb{Q}$. Our algorithm can be used to interpolate any scalar field over a uniform grid in $\mathbb{R}^k$ from a set of sample points. For instance, it can be used to interpolate density in a volumetric dataset.

LiDAR scanners provide dense (high-resolution) point clouds of elevation data at most locations, but there are gaps. In space, these gaps usually appear at large bodies of water or human-made objects that have been removed from the point cloud in a preprocessing step. In time, the gaps appear when surveys fail to cover exactly the same region as previous surveys. When these gaps are large, it is desirable to label the corresponding *gap* cells in the volumetric grid with *nodata* instead of interpolating elevation based on points that are very far away. We introduce the notion of the *region of influence* for each input point, similar to the one used in α-shapes [Edelsbrunner and Mücke 1994]. For a grid point q, we use only those points to compute its elevation whose regions of influence contain q; for spatiotemporal data the region of influence has a temporal component as well. Our algorithm allows for an almost arbitrary region of influence; if this region is sufficiently large, then the algorithm computes the standard NNI.

Euclidean distance may not necessarily be the appropriate function to measure the distance between two points, especially for spatiotemporal data. We therefore assume that we have a metric $d(\cdot, \cdot)$ and compute Voronoi diagrams under this metric. While computing Voronoi diagrams on the CPU for general metrics is hard, it is relatively straightforward on a GPU. We also show how the computation of Voronoi diagrams can be optimized for Euclidean distance using the so-called lifting transform.

TerraNNI for 2D data. Exploiting the fact that we are interpolating elevations at grid points, TerraNNI uses a clever "blocking" scheme to expedite the computation considerably. In contrast to the algorithm in Fan et al. [2005], which performs NNI interpolation at ≤ 32 points in one step (or ≤ 128 points depending on hardware properties of the GPU card), TerraNNI can answer more than 10^6 NNI queries at grid points in one step. It exploits CUDA to substantially improve its efficiency by performing the majority of the computation on the GPU, thereby minimizing the communication between main and GPU memory.

These techniques lead to an extremely fast algorithm for computing a grid DEM. For example, TerraNNI computes a grid DEM covering a 600km^2 region at $2m$ resolution (i.e., $\approx$ 150 million grid points) from 2×10^9 input points in less than 40 minutes on a 3GHz Intel Core 2 Duo processer with an NVIDIA GeForce GTX 470 graphics card. Our CPU-based linear-interpolation algorithm takes more than 5 1/2 hours on the same PC. Not only is this a significant speedup, but also NNI interpolation produces a smoother grid DEM than the linear-interpolation method. The more sophisticated RST-based algorithm takes about 34 hours on the same dataset, even after throwing away a fraction of the points for efficiency, and the output between NNI and RST-based interpolations is nearly indistinguishable. Another advantage of our algorithm

over linear or RST interpolation is that it can be trivially parallelized, so it could be implemented easily on a GPU cluster.

TerraNNI for spatiotemporal data. Since GPUs support only 2D color and depth buffers, we process one 2D slice of $\mathbb{Q}$ each step. This involves processing each input point multiple times. We present three algorithms, which provide tradeoffs between the number of times each point is processed and the space (on the GPU) used by the algorithm. We refer to Table I in Section 5 for an overview of the relative performance of our algorithms.

Our experimental results demonstrate that TerraNNI is both scalable and efficient on spatiotemporal data. Since big multiyear LiDAR data are not yet publicly available, we tested the scalability of our algorithm by interpolating synthetic data. TerraNNI interpolates a 5,000 by 5,000 by 20 grid from a dataset consisting of 10^{10} input points spanning 20 years (186GB) in only 5 hours and 17 minutes. Additionally, we tested our algorithm on a smaller, nonsynthetic dataset consisting of about 20 million data points spanning 11 years. Our algorithm can interpolate more than 4×10^8 points from a 3D point cloud of 10^{10} points in approximately 5 hours. In contrast, Interpolate3d [Hemsley 2009], an implementation of NNI on the CPU, was unable to handle point clouds larger than 10^4 points.

Overview of the article. This article is organized as follows. Section 2 gives a short introduction to the fundamentals of the GPU model of computation, and Section 3 presents an algorithm for computing Voronoi diagrams, primarily in 2D and 3D. Section 4 presents our 2D algorithm, and Section 5 extends the 2D algorithm to 3D. In Section 6, we discuss our experimental results, exploring the performance and output quality of our algorithms.

2. GPU MODEL OF COMPUTATION

This section gives a brief overview of the computing primitives of a typical GPU (e.g., those that are relevant for our article).

The graphics pipeline. The graphics pipeline is responsible for drawing 3D scenes, composed of a set Ω of objects, onto a 2D image plane of pixels as seen from a viewpoint o. Because of their simplicity and flexibility, Ω is almost always a set of triangles. For each pixel $\pi = (x, y)$, where x, y is a global coordinate, the GPU finds the subset $\Omega_\pi \subseteq \Omega$ of objects intersected by the ray $o\vec{\pi}$. To store the information about the scene, the GPU efficiently maintains 2D arrays of pixels called buffers. We will use the following two buffers:

—The *depth buffer* $\mathbb{D}$ stores the distance to the nearest object from o for each pixel π. Given that p_j is the point of intersection for ray $o\vec{\pi}$ and object $\omega_j \in \Omega_\pi$, the GPU calculates and stores $\mathbb{D}[\pi] = \min_{\omega_j \in \Omega_\pi} \|op_j\|$.
—The *color buffer* $\mathbb{C}$ stores the color of the scene as viewed from o. Modern GPUs offer several options to define $\mathbb{C}[\pi]$ for a pixel π, often conditioned on the value of $\mathbb{D}[\pi]$. Let χ_j be the color of $\omega_j \in \Omega_\pi$. We will use the following two options: (1) $\mathbb{C}[\pi]$ is the color of the foremost object of Ω_π at π. That is, if $\omega_i = \arg\min_{\omega_j \in \Omega_\pi} ||op_j||$, then $\mathbb{C}[\pi] = \chi_i$. (2) $\mathbb{C}[\pi] = \sum_{\omega_j \in \Omega_\pi} \alpha_j \chi_j$, where $\alpha_j \in [0, 1]$ is a blending parameter. In our application, χ_j will be of the form 2^k and $\chi_j \neq \chi_{j'}$ for all pairs $\omega_j, \omega_{j'} \in \Omega_\pi$ and we will set $\alpha_j = 1$, so the blending function will set $\mathbb{C}[\pi]$ to the bitwise-OR of colors of objects in Ω_π.

A pair $\mathbb{F} = (\mathbb{C}, \mathbb{D})$ is referred to as a *frame buffer*, and the GPU can store multiple frame buffers simultaneously (as long as there is enough GPU memory available). One of these frame buffers is marked as the *active* frame buffer, and drawing commands go to the active frame buffer. It is usually clear from the context which frame buffer is

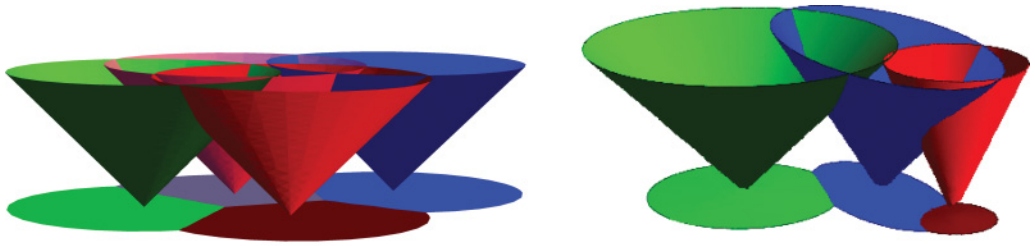


Fig. 4. Voronoi diagram as the lower envelope of a set of cones. The outer cells of a Voronoi diagram are infinite, but in this figure their sizes are limited because the cones are truncated.

active. We use the special value $\perp$ to denote the default values of the color and depth buffers. During graphical computations, all frame buffers and their associated color and depth buffers reside in memory on the graphics card. We let L denote the side length of the maximum buffer that the GPU can handle.

There are multiple libraries that provide APIs for working with GPU buffers and for rendering objects to these buffers; popular ones include OpenGL and Microsoft's DirectX. We use OpenGL. We also use CUDA, the parallel computing architecture by NVIDIA [2010], to perform general-purpose computation (not just rendering) directly on buffers in GPU memory, thereby reducing communication between the GPU and CPU.

3. COMPUTING VORONOI DIAGRAMS

This section defines a discretized version of the Voronoi diagram, suitable for computing on a GPU, and presents algorithms for computing it in $\mathbb{R}^2$ and $\mathbb{R}^3$.

3.1. Voronoi Diagrams

Let $S = \{p_1, \ldots, p_n\}$ be a set of points in $\mathbb{R}^k$, for $k \geq 1$, and let $d(\cdot, \cdot)$ be a convex distance function. For a point $p_i \in S$, we define its Voronoi cell $\mathrm{Vor}_S(p_i)$ with respect to $d(\cdot, \cdot)$ to be

$$\mathrm{Vor}_S(p_i) = \{a \in \mathbb{R}^k \mid d(a, p_i) \leq d(a, p_j) \quad 1 \leq j \leq n\}.$$

$\mathrm{Vor}(S)$ is the subdivision of $\mathbb{R}^k$ induced by the Voronoi cells of points in S; see Figure 3. We will omit the subscript S from $\mathrm{Vor}_S(p_i)$ when it is obvious from the context.

We introduce the notion of the *region of influence* $I \subset \mathbb{R}^k$, which is used to limit the region a point $p_i \in S$ can influence. We modify the distance function so that the distance $d(p_i, p)$ is infinity for points p outside $I + p_i$. Note that a point $p \in \mathbb{R}^k$ that does not lie in $I + p_i$ for any $p_i \in S$ does not belong to the Voronoi cell of any point of S. This implies that $\mathrm{Vor}(S)$ is not a subdivision of the entire set $\mathbb{R}^k$ but only of the region $\cup_{p \in S}(I + p)$. For simplicity, we will describe algorithms with the assumption that I is a disc in 2D and a cylinder in 3D. The algorithms can be modified to handle influence regions of more general convex shapes.

Vor(S) *as a lower envelope*. For a point $p \in \mathbb{R}^k$, let $f_p : \mathbb{R}^k \to \mathbb{R}$ be the function $f_p(u) = d(u, p)$. With a small abuse of notation, we define $f_i = f_{p_i}$ for $1 \leq i \leq n$. The *lower envelope* f of $F = \{f_1, \ldots, f_n\}$ is defined as

$$f(u) = \min_{1 \leq i \leq n} f_i(u).$$

That is, $f(u)$ is the distance from u to its nearest neighbor in S. Therefore, $\mathrm{Vor}(S)$ is a projection of the graph of f onto $\mathbb{R}^k$; see Figure 4.

If $d(\cdot,\cdot)$ is the Euclidean distance, that is,

$$d(a,b) = \left(\sum_{j=1}^{k}(a_j - b_j)^2\right)^{1/2},$$

then $\mathrm{Vor}(S)$ can be represented as the lower envelope of a family of linear functions using the so-called lifting transform [de Berg et al. 1997], as follows: let $p_i = (p_{i1}, \ldots, p_{ik})$ for $1 \le i \le n$. For a point $a = (a_i, \ldots, a_k) \in \mathbb{R}^k$,

$$\begin{aligned}\arg\min_i f_i(a) &= \arg\min_i f_i^2(a) = \arg\min_i \sum_{j=1}^{k} a_j^2 - 2a_j p_{ij} + p_{ij}^2 \\ &= \arg\min_i \sum_{j=1}^{k} -2a_j p_{ij} + p_{ij}^2 \\ &= \arg\min_i g_i(a),\end{aligned}$$

where

$$g_i(a) = -2\sum_{j=1}^{k} a_j p_{ij} + \sum_{j=1}^{k} p_{ij}^2.$$

The projection of the lower envelope of $G = \{g_1, \ldots, g_n\}$ is therefore identical to that of F, and the graph of each g_i is a hyperplane. As we will see later, working with G is substantially simpler and more efficient than working with F.

Discretized Voronoi diagram. Let $\mathbb{B}$ be an axis-aligned box in $\mathbb{R}^k$. We are interested in computing a discretized version of $\mathrm{Vor}(S)$ inside $\mathbb{B}$, defined later. We discretize $\mathbb{B}$ into a $N_1 \times \cdots \times N_k$ uniform k-dimensional grid of *voxels*. Each voxel is a (tiny) box and its volume is $\mu = \mathrm{Vol}(\mathbb{B})/(\Pi_{i=1}^{k} N_i)$. We use $\hat{v}$ to denote the center point of voxel v. With a slight abuse of notation, we will also use $\mathbb{B}$ to denote this k-dimensional grid of voxels. For a voxel $v \in \mathbb{B}$, let $\varphi(v, S)$ denote the point of S that is nearest to $\hat{v}$ under the distance function $d(\cdot,\cdot)$. If $d(\hat{v}, p) = \infty$ for all $p \in S$, then we consider $\varphi(v, S)$ to be undefined. We discretize $\mathrm{Vor}(S) \cap \mathbb{B}$ by assuming that the entire voxel v lies in the Voronoi cell of $\varphi(v, S)$. For a point $p_i \in S$, we define its *discretized Voronoi cell* as

$$\mathrm{Vor}^{\square}(p_i) = \{v \mid \varphi(v, S) = p_i\}.$$

The quantity $\mu|\mathrm{Vor}^{\square}(p_i)|$ approximates the volume of $\mathrm{Vor}(p_i)$ inside $\mathbb{B}$. This approximation improves as we increase the resolution, that is,

$$\lim_{N_1,\ldots,N_k \to \infty} \mu|\mathrm{Vor}^{\square}(p_i)| = \mathrm{Vol}(\mathrm{Vor}(p_i) \cap \mathbb{B}).$$

$\mathrm{Vor}^{\square}(S)$ is the subdivision of $\mathbb{B}$ induced by $\mathrm{Vor}^{\square}(p_i)$ for $1 \le i \le n$; see Figure 5(a). Note that, as in the continuous case, a voxel whose center does not lie in $I + p_i$, for any $p_i \in S$, does not belong to the discretized Voronoi cell of any point of S.

3.2. Computing Voronoi Diagrams in 2D

We now describe a GPU-based algorithm, which we refer to as GVor, for computing $\mathrm{Vor}^{\square}(S)$ for a point set $S \subset \mathbb{R}^2$. In $\mathbb{R}^2$, $\mathbb{B}$ is an $N_1 \times N_2$ rectangle and each voxel $\pi \in \mathbb{B}$ is a two-dimensional square with side length ρ; we call π a *pixel* of $\mathbb{B}$. We assume I, the region of influence, to be a disk of radius r.

