From Point Cloud to 2D and 3D Grids: A Natural Neighbor Interpolation Algorithm using the GPU

by

Alex Beutel

Department of Computer Science
Duke University

Date: ____________________

Approved:

____
Pankaj K. Agarwal, Advisor

____
Thomas Mølhave, Mentor

____
Bruce Maggs

____
Henry Greenside

Thesis submitted in partial fulfillment of the requirements for
graduating with distinction in the Department of Computer Science
of Duke University
2011

Abstract

With modern LiDAR technology the amount of topographic data, in the form of massive point clouds, has increased dramatically. One of the most fundamental GIS tasks is to construct a grid digital elevation model (DEM) from these point clouds. We present a simple yet very fast natural neighbor interpolation algorithm for constructing a grid DEM from massive point clouds. We use the graphics processing unit (GPU) to significantly speed up the computation. To handle the large data sets and to deal with graphics hardware limitations clever blocking schemes are used to partition the point cloud. This algorithm is about an order of magnitude faster than the much simpler linear interpolation, which produces a much less smooth surface. We also show how to extend our algorithm to higher dimensions, which is useful for constructing 3D grids, such as from spatial-temporal topographic data. We describe different algorithms to attain speed and memory trade-offs.

Contents

Acknowledgements

I would like to give a special thanks to my research advisor Professor Pankaj K. Agarwal and my mentor Thomas Mølhave for all of their help and guidance over the past two years. I would also like to thank the Army Research Office, COWI A/S, and Professor Helena Mitasova for providing terrain data.

1

Introduction

The revolution in sensing and mapping technologies is providing an unprecedented opportunity to characterize and understand the earth's surface and dynamics. For instance, modern airborne LiDAR technology can map the earth's surface at a 15-20cm horizontal resolution, and the future generation of LiDAR scanners are expected to generate high-resolution maps of other planets; see Figure 1.1(a). It is essential for many applications to exploit the high-resolution data sets that are available. An example of this can be seen in a simple flood mapping application. Figure 1.1(b,c) shows 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 five meter tall perimeter dike which protects it from the sea. Because of the small width of the perimeter dike, this feature is not present in low- or mid-resolution grids. Thus, when flood maps are constructed for a water level of 2 meters, it looks as if most of the island will be underwater; see Figure 1.1(b) for an example using a 90m grid (the SRTM grid available from NASA [13]). The same computation performed on a 2m-resolution grid, shown in Figure 1.1(c), correctly finds that the dikes, now present in the terrain model, block the water from entering the lower-lying areas inside the perimeter.

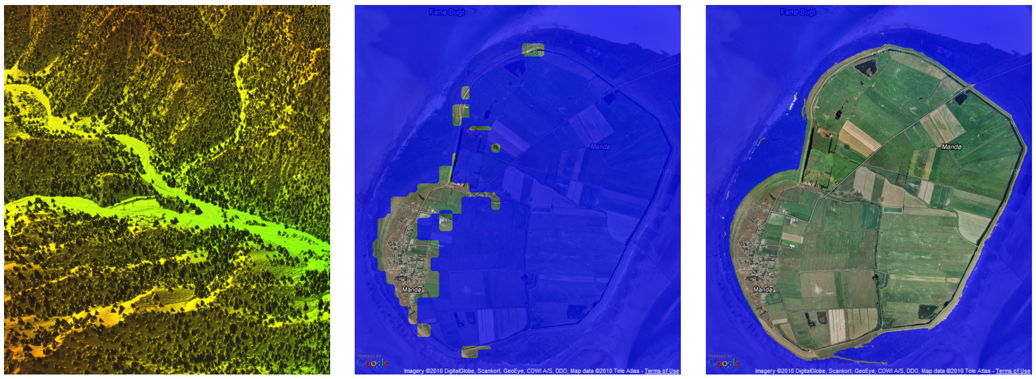

(a) 0.5m grid (b) 90m grid (c) 2m grid

FIGURE 1.1: (a) Grid DEM constructed from LiDAR data over a region in Afghanistan (data source: Army Research Office), trees are clearly visible. (b) A flood risk mapping of the island of Mandø in Denmark, using the 90m, and (c) the same using 2m grid; both figures are screenshots of a custom map application built on Google Maps.

Capitalizing on opportunities made feasible by high resolution data sets and transforming this massive amount of topographic data into useful information for vastly different types of users requires solving several challenging algorithmic problems. For example, in order to fully explore topographic data, one must often first extract a terrain from the scattered set of points generated by the LiDAR equipment. Most GIS applications do not work directly on the point cloud S gathered by a LiDAR scanner, but instead operate on a *digital elevation model (DEM)* of the terrain surface. Thus one of the most important of these problems is to generate a DEM from S.

Because of its simplicity and efficiency, the most widely used DEM is a uniform two-dimensional grid in which an elevation value is stored at each cell. However, point clouds are not acquired on a uniform grid but 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.

Additionally, as acquiring high resolution data sets becomes easier, it has also become more reasonable to have multiple scans of the same region over multiple years. In dynamic terrains, such as coastal regions, spatial-temporal data can offer interesting insight into how the terrain changes over time. To analyze the surface we still would like create grid DEMs based on the point cloud, but we now have a higher dimensional point cloud, set of points S in $\mathbb{R}^3$, and would like to produce DEMs for many slices in time, or more generally, interpolate to a higher dimensional grid. This provides additional algorithmic challenges.

1.1 Related Work and Contributions

Numerous interpolation methods, ranging from sophisticated but computationally expensive methods to simpler and efficient methods, for grid DEMs have been developed; see [20] for a review. Regularized splines with tension (RST) is a well-known method, which is sophisticated but computationally expensive due to its use of non-trivial polynomials [1, 21]. RST and similar highly sophisticated interpolation methods are especially good when the input data is sparse and lots of interpolation has to be performed. On the other hand, constructing a triangulation on input points and linearly interpolating the elevation of grid points across the triangles is one of the simplest interpolation methods. It, however, does not produce a smooth surface, especially when the data is relatively sparse. The resulting grid DEM can appear jagged both when viewed directly and also in derived products, such as contour maps.

In this thesis we use the well-known *natural neighbor interpolation* strategy [26]. Based on the Voronoi diagram of S, it produces a smoother surface than linear interpolation. Although NNI is more efficient than RST and other similar interpolation methods, its traditional implementations are slower than linear interpolation and are therefore not widely used for very large data sets.

Over the last decade modern PCs have started to become equipped with advanced and increasingly powerful graphics processing units (GPUs). Although originally designed for rapidly transforming 3D geometric scenes into pixels on the 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, and they have been used for a wide range of applications, e.g., geometric computing [3], robotic collision detection [14], database systems [15], fluid dynamics [19], and solving sparse linear systems [9, 8]. As GPUs have become more flexible and programmable (e.g. NVIDIA's CUDA [23] library), their applicability has also increased tremendously; see [24] for a recent survey. In the context of grid DEM construction, Fan *et al.* [12] have described a GPU based algorithm for natural neighbor interpolation (NNI), which is considerably faster than a CPU based algorithm.

Results. In this thesis we present a simple yet very fast GPU based algorithm for constructing a grid DEM from large LiDAR point clouds using a variant of the natural neighbor interpolation method. LiDAR scanners provide dense (high resolution) point cloud of elevation data at most locations, but there are gaps, usually at large bodies of water or human-made objects that have been removed from the point cloud in a preprocessing step. In such cases we wish to label the corresponding "gap" cells in the grid DEM with "no-data" instead of interpolating elevation based on points that are far away. We introduce the notion of *region of influence* for each input point, similar to the one used in α-shapes [11]. For a grid point, we use only those points to compute its elevation whose regions of influence contain the query point. See Section 2.2 for details. Although our algorithm is similar to that of Fan *et al.* [12], there are three main differences:

(i) Our algorithms handles gaps differently, as described above.

(ii) Exploiting the fact that we are interpolating elevations at grid points, it uses a clever "blocking" scheme to expedite the computation considerably. In contrast to the algorithm in [12], which performs NNI interpolation at $\leqslant 32$ points (or $\leqslant 128$ points depending on hardware properties of the GPU card) in one step, it can answer more than 10^6 NNI queries at grid points in one step.

(iii) 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, our algorithm computes a grid DEM covering a 600km^2 region at $2m$ resolution (i.e., ≈ 150 million grid points) from two billion input points in less than thirty-seven minutes on a 3GHz Intel Core 2 Duo processer with a NVIDIA GeForce GTX 470 graphics card. Our CPU based linear-interpolation algorithm takes more than five and a half hours on the same PC. Not only is this a significant speedup, NNI interpolation also produces a smoother grid DEM than the linear-interpolation method. The more sophisticated RST-based algorithm takes about thirty-four hours on the same data set, 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.

In extending the algorithm to handle higher dimensional grids, we exploit redundancy in interpolating on adjacent grids and discuss memory and time trade-offs in handling this redundancy. We find that at the expense of GPU memory we can reduce the time spent on interpolation to a third of the time, and without any loss of interpolation quality.

The thesis is organized as follows. Section 1.2 provides a brief overview of the GPU model of computation, which will be used and referenced through the rest of the thesis. In Chapter 2 we discuss the algorithm for constructing 2D grids. This chapter is based on an extended version of our 2010 ACM GIS paper [6]. Section 2.1 describes a GPU based algorithm for computing the Voronoi diagram of a set of points, and Section 2.2 describes a slight variant of the Fan *et al.* [12] algorithm for computing natural neighbor interpolation. Section 2.3 describes the new algorithm for computing NNI interpolation on a grid, and Section 2.4 contains implementation details and information about the speed and quality experiments we have performed. In Chapter 3 we dicuss extending the algorithm to constructing 3D grids. In Sections 3.2 and 3.3 we discuss computing the Voroonoi diagram and performing natural neighbor interpolation on the GPU in 3D. In Section 3.4 we discuss different algorithms for interpolating on 3D grids and in Section 3.5 we compare the results from these different algorithms.

1.2 GPU Model of Computation

Primarily designed to achieve high performance for interactive graphics applications, modern programmable GPUs consist of a large number of processors (e.g., up to 480 for the newest NVIDIA GeForce 4-series) with a high memory bandwidth (177.4 GB/sec for NVIDIA's GeForce GTX 480) and achieve higher floating-point throughput than the CPUs. This high throughput has led to a tremendous effort for developing GPU based numerical algorithms; see the recent survey by Owens *et al.* [24] and the references therein.

The graphics computation in all GPUs follows a similar pipeline, called the *graphics pipeline*, which draws a three dimensional scene, composed of many objects, onto a two dimensional image plane Π of pixels as seen from a specified viewpoint o. Because of their simplicity and flexibility, these objects are almost always triangles. For each pixel $\pi = (x, y)$ where x, y is a global coordinate, the GPU finds all objects $\Omega = \{\omega_1, \omega_2, \ldots \omega_n\}$ which ray $\vec{o\pi}$ intersects. To maintain high throughput, each stage of the computation is implemented in hardware and computation on different parts of Π is performed in parallel. See e.g. [18, 22] for details of NVIDIA's GeForce 6 series. Here we note that, a GPU contains several two dimensional arrays of pixels called buffers (or texture memory). We mention the two most basic ones, which we will use:

- 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 $\vec{o\pi}$ and object ω_j, the GPU calculates

  $$\mathbb{D}[\pi] = \min_{1 \leqslant j \leqslant n} \|op_j\|.$$

  The modern GPUs provide the flexibility of performing various *semigroup* operations on op_j's instead of simply computing the minimum and also of performing them in a conditional manner. For example, $\mathbb{D}$ can be in the *read-only* mode.