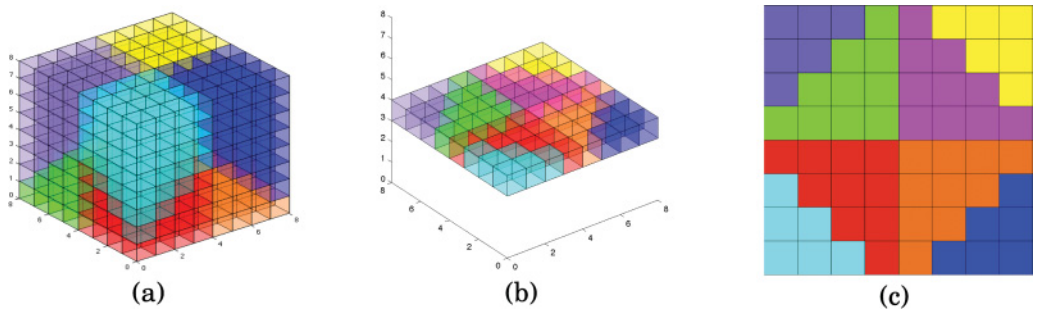


Fig. 5. Pixelized Voronoi diagram where $d(\cdot, \cdot)$ is the Euclidean metric. (a) $\mathrm{Vor}^{\square}(S)$ for a set of eight points; (b, c) $\mathrm{Vor}^{\square}{}_S(3)$, shown as a slice of $\mathbb{B}$ from (a).

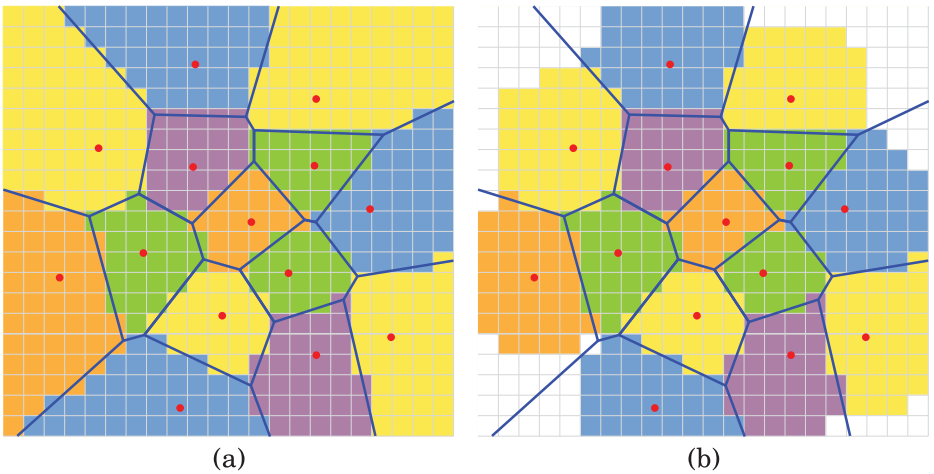


Fig. 6. Example of $k = 2$ using the Euclidean metric and setting I to a disk of radius 5: (a) Discretized Voronoi diagram $\mathrm{Vor}^{\square}(S)$ without a truncated d. (b) Discretized Voronoi diagram $\mathrm{Vor}^{\square}(S)$ with d truncated to a circular region of influence.

We map the image plane of Π of the GPU to $\mathbb{B}$ with each pixel of π mapping to a pixel in the image plane; thus, we do not distinguish between an entry in the color buffer $\mathbb{C}[\pi]$ and (the center of) the corresponding pixel $\pi \in \mathbb{B}$.[1] We use the color buffer to store $\mathrm{Vor}^{\square}(S)$ on the GPU by viewing each pixel as a single word (concatenation of R, G, B, A components). Specifically, if $\varphi(\pi, S) = p_i$ for some pixel π, then the color buffer stores $\mathbb{C}[\pi] = i$. If $\varphi(\pi, S)$ is undefined, the color buffer $\mathbb{C}[\pi]$ contains the uninitialized value $\perp$; this happens when $\hat{\pi}$, the center point of π, is outside the region of influence for all points of S. Figure 6 shows an example of two discretized Voronoi diagrams $\mathrm{Vor}^{\square}(S)$ corresponding to different regions of influence (and thus truncations of d).

Convex distance function. If $d(\cdot, \cdot)$ is a general convex function, we proceed as follows. For a point $p \in \mathbb{R}^k$, let γ_p denote the graph of the function f_p and let $\Gamma = \{\gamma_1, \ldots, \gamma_n\}$, where $\gamma_i = \gamma_{p_i}$. A point $a \in \mathbb{R}^2$ is in $\mathrm{Vor}^{\square}(p_i)$ if $f(a) = f(a_i)$; that is, the ray from a in the $+z$ direction hits γ_i before any other $\gamma' \in \Gamma$. In other words, $\varphi(\pi, S) = p_i$ if γ_i is the surface seen at $\hat{\pi}$ when the set Γ is viewed from $z = -\infty$. This suggests

[1]Here we assume that the size of Π is larger than that of $\mathbb{B}$, that is, that $N_1, N_2 < L$. We describe in Section 4 how to adapt the algorithm if $\mathbb{B}$ is larger than Π.

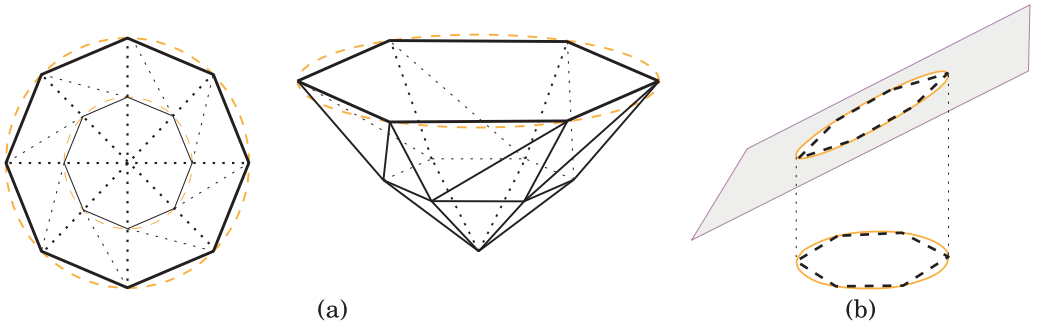


Fig. 7. (a) The triangulation of γ_i into $\gamma_i^{\diamond}$ using two triangle strips ($m = 2$). (b) In the case of the Euclidean metric, the lifting transform is used to produce a simple $\gamma_i^{\triangle}$.

that we can compute $\mathrm{Vor}^{\square}(S)$ by rendering the surfaces of Γ with the viewpoint set appropriately. However, since GPUs are optimized to render triangles, not an arbitrary surface, we construct a triangulated surface $\gamma_p^{\triangle}$ approximating γ_p. We use a simple triangulation that is defined by two integer parameters: k and m. We first compute m level-set curves (contour lines) at regular height intervals on γ_i, that is, intersect γ_i with m horizontal planes (e.g., dashed circles in Figure 7(a)). Since $d(\cdot, \cdot)$ is assumed to be convex, each level set is a convex closed curve, which we approximate by a convex k-gon, by choosing k points on the curve. We then create triangles between adjacent k-gons using $2k$ triangles. The resulting triangulated surface, denoted by $\gamma_p^{\triangle}$, is composed of $2km$ triangles; see Figure 7(a). Let $\Gamma^{\triangle} = \{\gamma_1^{\triangle}, \ldots, \gamma_n^{\triangle}\}$, where $\gamma_i^{\triangle} = \gamma_{p_i}^{\triangle}$. We can approximate $\gamma_p^{\triangle}$ as close to γ_p as desired by choosing the parameters m and k appropriately.

To compute $\mathrm{Vor}^{\square}(S)$, we set the colors of all triangles in $\gamma_i^{\triangle}$ to i and render the triangles of $\Gamma^{\triangle}$ onto Π using the GPU. We note that after rendering $\Gamma^{\triangle}$, the depth buffer encodes the distances in the sense that $\mathbb{D}[\pi]$ stores the (approximated) value of $d(\hat{\pi}, \varphi(\pi, S))$, the distance from the center of π, to its nearest neighbor in S. $\mathbb{C}$ is set to store the color of the foremost object at π (cf. Section 2).

Euclidean distance function. The number of triangles needed to get a good approximation $\gamma_i^{\triangle}$ of f_i can be large, depending on the complexity of $d(\cdot, \cdot)$ and the desired level of accuracy, and it can affect the efficiency of the algorithm. For Euclidean distance functions, much higher accuracy can be obtained with just a few triangles, by using the lifting transform, that is, working with the functions $g_1, \ldots, g_n$. Recall that we assume I to be a disk of radius r. Therefore, the (truncated) surface γ_i, the graph of function g_i truncated within I, is the planar elliptic region

$$\gamma_i = \{(u, g_i(u)) \mid u \in \mathbb{R}^2, ||up_i|| \leq r\};$$

see Figure 7(b). The ellipse γ_i can be triangulated using a convex k-gon as follows. Let σ be a regular k-gon in $\mathbb{R}^2$ inscribed in the disk of radius r centered at the origin. We set $\sigma_i = \sigma + p_i^*$, where p_i^* is the projection of p_i on the xy-plane. Lifting σ_i to h_i yields the surface

$$\gamma_i = \{(u, g_i(u)) \mid u \in \sigma_i\}.$$

For each vertex v of σ_i, there is a vertex $(v, g_i(v))$ in γ_i (Figure 7(b)). We can easily triangulate γ_i using k triangles, say, by drawing an edge from the center of $g_i(p_i)$ of γ_i to

each vertex of y_i. Note that the resulting triangulated surface $\gamma_i^{\triangle}$ encodes the function g_i exactly; it only approximates the influence region of p_i.

The GVOR(S) *procedure*. In the context of NNI, each point p_i of S is endowed with height $h(p_i)$. We set the color of all triangles of $\gamma_i^{\triangle}$ to be $h(p_i)$. Thus, for all pixels $\pi \in \text{Vor}^{\square}(p_i)$, $\mathbb{C}[\pi]$ stores $h(p_i)$. The GVOR(S) algorithm returns the frame buffer $\mathbb{F} = (\mathbb{C}, \mathbb{D})$; that is, for each pixel $\pi \in \text{Vor}^{\square}(p_i)$, we have its distance to p_i (from $\mathbb{D}$) and $h(p_i)$ (from $\mathbb{C}$).

3.3. 3D Voronoi Diagrams

We now extend the previous algorithm to compute $\text{Vor}^{\square}(S)$ in $\mathbb{R}^3$. We view $\mathbb{R}^3$ as xyt-space—x,y being spatial dimension and t being the time axis; each point $p = (x_p, y_p, t_p)$ is located in a 2D region, with (x_p, y_p) as its spatial coordinates, and has a time value t_p associated with it. We discretize the box $\mathbb{B}$ into an $N_1 \times N_2 \times N_3$ uniform 3D grid of voxels; N_3 is the temporal resolution of the grid. As mentioned earlier, we assume the region of influence I to be a cylinder of radius r in the xy-plane and height $2r$ along the t-axis, that is, $I = \{(x, y, t) \mid ||(x, y)|| \leq r, |t| \leq r\}$.

Recall from Section 3.1 that the problem of computing $\text{Vor}^{\square}(S)$ can be phrased as that of rendering a 4D scene onto a 3D hyperplane. However, GPUs can render only 3D scenes on a 2D plane, and they have only 2D buffers. We therefore decompose the problem into that of rendering each of the 2D slices of $\mathbb{B}$, one for each fixed time, separately and combining these N_3 slices at the end to construct $\text{Vor}^{\square}(S)$.

Fix a time slice τ of $\mathbb{B}$, that is, the set of voxels with time index τ, let Π_τ be the horizontal plane passing through the center points of the voxels of this slice. $\mathbb{B}_\tau = \mathbb{B} \cap \Pi_\tau$ is a rectangle with an $N_1 \times N_2$ 2D uniform grid induced on it. Each pixel υ_τ of this 2D grid, the intersection of a voxel υ with Π_τ, is a square; $\hat{\upsilon}$ is the center point of both υ and υ_τ. For $p_i \in S$, we define its *Voronoi cell slice* on Π_τ as

$$\text{Vor}^{\square}(p_i, \tau) = \{\upsilon_\tau \in \mathbb{B}_\tau \mid \upsilon \in \text{Vor}^{\square}(p_i)\}.$$

$\text{Vor}^{\square}(S, \tau)$ is the subdivision of $\mathbb{B}_\tau$ induced by these cell slices, which can be viewed as a discretization of the 2D slice of $\text{Vor}(S) \cap \Pi_\tau$ inside $\mathbb{B}$; see Figures 5(a) and 5(b).

We first describe how to compute $\text{Vor}^{\square}(S, \tau)$ for a convex distance function and then improve it for Euclidean distance.

Convex distance function. For a fixed $\tau \in \mathbb{R}$ and for a given i, $1 \leq i \leq n$, we define functions $f_i^\tau : \mathbb{R}^2 \to \mathbb{R}$ as follows. For $u \in \mathbb{R}^2$, $f_i^\tau(u) = d((u, \tau), p_i)$; that is, f_i^τ is the restriction of the distance function from p_i at $t = \tau$.

Let $f^\tau(u) = \min_i f_i^\tau(u)$ be the lower envelope of $F^\tau = \{f_1^\tau, \ldots, f_n^\tau\}$. $\text{Vor}(S, \tau)$, the 2D slice of $\text{Vor}(S)$ for $t = \tau$, is the projection of the graph of f^τ on the xy-plane. As in Section 3.2, $\text{Vor}^{\square}(\tau)$ can be computed by viewing $\mathbb{B}_\tau$ as the 2D image plane and rendering the triangulated surfaces $\gamma_1^{\triangle}, \ldots, \gamma_n^{\triangle}$ with $(0, 0, -\infty)$ as the viewpoint; here, $\gamma_i^{\triangle}$ is a triangulated surface approximating the graph of f_i^τ. By repeating this for all slices $0 \leq \tau < N_3$, we can compute $\text{Vor}^{\square}(S)$; however, this requires rendering each of the surfaces $\gamma_i^{\triangle}$ a total of N_3 times (once for each slice of $\mathbb{B}$). This can be reduced to $2r$ by exploiting the region of influence as follows: for a fixed τ, let $S^\tau = \{p \in S \mid \tau - r \leq t_p \leq \tau + r\}$. Then $\text{Vor}^{\square}(S, \tau) = \text{Vor}^{\square}(S^\tau, \tau)$. So while computing $\text{Vor}^{\square}(S, \tau)$, we only render $\gamma_i^{\triangle}$s for $p_i \in S^\tau$.

Euclidean distance function. For the Euclidean distance function, we again compute $\text{Vor}^{\square}(S, \tau)$ using the functions $G^{\tau} = \{g_1, \ldots, g_n\}$, defined as follows. For $1 \leq i \leq n$, define $g_i^{\tau} : \mathbb{R}^2 \to \mathbb{R}$ as

$$g_i^{\tau}(x, y) = f_i^{\tau}(x, y)^2 - x^2 - y^2 = -2xx_i - 2yy_i + x_i^2 + y_i^2 + (t_i - \tau)^2,$$

and let $g^{\tau}(x, y) = \min_i g_i^{\tau}(x, y)$ be the lower envelope of $\{g_i^{\tau}, \ldots, g_n^{\tau}\}$. $\text{Vor}(S, \tau)$ is the projection of the graph of g^{τ} onto the xy-plane. Note that each g_i^{τ} is a linear function. Since we assume the influence region I to be a cylinder, its cross-section for $t = \tau$ is a disk. As in $\mathbb{R}^2$, the truncation of the graph of g_i^{τ} within the influence region of p_i is an ellipse. We therefore approximate it to a convex k-gon $\gamma_i^{\triangle}$ as in $\mathbb{R}^2$.

The GVOR(S, τ) *procedure*. We use GVOR(S, τ) to refer to the algorithm that computes the slice of $\text{Vor}^{\square}(S, \tau)$. As in 2D, the algorithm returns the frame buffer containing the color buffer and depth buffer $\mathbb{F} = (\mathbb{C}, \mathbb{D})$. In the context of NNI, we assign the color $h(p_i)$ to the surface $\gamma_i^{\triangle}$, so for all pixels $\pi \in \text{Vor}^{\square}(p_i, \tau)$, $\mathbb{C}[\pi]$ stores $h(\pi)$.

4. NATURAL NEIGHBOR INTERPOLATION IN 2D

We now describe a GPU-based algorithm for computing NNI on a set $\mathbb{Q}$ of $M \times M$ grid points. We will be using the discretized Voronoi diagram, described in Section 3, instead of the standard Voronoi diagram, so we redefine the NNI function $h : \mathbb{R}^2 \to \mathbb{R}$ as follows. For a point $q \in \mathbb{R}^2$,

$$h(q) = \frac{\sum_{p \in S} |\text{Vor}_S^{\square}(p) \cap \text{Vor}_{S \cup \{q\}}^{\square}(q)| \; h(p)}{|\text{Vor}_{S \cup \{q\}}^{\square}(q)|} := \frac{N(q)}{D(q)}. \tag{3}$$

We note that Equation (3) is an approximation of Equations (1) and (2) because of the following:

(i) The tessellation error from the triangulated $\gamma^{\triangle}$ surfaces for non-Euclidean metrics
(ii) The discretization error
(iii) The limited precision of the depth buffer $\mathbb{D}$ (which can cause problems at the boundaries between two Voronoi cells)
(iv) The region of influence

By adjusting the corresponding parameters, we can bring Equation (3) as close to Equation (1) as we wish.[2] Here is a brief outline of the algorithm. We choose a scaling parameter s so that each grid cell of $\mathbb{Q}$ contains $s \times s$ grid cells of the box $\mathbb{B}$ on which $\text{Vor}^{\square}(S)$ is computed. The algorithm works in two stages. The first stage computes $\text{Vor}^{\square}(S)$ using the GVOR(S) algorithm, described in Section 3. The second phase computes $\text{Vor}^{\square}{}_{S \cup \{q\}}(q)$ for every $q \in \mathbb{Q}$. Instead of computing $\text{Vor}^{\square}{}_{S \cup \{q\}}(q)$ sequentially for each $q \in \mathbb{Q}$, one by one, we use the features of the GPU to compute $\text{Vor}^{\square}{}_{S \cup \{q\}}(q)$ for several grid points, and possibly for all points in $\mathbb{Q}$ if $\mathbb{Q}$ is not very large, in one pass. This is accomplished using two ideas. First, as in Fan et al. [2005], if the bit depth of the color buffer is w, we compute $\text{Vor}^{\square}{}_{S \cup \{q\}}$ for w points of $\mathbb{Q}$ by packing the bits of the color buffer carefully. Second, we observe that if two grid points $q_1, q_2 \in \mathbb{Q}$ are not very close to each other, then $\text{Vor}^{\square}{}_{S \cup \{q_1\}}(q_1)$ and $\text{Vor}^{\square}{}_{S \cup \{q_2\}}(q_2)$ are independent and thus can be computed in parallel using the GPU. We make these statements precise later. Before describing the algorithm in detail, we introduce some notation.

[2]For instance, the tesselation error can be reduced by increasing the number of triangles in $\gamma^{\triangle}$, thus achieving a smaller difference between $\gamma^{\triangle}$ and γ. Similarly, the region-of -influence approximation can be reduced by carefully truncating the $\gamma^{\triangle}$ surfaces and tweaking the range of the depth buffer.

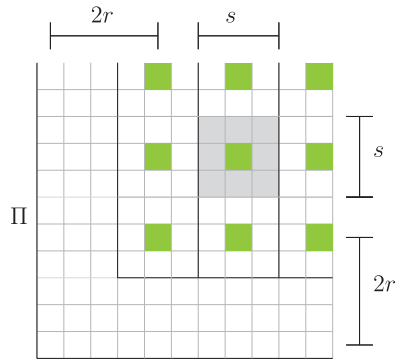


Fig. 8. Embedding $\mathbb{Q}$ (green points) on Π; $s = 3$ and $r = 2$. The shaded area shows the $s \times s$ pixels around a point of $\mathbb{Q}$.

For simplicity, we assume that the length of each grid cell $\mathbb{Q}$ is 1, and we use $\mathbb{Q}[i, j]$ to denote the (i, j)'th query point of $\mathbb{Q}$ for $0 \leq i, j < M$. Let s be the scaling parameter and w the bit depth of the color buffer, as mentioned earlier. For simplicity, we assume s to be odd, M to be a power of 2, and $B = \sqrt{w}$ also to be a power of 2. The length of each pixel is $1/s$. Suppose the frame buffer of the GPU has size $N \times N$. $\mathbb{Q}$ is mapped to the image plane Π so that each grid point of $\mathbb{Q}$ lies at the center of an $s \times s$ array of pixels of Π. As in Section 3, we assume that the region of influence is a disk of radius r, and thus we can assume that all points of S lie within distance $2r$ from $\mathbb{Q}$. Let $R(\mathcal{Q})$ denote the square $\mathcal{Q} \oplus [-2r, 2r]^2$, where $\mathcal{Q}$ is the bounding square of $\mathbb{Q}$; that is, $R(\mathcal{Q})$ is the square of side length $M + 4r$. Let $\alpha = (s - 1)/2 + 2rs$. We map the bottom left corner of $\mathcal{Q}$ to that of $\mathbb{B}$, so the query point $\mathbb{Q}[i, j]$ maps to the pixel $(is + \alpha, js + \alpha)$; see Figure 8.

We first define the algorithm under the following two assumptions:

(A) (A1) $L \geq (M + 4r)s$, that is, $M \leq L/s - 4r$
(B) (A2) $r \leq B/2$

Recall, L denotes the side length of the maximum buffer that the GPU can handle. The first assumption (A1) ensures that $R(\mathcal{Q})$ can be mapped to $\mathbb{B}$ without exceeding the amount of available memory on the GPU. For a graphics card with $w = 32$ and with $s = 5$, the second assumption implies that $r \leq 5$ meters if the grid $\mathbb{Q}$ has a resolution of 2 meters. In other words, the height of a grid point of $\mathbb{Q}$ is not affected by an input point that is more than 10 meters away. With high-resolution LiDAR datasets having potentially better than submeter and near-uniform density, this is a reasonable assumption. Nevertheless, we will discuss at the end how we adapt our algorithm if (A1) and (A2) do not hold.

We are now ready to describe the algorithm.

First phase. We create a new active frame buffer $\mathbb{F} = (\mathbb{C}^{(1)}, \mathbb{D})$ and call GVor(S). Recall that the color of the surface corresponding to $p_i \in S$ is set to $h(p_i)$. After the first phase, $\mathbb{C}^{(1)}[\pi]$ stores $h(p_i)$ for all pixels $\pi \in \text{Vor}^{\square}(p_i)$. Before exiting from phase 1, we create a new blank color buffer $\mathbb{C}^{(2)}$, replacing $\mathbb{C}^{(1)}$ in the active frame buffer, but we leave $\mathbb{C}^{(1)}$ in the GPU memory. We leave the depth buffer intact in the active frame buffer; that is, $\mathbb{D}[\pi]$ continues to store $\|\pi\varphi(\pi, S)\|$, the distance from the center of π to $\varphi(\pi, S)$.

Second phase. The second phase computes $N(q)$ and $D(q)$, as defined in Equation (3), for every $q \in \mathbb{Q}$. We proceed as follows: we set the depth buffer in read-only mode, but

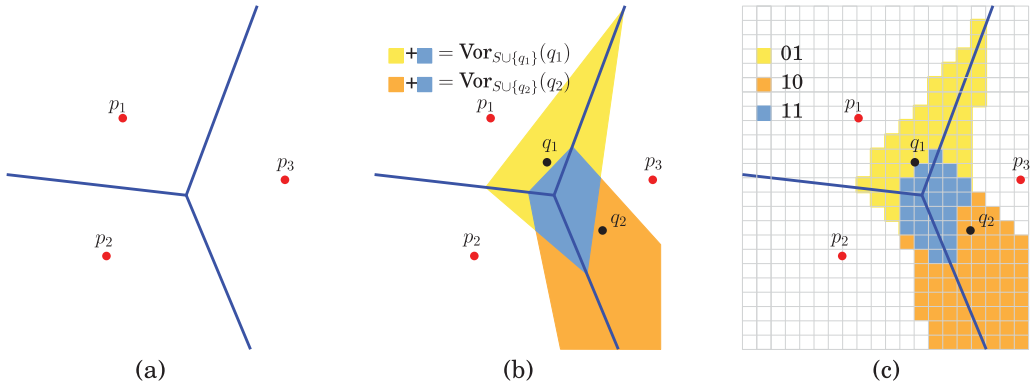


Fig. 9. (a) $\mathrm{Vor}(S)$ of a set S. (b) $\mathrm{Vor}(S\cup\{q_1\})$ and $\mathrm{Vor}(S\cup\{q_2\})$ for two query points q_1 and q_2. (c) $\mathrm{Vor}^{\square}(S\cup\{q_1\})$ and $\mathrm{Vor}^{\square}(S\cup q_2)$. The colors correspond to the bitwise-OR colors of the query point. Query point q_1 and q_2 have colors 01 and 10, respectively; the pixels in $\mathrm{Vor}^{\square}{}_{S\cup\{q_1\}}(q_1)\cap\mathrm{Vor}^{\square}{}_{S\cup q_2}(q_2)$ thus get the color $01\vee 10 = 11$, where $\vee$ is bitwise-OR.

recall that it stores the depth values computed in the first phase. For each $q \in \mathbb{Q}$, we do the following. We render all triangles of the surface $\gamma_q^{\triangle}$. Since $\mathbb{D}$ is left intact and the color buffer $\mathbb{C}^{(2)}$ is blank, this step computes $\mathrm{Vor}^{\square}{}_{S\cup\{q\}}(q)$. We compute $N(q)$ by adding the values of $\mathbb{C}^{(1)}[\pi]$ for all π for which $\mathbb{C}^{(2)}[\pi] \neq \perp$. $D(q)$ is the number of pixels π with $\mathbb{C}^{(2)}[\pi] \neq \perp$. After processing each q, we reset all entries of $\mathbb{C}^{(2)}$ to $\perp$. Note that since $\mathbb{D}$ is in read-only mode, it is not modified while processing q.

Next, we describe how to parallelize the computation of $\mathrm{Vor}^{\square}{}_{S\cup\{q\}}(q)$ by using the bit-packing trick and exploiting the independence of influence region of faraway points.

Packing bits of $\mathbb{C}^{(2)}$. Let $q_1, q_2, \ldots, q_m$, for $m < w$, be a subset of points in $\mathbb{Q}$. We show how to compute the height at all these points in one pass. Let $\gamma_i^{\triangle}$ be the triangulated surface that approximates the function f_{q_i}. The color of all triangles in the surface $\gamma_i^{\triangle}$ is set to 2^i. This ensures that colors of $\gamma_1^{\triangle}, \ldots, \gamma_m^{\triangle}$ are bitwise disjoint.

Suppose a triangle T of color χ is being rendered. If the depth of T at a pixel π is larger than $\mathbb{D}[\pi]$, then nothing happens. Otherwise, $\mathbb{C}^{(2)}[\pi]$ is set to $\mathbb{C}^{(2)}[\pi] \leftarrow \mathbb{C}^{(2)}[\pi]\vee\chi$, where $\vee$ is the bitwise-OR operation. Recall that $\mathbb{D}[\pi]$ is not updated, as it is in read-only mode, and that $\mathbb{D}[\pi]$ stores $\|\pi\varphi(\pi)\|$. After rendering all $\gamma_1^{\triangle}, \ldots, \gamma_m^{\triangle}$, the ith bit of $\mathbb{C}[\pi]$ is 1 if $\pi \in \mathrm{Vor}^{\square}_{S\cup\{q_i\}}(q_i)$; see Figure 9(c).

We let $\mathbb{C}^{(2)}$ denote the contents of the buffer after the second phase. We compute $N(q_i)$ by summing the values of $\mathbb{C}^{(1)}[\pi]$ for all pixels π for which the i^{th} bit of $\mathbb{C}^{(2)}[\pi]$ is 1 and then dividing the sum by the number of such pixels. We have computed $N(q_i)$ and $D(q_i)$, and thus $h(q_i)$, for all $1 \le i \le m$ by rendering all surfaces in one pass.

Processing independent points. We call two points $p, q \in \mathbb{R}^2$ *independent* if $||pq|| > 2r$ (i.e., their influence regions are disjoint). A useful property of independent points is that $\mathrm{Vor}^{\square}{}_{S\cup\{p\}}(p)$ and $\mathrm{Vor}^{\square}{}_{S\cup\{q\}}(q)$ are disjoint, so a pixel belongs to at most one of them. Given a pixel $\pi \in \Pi$ that is known to lie in one of them, we can determine in $O(1)$ time whether $\pi \in \mathrm{Vor}^{\square}{}_{S\cup\{p\}}(p)$ or $\pi \in \mathrm{Vor}^{\square}{}_{S\cup\{q\}}(q)$—the former if p is closer to π than q. We can therefore render both $\gamma_p^{\triangle}$ and $\gamma_q^{\triangle}$ in one pass using the same color. With this observation at hand, we proceed as follows.