- The *color buffer* $\mathbb{C}$ stores the color of the scene as viewed from o. If for each object $\omega_j \in \Omega$ we have a color χ_j, we define a blending function that computes the color at each pixel:

  $$\mathbb{C}[\pi] = \sum_{1 \leqslant j \leqslant n} \alpha_j \chi_j,$$

  where $\alpha_j \in [0, 1]$ is the blending parameter of ω_j. Typically, α_j is based on depth buffer so that $\mathbb{C}$ stores the color of the foremost object. Again, the modern GPUs provide several other binary operations on the colors. We will like our blending function to compute the bitwise-OR of the colors. This can be done by setting $\alpha_j = 1$ for $1 \leqslant j \leqslant n$, as long as χ_j are bitwise-disjoint.

During graphical computations, the color and depth buffers reside in memory on the graphics card. Objects can be drawn onto these buffers with specific APIs such as OpenGL[25] or Microsoft DirectX [7]. However, in some cases we will want to use the values in these buffers for computation on the CPU. For this, we have to read the buffer back to the computer's main memory. Unfortunately, since this involves transferring large amounts of data over the relatively slow bus systems, read backs are very slow.

For using the GPUs parallel processing capabilities, the popular graphics card manufacturer NVIDIA has created the CUDA parallel computing architecture. CUDA makes it easy to divide a task into many threads, where threads can work in parallel but also when necessary share memory and work together on procedures that aren't trivially parallelizable. Additionally advantageous is that CUDA operations are performed directly on the graphics card and can efficiently access buffers, which reside in nearby GPU memory. As an example, CUDA can split into one thread for each pixel of a buffer and read from each pixel simultaneously without reading the buffer back to main memory. Inversely, modern GPUs allow multiple threads to write to the same memory with atomic functions that provide thread synchronization and serialization.

2

Grid DEM Construction using a GPU

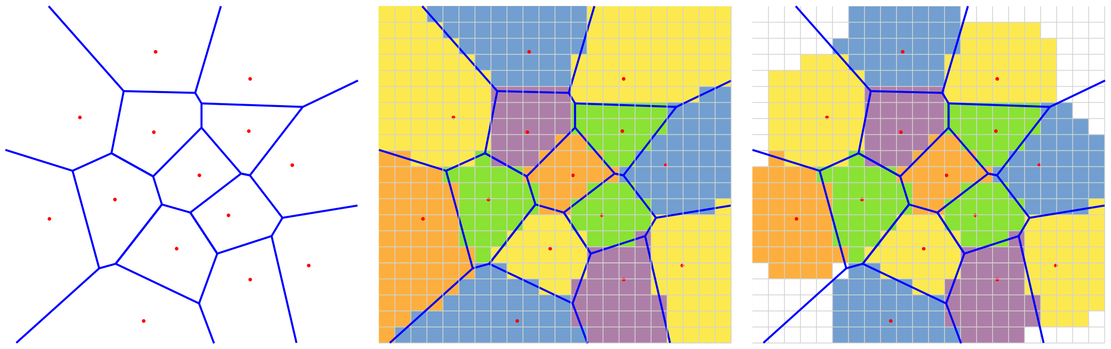
2.1 Pixelized Voronoi Diagram

Let $S = \{p_1, \dots p_n\}$ be a set of n points in $\mathbb{R}^2$. For each point $p \in S$, its *Voronoi cell*, denoted by $\mathrm{Vor}_S(p)$, is defined as

$$\mathrm{Vor}_S(p) = \{x \in \mathbb{R}^2 \mid \|xp\| \leqslant \|xq\| \forall q \in S\},$$

where $\| \cdot \|$ is the Euclidean distance, i.e., a point $x \in \mathrm{Vor}_S(p)$ if p is the point in S closest to x. The *Voronoi diagram* of S, $\mathrm{Vor}(S)$, is the planar subdivision induced by the Voronoi cells of points in S. See Figure 2.1 (a).

Hoff *et al.*[17] have described a GPU based algorithm for computing the Voronoi diagram of a set of points. Since we use a slightly different algorithm, we describe the algorithm for the sake of clarity and completeness. An image plane Π consisting of $N \times N$ pixels can be regarded as the square $[0, N-1] \times [0, N-1]$ in $\mathbb{R}^2$. Any square $R \subseteq \mathbb{R}^2$ can be mapped to Π using an affine transformation. Given the set S and a square R, we are interested in computing a *pixelized* (*discretized*) *Voronoi diagram* of S within R, which we define below. We assume that R is mapped to the image plane Π. Each pixel of Π corresponds to a (tiny) square of area $\rho^2 = \mathrm{Area}(R)/N^2$. We refer to ρ as the *resolution*



(a) (b) (c)

FIGURE 2.1: (a) Voronoi diagram $\mathrm{Vor}(S)$ of a set S of points. (b) Pixelized Voronoi diagram $\mathrm{PVor}(S)$. (c) Truncated pixelized Voronoi diagram $\mathrm{TPVor}(S)$ of S with $r = 5$, k was set high enough for the k-gons to be indistinguishable from cones.

of Π. For a pixel $\pi \in \Pi$, let $\varphi(\pi, S)$ be the point in S whose Voronoi cell contains π. Since π is a (tiny) square region, it may intersect multiple Voronoi cells, in which case $\varphi(\pi, S)$ is assigned to one of the points using standard methods. For a point $p \in S$, we define the *pixelized Voronoi cell* of p to be

$$\mathrm{PVor}_S(p) = \{\pi \mid \varphi(\pi, S) = p\},$$

i.e., the set of pixels that lie in $\mathrm{Vor}_S(p)$; see Figure 2.1 (b). The quantity $\rho^2|\,\mathrm{PVor}_S(p)|$ approximates the area of $\mathrm{Vor}_S(p)$ within R. The approximation error depends on ρ. For a fixed R, the error decreases as we increase N, namely,

$$\lim_{N\to\infty} \rho^2|\,\mathrm{PVor}_S(p)| = \mathrm{Area}(\mathrm{Vor}_S(p)).$$

For our purpose, we assume that $\mathrm{PVor}(s)$ is stored as follows. If $\varphi(\pi, S) = p_i$, then the color buffer $\mathbb{C}[\pi] = i$, i.e., we view each cell of the color buffer as a single word (concatenation of R, G, B, A components) that stores the index of the point; $\mathbb{D}[\pi]$ stores the value of $\|\pi\varphi(\pi, S)\|$, the distance from the center of π to its nearest neighbor in S.

The problem of computing $\mathrm{PVor}(S)$ can be formulated as that of rendering a 3D scene. For a point $p_i \in S$, let $f_i : \mathbb{R}^2 \to \mathbb{R}$ be defined as $f_i(x) = \|xp_i\|$. The *lower envelope* f of

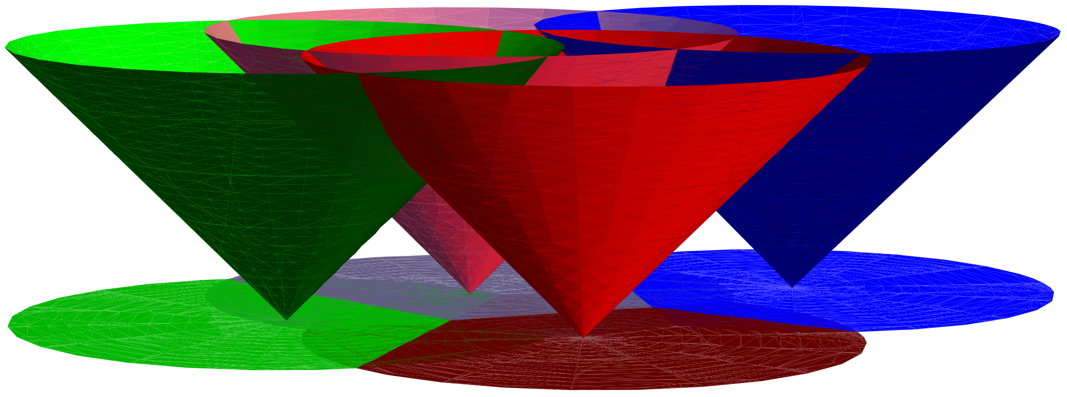


FIGURE 2.2: Voronoi diagram as the lower envelope of a set of cones. The outer cells of a Voronoi diagram are infinite, but in this figure their sizes are limited because the cones are truncated.

$\{f_1, \ldots, f_n\}$ is defined to be

$$f(x) = \min_{1 \leqslant i \leqslant n} f_i(x)\,,$$

which is the distance from x to its nearest neighbor. $\mathrm{Vor}(S)$ is the projection of the graph of f on the xy-plane. Let C be the circular cone $C : z = \sqrt{x^2 + y^2}$ in $\mathbb{R}^3$. Then the graph of each f_i is a circular cone $C_i = C + p_i$, with p_i as its apex. See Figure 2.2. Let $\mathcal{C} = \{C_1, \ldots, C_n\}$. A point $x \in \mathrm{Vor}_S(p_i)$ if $f(x)$ is realized by the function f_i at x, i.e., the line oriented in the $+z$ direction hits C_i first. In other words, $\varphi(\pi, S) = p_i$ if C_i is the cone seen at pixel π when the set C is viewed from $z = -\infty$. If we set the color of C_i to i, then the color and depth buffers store the desired information.

It is not easy to render a circular cone using a GPU, so we approximate a circular disk by a regular k-gon and approximate the circular cone by using this k-gon as its base (see Figure 2.3). The resulting polygonal cone $C^\Diamond$ is composed of k triangles. We replace C_i by $C_i^\Diamond = C^\Diamond + p_i$. The error in tessellation induced by this approximation can be controlled by choosing the value of k appropriately.

Finally, we note that we want to limit the *region of influence* for the points. We do this by using a truncated Voronoi diagram. We define the *radius of influence* r of each point in

S and the notion of a *truncated* pixelized Voronoi cell:

$$\mathrm{TPVor}_S(p) = \{\pi \mid \varphi(\pi) = p \wedge \|p\pi\| < r\}.$$

See Figure 2.1 (c). Thus, a pixel π that is farther than r away from all points of S does not belong to the Voronoi cell of any point. Let D_r denote the disk of radius r centered at origin. We can assume that $S \subset R + D_r$, as no point outside this region will contain any pixel of Π in its Voronoi cell. This truncation is realized by limiting the height of the cones $C_i^\diamond$; with a slight abuse of notation we use $C_i^\diamond$ to denote the truncated cone as well. For each $p_i \in S$, we set the color of each triangle of $C_i^\diamond$ to i and pass them to the graphics pipeline with $z = -\infty$ as the viewpoint. $\mathbb{C}$ and $\mathbb{D}$ together contain $\mathrm{TPVor}(S)$. We refer to this algorithm as GPUVORONOI (S). As mentioned above, there might be pixels that are not touched by GPUVORONOI (S). We assume that $\mathbb{C}$ is initialized with a value that allows us to distinguish these pixels from the pixels that are part of the truncated diagram, e.g., we set their color to 0.