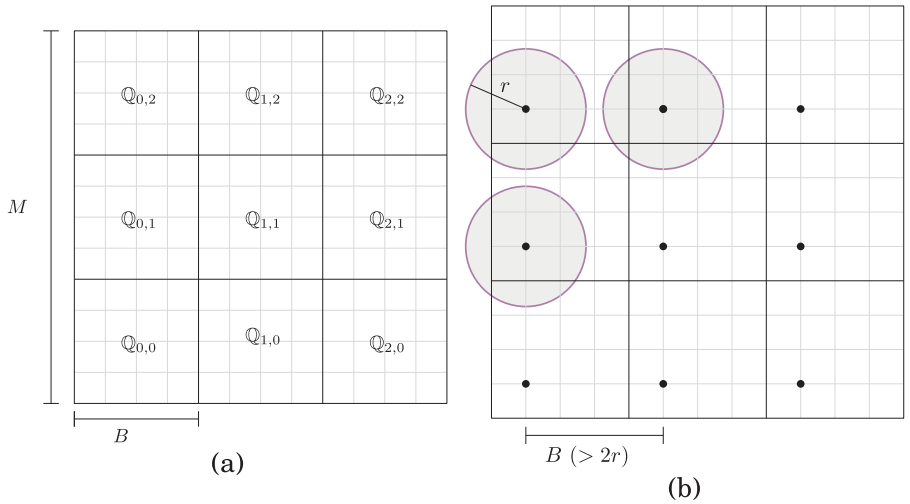


Fig. 10. (a) Splitting $\mathbb{Q}$ into query tiles of size $B \times B = 16$ query points, with $B = 4$. (b) All the $\mathbb{Q}^*_{[1,1]}$ query points are independent since their areas of influence (depicted by circles in the figure) are disjoint.

We partition $\mathbb{Q}$ into $(M/B)^2$ query *tiles*, each of size $B \times B$. For $\mathbf{i} = (i_1, i_2) \in [0, M/B)^2$, the $\mathbf{i}$ tile denoted by $\mathbb{Q}_\mathbf{i}$ is $\mathbb{Q}_\mathbf{i} = \{\mathbb{Q}[j_1, j_2] \mid i_1 B \le j_1 < (i_1 + 1)B, i_2 B \le j_2 < (i_2 + 1)B\}$. See Figure 10(a). A query point $q \in \mathbb{Q}$ can be represented by a pair $(\mathbf{a}, \mathbf{t})$, where $\mathbf{a} \in [0, M/B - 1]^2$ is the index of the query tile that contains q and $\mathbf{t} \in [0, B-1]^2$ is the offset of q within that query tile. For a fixed offset $\mathbf{t}$, we define $\mathbb{Q}^*_\mathbf{t} = \{(\mathbf{a}, \mathbf{t}) \mid \mathbf{a} \in [0, M/B - 1]^2\}$ to be the set of points of $\mathbb{Q}$ with offset $\mathbf{t}$. By assumption (A2), $r < B/2$. Therefore, for any $\mathbf{t} \in [0, B-1]^2$, the points in $\mathbb{Q}^*_\mathbf{t}$ are pairwise independent—the distance between points of $\mathbb{Q}^*_\mathbf{t}$ lying in adjacent tiles is $B > 2r$; see Figure 10(b).

For $\mathbf{t} = (t_1, t_2) \in [0, B)^2$, we set $\chi(t) = 2^{t_1 B + t_2}$ and assign the color $\chi(t)$ to the query surfaces $\gamma_q^\triangle$ for all $q \in \mathbb{Q}^*_\mathbf{t}$. We render the surface $\gamma_q^\triangle$ for all $q \in \mathbb{Q}$ one by one while keeping the depth buffer $\mathbb{D}$ in read-only mode. Let $\mathbb{C}^{(2)}$ denote the content of the color buffer after rendering all of these surfaces. We process each pixel $\pi \in \mathbb{C}^{(2)}$ as follows: If the ℓth bit of π is 1 (i.e., it has been rendered by a query point with the offset $\mathbf{l} = (\lfloor \ell / B \rfloor, \ell \bmod B)$), $\pi \in \mathrm{Vor}^\square_{S \cup \{q\}}(q)$ for some $q \in \mathbb{Q}^*_\mathbf{l}$. Then we compute in $O(1)$ time the point $q_\pi \in \mathbb{Q}^*_\mathbf{l}$ that is nearest to π; let $\mathrm{NN}(\pi, \mathbf{l})$ denote this procedure. As stated earlier, among all points of $\mathbb{Q}^*_\mathbf{l}$, $\pi \in \mathrm{Vor}^\square_{S \cup \{q_\pi\}}(q_\pi)$. For each $q \in \mathbb{Q}$, we now compute $N(q)$ and $D(q)$ as before. See Algorithm 1 for the pseudo-code.

This completes the description of the algorithm. Hence, if assumptions (A1) and (A2) hold, the height of all points in $\mathbb{Q}$ can be computed in one pass. Next we describe how to adapt the algorithm if (A1) or (A2) does not hold.

Handling larger regions of influence. If the assumption (A2) does not hold (i.e., the influence region is too big), the first phase of the algorithm, which computes $\mathrm{Vor}^\square(S)$, remains the same, but the second phase works in multiple passes, each computing the height of a subset of grid points.

We set $\overline{B} = 2^{\lceil \log_2 2r \rceil}$ and partition $\mathbb{Q}$ into *blocks* of size $\overline{B} \times \overline{B}$. Two grid points of $\mathbb{Q}$ with the same offset remain independent and thus can be processed simultaneously as before. But each block has more than w grid points, so one cannot process all of them in one pass using the bit-packing approach.

ALGORITHM 1: Computing $N(q)$ and $D(Q)$.

```
for i ← 0 to M − 1 do
  for j ← 0 to M − 1 do
    N[i, j] ← 0, D[i, j] ← 0
for all π ∈ C^(2) do
  if C^(2)[π] ≠ ⊥ and C^(1)[π] ≠ ⊥ then
    v = C^(2)[π]
    for ℓ ← 0 to w − 1 do
      v_ℓ = ℓ'th bit of v
      if v_ℓ = 1 then
        (i, j) = NN(π, ℓ)
        N[i, j] ← N[i, j] + C^(1)[π]
        D[i, j] ← D[i, j] + 1
return (N, D)
```

We partition each block $\mathbf{j}$ into $\overline{B}^2/B^2$ tiles, each of size $B \times B$. Each tile is represented by a pair $\mathbf{i} = (i_1, i_2) \in [0, \overline{B}/B)^2$ and each grid point in a tile is represented by its offset $\mathbf{t} \in [0, B)^2$. Therefore, a grid point q with offset $\mathbf{t}$, lying in the tile $\mathbf{i}$, of a block $\mathbf{j}$ is represented as the triple $(\mathbf{j}, \mathbf{i}, \mathbf{t})$.

For a fixed $\mathbf{i} \in [0, \overline{B}/B)^2$, let

$$\mathbb{Q}^*_{\mathbf{i}} = \{(\mathbf{j}, \mathbf{i}, \mathbf{t}) \mid \mathbf{j} \in [0, M/B)^2, \mathbf{t} \in [0, B)^2\}$$

be the set of all grid points of $\mathbb{Q}$ that lie in a tile whose index is $\mathbf{i}$.

We note that two points in $\mathbb{Q}^*_{\mathbf{i}}$ with the same offset but lying in two different blocks are independent because $\overline{B} > 2r$. Furthermore, there are at most w points of the same block in $\mathbb{Q}^*_{\mathbf{i}}$. Hence, we can process all points of $\mathbb{Q}^*_{\mathbf{i}}$ in one pass. Namely, for any point $q \in \mathbb{Q}^*_{\mathbf{i}}$ with offset $\mathbf{t} = (t_1, t_2)$, we set the color of $\gamma_q^{\triangle}$ to $\chi(\mathbf{t}) = 2^{t_1\sqrt{w}+t_2}$. We render the surfaces in $\{\gamma_q^{\triangle} \mid q \in \mathbb{Q}^*_{\mathbf{i}}\}$ one by one in one pass. After rendering all of them, we compute $N(q)$ and $D(q)$ for all $q \in \mathbb{Q}^*_{\mathbf{i}}$ using Algorithm 1. Since all points in $\mathbb{Q}^*_{\mathbf{i}}$ with the same color are independent, the algorithm computes $h(q)$ for all $q \in \mathbb{Q}^*_{\mathbf{i}}$ correctly. Repeating this procedure for all $\mathbf{i} \in [0, \overline{B}/B)^2$, we compute the elevation of all points of $\mathbb{Q}$ in $(\overline{B}/B)^2$ passes.

Handling larger grids. The preceding algorithm assumed $\mathbb{Q}$ to be small enough (assumption (A1)) to be mapped to Π. This is not always realistic for the value of L is limited by the graphics hardware. For example, $L \leq 2^{14} = 16,384$ on modern graphics cards such as the NVIDIA GeForce GTX 660. With a scaling parameter $s = 5$, we have $M \leq \lfloor 2^{14}/5 \rfloor = 3,276$, implying that we can process $3,276^2 \approx 10^7$ grid query points in a single pass. Recall that each pass consists of two rendering phases and the subsequent buffer analysis. For $s = 2$ and a 2m grid spacing of $\mathbb{Q}$, this corresponds to computing a grid DEM for a region of roughly $70 \times 70\text{km}^2$ in area. However, we often want to generate grid DEMs that are considerably larger, in which case we proceed as follows.

Let $\mu = (L - 4r)/s$; the largest grid of query points that can be handled in one pass is $\mu \times \mu$. Thus, if $M > \mu$, we partition $\mathbb{Q}$ into $\mu \times \mu$ *parcels* and process these parcels individually. For $\mathbf{k} = (k_1, k_2) \in [0, \lceil M/\mu \rceil)^2$, we define the parcel $\mathbb{Q}^{\mathbf{k}} = \{\mathbb{Q}[k_1\mu + l_1, k_2\mu + l_2] \mid (l_1, l_2) \in [0, \mu)^2\}$.

We process each parcel $\mathbb{Q}^{\mathbf{k}}$ independently. For each $\mathbf{k}$, let $S^{\mathbf{k}} \subseteq S$ be the subset of points relevant for $\mathbb{Q}^{\mathbf{k}}$, that is, $S^{\mathbf{k}} = S \cap R(\mathbb{Q}^{\mathbf{k}})$. We compute the elevation of points in $\mathbb{Q}^{\mathbf{k}}$ with respect to $S^{\mathbf{k}}$ using the previous algorithm. For now assume that we have $S^{\mathbf{k}}$ at our disposal for all $\mathbf{k}$.

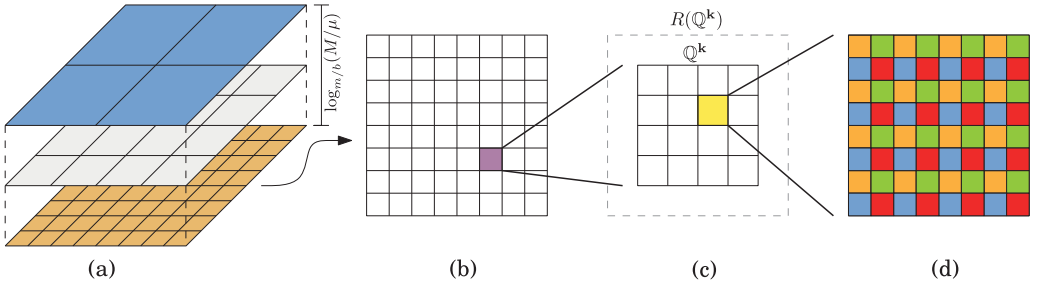


Fig. 11. (a) Recursive partition to compute $S^{\mathbf{k}}$. (b) Partition of $\mathbb{Q}$ into parcels. (c) Partition of a parcel $\mathbb{Q}^{\mathbf{k}}$ into blocks; the solid region corresponds to $R(\mathbb{Q}^{\mathbf{k}})$. (d) Partition of the blocks of $\mathbb{Q}^{\mathbf{k}}$ into tiles; all same-colored tiles are processed in one pass.

Putting everything together, there are three levels of partitioning of $\mathbb{Q}$ (cf. Figure 11):

$\mathbb{Q}^{\mathbf{k}}$: Partition $\mathbb{Q}$ into $\lceil M/\mu \rceil^2$ parcels, each of size at most $\mu \times \mu$, where $\mu = (L-4r)/s$.
$\mathbb{Q}^{\mathbf{k}}_{\mathbf{j}}$: Partition each parcel $\mathbb{Q}^{\mathbf{k}}$ into $\lceil \mu/\overline{B} \rceil^2$ blocks, each of size at most $\overline{B} \times \overline{B}$, where $\overline{B} = 2^{\lceil \log_2 r \rceil}$.
$\mathbb{Q}^{\mathbf{k}}_{\mathbf{j},\mathbf{i}}$: Partition each block $\mathbb{Q}^{\mathbf{k}}_{\mathbf{j}}$ into tiles each of size at most $B \times B$, where $B = \sqrt{w}$.

A point $q \in \mathbb{Q}$ is represented as a 4-tuple $(\mathbf{k}, \mathbf{j}, \mathbf{i}, \mathbf{t})$ for $\mathbf{k} \in [0, \lceil M/\mu \rceil)^2$, $\mathbf{j} \in [0, \lceil \mu/\overline{B} \rceil)^2$, $\mathbf{i} \in [0, \lceil \overline{B}/B \rceil)^2$, and $\mathbf{t} \in [0, B)^2$. Here $\mathbf{k}$ is the index of the parcel that contains q, $\mathbf{j}$ is the index of the block in the parcel $\mathbb{Q}^{\mathbf{k}}$ that contains q, and $\mathbf{t}$ is the offset of q within the tile $\mathbb{Q}^{\mathbf{k}}_{\mathbf{j},\mathbf{i}}$.

The height of points in $\mathbb{Q}$ is computed in $\lceil M/\mu \rceil^2 \times \lceil \overline{B}/B \rceil^2$ passes, as follows. Fix a pair $\mathbf{k} \in [0, \lceil M/\mu \rceil)^2$, $\mathbf{i} \in [0, \lceil \overline{B}/B \rceil)^2$. Let

$$\mathbb{Q}^{*}_{(\mathbf{k},\mathbf{i})} = \{(\mathbf{k}, \mathbf{j}, \mathbf{i}, \mathbf{t}) \mid \mathbf{j} \in [0, \lceil \mu/\overline{B} \rceil)^2, \mathbf{t} \in [0, B)^2\}.$$

All points of $\mathbb{Q}^{*}_{(\mathbf{k},\mathbf{i})}$ are processed in one pass using the points in $S^{\mathbf{k}}$. For all points $q \in \mathbb{Q}^{*}_{(\mathbf{k},\mathbf{i})}$ whose offset is $\mathbf{t} = (t_1, t_2)$, the color of all triangles in $\gamma_q^{\triangle}$ is set to $2^{t_1 B + t_2}$. Note that some of the points of S are sent to the GPU multiple times. In particular, if the points of S are uniformly distributed, then on average each point of S is processed at most $(1 + \frac{2r}{2r+\mu})$ times.

This completes the description of the algorithm except how to compute the set $S^{\mathbf{k}}$ for each parcel $\mathbb{Q}^{\mathbf{k}}$, which we describe next.

Computing $\mathbf{S}^{k}$. If we only care about the CPU and GPU efficiency of the algorithm and are not concerned about the time spent in accessing the data, which may reside on disk, computation of $S^{\mathbf{k}}$ is straightforward. However, if S is too large to fit in main memory and resides on disk, the time spent in transferring the data between disk and main memory is the bottleneck. In our application, S is indeed too large to fit in main memory, so we describe an algorithm that minimizes the data transfer between disk and main memory, that is, the number of *I/O-operations*. This algorithm will be referred to as the *binning* procedure.

Using the two-level I/O model introduced by Aggarwal and Vitter [1988], let m be the size of main memory and b be the disk block size (i.e., the number of words that are transferred between disk and main memory in each disk read/write operation). The disk is assumed to have infinite size. We describe a recursive algorithm in this model

for computing $S^{\mathbf{k}}$ for every parcel $\mathbb{Q}^{\mathbf{k}}$. We assume that S is too large to fit in main memory.

Set $u = \min\{\frac{M}{\mu}, \sqrt{m/b}\}$. We partition $\mathbb{Q}$ into u^2 subgrids, each of size $\lceil M/u \rceil \times \lceil M/u \rceil$. If $u = M/\mu$, then each subgrid is a parcel. We create a bin for each subgrid. We scan S once. For each point, we find in $O(1)$ time the (at most four) subgrids $\mathbb{Q}^{(i)}$ such that $p \in R(\mathbb{Q}^{(i)})$. If $p \in R(\mathbb{Q}^{(i)})$, we add p to the bin of $\mathbb{Q}^{(i)}$. Once a bin gets full, we write it to the disk and create a new bin for that subgrid. The total number of I/Os performed is $O(|S|/b)$. Let $S^{(i)} = S \cup R(\mathbb{Q}^{(i)})$ be the set of points assigned to $\mathbb{Q}^{(i)}$. If $u < M/\mu$, then each subgrid is a parcel, so we have the relevant points for each parcel and we are done. Otherwise, if $u > M/\mu$, we recursively solve the subproblem for each subgrid $\mathbb{Q}^{(i)}$ using the points of $S^{(i)}$. The algorithm stops after $O(\log_{m/b} \frac{M}{\mu})$ recursive steps. Hence, the algorithm performs a total of $O(\frac{N}{b} \log_{m/b} \frac{M}{\mu})$ I/Os.

5. NATURAL NEIGHBOR INTERPOLATION ON 3D GRIDS

Let S be a set of n points in $\mathbb{R}^3$, each point $p \in S$ endowed with an elevation value $h(p)$, and let $\mathbb{Q}$ be an $M \times M \times T$ grid of points in $\mathbb{R}^3$. We describe a GPU-based algorithm for constructing the elevation at all points of $\mathbb{Q}$ using NNI as defined in Equation (3). As in Section 3, we assume that the influence region of a point $p \in \mathbb{R}^3$ is the cylinder of radius r and height $2r$, centered at p, and its base parallel to the xy-plane.

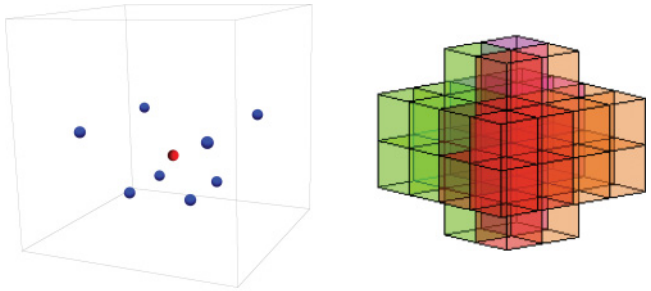
Analogous to the computation of the Voronoi diagram in $\mathbb{R}^3$, described in Section 3, we compute the elevation of points in $\mathbb{Q}$ slices by slice—at each stage we process a 2D slice of $\mathbb{Q}$ with a fixed value of t. Each stage involves computing a portion of $\mathrm{Vor}^{\square}(S)$—roughly speaking, $O(r)$ 2D slices of $\mathrm{Vor}^{\square}(S)$. Hence, each 2D slice of $\mathrm{Vor}^{\square}(S)$ is used $O(r)$ times. The question is whether we recompute this slice every time or compute it once and store it. We present three algorithms, which provide a space-time tradeoff. The first algorithm recomputes a 2D slice of $\mathrm{Vor}^{\square}(S)$ every time it is needed, resulting in an algorithm that needs only one copy of the frame buffer $\mathbb{F}$ of the GPU, but renders each point r^2 times. The second algorithm computes a 2D slice of $\mathrm{Vor}^{\square}(S)$ only once and then stores it on the GPU for future use. It needs $2r + 1$ copies of $\mathbb{F}$ but renders each point of S only r times. Finally, we describe a third algorithm that works for time-series data; that is, input is collected at fixed time intervals, which is often the case for GIS applications. It also assumes that the GPU has certain capabilities (see later for details). Under these assumptions, it renders each point only once and needs $2(2r + 1)$ copies of $\mathbb{F}$.

Before describing these algorithms, we fix a few parameters and notation. We assume that $\mathbb{Q}$ has origin $(0, 0, 0)$ and resolution 1 in each dimension, that is, $\mathbb{Q} = [0, M)^2 \times [0, T)$. Any 3D grid can be mapped to $\mathbb{Q}$ using an affine transformation and using an adjusted distance function to preserve the structure of the Voronoi diagram. We define $\mathbb{B} = \mathbb{Q} + [-2r - 1/2, 2r + 1/2]^3$—large enough to contain all the points of S that are of interest, and we can assume that $S \subset \mathbb{B}$. Similarly to Section 4, we let $s \in \mathbb{N}$ be a scaling parameter. We discretize $\mathbb{B}$ into an $sM \times sM \times T$ 3D grid of voxels; each voxel is a box of size $\frac{1}{s} \times \frac{1}{s}$ and height 1, and its volume is $\frac{1}{s^2}$ (cf. Figure 12). By our definition of $\mathbb{B}$ and the voxels, each grid point of $\mathbb{Q}$ lies at the center of an $s \times s \times 1$ array of voxels of $\mathbb{B}$; that is, the temporal resolution of $\mathbb{B}$ and $\mathbb{Q}$ is the same, but the spatial resolution of $\mathbb{B}$ is s times that of $\mathbb{Q}$. One can choose the temporal resolution of $\mathbb{B}$ higher than that of $\mathbb{Q}$, but choosing the two to be the same simplifies the description of the algorithm.

We observe that the region of influence, being a cylinder with its axis parallel to the time axis, implies that only voxels υ of $\mathrm{Vor}^{\square}(S)$ for which $|t_\upsilon - t_q| \leq r$ are relevant for computing $N(q)$ and $D(q)$ for any point $q \in \mathbb{Q}$. Thus, we only need to consider $2r + 1$

Table I. Performance of the Three Algorithms: $|S| = n$, and Each Copy of $\mathbb{F}$ Requires Storing a Color Buffer and Associated Depth Buffer on the GPU

Algorithm # of Passes	# Copies of $\mathbb{F}$	Triangles Rendered per $p \in S$	
		General	Weighted Euclidean
r^2	1	$O(r^2kmn)$	$O(r^2n)$
r	$2r+1$	$O(rkmn)$	$O(rn)$
1	$2(2r+1)$	$O(kmn)$	$O(n)$



(a) (b)

Fig. 12. (a) A query point in the middle of the point cloud of eight points in $\mathbb{R}^3$, same dataset as in Figure 5. (b) The voxels of the query's Voronoi cell colored based on the cells they stole volume from.

slices of $\mathbb{B}$ corresponding to $t \in [t_q - r, t_q + r]$. Applying this to Equation (3), we obtain

$$D(q) = \sum_{\tau=t_q-r}^{t_q+r} \left|\mathrm{Vor}^{\square}_{S\cup\{q\}}(q,\tau)\right| = \sum_{\tau=t_q-r}^{t_q+r} D(q,\tau), \tag{4}$$

$$N(q) = \sum_{\tau=t_q-r}^{t_q+r} \sum_{\pi\in\mathrm{Vor}^{\square}_{S\cup\{q\}}(q,\tau)} h(\varphi(\pi,S)) = \sum_{\tau=t_q-r}^{t_q+r} N(q,\tau). \tag{5}$$

Recall that for any fixed τ, let $S^\tau = \{p \in S \mid |t_p - \tau| \le r\}$ and that $\mathrm{Vor}^{\square}(S,\tau) = \mathrm{Vor}^{\square}(S^\tau,\tau)$. We are now ready to describe the three algorithms. For each algorithm, we describe how we compute the elevation of points in a fixed 2D slice $\mathbb{Q}_i$ of $\mathbb{Q}$.

The r^2-pass algorithm. Equations (4) and (5) suggest that we compute $N(q,\tau)$ and $D(q,\tau)$ for $\tau = i-r,\ldots,i+r$ and for all $q \in \mathbb{Q}_i$. For each τ, we first compute $\mathrm{Vor}^{\square}(S,\tau) = \mathrm{Vor}^{\square}(S^\tau,\tau)$ using the algorithm GVor(S^τ,τ), described in Section 3. It returns a copy of the frame buffer, which we denote by $\mathbb{F}_\tau = (\mathbb{C}_\tau, \mathbb{D}_\tau)$. Next we adapt the second phase of the 2D NNI algorithm of Section 4. It uses $\mathbb{F}_\tau$ and renders the surfaces $T_{\tau,i}(\Gamma) = \{\gamma_q^{\triangle} \mid q \in \mathbb{Q}_i\}$, where $\gamma_q^{\triangle}$ approximates the graph of f_q^τ—the distance function from q in the plane $t = \tau$—so the color buffer $\mathbb{C}_{i,\tau}$ encodes $\mathrm{Vor}^{\square}{}_{S\cup\{q\}}(q,\tau)$ for all $q \in \mathbb{Q}_i$. Then, using $\mathbb{F}_\tau$ and $\mathbb{C}_{i,\tau}$, it computes $N(q,\tau)$ and $D(q,\tau)$. Depending on the size of M, r, and s and GPU constraints, this may require several rendering passes and the I/O-efficient algorithm for distributing the input points described in Section 4. We call this modified 2D NNI algorithm Interpolate$(\mathbb{F}, i, \tau)$. It takes the frame buffer computed after phase I of the algorithm and the values of i and τ as input and returns two $M \times M$ arrays N^∇ and D^∇ such that $N^\nabla[q] = N(q,\tau)$ and $D^\nabla[q] = D(q,\tau)$. The pseudo-code of the overall r^2-pass algorithm is described in Algorithm 2.

ALGORITHM 2: r^2-Pass

```
for i ← 0 to T − 1 do
    H_i ← N_i ← D_i ← 0
    for τ ∈ {i − r, ... i + r} do
        𝔽 ← GVor(S^τ, τ)
        (N^▽, D^▽) = Interpolate(𝔽, i, τ)
        N_i ← N_i + N^▽, D_i ← D_i + D^▽
    H_i ← N_i / D_i
return H
```

ALGORITHM 3: r-Pass

```
Create 𝔽_{−r−1} ... 𝔽_{r−1}
for i ← 0 to T − 1 do
    H_i ← N_i ← D_i ← 0
    Delete 𝔽_{i−r−1} from GPU
    𝔽_{i+r} ← GVor(S^{i+r}, i + r)
    for τ ∈ {i − r, ... i + r} do
        (N^▽, D^▽) = Interpolate(𝔽_τ, i, τ)
        N_i ← N_i + N^▽, D_i ← D_i + D^▽
    H_i ← N_i / D_i
return H
```

Each point $p \in S$ is passed to the computation of $O(r)$ slices of $\text{Vor}^{\square}(S^{\tau}, \tau)$ for $\tau \in \{\lfloor t_p \rfloor - r, \ldots, \lceil t_p \rceil + r\}$. Furthermore, each Voronoi diagram slice $\text{Vor}^{\square}(S, \tau)$ is computed $2r + 1$ times for $\mathbb{Q}_{\tau - r}, \ldots, \mathbb{Q}_{\tau + r}$. Hence, each point of S is passed $O(r^2)$ times.

The r-pass algorithm. Some computations in the previous algorithm can be saved by storing what has already been computed and reusing it: as before, let frame buffer $\mathbb{F}_{\tau} = \text{GVor}(S, \tau)$ refer to the combination of $\mathbb{C}_{\tau}$ and $\mathbb{D}_{\tau}$ describing $\text{Vor}^{\square}(S, \tau)$. The previous algorithm generates $\mathbb{F}_{i-r}, \ldots \mathbb{F}_{i+r}$ and performs the interpolation on each time slice. It must generate $\mathbb{F}_{i+1-r}, \ldots \mathbb{F}_{i+1+r}$ to process $\mathbb{Q}_{i+1}$. All these frame buffers, except $\mathbb{F}_{i+1+r}$, are already available from the processing of $\mathbb{Q}_i$, and thus they can be reused. Thus, we only need to compute $\text{Vor}^{\square}(S, i + r + 1) = \text{Vor}^{\square}(S^{i+r+1}, i + r + 1)$ by calling $\text{GVor}(S^{i+1+r}, i + 1 + r)$. See Algorithm 3 for the pseudo-code. This algorithm requires storing $2r + 1$ frame buffers: $2r$ to save values from the previous stage, and one to generate the new Voronoi diagram slice. Doing this for all $0 \leq i < M$ *in order* lets us generate each $\mathbb{F}_i$ only once. This method renders each point $p \in S$ $O(r)$ times: once for each $\text{Vor}^{\square}(S^{\tau}, \tau)$ for $\tau \in \{\lfloor t_p \rfloor - r, \ldots, \lceil t_p \rceil + r\}$.