2.2 Natural Neighbor Interpolation

In this section we first formally define natural neighbor interpolation (NNI), then describe a GPU algorithm for answering NNI queries, which is a small variant of the algorithm by Fan *et al.* [12]. A height function $h : S \to \mathbb{R}$ can be extended to entire $\mathbb{R}^2$ using natural

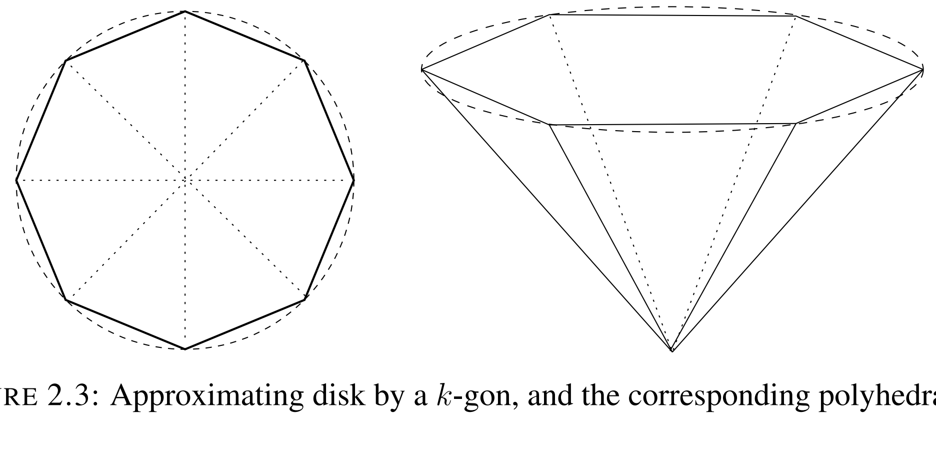


FIGURE 2.3: Approximating disk by a k-gon, and the corresponding polyhedralcone.

neighbor interpolation. In particular, for a point $x \in \mathbb{R}^2$,

$$h(x) = \sum_{p \in S} w_p(x) h(p),$$

where $w_p(x)$ is the fractional area of $\mathrm{Vor}_{S \cup \{x\}}(x)$ that belongs to $\mathrm{Vor}_S(p)$ (Figure 2.4), i.e.,

$$w_p(x) = \frac{\mathrm{Area}(\mathrm{Vor}_S(p) \cap \mathrm{Vor}_{S \cup \{x\}}(x))}{\mathrm{Area}(\mathrm{Vor}_{S \cup \{x\}}(x))}.$$

Since we use truncated pixelized Voronoi diagrams, we redefine the height function as

$$h(x) = \sum_{p \in S} \overline{w}_p(x) \cdot h(p) \tag{2.1}$$

where

$$\overline{w}_p(x) = \frac{|\,\mathrm{TPVor}_S(p) \cap \mathrm{TPVor}_{S \cup \{x\}}(x)|}{|\,\mathrm{TPVor}_{S \cup \{x\}}(x)|}. \tag{2.2}$$

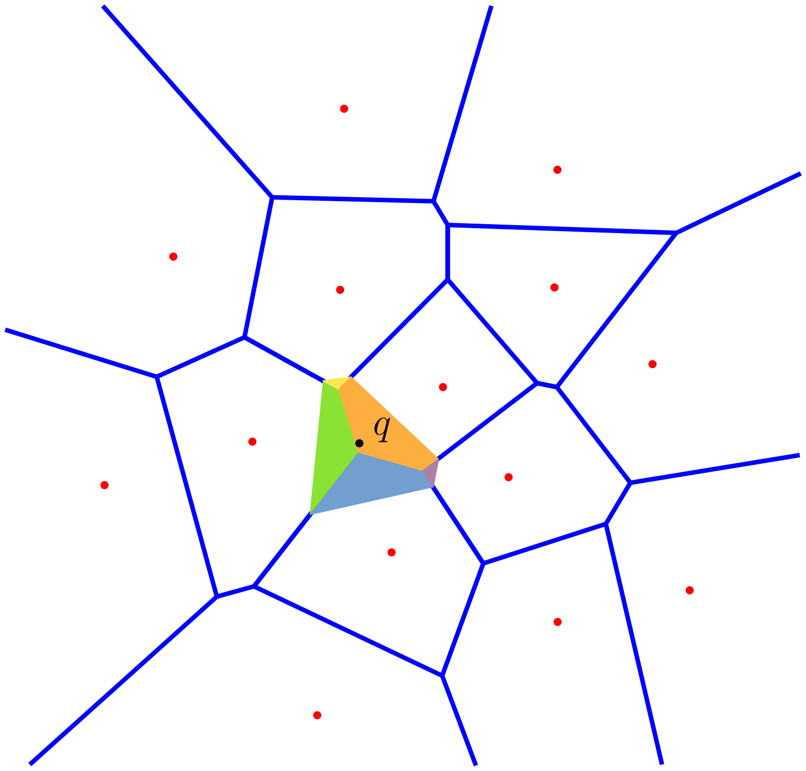


FIGURE 2.4: Natural neighbor interpolation. Shaded cell is $\mathrm{Vor}_{S \cup \{q\}}(q)$, and each color denotes the area stolen from each cell of $\mathrm{Vor}(S)$.

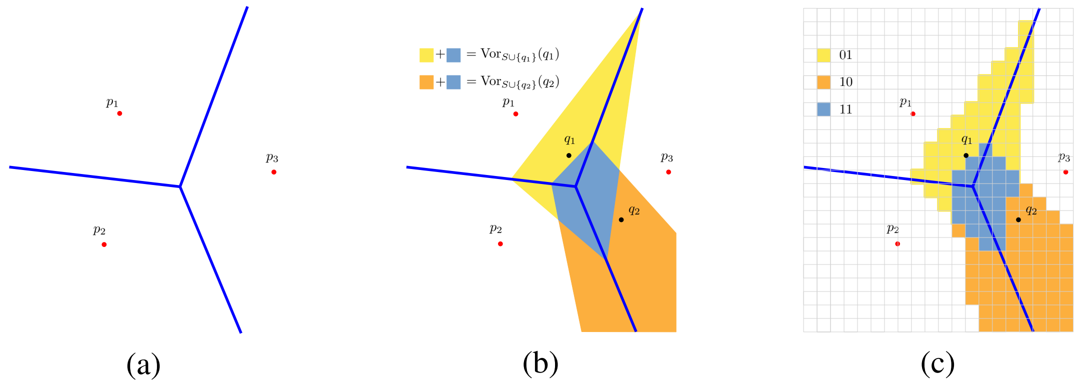


FIGURE 2.5: (a) $\mathrm{Vor}(S)$ of a set S. (b) $\mathrm{Vor}(S \cup \{q_1\})$ and $\mathrm{Vor}(S \cup \{q_2\})$ for two query points q_1 and q_2. (c) $\mathrm{PVor}(S \cup \{q_1\})$ and $\mathrm{PVor}(S \cup q_2)$ The colors correspond to the bitwise-OR colors of the query point. Query point q_1 and q_2 have colors 01 and 10 respectively, the pixels in $\mathrm{PVor}_{S\cup\{q_1\}}(q_1) \cap \mathrm{PVor}_{S\cup q_2}(q_2)$ thus get the color $01 \vee 10 = 11$, where $\vee$ is bitwise or.

Answering an NNI query. For a point $x \in \mathbb{R}^2$, let $D_r(x) = D_r + x$ denote the disk of radius r centered at x. Let q be a query point such that $D_r(q) \subseteq R$. The algorithm for computing $h(q)$ works in two phases. The first phase calls GPUVORONOI (S) with the following twist: the color of each triangle of the cone $C_i^\diamondsuit$ is set to $h(p_i)$ (instead of i). After the first phase $\mathbb{C}[\pi]$ stores $h(p_i)$ for all pixels $\pi \in \mathrm{TPVor}_S(p_i)$. We read back the color buffer; let $\mathcal{C}^1$ denote the resulting two-dimensional array. We then clear the color buffer. The depth buffer $\mathbb{D}$ is is not touched, i.e., $\mathbb{D}[\pi]$ continues to store $\|\pi\varphi(\pi, S)\|$, the distance from the center of π to $\varphi(\pi, S)$. In the second phase, we set $\mathbb{D}$ to read-only mode so that it is not overwritten and draw a polygonal cone $qC^\diamondsuit = C^\diamondsuit + q$ with q as the apex. Adding $qC^\diamondsuit$ is the same as computing $\mathrm{TPVor}_{S\cup\{q\}}(q)$. However, the color buffer was cleared before the second phase and thus has non-zero entries[1] (corresponding to the color of $qC^\diamondsuit$) only for $\mathrm{TPVor}_{S\cup\{q\}}(q)$. Let $\mathbb{C}^2$ denote the color buffer contents after the second phase, and $\mathcal{C}^2$ the array resulting from reading back $\mathbb{C}^2$ into memory. The value of $h(q)$ can be computed by adding the values of $\mathcal{C}^1[\pi]$ for all π for which $\mathcal{C}^2[\pi] \neq 0$ and finally dividing the sum by the number of non-zero values in $\mathcal{C}^2$ (this is the denominator of (3.1)).

[1] We assume without loss of generality that all points of S have a positive height.

Algorithm 1 BUFFERANALYSIS($\mathcal{C}^1$, $\mathcal{C}^2$)

```
for all $\pi \in \mathcal{C}^2$ do
    if $\mathcal{C}^2[\pi] \neq 0$ and $\mathcal{C}^1[\pi] \neq 0$ then
        $N_q \leftarrow N_q + \mathcal{C}^1[\pi]$
        $D_q \leftarrow D_q + 1$
return $N_q/D_q$
```

We refer to this final step of the algorithm as BUFFERANALYSIS; Algorithm 1 gives the pseudo code.

There are four main sources of difference between our method and the traditional natural neighbor interpolation:

(i) The tessellation error caused by using k-gons instead of cones.

(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 radius of influence r.

Batching the queries. The above algorithm is very inefficient if we want to compute heights at many points, especially since reading back a buffer is a slow step. Fan *et al.* [12] have shown that by exploiting the power of modern GPUs, many queries can be batched and answered in one pass. More precisely, if each pixel of the color buffer $\mathbb{C}$ has w bits, then $m \leqslant w$ queries can be batched in one pass by encoding the colors cleverly in $\mathbb{C}$: One bit of $\mathbb{C}[\pi]$ is assigned for each query point. Let $q_1, q_2, \ldots, q_m$ be the m query points, and let $qC_i^\Diamond = C^\Diamond + q_i$ be the cone corresponding to the query point q_i. The color of all triangles in $qC_i^\Diamond$ is set to 2^i. This ensures that colors of the m query points are bitwise-disjoint.

The first phase of the algorithm is the same as before. Let $\mathcal{C}^1$ be the same as above. In the second phase, we again set the depth buffer to the read-only mode. We draw

Algorithm 2 BUFFERANALYSIS($\mathcal{C}^1$, $\mathcal{C}^2$,m)

```
for $i \leftarrow 1$ to $m$ do
    $N_q[i] \leftarrow 0$
    $D_q[i] \leftarrow 0$
for all $\pi \in \mathcal{C}^2$ do
    if $\mathcal{C}^2[\pi] \neq 0 \wedge \mathcal{C}^1[\pi] \neq 0$ then
        $v = \mathcal{C}^2[\pi]$
        for $i \leftarrow 1$ to $m$ do
            if $v_i = 1$ then $\{\pi \in \text{TPVor}_{S \cup \{q_i\}}\}$
                $N_q[i] \leftarrow N_q[i] + \mathcal{C}^1[\pi]$
                $D_q[i] \leftarrow D_q[i] + 1$
for $i \leftarrow 1$ to $m$ do
    $H[i] \leftarrow N_q[i]/D_q[i]$
return H
```

$qC_1^\diamondsuit, qC_2^\diamondsuit, \ldots, qC_m^\diamondsuit$ but the color buffer now operates as follows. Suppose the graphics pipeline is rendering a triangle t of color χ_t. If the depth of t at a pixel π is larger than $\mathbb{D}[\pi]$, then nothing happens. Otherwise $\mathbb{C}[\pi]$ is set to $\mathbb{C}[\pi] \leftarrow \mathbb{C}[\pi] \vee \chi_t$, where $\vee$ is the bitwise OR operation. Recall that $\mathbb{D}[\pi]$ is not updated, as it is in the read-only mode and $\mathbb{D}[\pi]$ stores $\|\pi\varphi(\pi)\|$. After the second phase, the i'th bit of $\mathbb{C}[\pi]$ is 1 if $\pi \in \text{TPVor}_{S \cup \{q_i\}}(q_i)$ see Figure 2.5(c).