The 1*-pass algorithm*. We now show that the number of passes can be reduced to 1 if S is time-series data, $d(\cdot, \cdot)$ satisfies certain properties, and the GPU supports certain primitives. For simplicity, we assume that $S = \Sigma^1 \cup \Sigma^2 \cup \cdots$, where the t-coordinate of all points in Σ^i is i, the time coordinate of the ith slice of $\mathbb{Q}$, and $d(\cdot, \cdot)$ is the weighted Euclidean distance. Recall that the r-pass algorithm computes $\text{Vor}^{\square}(S^{i+r}, i + r)$ in the ith stage, which computes the elevation of points in $\mathbb{Q}_i$. Note that $S^{i+r} = \bigcup_{j=0}^{2r} \Sigma^{i+j}$, so each point belongs to $2r + 1$ different S^is, and $S^{i+r+1} = (S^{i+r} \setminus \Sigma^i) \cup \Sigma^{i+2r+1}$. In order to ensure that each point is rendered only once, we rely on two ideas. First, in principle, if we have $\text{Vor}^{\square}(\Sigma^i, i)$, stored in a frame buffer, $\text{Vor}^{\square}(\Sigma^i, j)$ can be computed without rendering any points. Second, if we have $\text{Vor}^{\square}(\Sigma^i, j)$, for $j - r \leq i \leq j + r$, $\text{Vor}^{\square}(S^i, j)$ can be computed without rendering any points.

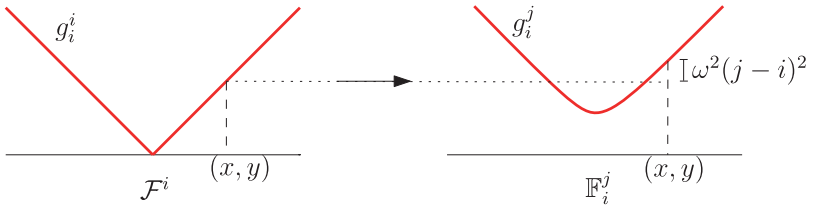


Fig. 13. Computing $\mathbb{F}_i^j$ from $\mathcal{F}^i$ using $\textsc{Shift}(\mathcal{F}^i, (j-i)^2)$.

We first show how each of these two operations are performed and then describe how to process each slice $\mathbb{Q}_i$ so that any point of S is rendered only once.

First, we describe how to compute $\text{Vor}^{\square}(\Sigma^i, j)$ from $\text{Vor}^{\square}(\Sigma^i, i)$. For a point $p \in \Sigma^i$, that is, $t_p = i$, we define the function $g_{j,p} : \mathbb{R}^2 \to \mathbb{R}$ as

$$g_{j,p}(x, y) = -2xx_p - 2yy_p + x_p^2 + y_p^2 + (j-i)^2 \,. \tag{6}$$

Let $g_j^i(x, y) = \min_{p\in\Sigma^i} g_{j,p}(x, y)$ be the lower envelope of $\{g_{j,p} \mid p \in \Sigma^i\}$. Then $\text{Vor}^{\square}(\Sigma^i, j)$ is the projection of g_j^i. Since the last term in Equation (6) does not depend on p, it immediately implies that $\text{Vor}^{\square}(\Sigma^i, j)$ is the same for all j; that is, if a pixel $\pi \in \text{Vor}^{\square}{}_{\Sigma^i}(p, i)$, then $\pi \in \text{Vor}^{\square}{}_{\Sigma^i}(p, j)$ for all $j \geq i$. Furthermore,

$$g_j^i(x, y) = g_i^i(x, y) + (j-i)^2.$$

Let $\mathbb{F}_j^i = (\mathbb{C}_j^i, \mathbb{D}_j^i) = \textsc{GVor}(\Sigma^i, j)$. $\mathbb{C}_j^i[\pi]$ stores the elevation of the point nearest to π and $\mathbb{D}_j^i$ stores the distance from π to its nearest neighbor. Set $\mathcal{F}^i = \mathbb{F}_i^i$. By the previous observation, $\mathbb{C}_j^i[\pi] = \mathbb{C}_i^i[\pi]$ and $\mathbb{D}_j^i[\pi] = \mathbb{D}_i^i[\pi] + (j-i)^2$. Then $\mathbb{F}_j^i$ can be computed from $\mathcal{F}^i$ by adding the value $(j-i)^2$ to every pixel in $\mathbb{D}_i^i$. Let $\textsc{Shift}(\mathbb{F}, \delta)$ be the procedure that adds the value δ to all pixels in $\mathbb{D}$; see Figure 13. Then

$$\mathbb{F}_j^i = \textsc{Shift}(\mathcal{F}^i, (j-i)^2).$$

Hence, $\mathbb{F}_j^i$ can be computed from $\mathbb{F}_i^i$ without rendering any points provided the $\textsc{Shift}$ procedure is available. Next, assuming we have $\mathcal{F}^\ell$ for $j-r \leq \ell \leq j+r$, we can compute $\text{Vor}^{\square}(S^i, j)$, which is stored in a copy of the frame buffer referred to as $\mathbb{F}_j = (\mathbb{C}_j, \mathbb{D}_j)$, as follows. We first compute $\mathbb{F}_j^\ell$, for $j-r \leq \ell \leq j+r$, as earlier. Then set

$$\mathbb{D}_j[\pi] = \min_{j-r\leq\ell\leq j+r} \mathbb{D}_j^\ell[\pi]$$

and set $\mathbb{C}_j[\pi]$ to the contents of $\mathbb{C}_j^\ell[\pi]$ if $\mathbb{D}_j[\pi] = \mathbb{D}_j^\ell[\pi]$. We call this procedure $\textsc{Combine}(\mathcal{F}^{j-r}, \ldots, \mathcal{F}^{j+r})$. The pseudo-code can be found in Algorithm 4.

With these two procedures at hand, we modify the r-pass algorithm as follows. The algorithm maintains the invariant that while processing $\mathbb{Q}_i$, it maintains $\text{Vor}^{\square}(S^{i-r}, i-r), \ldots, \text{Vor}^{\square}(S^{i+r}, i+r)$ in the copies of the frame buffer $\mathbb{F}_{i-r}, \ldots, \mathbb{F}_{i+r}$ and also $\text{Vor}^{\square}(\Sigma^i, i), \text{Vor}^{\square}(\Sigma^{i+1}, i+1), \ldots, \text{Vor}^{\square}(\Sigma^{i+2r}, i+2r)$ in $\mathcal{F}^i, \ldots, \mathcal{F}^{i+2r}$. While processing $\mathbb{Q}_i$, the algorithm first computes $\text{Vor}^{\square}(\Sigma^{i+2r}, i+2r)$ using $\textsc{GVor}(\Sigma^{i+2r}, i+2r)$, storing the result in $\mathcal{F}^{i+2r}$. Next, instead of computing $\textsc{GVor}(S^{i+r}, i+r)$ by invoking $\textsc{GVor}(S^{i+r}, i+r)$, it uses the procedure $\textsc{Combine}(\mathcal{F}^i, \ldots, \mathcal{F}^{i+2r})$. The rest of the procedure is the same. See Algorithm 5 for the pseudo-code and Figure 14 for an overview of the procedure. Note that each point is processed only once, and that it stores $2(2r+1)$ copies of the frame buffer.

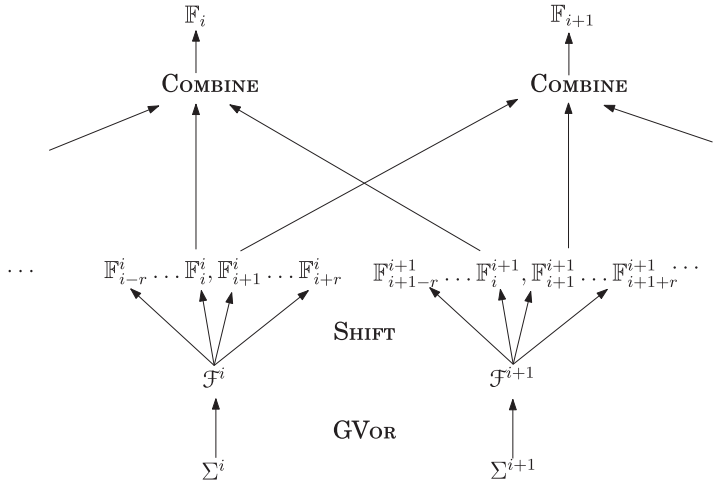


Fig. 14. The process of creating $\mathbb{F}_i$ from sets of data Σ^i rendering each point only once.

ALGORITHM 4: COMBINE($\mathcal{F}^{j-r}, \ldots, \mathcal{F}^{j+r}$)

for all $\pi \in \mathbb{F}_j$ **do**
  $\mathbb{C}_j[\pi] \leftarrow 0, \mathbb{D}_j[\pi] \leftarrow \infty$
for $i \in j-r, \ldots, j+r$ **do**
  $\mathbb{F}^i_j \leftarrow$ SHIFT($\mathcal{F}^i, (j-i)^2$)
  for all $\pi \in \mathbb{F}_j$ **do**
    if $\mathbb{D}^i_j[\pi] < \mathbb{D}_i[\pi]$ **then**
      $\mathbb{D}_j[\pi] \leftarrow \mathbb{D}^i_j[\pi]$
      $\mathbb{C}_j[\pi] \leftarrow \mathbb{C}^i_j[\pi]$
return $\mathbb{F}_j$

ALGORITHM 5: 1-Pass

Create $\mathcal{F}^{-1} \ldots \mathcal{F}^{2r-1}$ and $\mathbb{F}_{-r-1} \ldots \mathbb{F}_{r-1}$
for $i \leftarrow 0$ to $T-1$ **do**
  $H(\mathbb{Q}_i) \leftarrow N(\mathbb{Q}_i) \leftarrow D(\mathbb{Q}_i) \leftarrow 0$
  Delete $\mathcal{F}^{i-1}$ and $\mathbb{F}_{i-r-1}$ from GPU
  $\mathcal{F}^{i+2r} \leftarrow$ GVOR($\Sigma^{i+2r}, i+2r$)
  $\mathbb{F}_{i+r} \leftarrow$ COMBINE($\mathcal{F}_i \ldots \mathcal{F}_{i+2r}$)
  for $\tau \in \{i-r, \ldots i+r\}$ **do**
    $(N^\nabla, D^\nabla) =$ INTERPOLATE($\mathbb{F}_\tau, i, \tau$)
    $N_i \leftarrow N_i + N^\nabla$, $D_i \leftarrow D_i + D^\nabla$
  $H_i \leftarrow N_i / D_i$
return H

6. EXPERIMENTS

We describe here our implementation of the algorithm given earlier and offer empirical results to demonstrate the quality and efficiency of the algorithm on real-world as well as synthetic terrains.

6.1. Implementation

TerraNNI was implemented in C++ using OpenGL to interact with the graphics card. The source code is available at https://github.com/thomasmoelhave/TerraNNI. The color and depth buffers were implemented using OpenGL's frame-buffer and render-buffer objects; the display list feature of OpenGL was used to render the surfaces. We optimized some steps of the algorithm that greatly improve the efficiency of our implementation in practice.

GPU-to-CPU communication. Recall that besides drawing the surfaces, the INTERPOLATE procedure of the algorithm performs certain computations on the data stored in color and depth buffers. In particular, it aggregates the data to compute the elevation at query points. Since the bus connecting the GPU to the CPU is relatively slow, it is important to perform certain operations on the GPU itself and minimize the data transfer, ideally only transferring the final interpolated values for each query point of $\mathbb{Q}$ back to main memory. To do this, we use CUDA [NVIDIA 2010], first to implement INTERPOLATE and then to calculate the interpolated values directly on the graphics card. Because performing INTERPOLATE in CUDA parallelizes the computation of N and D as GPU buffers, we use CUDA's atomic addition operation to prevent the same location of N and D from being incremented by multiple threads simultaneously. With N and D stored in GPU memory, we also calculate the interpolated values H on the graphics card. As a result, only one read operation is performed to transfer data from the GPU to the CPU, with only one 32-bit word for each query point of $\mathbb{Q}$.

Minimizing disk transfers. Recall that we have described an algorithm to compute $S^k \subseteq S$, the subset of relevant points for each layer $\mathbb{Q}^k$. This algorithm is implemented using the Templated Portable I/O Environment (TPIE) library [TPIE Development Team 2014]. TPIE is also used to store the output, the elevation of the query grid points, in a row-major 2D grid, which involved sorting the output.

6.2. Experimental Setup

We ran our experiments on an Intel Core i7-3770 CPU at 3.40GHz with 24GB of internal memory and two 2TB 7200RPM hard drives in a RAID 0 setup. The machine has Ubuntu 13.10. Additionally, the machine contained an NVIDIA GeForce GTX 660 graphics card running CUDA 5.0. This card has 2 gigabytes of memory and 960 CUDA cores.

Datasets. We tested our algorithm on the following 2D and 3D datasets:

Lake Tahoe: The first 2D dataset we used was a large public LiDAR dataset that covers Lake Tahoe in the United States.[3] The dataset is 56GB, covering $34 \times 69\text{km}^2$ with over 2.1×10^9 points. The region includes the 490km^2 Lake Tahoe in the center, which has no LiDAR points because it is a body of water. Outside the lake, the ground point density was about 2.2 points/m^2.

Afghanistan: The second dataset is a point cloud of a mountainous region in the Paktika province of Afghanistan.[4] This dataset is 3.5GB, covering an area of $4,000\ \text{m}^2$, with over 1.86×10^8 points. This is approximately 6.5 points/m^2 on average. Because the Afghanistan dataset comes from a mountainous region, the data is useful for comparing how different algorithms handle steep slopes and ridges.

This dataset contains many nonground points. For part of the quality tests we used a subset of the points with most of the nonground points removed.[5] The resulting subset,

[3]Available at http://www.opentopography.org/index.php/news/detail/lake_tahoe_basin_lidar_data_released.
[4]Afghanistan data courtesy of ARO.
[5]This was done by only using the "last return" points for each pulse.

which we refer to as Afghanistan-1, has a point density of 0.26 points/m^2. We also removed a portion of the points, keeping only one out of 16 data points at random. This produces a point density of 0.016 points/m^2. We refer to this dataset as Afghanistan-2.