We read back the color buffer; let $\mathcal{C}^2$ denote the contents of the buffer after the second phase. We compute $h(q_i)$ by summing the values of $\mathcal{C}^1[\pi]$ for all pixels π for which the i^{th} bit of $\mathcal{C}^2[\pi]$ is 1 and then dividing the sum by the number of such pixels. Algorithm 2 gives the pseudo code for doing this step efficiently; N_q and D_q are arrays of length m; for a bit-vector v, v_i denotes its i'th bit; and H is an array of length m, where $H[i] = h(q_i)$.

2.3 NNI on Grids

We now describe a faster algorithm for answering NNI queries when the query points lie on an $M \times M$ grid $\mathbb{Q}$. The algorithm can easily be extended to handle rectangular grids. For convenience we will use $\mathbb{Q}[i, j]$ to denote the (i, j)'th query point of $\mathbb{Q}$ for $0 \leqslant i, j < M$. Let $s\rho$ denote the size of each grid cell in $\mathbb{Q}$, for some positive integer s. We refer to s as the *scaling parameter*, and for simplicity we assume that s is odd and that

$\rho(s-1)/2 < 2r$. $\mathbb{Q}$ can be 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. Since the radius of influence is r, we can assume that all points of S lie within distance $2r$ from $\mathbb{Q}$; see below for an explanation. Let $\sigma = [-2r, +2r]^2$ and $\mathcal{Q} = \mathbb{Q} + \sigma$, so $S \subset \mathcal{Q}$. For now we assume that $N \geqslant sM + 4r/\rho$, i.e., $M \leqslant (N - 4r/\rho)/s$. Later we will describe how to handle larger query grids. Our assumption ensures that $\mathcal{Q}$ can be mapped to Π with ρ being the resolution of each pixel. Let $\alpha = (s-1)/2 + 2r/\rho$. We map the bottom left corner of $\mathcal{Q}$ to that of Π, so the query point $\mathbb{Q}[i, j]$ maps to the pixel $(s \cdot i + \alpha, s \cdot j + \alpha)$; see Figure 2.6.

Let $B = \lfloor\sqrt{w}\rfloor$, where w, as above, is the number of bits in the color buffer. For simplicity, we assume that B is a divisor of M. We partition $\mathbb{Q}$ into $(M/B)^2$ query blocks, each of size $B \times B$, with the $(i, j)^{th}$ block, for $0 \leqslant i, j < M/B$, being

$$\mathbb{Q}_{i,j} = \mathbb{Q}[iB, (i+1)B - 1][jB, (j+1)B - 1].$$

See Figure 2.7 (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 block that contains q, $\mathbf{t} \in [0, B-1]^2$ is the offset of q within that query block.

We process all $B^2 \leqslant w$ NNI queries in each block in one pass, using the algorithm

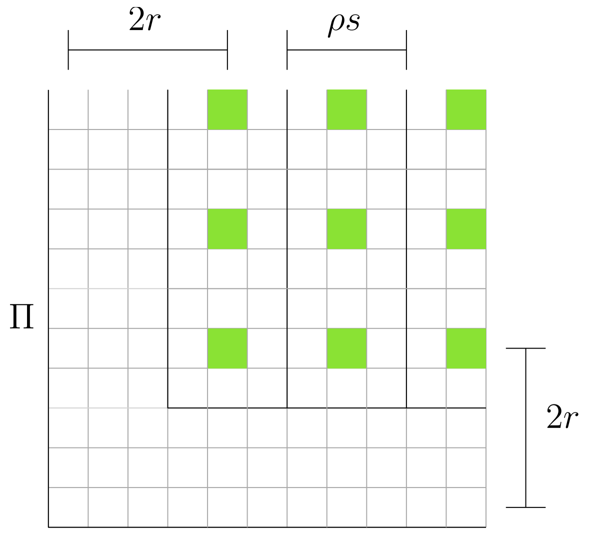


FIGURE 2.6: Embedding $\mathbb{Q}$ on Π; $s = 3$ and $r = 2\rho$.

described in Section 2.2. Processing all queries in $\mathbb{Q}$ thus requires $(M/B)^2$ passes, each involving the expensive operation of reading back the GPU memory to the main memory.

One-pass algorithm. By exploiting the grid structure of query points and the fact that a query q is affected by only those points p for which $D_r(p)$ and $D_r(q)$ intersect (i.e., $\|pq\| \leqslant 2r$), we show that we can answer all queries in one pass provided that

(A1) $M \leqslant (N - 4r/\rho)/s$, and

(A2) $r \leqslant s\rho B/2$.

The second assumption is reasonable for high resolution LiDAR data sets because the height of a point can be interpolated from the nearby sampled points. For a graphics card with $w = 32$ and with $s = 5$ we will assume that $r < 5$ meter for an output grid with a resolution of $s\rho = 2$ meter. In other words, the result of a query is not affected by an input point that is more than 20 meters away. With high-resolution LiDAR data sets this is not a bad assumption. Such gaps usually appear when buildings and other features are removed using a classification algorithm, or at lakes and similar features.

Here is the key idea that enables us to answer all queries in one pass assuming that (A1) and (A2) hold. We call two points $p, q \in S$ *independent* if $\mathrm{TPVor}_S(p)$ and $\mathrm{TPVor}_S(q)$ are disjoint. If $\|pq\| > 2r$, then, by definition of $\mathrm{TPVor}(S)$, p and q are independent. Let $q_1, \ldots, q_u$ be a set of query points such that $\|q_iq_j\| > 2r$ for any pair $i \neq j$. Then $qC_i^\Diamond$ and $qC_j^\Diamond$ are disjoint. We can potentially use the same color for all of these query points because no pixel will be rendered by two such cones in the second pass of the NNI query-answering algorithm. The difficulty with using the same color for all q_i's is that the color of $qC_i^\Diamond$ no longer encodes the value of i, so the algorithm does not know which query point colored a given pixel. However, q_i's being independent implies that there is at most one query point for each pixel π that could color π, namely, the query point closest to π and it lies within distance r from π.

In our case, the query points lie on a grid $\mathbb{Q}$ and we assume that $r < s\rho B/2$. Therefore for any two query blocks $\mathbb{Q}_{\mathbf{a}}$ and $\mathbb{Q}_{\mathbf{b}}$ and an offset $\mathbf{t}$, the query points $(\mathbf{a}, \mathbf{t})$ and $(\mathbf{b}, \mathbf{t})$ are independent, as the distance between query points with the same offset in two adjacent blocks is $s\rho B$. In other words, for any $\mathbf{t} \in [0, B-1]^2$, all points in

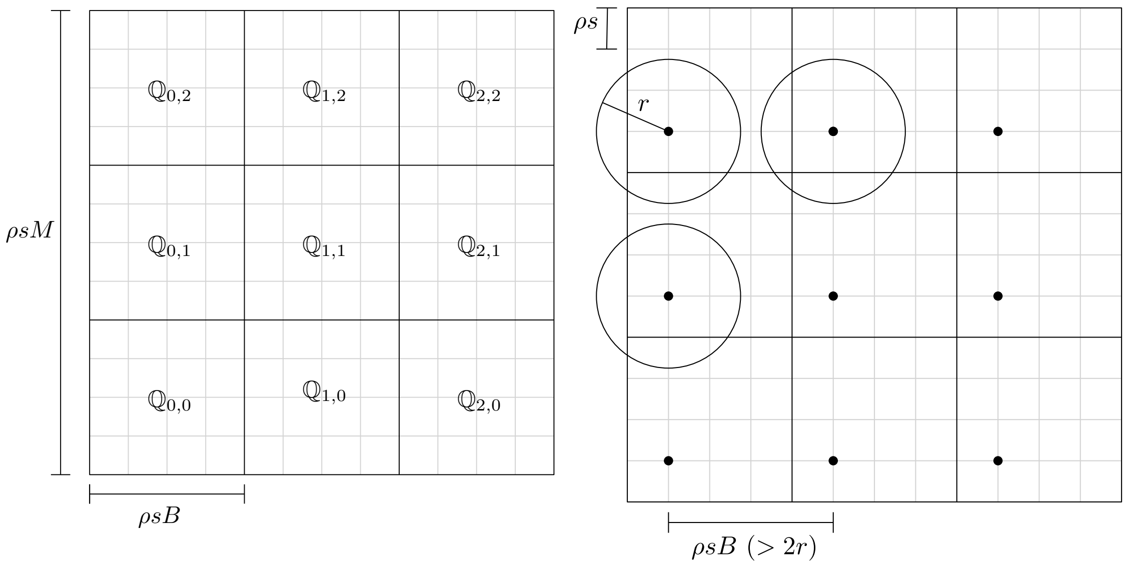
$$\mathbb{Q}_{|\mathbf{t}} = \{(\mathbf{a}, \mathbf{t}) \mid \mathbf{a} \in [0, M/B - 1]^2\},$$

the set of all query points with offset $\mathbf{t}$, are independent; see Figure 2.7 (b). We can therefore assign the same color, say, χ, to all points in $\mathbb{Q}_{|\mathbf{t}}$. If a pixel π is colored χ, we can determine in $O(1)$ time which point in $\mathbb{Q}_{|\mathbf{t}}$ colored π. Hence, we proceed as follows.

For $\mathbf{t} = (t_1, t_2) \in [0, B-1]^2$, we set $\chi(\mathbf{t}) = 2^{t_1 B + t_2}$ and assign the color $\chi(q)$ to all triangles of the query cone $C^\diamond + q$ for $q \in \mathbb{Q}_{|\mathbf{t}}$. Let

$$\mathcal{C} = \{qC^\diamond_{ij} = C^\diamond + \mathbb{Q}[i,j] \mid 0 \leqslant i, j < M\}$$

be the set of all query cones.



(a) (b)

FIGURE 2.7: (a) Splitting $\mathbb{Q}$ into query blocks of size $B = 4$ query points. (b) All the $\mathbb{Q}_{|[1,1]}$ query points are independent since their areas of influence (depicted by circles in the figure) are disjoint.

Algorithm 3 BUFFERANALYSIS($\mathcal{C}^1$, $\mathcal{C}^2$,$\mathbb{Q}$)

```
for $i \leftarrow 0$ to $M-1$ do
    for $j \leftarrow 0$ to $M-1$ do
        $N_q[i,j] \leftarrow 0$, $D_q[i,j] \leftarrow 0$
for all $\pi \in \mathcal{C}^2$ do
    if $\mathcal{C}^2[\pi] \neq 0$ and $\mathcal{C}^1[\pi] \neq 0$ then
        $v = \mathcal{C}^2[\pi]$
        for $\ell \leftarrow 0$ to $w-1$ do
            if $v_\ell = 1$ then
                $(i,j) = \mathsf{NN}(\pi, \ell)$
                $N_q[i,j] \leftarrow N_q[i,j] + \mathcal{C}^1[\pi]$
                $D_q[i,j] \leftarrow D_q[i,j] + 1$
for $i \leftarrow 0$ to $M-1$ do
    for $j \leftarrow 0$ to $M-1$ do
        $H[i,j] \leftarrow N_q[i,j]/D_q[i,j]$
return H
```

The first pass of the algorithm is the same as in Section 2.2, i.e., we compute $\mathrm{TPVor}(S)$. In the second pass, we render all cones in $\mathcal{C}$ one by one while keeping the depth buffer in the read-only mode. Next, we read the color buffer back to main memory, and let $\mathcal{C}^2$ denote its content. We process each pixel π as follows: If the ℓ^{th} bit of π is 1, i.e., it has been rendered by a query point with the offset $\mathbf{l} = (\lfloor \ell/B \rfloor, \ell \mod B)$, then we compute in $O(1)$ time the query point $\mathbb{Q}[i,j]$ that rendered π, i.e., the nearest point to π in the set $\mathbb{Q}_\mathbf{l}$. Let $\mathsf{NN}(\pi, \ell)$ denote this procedure. We update the height of $\mathbb{Q}[i,j]$ appropriately. Algorithm 3 gives the pseudo-code of this step.

Expanding the region of influence. The previous description bounds the region of influence r by $s\rho B/2$. However, we can make r be of any size through expanding the size of a query block and then handling subsets of queries in each block in separate passes of BUFFERANALYSIS. We expand the size of a query block by making $B = c\lfloor\sqrt{w}\rfloor$ for $c \geqslant 1$. For simplicity we will use $c = 2^k$ for some power k. Expanding B such that (A2) holds means that for any two query blocks $\mathbb{Q}_\mathbf{a}$ and $\mathbb{Q}_\mathbf{b}$ and an offset $\mathbf{t}$, the query points $(\mathbf{a}, \mathbf{t})$ and $(\mathbf{b}, \mathbf{t})$ are still independent.

However, we run into the issue that $B^2 \geqslant w$, and as a result, we do not have a unique bit in the color for each query in the query block. To handle this we perform multiple passes of BUFFERANALYSIS with a slight modification of NN. If $c = 2$ then we see that we will need $c^2 = 4$ passes. For pass j for $0 \leqslant j < c^2$ we only consider offsets $\mathbf{t} = (t_1, t_2)$ such that $t_1 B + t_2 \bmod c^2 = j$, and as such, we set $\chi(\mathbf{t}, j) = 2^{(t_1 B + t_2 - j)/c^2}$. We can again reverse this procedure in $\mathsf{NN}(\pi, \ell, j)$. A pixel π with the ℓ^{th} bit set to 1 in pass j of c^2 was rendered by a query point with offset

$$\mathbf{l} = (\lfloor (\ell \cdot c^2 + j)/B \rfloor, (\ell \cdot c^2 + j) \bmod B).$$

We can of course again compute in $O(1)$ time the query point $\mathbb{Q}[i, j]$ that rendered π, i.e., the nearest point to π in the set $\mathbb{Q}_\mathbf{l}$. Keeping $N_q[i, j]$ and $D_q[i, j]$ from Algorithm 3 through all query passes, we can at the end calculate and return the heights H for the entire $M \times M$ grid.

This general procedure of expanding B enables us to set a variable region of influence based on the properties of the input data set and desired interpolation. The particular assignment $\chi(\mathbf{t}, j)$ and procedure $\mathsf{NN}(\pi, \ell, j)$ are merely examples of methods of dividing and labeling subsets of each query block. As long as we have a deterministic process $\mathsf{NN}(\pi, \ell, j)$ for any size B, this method will work. Thus, we can use more creative numbering methods to make c any constant $\geqslant 1$ rather than just 2^k.

Handling larger grids. The preceding algorithm assumed that $\mathbb{Q}$ was small enough that the entire $\mathbb{Q}$ could be mapped to Π. This is not always realistic since the value of N is limited by the graphics hardware. For example, $N \leqslant 2^{14} = 16384$ on modern graphics cards such as the NVIDIA GeForce GTX 470. With a scaling parameter $s = 5$, $M \leqslant \lfloor 2^{14}/5 \rfloor = 3276$, implying that we can process $3276^2 \approx 10^7$ grid query points in a single pass. Recall that each pass consists of two rendering phases and the subsequent buffer analysis. For $s\rho = 2$ meter, this corresponds to computing a grid DEM for a region of roughly 70×70 km^2 in area. However, we often want to generate grid DEMs that are considerably larger, in which case we proceed as follows.

Let $\mu = (N - 4r/\rho)/s$, the largest grid of query points we can handle in one pass is

$\mu \times \mu$. Thus, if $M > \mu$ we partition $\mathbb{Q}$ into $\mu \times \mu$ sub-grids; see Figure 2.8(a) and process these sub-grids individually. Let $m = M/\mu$, then m^2 is the number of sub-grids. We define

$$\mathbb{Q}^{i,j} = \bigcup_{(l,k)\in[0,\mu-1]^2} \mathbb{Q}[l + i\mu, k + j\mu]$$

to be the (i, j)'th sub-grid, for $i, j < m$. For sub-grid $\mathbb{Q}^l$ we let $\mathcal{Q}^l = \mathbb{Q}^l + \sigma$. We interpolate each $\mathbb{Q}^l$ on the GPU independently, using the algorithm described above. Thus we need to find the set $S^l = \mathcal{Q}^l \cap S$ of input points relevant for the queries in $\mathbb{Q}^l$. Note that the S^l's are not disjoint; see Figure 2.8(b). Let $\mathcal{M}$ and $\mathcal{B}$ be the amount of points that fit in main memory and in a disk block respectively. For simplicity we assumes that $\mathcal{B}$ divides $\mathcal{M}$.

If $|S| < \mathcal{M}$, then extracting the set S^l is not hard — we can construct a two-dimensional table to store S and extract S^l efficiently for each sub-grid $\mathbb{Q}^l$. This is, however, more challenging and expensive when, as is typically the case, S is too large to fit in the main memory. For example, the Denmark data set we have consists of 25 billion points. In this case, we preprocess S as follows.

We can keep up to $\mathcal{M}/\mathcal{B}$ streams of points by storing a block of $\mathcal{B}$ elements for each stream in memory, and the rest on disk. If $m^2 \leqslant \mathcal{M}/\mathcal{B}$, then we construct a stream for each bin $\mathbb{Q}^l$. We then distribute each point $p \in S$ to the bins for the sub-grids $\{\mathbb{Q}^l | p \in \mathcal{Q}^l\}$. If $m^2 > \mathcal{M}/\mathcal{B}$ we cannot hold enough streams in memory and instead use a recursive procedure. We partition the m^2 sub-grids into $\mathcal{M}/\mathcal{B}$ square partitions of dimension $n = m/\sqrt{\mathcal{M}/\mathcal{B}}$.

$$\mathbb{P}^{i,j} = \bigcup_{(l,m)\in[0,n-1]^2} \mathbb{Q}^{l+in,k+jn},$$

for $0 \leqslant i, j < \sqrt{\mathcal{M}/\mathcal{B}}$ (see Figure 2.8(b)). Since there are $\mathcal{M}/\mathcal{B}$ partitions, we can construct a stream for each of them. We then distribute each point $p \in S$ into the stream representing partition $\mathbb{P}^x$ if $p \in \mathcal{P}^x = \mathbb{P}^x + \sigma$, where $\sigma = [-2r, +2r]^2$. Following this

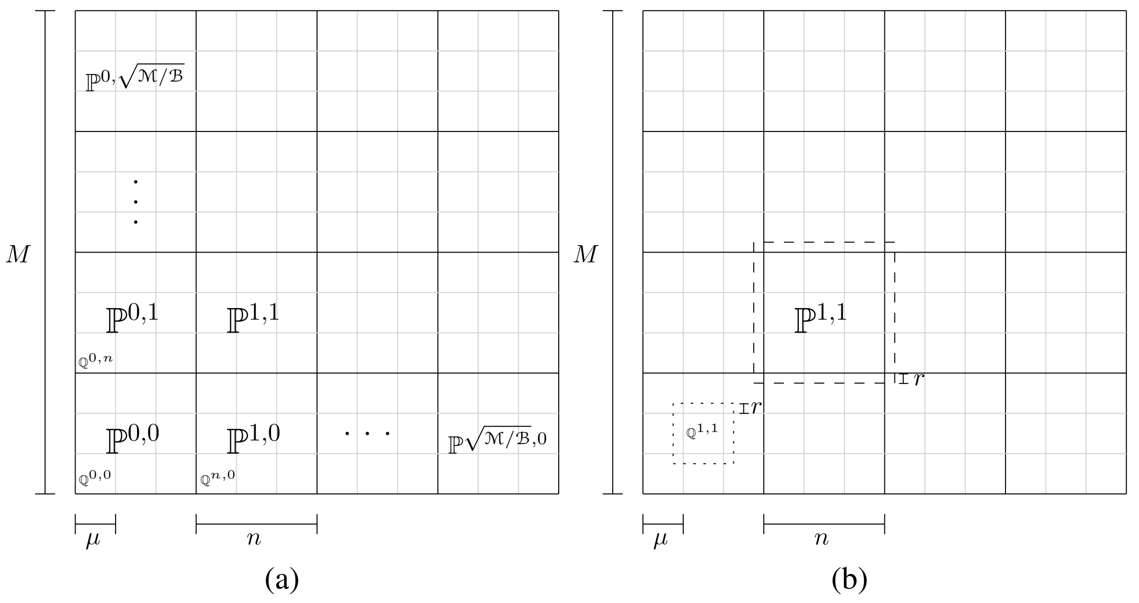


FIGURE 2.8: (a) When $\mathbb{Q}$ is too big to handle in one pass, we split it up into sub-grids of size μ. To filter the points of S to the right place we may need several passes where the grids are split into $\mathcal{M}/\mathcal{B}$ partitions of size n by n. (b) This figure shows $\mathcal{Q}^{1,1} = \mathbb{Q}^{1,1} + \sigma$ (contained in the dotted square) and $\mathcal{P}^{1,1} = \mathbb{P}^{1,1} + \sigma$ (contained in the dashed square).

distribution step we recurse on each partition individually. The depth of the recursion is $O(\log_{\sqrt{\mathcal{M}/\mathcal{B}}} m) = O(\log_{\mathcal{M}/\mathcal{B}} M/\mu)$.

2.4 Implementation and Experimentation

Here we describe implementation details that contribute to the efficiency and quality of our algorithm. We subsequently offer empirical results for tests of the algorithm's speed and quality on real-world terrains.

Platform. We ran our experiments on an Intel Core2 Duo CPU E6850 at 3.00GHz with 4GB of internal memory. We used Ubuntu 10.4 and two 1TB SATA disk drives in a RAID0 configuration. Additionally, the machine contained a NVIDIA GeForce GTX 470 graphics card running CUDA 3.0. This card has 1.2 gigabytes of memory, 448 CUDA cores, and 14 multiprocessors.