Nags Head: To test our 3D algorithm, we use LiDAR data collected from the Nags Head, NC, region for the years 1997–1999, 2001, 2004, 2005, 2007, and 2008. The data ranges from 10^5 points/year to nearly 3×10^6 points/year, totaling just under 2×10^7 million points and 0.38GB. The data is freely available at NOAA [2014] and was provided to us by Dr. Helena Mitasova.

Synthetic: We also tested our 3D algorithm on synthetic data generated by Qhull's *rbox* tool [Barber et al. 1996]. We used this to create numerous datasets, each a random point cloud in a 3D cube of side length 10^4 along with an elevation value for each point. They have 10^3 to 10^{10} points, and their sizes vary from 20KB to 186GB.

Parameter choices. There are a few parameters that can be adjusted to tune TerraNNI's speed and quality tradeoffs. We list later the default parameters used in the following experiments, which offer a sufficiently high quality of output without significantly slowing the running time. We set the scaling parameter $s = 5$, the side length of a pixel $\rho = 0.4$ meters, the radius of the influence region of input points $r_s = 5$ meters, and the radius of the query influence region $r_q = s\rho B/2$. The word size w of the color buffers is 32 bits (though this can easily be increased to 128 bits on modern graphics cards), so $B = 5$ and then $r_q = 5$ meters.

We use the Euclidean metric as the distance function for most of our tests. We tested our algorithm by treating the distance function as a general convex distance function as well as using the lifting transform. In rendering general convex distance functions, we generate the surfaces $\gamma_i^{\triangle}$ by setting $m = 5$ and $k = 50$. As a result, each $\gamma_i^{\triangle}$ is composed of 500 triangles in total. When using the lifting transform, we clip the plane γ_i to an ellipse $\gamma_i^{\diamond}$, approximate it by a rectangle $\gamma_i^{\triangle}$ circumscribing $\gamma_i^{\diamond}$, and partition it into two triangles.

We tested the algorithm with varying values for especially s and k and m, but values larger than the ones chosen did not have much impact on the output, implying that increasing the values further is not worth the tradeoff in computation time.

6.3. Output Quality

To test the quality of the output, we constructed DEMs of the full Afghanistan dataset with a resolution of 1 meter using three interpolation schemes: the spline-based RST scheme, linear interpolation, and NNI. We choose the Afghanistan dataset because it has the most variation in elevation. Figure 15 shows the resulting DEMs. We used the full Afghanistan point cloud and thus, trees are visible in the model. The trees represent high-frequency components in the data, and they exhibit the largest difference between the three interpolation methods. The three DEMs are indistinguishable at a high level, and the differences are minor in the detailed view.

We then used the Afghanistan-1 dataset, as well as the sparser Afghanistan-2 dataset, to focus on elevation at the ground level and see how the density of data affects the quality. Using the GRASS GIS system [GRASS Development Team 2014], we compute the contour lines from the interpolated data with a 1-meter increment between contour levels.

Figure 16 shows sample images from the contour maps of Afghanistan-1 and Afghanistan-2. The linear interpolation produces jagged contour results, while the NNI generates smooth contours, especially around portions of large curvature. As the input data becomes sparser (see Afghanistan-2), the linear interpolation becomes increasingly jagged, while the NNI output remains smooth. The contours generated using RST interpolation were indistinguishable from the contours generated using NNI.

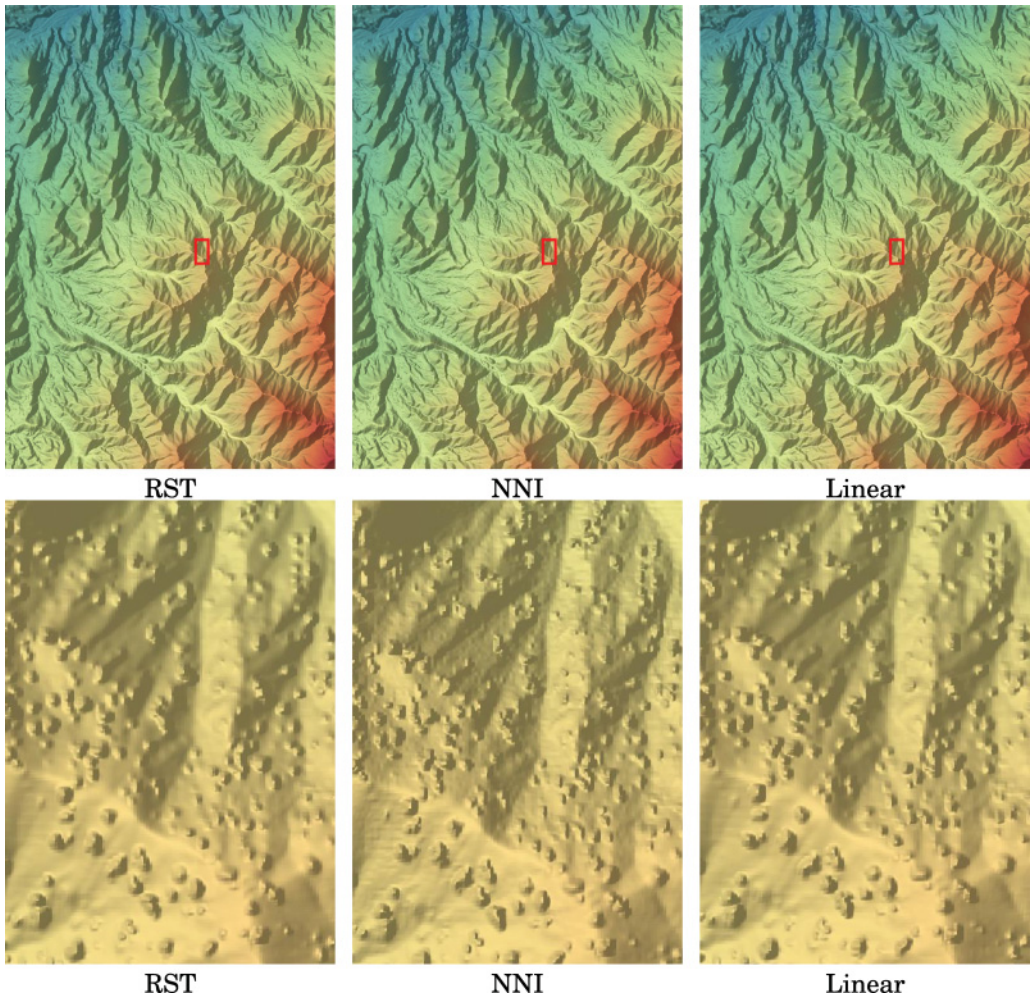


Fig. 15. Afghanistan 1m resolution DEMs, as constructed using different interpolation schemes. The top row shows a high-level overview; the area marked by the red box is displayed in detail in the bottom row.

6.4. Efficiency

We now discuss the impact of parameter settings on the efficiency of the 2D and the 3D algorithm.

2D performance. We tested the effect of varying the main parameters—grid size, scaling, and radius of the query influence region—on each phase of the algorithm: binning, GVor, drawing the query surfaces, Interpolate, and writing the output to disk, and saving the final results. By default, we use a 2m grid of size 2,027 by 4,700 with $q_r = 5$ and $s = 5$. Table II summarizes the results on the Afghanistan dataset. We note that the binning procedure is the same for all runs since we use the same dataset in each test. This step, consisting almost entirely of disk I/O, takes a majority of the running time. Since this step is not the primary focus of our algorithm, we didn't try to optimize it.

As we decrease the size of the query grid cells from 2 meters down to 1 meter to 0.5 meter, the number of grid points increases to 4 and 16 times, respectively.

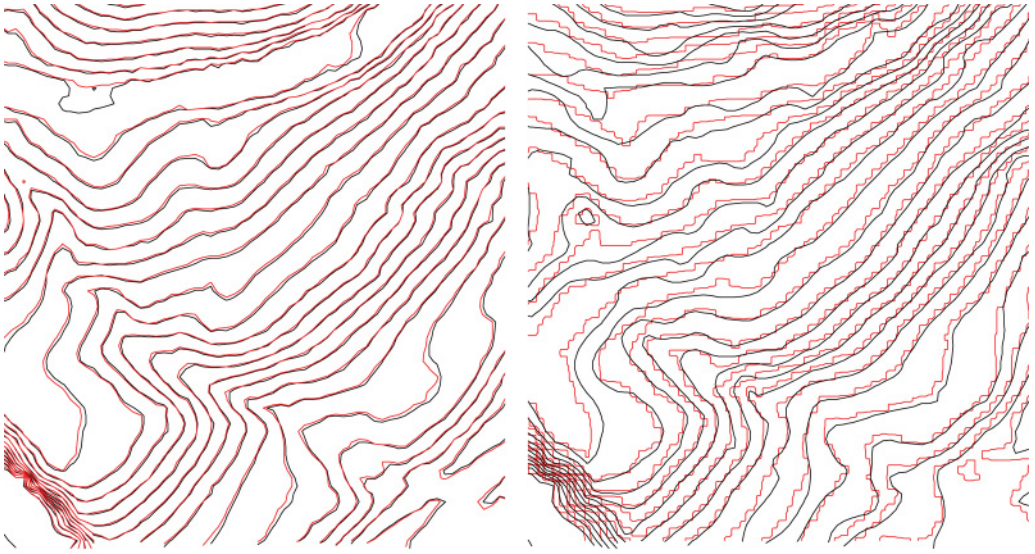


Fig. 16. Contour map comparison. In both figures, the grid used for the red (black, respectively) contours was generated from the Afghanistan data using linear (natural neighbor, respectively) interpolation.

Table II. Comparison of Running Times of Different Variants of TerraNNI on the Afghanistan Data Set (by Default We Have a 2m Grid, $q_r = 5$ and $s = 5$)

	0.5m Grid	1m Grid	2m Grid	$q_r = 50$	$s = 15$
Binning time	3m 20s	3m 25s	3m 37s	3m 36s	3m 26s
GVor(S)	1m	1m 3s	1m 2s	1m 14s	1m 6s
Draw query surface	47s	14s	4.07s	34s	3.46s
Interpolate	20s	5.0s	1.28s	15s	3.08s
Write points	12s	2.39s	0.82s	0.69s	0.74s
Save results	3m	43s	10s	10s	10s
Total interpolation time	**2m 19s**	**1m 24s**	**1m 8s**	**2m 5s**	**1m 14s**
Total running time	8m 40s	5m 31s	4m 56s	5m 51s	4m 50s

Unsurprisingly, the time to draw the Voronoi diagrams is largely unaffected since we are drawing approximately the same number of triangles for this step. The slight increase in GVor timing is because the number of grid points increases, the number of parcels increases, and therefore more input points lie in multiple parcels.

Drawing query surfaces, Interpolate, and writing the output to disk all increase in time approximately linearly with the number of grid points because each additional grid point requires drawing one more query surface, interpolating one more point, and saving one more grid point of output.

Next, increasing the radius of the query influence region 10-fold, from 5 meters to 50 meters, causes the time for both drawing the query surfaces and Interpolate to increase significantly, 8.4 times and 11.4 times, respectively, because we now must take 121 passes of drawing query surfaces and Interpolate, thus reducing parallelism. This demonstrates the tradeoff of covering regions with no data as well as the importance of our choice to limit the query radius.

Table III. Comparison of Our 3D Algorithm on the NC Coastal Dataset Under Different Settings (for Clarity, We Include Only One Parameter Setting for the Lifting Transform)

	Lifting Transform		General Convex Metric			
	r-algorithm	r^2-algorithm	r-algorithm		r^2-algorithm	
	$r=1$	$r=1$	$r=1$	$r=2$	$r=1$	$r=2$
Binning time	27s	21s	29s	21s	21s	22s
GVOR(S)	22s	1m 24s	29s	47s	1m 44s	4m 58s
Draw query surfaces	2m 32s	2m 38s	3m 31s	5m 41s	3m 36s	5m 43s
INTERPOLATE	34s	25s	26s	48s	21s	38s
Write points	7.07s	7.02s	7.07s	7.01s	6.96s	6.97s
Save results	1m 48s	1m 49s	1m 49s	1m 48s	1m 48s	1m 48s
Total interpolation time	**3m 36s**	**4m 35s**	**4m 34s**	**7m 24s**	**5m 49s**	**11m 27s**
Total running time	5m 52s	7m 59s	6m 51s	9m 34s	7m 58s	13m 36s

Finally, as described, the scaling parameter changes the number of pixels between grid points; thus, a larger s means higher precision. We see that the time to draw the Voronoi diagram and the query surfaces is largely unaffected by the increase in s from 5 to 15, despite the fact that more pixels are rasterized by triangles; this indicates that rasterization is not a bottleneck in the rendering of the query surfaces. However, we do see a nearly 2.4 times increase in the INTERPOLATE time. This is because with more pixels in the same area, there are more atomic adds happening concurrently and thus more contention, reducing the parallelism in INTERPOLATE. However, given that this step is already fast, the increase in time is relatively insignificant.

3D performance. We tested the r-pass and r^2-pass algorithms with $r=1$ and $r=2$ on the Nags Head dataset. Each test produced a 3D grid with a spatial 1-meter resolution over a $2.3\times1.8\text{km}^2$ region. The 3D grid has 12 slices in time, corresponding to producing an output every half year for 6 years in total, and this 3D grid has a total of about 4.8×10^7 voxels. For each of these parameter settings, we used the Euclidean metric but tested both for treating it as a convex distance function as well as using the lifting transform. See Table III.

First, we observe a significant improvement from the r^2-pass algorithm to the r-pass algorithm, especially in the GVOR step, taking 16% of the time when $r=2$ and 28% of the time when $r=1$. Overall, the r-pass algorithm takes approximately 66% of the interpolation time of the r^2-pass algorithm when $r=2$ and 79% of the interpolation time when $r=1$.

For both the r and r^2 algorithms, increasing the time radius r results in more time needed for GVOR, drawing query surfaces, and INTERPOLATE. By increasing r, each input point is drawn approximately 5 rather than 3 times during the r-algorithm, resulting in GVOR taking about 1.4 times as much time with the larger radius. When running the r^2-algorithm, this change in r is exacerbated with points being drawn 25 times rather than 9 times, and GVOR takes approximately 2.3 times as long for the larger radius. By increasing r, we also now render each query point and run the INTERPOLATE algorithm more times, resulting in similar increases in time for both of these steps.