The algorithm was implemented in C++ using OpenGL to interact with the graphics

	Afghanistan	DKPART	Fort Leonard Wood
Size of input (10^6)	186	1038	2180
Size of output (10^6)	9.5	213	151
NNI with CUDA	163	1238	2190
Binning Time	67	558	1030
Interpolation Time	96	680	1160
NNI without CUDA	1252	14323	11164
Binning Time	91	569	1036
Interpolation Time	1161	13754	10128
Linear	962	7377	20307
RST	5698	66729	122305

Table 2.1: Time comparison of competing interpolation algorithms (times in seconds)

card. The $\mathbb{C}^1$, $\mathbb{C}^2$, and $\mathbb{D}$ buffers were implemented using OpenGL's frame-buffer and render-buffer objects. Additionally, we used a display list to render the cones $C^\diamondsuit$. As described, our algorithm uses the same radius of influence r for all $C^\diamondsuit$, however, for flexibility our implementation supports using one radius for input points r_s and another one for queries r_q. Adjusting these values separately allows us to optimize r_q based on hardware bounds imposed by w, while adjusting r_s based on properties of our data sets.

2.4.1 Reducing communication complexity

The computational efficiency is one of two important factors in the real-world performance of our algorithm. We have taken great care to minimize the other major component, communication cost, as well. As mentioned in the introduction, the cost of transferring buffers between GPU and main memory is substantial, but the cost of transferring data between the hard drive and the main memory is also substantial, especially for large data sets that do not fit in main memory.

GPU to CPU communication. As described in Section 2.2, the colors buffers $\mathbb{C}^1$, $\mathbb{C}^2$ are read back into memory resulting in $\mathcal{C}^1$ and $\mathcal{C}^2$. They are then used by the BUFFERANALYSIS algoritm, the final step in the interpolation. However, each of these buffers contains an $s \times s$ square of pixels for each query point of $\mathbb{Q}$, which means that each buffer is a factor of s^2 larger than $\mathbb{Q}$. Thus, we are transferring far more data between GPU and CPU memory than would be required if we could just transfer the final interpolated values (i.e. the buffer H from BUFFERANALYSIS).

Therefore, we have used CUDA to implement BUFFERANALYSIS directly on the graphics card. As discussed previously, CUDA can directly access the color buffers from GPU memory and likewise can keep two dimensional arrays of its own for N_q, D_q, and H. In performing this final summation step in CUDA, each pixel in the color buffer is accessed in parallel, and for each bit that is set to one, the appropriate values in N_q and D_q are incremented. Because this may cause the same memory location to be written to by multiple threads simultaneously, we use a CUDA function for atomic addition, providing serialization and synchronization of the many threads. Thus, we only perform one read from GPU memory to main memory, with only one 32-bit word being transferred for each query point. This has a drastic effect on running times of our algorithm, which will be demonstrated in our experiments.

Reducing disk-transfers. As described in Section 2.3 we preprocess S by binning the data into sub-grids of size $\mu \times \mu$, before feeding them to the GPU one by one. We have used the efficient disk-based stream abstractions provided by the Templated Portable I/O Environment (TPIE) [5] library to implement the recursive algorithm that performs this binning.

Additionally, the final output from the interpolation routine is a stream of points $(i, j, h(i, j))$ which typically need to be converted into a row-major raster grid sorted on i and j. For this we use the external-memory sorting algorithm [4] from TPIE, which is asymptotically optimal with respect to disk-memory transfers. The time taken by the final step is not included in the results presented in this section since this step is the same for all of the algorithms presented and it is not the bottleneck in the running time.

2.4.2 Performance

In analyzing our algorithm, we compared our results against results from applying a regularized spline with tension (RST) and linear interpolation based on the global Delaunay triangulation of the data. The RST implementation is from Danner *et al.*'s previous work on TerraSTREAM[10]. We also tested our NNI implementation against the linear interpolation, in which we compute the Delaunay triangulation of the entire data set, as presented in [2], followed by interpolating the value of each query point. Both of these algorithms are sequential.

Data sets. For our tests, we ran our different interpolation schemes on three main data sets. The first was LiDAR data that covers most of Denmark supplied by COWI A/S. The entire point cloud is 1.5 terabytes in size with 26 billion data points. For a number of our experiments we used a portion of this data set, which we will refer to as DKPART. DKPART contains 1 billion data points over a 10 kilometer by 90 kilometer region and is 27 gigabytes on disk. This gives a point density of 0.9 points per square meter on average.

Because Denmark is relatively flat, we also ran tests for both speed and quality on a point cloud of a mountainous region in the Paktika province of Afghanistan (data courtesy of ARO). This data set is 3.5 gigabytes on disk and contains 186 million data points over an approximately $4000\ \mathrm{m}^2$ region. This is approximately 6.5 points/m^2 on average. Because the Afghanistan data set comes from a mountainous region, the data is useful for comparing how different algorithms handle steep slopes and ridges.

The third dataset covers an approximately 600km^2 region around Fort Leonard Wood in Missouri with a dense point cloud consisting of about 2.2 billion points (data courtesy of ARO) and takes up 57 gigabytes on disk. That is about 3.6 points /m^2 on average.

The Afghanistan point cloud has not been substantially filtered and contains many non-ground points (such as points on vegetation). For the quality tests we used a subset of the points without many of the non-ground points[2]. This data set, which we refer to as Afghanistan-1, has a point density of 0.26 points per square meter. To test the significance of the density of the points, we removed a portion of the points, keeping only one out of sixteen data points at random. This produces a point density of 0.016 points/m^2. We denote this data set as Afghanistan-2.

Parameter choices. Within our algorithm there are numerous parameters that can be adjusted to shift speed and quality trade-offs. The algorithm's precision can be modified through its scaling parameter s, the number of faces k of the $C^{\diamondsuit}$'s, and the cone radii r_s and r_q. For our tests we set $k = 6$ and $r_s = 20$ meters. We set r_q based on ρ and w, such that $r_q = s\rho B/2$. In our tests the word size w of the color buffers was set to 32 bits (though this can easily be increased to 128 bits on modern graphics cards), and thus $B = 5$. For the scaling parameter, we tested the algorithm with varying values, but in experiments below we used $s = 5$. These parameters were chosen to offer a sufficiently high quality of output without unnecessarily slowing the implementation.

Efficiency. We ran the various interpolation algorithms on our Afghanistan, Fort Leonard Wood and DKPART data sets. For these general tests, we used a grid resolution of 2 meters. As shown in Table 2.1, our NNI implementations run significantly faster than the linear interpolation and the RST based interpolation. The NNI algorithm takes only 17% of the time of the linear interpolation for both the Afghanistan and DKPART data sets, and only 11% of the time on the Fort Leonard Wood data set. In comparison with the RST algorithm, the NNI implementation takes only 2.8% of the time on the Afghanistan data set and approximately 1.8% of the time on both the DKPART and Fort Leonard Wood data sets.

Worth noting is the breakdown of the time spent binning the data and the time spent performing the interpolation. On all three data sets when running the CUDA-based implementation, the binning takes a little under half of the running time of the algorithm. While

[2] This was done by only using the "last return"-points for each pulse.

necessary in some cases, many data sets are already stored in tiles such that binning could be skipped. In contrast, the RST method spends most of the time on interpolation, while the linear-interpolation algorithm spends most of the time on constructing the TIN.

The time results also show the significant advantage provided by CUDA, with the CUDA based implementation taking 10%-20% of the time of the non-CUDA based implementation.

It it worthwhile to compare the bottlenecks in the two different NNI implementations. The experiments were again done on the Afghanistan data set. As shown in Table 2.2, without using CUDA the bottleneck in the implementation is reading $\mathbb{C}$ to main memory. At both grid resolutions, reading $\mathbb{C}$ takes approximately 70% of the total running time of the interpolation. And, since a smaller grid resolution requires more tiles, there are more read backs on the 0.8 meter resolution and thus the time of the algorithm increases significantly.

With our CUDA implementation, we remove the high cost of GPU memory reads since we are reading back a fraction of the data, but the BUFFERANALYSIS step is highly dependent on the grid resolution. With a 2 meter resolution, the bottleneck is clearly merely drawing the cones for the Voronoi diagrams, and BUFFERANALYSIS takes negligible time. With a higher grid resolution of 0.8 meters, the expensive atomic add operation is performed many more times in the BUFFERANALYSIS step and thus it takes far more time. However, even with the bottleneck shifting to the CUDA step, the algorithm is still much faster with CUDA than without it.

Quality of output. It is, of course, necessary to compare not just the speed but also the quality of these different interpolation schemes. To do this we compared the smoothness of the contour maps from the NNI and linear interpolation over the Afghanistan data set. The entire Afghanistan data set is very dense, with 6.5 points per square meter. Even at a 0.8 meter grid resolution, very few differences in contour maps were seen between the two interpolation methods. This was also obstructed by the many trees in the region which

	Without CUDA		With CUDA	
Grid Resolution (m.)	0.8	2	0.8	2
GPUVORONOI (S)	411	73	76	74
Read $\mathcal{C}^1$	814	116	N/A	N/A
Draw Query Cones	51	5.84	39	6.96
Read $\mathcal{C}^2$	875	135	N/A	N/A
BUFFERANALYSIS	102	9.57	183	0.46
Write points	4.01	0.92	4.2	0.8
Total running time	2289	371	337	105

Table 2.2: CUDA's effect on NNI algorithm timing, running on the Afghanistan data set (times in seconds)

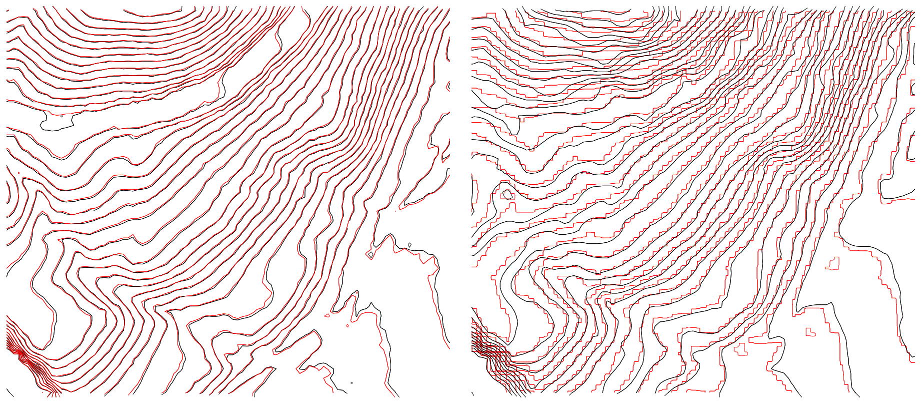


FIGURE 2.9: Contour map comparison. In both figures, the grid used for the red (resp. black) contours were generated from the Afghanistan data using linear (resp. natural neighbor) interpolation.

cause frequent changes in elevation in the grid. Therefore, to perform these quality tests we used the Afghanistan-1 and Afghanistan-2 data sets.

We ran both the linear interpolation and NNI on the data, with a scaling parameter of 5 and a grid resolution of 2 meters. Finally, we use the GRASS GIS system [16] to compute the contour lines from the interpolated data with a 1 meter increment between contour levels.

We present in Figure 2.9 sample images from each pair of contour maps. Comparing the maps, it can be clearly seen that at certain points, especially around curves, the linear interpolation from the triangulation produces jagged results while the NNI maintains its smoothness. It is clear that as the input data becomes sparser in Afghanistan-2, the linear interpolation becomes increasingly jagged, while the NNI output remains smooth.

3
3D Grid Construction

3.1 Introduction

As mentioned in the introduction, the new access to high resolution data sets provides interesting insight into terrains. We now look at how we would handle spatial-temporal data, such as if we had LiDAR scans of the same region over multiple years. These scans provide a point cloud S in $\mathbb{R}^3$ with time as the third dimension. When discussing a point $p \in S$ we will refer to x_p, y_p, and t_p as the points position in xyt-space. As described previously, for the case of terrain data, the point cloud has an associated elevation function $h : S \to \mathbb{R}$. In order to make use of a 3D point cloud we revisit the ideas of a pixelized Voronoi diagram and natural neighbor interpolation, but now in higher dimensions. We subsequently discuss memory and time trade-offs for interpolating over a 3D grid, and finally compare results from running different versions of our interpolation algorithm. Although our algorithm is described in terms of spatial-temporal terrain data, it can be applied to any 3D point cloud, and could be easily extended to higher dimensions.

3.2 Discretized Voronoi Diagram

We first redefine the Voronoi diagram in $\mathbb{R}^3$. Let $S = \{p_1, \dots p_n\}$ be a set of n points in $\mathbb{R}^3$. For each point $p \in S$, its *Voronoi cell*, denoted by $\mathrm{Vor}_S(p)$, is defined as

$$\mathrm{Vor}_S(p) = \{x \in \mathbb{R}^3 \mid \|xp\| \leqslant \|xq\| \forall q \in S\},$$

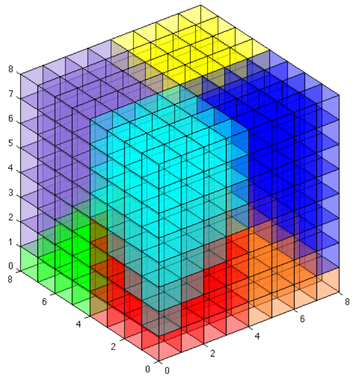
where $\|\cdot\|$ is the Euclidean distance, i.e., a point $x \in \mathrm{Vor}_S(p)$ if p is the point in S closest to x. The *Voronoi diagram* of S, $\mathrm{Vor}(S)$, is the subdivision of space induced by the Voronoi cells of points in S.

Hoff *et al.*[17] have described a GPU based algorithm for computing the Voronoi diagram of a set of points. Since we use a slightly different algorithm, we describe the algorithm for the sake of clarity and completeness. A cube $\mathcal{K}$ consisting of $N \times N \times N$ voxels can be regarded as the cube $[0, N-1]^3$ in $\mathbb{R}^3$. Any cube $R \subseteq \mathbb{R}^3$ can be mapped to $\mathcal{K}$ using an affine transformation. Given the set S and a cube R, we are interested in computing a *discretized Voronoi diagram* of S within R, which we define below. We assume that R is mapped to the image cube $\mathcal{K}$. Each voxel $\pi \in \mathcal{K}$ corresponds to a (tiny) cube $R_\pi \subset \mathbb{R}^3$ where $\mathrm{Volume}(R_\pi) = \rho^3 = \mathrm{Volume}(R)/N^3$. We refer to ρ as the *resolution* of $\mathcal{K}$. All of R_π is congruent to a sampling of $\mathbb{R}^3$ at the center of the cube. Although it is a slight deviation from the typical notion, we will consider the voxel π to be precisely the center of the cube R_π. For example, for $\pi \in \mathcal{K}$ let $\varphi(\pi, S)$ be the point in S whose Voronoi cell contains voxel π. If multiple Voronoi cells intersect R_π, the entire volume of R_π is described based on the sampling from π. For a point $p \in S$, we define the *discretized Voronoi cell* of p to be

$$\mathrm{DVor}_S(p) = \{\pi \mid \varphi(\pi, S) = p\},$$

i.e., the set of voxels that lie in $\mathrm{Vor}_S(p)$; see Figure 3.1(a). The quantity $\rho^3|\,\mathrm{DVor}_S(p)|$ approximates the volume of $\mathrm{Vor}_S(p)$ within R. The approximation error depends on ρ. For a fixed R, the error decreases as we increase N, namely,

$$\lim_{N\to\infty} \rho^3|\,\mathrm{DVor}_S(p)| = \mathrm{Volume}(\mathrm{Vor}_S(p)).$$


(a)

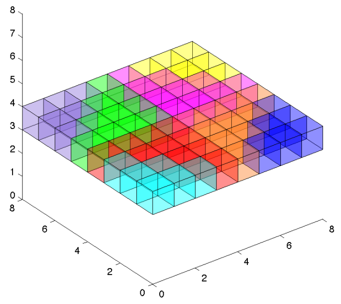

(b)

FIGURE 3.1: (a) A discretized Voronoi diagram for eight points. (b) The pixelized Voronoi diagram for $t = 3$, $\mathrm{PVor}(S, 3)$, shown as a slice of $\mathcal{K}$ from (a).

Similar to our approach for 2D Voronoi diagrams, the problem of computing $\mathrm{DVor}(S)$ can be formulated as that of rendering a scene. For a point $p_i \in S$, let $f_i : \mathbb{R}^3 \to \mathbb{R}$ be defined as $f_i(\pi) = \|\pi p_i\| = \sqrt{(x_\pi - x_{p_i})^2 + (y_\pi - y_{p_i})^2 + (t_\pi - t_{p_i})^2}$ where $\pi \in \mathbb{R}^3$. The *lower envelope* f of $\{f_1, \ldots, f_n\}$ is defined to be

$$f(\pi) = \min_{1 \leqslant i \leqslant n} f_i(\pi)\,,$$

which is the distance from q to its nearest neighbor. $\mathrm{Vor}(S)$ is the projection of the graph of f on the xyt-hyperplane.

This method requires drawing a four dimensional object, which is unfortunately not possible on the GPU. To formulate the problem as that of drawing a 3D scene, we must reduce the dimensionality by one. To do this we only examine a plane of the cube $\mathcal{K}$ with the time set constant to τ. We will refer to the plane $t = \tau$ in $\mathbb{R}^3$ as Π_τ. For a point $p \in S$ we define the *pixelized Voronoi cell* of p for time τ to be

$$\mathrm{PVor}_S(p, \tau) = \{\pi | \phi(\pi, S) = p \wedge t_\pi = \tau\}$$

i.e., the set of pixels that lie in $\mathrm{Vor}_S(p) \cap \Pi_\tau$. We refer to the *pixelized Voronoi diagram* for time τ as $\mathrm{PVor}(S, \tau)$. See Figure 3.1(b).

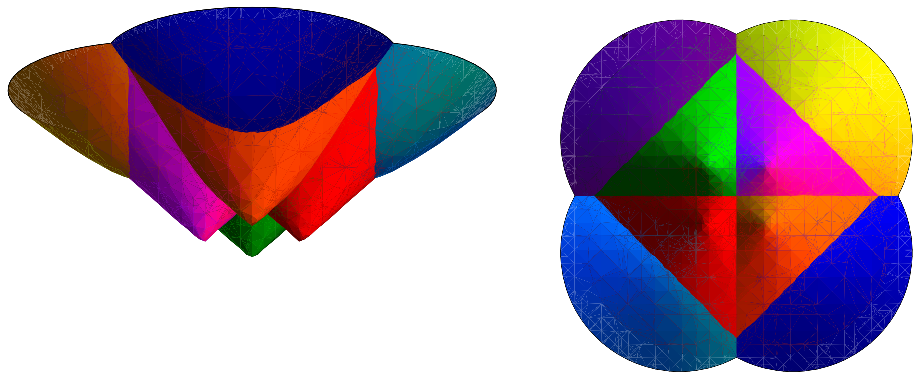


FIGURE 3.2: Voronoi diagram as the lower envelope of a set of hyperboloids. (a) shows the hyperboloids from a side view. (b) looks at the hyerboloids from below revealing the Voronoi diagram. The outer cells of a Voronoi diagram are infinite, but in this figure their sizes are limited because the hyperboloids are of a limited height.

The problem of generating $\mathrm{PVor}(S, \tau)$ from a 3D scene is now considerably easier. Setting $t = \tau$, we find that the distance function is

$$f_{i,\tau}(\pi) = \sqrt{(x_\pi - x_{p_i})^2 + (y_\pi - y_{p_i})^2 + (\tau - t_{p_i})^2},$$

which is one sheet of a hyperboloid of revolution of two sheets. Thus, $\mathrm{PVor}(S, \tau)$ is the projection of the graph of $f(x, \tau) = \min_i f_{i,\tau}(x)$ onto the xy-plane. We define $H(t_o) : z = \sqrt{x^2 + y^2 + t_o^2}$ in xyz-space. Therefore, the graph of each $f_{i,\tau}$ is hyperboloid $H_{i,\tau} = H(\tau - t_{p_i}) + P_\tau(p_i)$ where $P_\tau(p_i)$ is the projection of p_i onto Π_τ. Let $\mathcal{H}_\tau = \{H_{1,\tau}, \ldots, H_{n,\tau}\}$. A point x is in $\mathrm{Vor}_S(p_i) \cap \Pi_\tau$ if $f(x, \tau)$ is realized by the function $f_{i,\tau}$ at x, i.e., the line oriented in the $+z$ direction hits $H_{i,\tau}$ first. In other words, $\varphi(\pi, S) = p_i$ for $\pi \in \Pi_\tau$ if $H_{i,\tau}$ is the cone seen at pixel π when the set $\mathcal{H}_\tau$ is viewed from $z = -\infty$. If we set the color of $H_{i,\tau}$ to i, then the color buffer $\mathbb{C}_\tau[\pi]$ stores the index of the point and the depth buffer $\mathbb{D}_\tau[\pi] = \|\pi\varphi(\pi, S)\|$, where both $\mathbb{C}_\tau$ and $\mathbb{D}_\tau$ represent the image plane Π_τ. (We view each cell of the color buffer and depth buffer as a single word, concatenation of R, G, B, A components.) With this method we can generate $\mathrm{PVor}(S, \tau)$ using GPU rendering; see Figure 3.2. In order to calculate $\mathrm{DVor}(S)$ we need

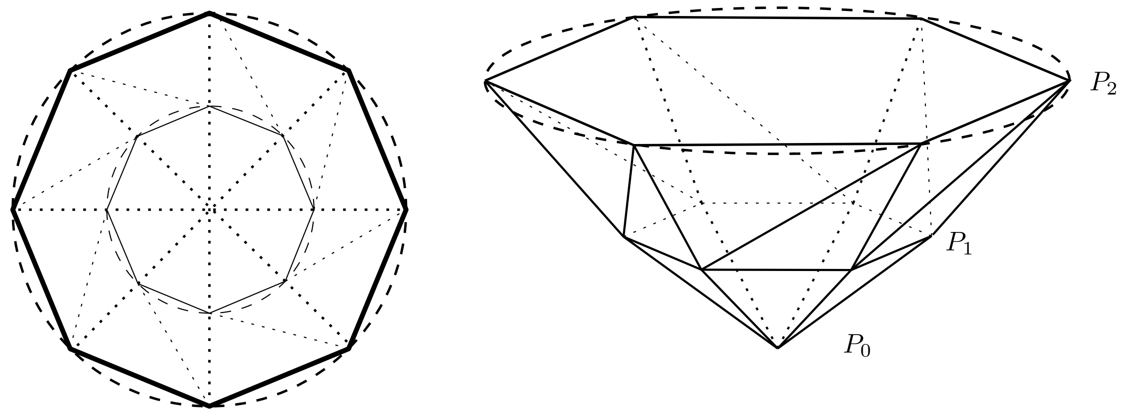


FIGURE 3.3: The triangulation of hyperboloid into two triangle strips.

only calculate $\mathrm{PVor}(S, \tau)$ for all τ at ρ increments.