Finally, we note that the lifting transform significantly reduces the time spent in drawing input and query surfaces because each surface consists of only four triangles. But INTERPOLATE takes longer because we are approximating an ellipse with its circumscribing rectangle and thus a larger influence region, which causes INTERPOLATE to aggregate the values over more pixels. Overall, the lifting transform saves 30% to 40% the time. Overall, we find that using the Euclidean metric results in the

Table IV. Running Time of TerraNNI, RST, and Linear Interpolation on the Lake Tahoe and Afghanistan Datasets

Dataset	TerraNNI	RST	Linear
Afghanistan	5m 16s	36m 10s	7m 0.8s
Lake Tahoe	85m 51s	867m 32s	160m 45s

interpolation time being between 67% and 79% of the interpolation time with a general convex metric.

6.5. Scalability Comparison

To test the scalability of our algorithm, we run experiments on both the 2D and 3D versions of the algorithm.

Real-world datasets. For the 2D algorithm, we compared against results from applying a RST and linear interpolation based on the global Delaunay triangulation of large real-world datasets. The RST implementation is from Danner et al.'s previous work on TerraSTREAM [Danner et al. 2007]. The linear interpolation computes the Delaunay triangulation of the entire dataset, as presented in Agarwal et al. [2005], and then interpolates the value of each query point. Both of these algorithms are sequential. We use two large real-world datasets with varied terrain for our experiments: the mountainous Afghanistan dataset and the Lake Tahoe dataset.

The results are summarized in Table IV. For the Afghanistan dataset, TerraNNI takes 75% of the time of linear interpolation and 14.6% of the time of RST. However, for the Lake Tahoe dataset, we see that the large lake in the middle of the region is time consuming for RST and linear interpolation but not as much so for TerraNNI; here our algorithm benefits greatly from the parallel processing of query points. In particular, TerraNNI takes only 53.4% of the time of the linear interpolation and 9.9% of the time of RST for the Lake Tahoe dataset.

Synthetic datasets. We compared our 3D implementation against traditional (exact) CPU algorithms. Note that, in addition to being confined to the CPU, these algorithms robustly and exactly compute the Voronoi diagrams $\mathrm{Vor}(S)$, which is a more complicated structure than $\mathrm{Vor}^{\square}(S)$. Common for all the algorithms later is that attempt to store the entire Voronoi diagram in memory, which in practice means that the available system memory determines how large datasets they can handle. Since our system has 24 gigabytes of main memory, these CPU algorithms were able to handle fairly sizable datasets but still slow down significantly on larger inputs.

Table V shows the results of running times on the synthetic datasets. For all datasets, TerraNNI interpolates a grid of 5,000 by 5,000 by 20. On small to medium-sized datasets, TerraNNI takes approximately constant time, because the time taken to draw the query surfaces and write the output to disk is independent of the data size and dwarfs the handling of the input point small datasets. We also look at the time it takes for TerraNNI to perform the interpolation, without including I/O costs. The results illustrate that several phases of the algorithm are independent of the number of input points (e.g., the drawing of the query surfaces). As such as the data size increases, the interpolation time increases more slowly than the general running time, which includes the I/O time.

We compare against CPU-based algorithms by using CGAL [CGAL Development Team 2014] to perform 2D NNI on a grid of 5,000 by 5,000 and Interpolate3d [Hemsley 2009] to perform 3D NNI on a grid of 5,000 by 5,000 by 20. CGAL finds the natural neighbors for queries based on the 2D Voronoi diagram. As can be seen in Table V, CGAL is quite

Table V. Comparison of Running Times Against CPU-Based Methods
(Note CGAL and Qhull Only Compute the Triangulation)

	Triangulation		Natural Neighbor Interpolation		
Number of Points	CGAL	qhull	TerraNNI	interp3d	CGAL NNI (2D)
10^3	0.02s	0.064s	14m 9s	23h 33m 54s	2h 50m 32s
10^4	0.031s	0.25s	13m 53s	> 2d 18h	1d 3h 54m 11s
10^5	0.26s	2.597s	13m 52s	–	> 2d 2h
10^6	4.182s	29s	14m 10s	–	–
10^7	1m 9s	7m 59s	14m 9s	–	–
10^8	–	–	15m 52s	–	–
10^9	–	–	41m	–	–
10^{10}	–	–	5h 10m 13s	–	–

slow for interpolating on even small datasets. This is primarily because even for small datasets, there are many query points (2×10^7 because it is 2D NNI) to interpolate, which CGAL does not handle efficiently. At about 10^5 input points, we stopped the run after 3 days of running, at which point about 25% of the grid query points had been processed. Interpolate3d is freely available and performs NNI of 3D vector fields on the CPU. Similar to CGAL, Interpolate3d has trouble scaling to a large number of query points. At 10^4 input points, we aborted the experiments after they had run for over 2 days without completing.

We also compared TerraNNI against state-of-the-art CPU-based algorithms for computing the Delaunay triangulation, the dual of the Voronoi diagram, and thus a fundamental part of computing NNI. We used the algorithm from the CGAL [CGAL Development Team 2014] library as well as the one available in Qhull [Barber et al. 1996].

Both CGAL and Qhull were efficient in running on the smaller datasets but had trouble scaling to larger datasets. For small datasets up to 10^6 points, CGAL was very fast, taking less than 5 seconds to construct the Delaunay triangulation. This is significantly faster than the time it takes TerraNNI to perform the full interpolation. (However, in practice, TerraNNI spends only 2.07 seconds to read the data and draw the Voronoi diagram for 10^6 input points.)

On larger datasets, CGAL's speed decreases, taking just over a minute to construct the Delaunay triangulation for 10^7 data points. For larger datasets, CGAL runs out of memory and crashes. Similarly, Qhull is very fast on relatively small datasets, taking less than 5 seconds to perform the Delaunay triangulation for up to 10^5 points. However, beyond this size, the speed starts decreasing, taking nearly 8 minutes for 10^7 points. When Qhull attempts to run on 10^8 points, it too crashes.

From these experiments, we see that while some CPU methods are more efficient for small datasets, they all quickly slow down for larger datasets and ultimately cannot scale to large datasets due to memory constraints. Additionally, CPU methods do not scale well to handle many queries. TerraNNI scales well to very large datasets, partially because it computes approximate NNI, and partly because of its ability to massively parallelize the processing of both input and query points.

7. CONCLUSION

In this article, we have demonstrated that GPUs can effectively construct large grid DEMs using natural neighbor interpolation, even for spatiotemporal and 3D density data. Our algorithm uses several levels of blocking to maximize the resource usage, and the 2D algorithm is used as a fundamental building block in the 3D algorithm.

TerraNNI uses a 3D rendering API, in this case, OpenGL, to compute the Voronoi diagrams that form the basis of the interpolation, and the more general CUDA GPU processing technology is used to process the rendered output. A recent paper has used new features available in CUDA to avoid using a 3D rendering API for the case of 2D NNI [You and Zhang 2012]. An interesting question is how to combine the two approaches to develop a fast 3D algorithm for large datasets.

There are many interesting compute-intensive terrain analysis problems defined on GRID DEMs that could lend themselves to GPU computations. Some work has already been done in that direction (e.g., Lebeck et al. [2013]), but many more problems can benefit from the parallelism offered by a GPU. We are particularly interested in problems involving topological persistence [Edelsbrunner et al. 2000] and hydrological analysis [Danner et al. 2007].

ACKNOWLEDGMENTS

The authors thank Helena Mitasova and the U.S. Army Corps of Engineers for access to data, and Tamal Dey and Danny Halperin for helpful discussions.

REFERENCES

P. K. Agarwal, L. Arge, and K. Yi. 2005. I/O-efficient construction of constrained Delaunay triangulations. In *Proceedings of the European Symposium on Algorithms*. 355–366.

A. Aggarwal and J. S. Vitter. 1988. The input/output complexity of sorting and related problems. *Communications of the ACM* 31, 9 (1988), 1116–1127.

D. Attali and J.-D. Boissonnat. 2004. A linear bound on the complexity of the Delaunay triangulation of points on polyhedral surfaces. *Discrete & Computational Geometry* 31, 3 (2004), 369–384.

C. B. Barber, D. P. Dobkin, and H. Huhdanpaa. 1996. The Quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software* 22, 4 (1996), 469–483.

J.-D. Boissonnat and F. Cazals. 2000. Smooth surface reconstruction via natural neighbour interpolation of distance functions. In *Proceedings of the 16th Annual ACM Symposium on Computer Geometry*. 223–232.

CGAL Development Team. 2014. CGAL, Computational Geometry Algorithms Library. (2014). http://www.cgal.org.

A. Danner, A. Breslow, J. Baskin, and D. Wilikofsky. 2012. Hybrid MPI/GPU interpolation for grid DEM construction. In *Proceedings of the 20th International Conference on Advances in Geographic Information Systems (SIGSPATIAL'12)*. ACM, New York, NY, 299–308. DOI:http://dx.doi.org/10.1145/2424321.2424360

A. Danner, T. Mølhave, K. Yi, P. K. Agarwal, L. Arge, and H. Mitasova. 2007. TerraStream: From elevation data to watershed hierarchies. In *Proceedings of the 15th Annual ACM International Symposium on Advances in Geographic Information Systems (GIS'07)*. ACM, New York, NY, 1–8. DOI:http://dx.doi.org/10.1145/1341012.1341049

M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. 1997. *Computational Geometry – Algorithms and Applications*. Springer Verlag.

H. Edelsbrunner, D. Letscher, and A. Zomorodian. 2000. Topological persistence and simplification. In *Proceedings of the IEEE Symposium on Foundations in Computer Science* 454–463.

H. Edelsbrunner and E. P. Mücke. 1994. Three-dimensional alpha shapes. *ACM Transactions on Graphics* 13, 1 (1994), 43–72. DOI:http://dx.doi.org/10.1145/174462.156635

Q. Fan, A. Efrat, V. Koltun, S. Krishnan, and S. Venkatasubramanian. 2005. Hardware-assisted natural neighbor interpolation. In *Proceedings of the 7th Workshop on Algorithm Engineering and Experiments (ALENEX'05)*.

T. G. Farr, P. A. Rosen, E. Caro, R. Crippen, R. Duren, S. Hensley, M. Kobrick, M. Paller, E. Rodriguez, L. Roth, D. Seal, S. Shaffer, J. Shimada, J. Umland, M. Werner, M. Oskin, D. Burbank, and D. Alsdorf. 2007. The shuttle radar topography mission. *Reviews in Geophysics* 45 (2007). http://dx.doi.org/10.1029/2005RG000183

S. Ghosh, A. E. Gelfand, and T. Mølhave. 2012. Attaching uncertainty to deterministic spatial interpolations. *Statistical Methodology* 9, 1–2 (January–March 2012), 251–264. DOI:http://dx.doi.org/10.1016/j.stamet.2011.06.001

GRASS Development Team. 2014. GRASS GIS Homepage. http://www.baylor.edu/grass/. (2014).

R. Hemsley. 2009. Interpolation on a magnetic field. (2009). http://code.google.com/p/interpolate3d/.

P. C. Kyriakidis and A. G. Journel. 1999. Geostatistics space-time models: A review. *Mathematical Geology* 31, 6 (1999), 651–684. http://www.springerlink.com/index/H5L4K7V760045748.pdf.

N. Lebeck, T. Mølhave, and P. K. Agarwal. 2013. Computing highly occluded paths on a terrain. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL'13)*. ACM, New York, NY, 14–23. DOI: http://dx.doi.org/10.1145/2525314.2525363

L. Li and P. Revesz. 2002. A comparison of spatio-temporal interpolation methods. In *Geographic Information Science*. Vol. 2478. Springer, 145–160.

J. Mateu, F. Montes, and M. Fuentes. 2003. Recent advances in space-time statistics with applications to environmental data: An overview. *Geophysical Research* 108, 8774 (2003).

E. Miller. 1997. Towards a 4d-GIS: Four dimensional interpolation utilizing kriging. In *Innovations in GIS 4*, Z. Kemp (Ed.). 181–197.

H. Mitasova, L. Mitas, W. M. Brown, D. P. Gerdes, I. Kosinovsky, and T. Baker. 1995. Modeling spatially and temporally distributed phenomena: New methods and tools for GRASS GIS. *International Journal of Geographical Information Systems* 9, 4 (1995), 433–446.

H. Mitasova, M. Overton, and R. S. Harmon. 2005. Geospatial analysis of a coastal sand dune field evolution: Jockey's ridge, North Carolina. *Geomorphology* 72, 1–4 (2005), 204–221.

T. Mølhave, P. K. Agarwal, L. Arge, and M. Revsbæk. 2010. Scalable algorithms for large high-resolution terrain data. In *Proceedings of the 1st International Conference and Exhibition on Computing for Geospatial Research & Application*. ACM.

NOAA. 2014. NOAA Coastal Services Center, LIDAR Data Retrieval Tool. (2014). http://csc-s-maps-q.csc.noaa.gov/dataviewer/viewer.html?keyword=lidar.

NVIDIA. 2010. CUDA Homepage. http://nvidia.com/cuda. (2010). 3.0.

S. J. Owen. 1992. *An Implementation of Natural Neighbor Interpolation in Three Dimensions*. Master's thesis.

J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. Lefohn, and T. J. Purcell. 2007. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum* 26, 1 (2007), 80–113.

S. Park, L. Linsen, O. Kreylos, J. D. Owens, and B. Hamann. 2006. Discrete Sibson interpolation. *IEEE Transactions on Visualization and Computer Graphics* 12, 2 (March 2006), 243–253.

S. Shekhar and H. Xiong (Eds.). 2008. *Encyclopedia of GIS*. Springer.

R. Sibson. 1981. A brief description of natural neighbour interpolation. In *Interpreting Multivariate Data*, V. Barnet (Ed.). John Wiley & Sons, Chichester, 21–36.

N. Sukumar, B. Moran, and T. Belytschko. 1998. The natural element method in solid mechanics. *International Journal of Numerical Methods in Engineering* 43, 5 (1998), 839–887.

TPIE Development Team. 2014. Templated Portable I/O Environment (TPIE). http://madalgo.au.dk/tpie. (2014).

D. Watson. 1992. *Contouring: A Guide to the Analysis and Display of Spatial Data*. Oxford, Pergamon.

C. K. Wikle, L. M. Berliner, and N. Cressie. 1998. Hierarchical Bayesian space-time models. *Environmental and Ecological Statistics* 5, 2 (1998), 117–154.

S. You and J. Zhang. 2012. Constructing natural neighbor interpolation based grid DEM using CUDA. In *Proceedings of the 3rd International Conference on Computing for Geospatial Research and Applications (COM.Geo'12)*. ACM, New York, NY, Article 28, 6 pages. DOI: http://dx.doi.org/10.1145/ 2345316.2345349

Received August 2014; revised March 2015; accepted May 2015