It is not easy to render a hyperboloid using a GPU, so we approximate the shape with triangle strips. We first define planes P_j in xyz-space where P_0 is the plane through the vertex at $z = t_o$ and $P_j = P_0 + h \cdot j$ where h is some increment in z. For each plane P_j we take the intersection $H_{i,\tau} \cap P_j$ and approximate the circular intersection by a regular k-gon. We can then create triangles between adjacent k-gons using non-intersecting $2k$ triangles. The resulting polygonal hyperboloid $H_\tau^\Diamond$ is composed of $2kj_{\max}$ triangles, as seen in Figure 3.3. We replace $H_{i,\tau}$ by $H_{i,\tau}^\Diamond = H_\tau^\Diamond + P_\tau(p_i)$. The error in tessellation induced by this approximation can be controlled by choosing the value of h and k appropriately.

Finally, we note that we want to limit the *region of influence* for the points. In 3D we define the region of influence to be cylindrical with *radius of influence* r such that each point in S can only influence pixels within a radius r in the xy-plane and a distance r in time. We limit the region of influence through the truncated Voronoi diagram. A *truncated* pixelized Voronoi cell is defined as

$$\mathrm{TPVor}_S(p, \tau) = \{\pi \mid \varphi(\pi) = p \wedge \|P_{t_\pi}(p)\pi\| < r \wedge |t_\pi - t_p| \leqslant r\}.$$

Thus, a pixel π that is not within the cylindrical region of influence of all points of S does not belong to the Voronoi cell of any point. Let C_r denote the cylinder of radius r and height $2r$ centered at origin. We can assume that $S \subset R + C_r$, as no point outside this region will contain any pixel of $\mathcal{K}$ in its Voronoi cell. This truncation is realized by limiting the range of times in which points are considered as well as the height of the hyperboloids $H^{\diamondsuit}_{i,\tau}$. When rendering $\mathrm{TPVor}(S,\tau)$ we only consider the subset $S(\tau) = \{p \in S||t_p - \tau| < r\}$. With a slight abuse of notation we use $H^{\diamondsuit}_{i,\tau}$ to denote the truncated hyperboloid as well. For each $p_i \in S$, we set the color of each triangle of $H^{\diamondsuit}_{i,\tau}$ to i and pass them to the graphics pipeline with $z = -\infty$ as the viewpoint. $\mathbb{C}$ and $\mathbb{D}$ together contain $\mathrm{TPVor}(S,\tau)$. We refer to this algorithm as GPUVORONOI (S,τ). As mentioned above, there might be pixels that are not touched by GPUVORONOI (S,τ). We assume that $\mathbb{C}$ is initialized with a value that allows us to distinguish these pixels from the pixels that are part of the truncated diagram, e.g., we set their color to 0. The collection of of buffers for $\mathrm{TPVor}(S,\tau)$ for all τ determines $\mathrm{TPVor}(S)$.

3.3 Natural Neighbor Interpolation

In this section we first formally define natural neighbor interpolation (NNI), then describe a GPU algorithm for answering NNI queries, which is an extension of the algorithm presented in Section 2.2. A height function $h : S \to \mathbb{R}$ can be extended to the entire $\mathbb{R}^3$ using natural neighbor interpolation. In particular, for a point $q \in \mathbb{R}^3$,

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

where $w_p(q)$ is the fractional volume of $\mathrm{Vor}_{S\cup\{q\}}(q)$ that belongs to $\mathrm{Vor}_S(p)$ (Figure 2.4), i.e.,

$$w_p(q) = \frac{\mathrm{Volume}(\mathrm{Vor}_S(p) \cap \mathrm{Vor}_{S\cup\{q\}}(q))}{\mathrm{Volume}(\mathrm{Vor}_{S\cup\{q\}}(q))}.$$

Since we use truncated pixelized Voronoi diagrams, we redefine the height function as

$$h(q) = \sum_{p \in S} \overline{w}_p(q) \cdot h(p)$$

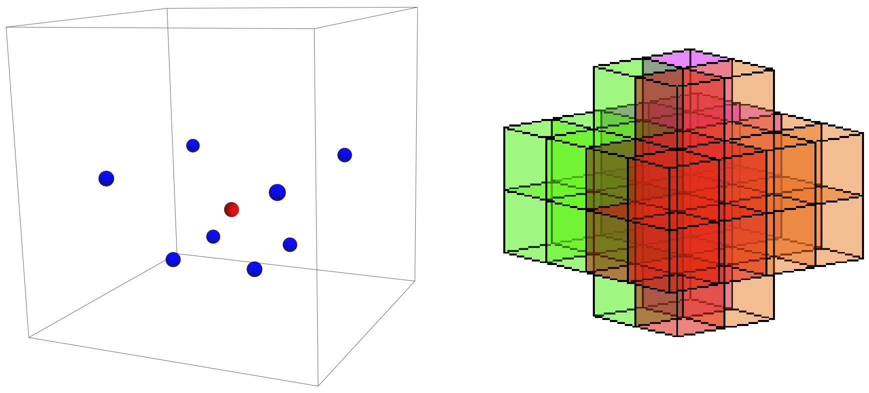

(a) (b)

FIGURE 3.4: (a) A query point in the middle of a point cloud of eight points in $\mathbb{R}^3$. (b) The voxels of the query's Voronoi cell colored based on the cells they stole volume from.

where

$$\overline{w}_p(q) = \frac{|\,\mathrm{TPVor}_S(p) \cap \mathrm{TPVor}_{S\cup\{q\}}(q)|}{|\,\mathrm{TPVor}_{S\cup\{q\}}(q)|}. \tag{3.1}$$

See Figure 3.4.

To calculate $\mathrm{TPVor}(S)$ and $\mathrm{TPVor}(S \cup \{q\})$, we must discretize the space into time-slices and handle each separately as previously explained. We rearrange equation 3.1 as summations over each Π_τ that contributes to the Voronoi cell of q:

$$\begin{aligned} h(q) &= \frac{\sum_{p\in S} |\,\mathrm{TPVor}_S(p) \cap \mathrm{TPVor}_{S\cup\{q\}}(q)| \cdot h(p)}{|\,\mathrm{TPVor}_{S\cup\{q\}}(q)|} \\ &= \frac{\displaystyle\sum_{\tau=t_q-r}^{t_q+r} \sum_{\pi\in\mathrm{TPVor}_{S\cup\{q\}}(q,\tau)} h(\phi(\pi, S))}{\displaystyle\sum_{\tau=t_q-r}^{t_q+r} |\,\mathrm{TPVor}_{S\cup\{q\}}(q,\tau)|} \end{aligned}$$

In the equation above we see that the denominator, which we will refer to as $D(q)$, is the simply the number of voxels in the truncated pixelized Voronoi cell of q. The numerator, referred to as $N(q)$, is a sum of the heights represented by each voxel in $\mathrm{TPVor}(S)$ from the Voronoi cell of q. We note that we only sum over τ in the range $t_q - r$ to $t_q + r$. This is because this range is the query point's region of influence in time, and as such no voxel π for which $|t_\pi - t_q| > r$ could be in the Voronoi cell of q. We increment τ by ρ as this is the separation between voxels adjacent in time in $\mathcal{K}$.

Answering an NNI query. The algorithm for computing $N(q)$ and $D(q)$ requires iterating over each value for τ and in each iteration performing two phases of GPU rendering and analysis. Because we iterate τ by ρ, the algorithm requires $\Delta = 2\frac{r}{\rho} + 1$ iterations. We describe here the two phases for a given iteration τ. The first phase calls GPUVORONOI (S,τ) with the following twist: the color of each triangle of the hyperboloid $H_i^\Diamond$ is set to $h(p_i)$ (instead of i). After the first phase $\mathbb{C}_\tau[\pi]$ stores $h(p_i)$ for all pixels $\pi \in \text{TPVor}_S(p_i, \tau)$. We read back the color buffer; let $\mathcal{C}_\tau^1$ denote the resulting two-dimensional array. We then clear the color buffer. The depth buffer $\mathbb{D}_\tau$ is is not touched, i.e., $\mathbb{D}_\tau[\pi]$ continues to store $\|\pi\varphi(\pi, S)\|$, the distance from the center of π to $\varphi(\pi, S)$.

In the second phase, we set $\mathbb{D}_\tau$ to read-only mode so that it is not overwritten and draw a polygonal hyperboloid $qC_\tau^\Diamond = H_\tau^\Diamond + q$ with q as the apex. Adding $qC_\tau^\Diamond$ is the same as computing $\text{TPVor}_{S \cup \{q\}}(q, \tau)$. However, the color buffer was cleared before the second phase and thus has non-zero entries[1] (corresponding to the color of $qC_\tau^\Diamond$) only for $\text{TPVor}_{S \cup \{q\}}(q, \tau)$. Let $\mathbb{C}_\tau^2$ denote the color buffer contents after the second phase, and $\mathcal{C}_\tau^2$ the array resulting from reading back $\mathbb{C}_\tau^2$ into memory. We build the value for $N(q)$ and $D(q)$ over all iterations. The value of $N(q)$ is computed by adding the values of $\mathcal{C}_\tau^1[\pi]$ for all π for which $\mathcal{C}_\tau^2[\pi] \neq 0$ and the value for $D_\tau(q)$ can be found by summing the number of non-zero values in $\mathcal{C}_\tau^2$. We refer to this step of the algorithm as BUFFERANALYSIS $_\tau$.

Once this sequence of GPUVORONOI (S,τ) and BUFFERANALYSIS $_\tau$ have been completed for all necessary values of τ we can perform the final calculation for $h(q)$ by merely dividing $N(q)$ by $D(q)$.

While this algorithm discusses the process of performing a natural neighbor interpolation query for a single point, we can use the same optimizations outlined in Section 2.2 to

[1] We assume without loss of generality that all points of S have a positive height.

perform an $N \times N$ grid of queries at time τ simultaneously.

3.4 NNI on Cubes

We now consider the problem of performing N^3 natural neighbor interpolation queries on a $N \times N \times N$ grid. Given we can compute an $N \times N$ grid of queries for some time τ, we would now like to do this for all N times. We outline below three different algorithms for doing this. Each discusses the trade-off of time, which is spent through the rendering of hyperboloids, against memory. We describe algorithms below based on the number of times each point $p \in S$ is rendered on the GPU.

Rendering $O(\Delta^2)$ hyperboloids per point. We first examine the naïve method for performing the N^3 NNI queries. Here, we can perform the algorithm described above to calculate the $N \times N$ grid of queries for each time independently. We will refer to the grid of queries for a given time τ as $\mathbb{Q}_\tau$.

For performing NNI for $\mathbb{Q}_{t_i}$ for any time t_o we must render $\mathrm{TPVor}(S, \tau)$ for $t_i - r \leqslant \tau \leqslant t_i + r$. All points p for which $t_p = t_i$ must be used to render a hyperboloid for each $\mathrm{TPVor}(S, \tau)$ and therefore each point is used Δ times. Additionally, we must render $\mathrm{TPVor}(S, \tau)$ for some τ for all $\mathbb{Q}_t$ for which $|\tau - t| \leqslant r$. Thus, each $\mathrm{TPVor}(S, \tau)$ is also computed Δ times. Because each point is used for Δ different pixelized Voronoi diagrams and each pixelized Voronoi diagram is generated Δ times, each point is used $O(\Delta^2)$ times.

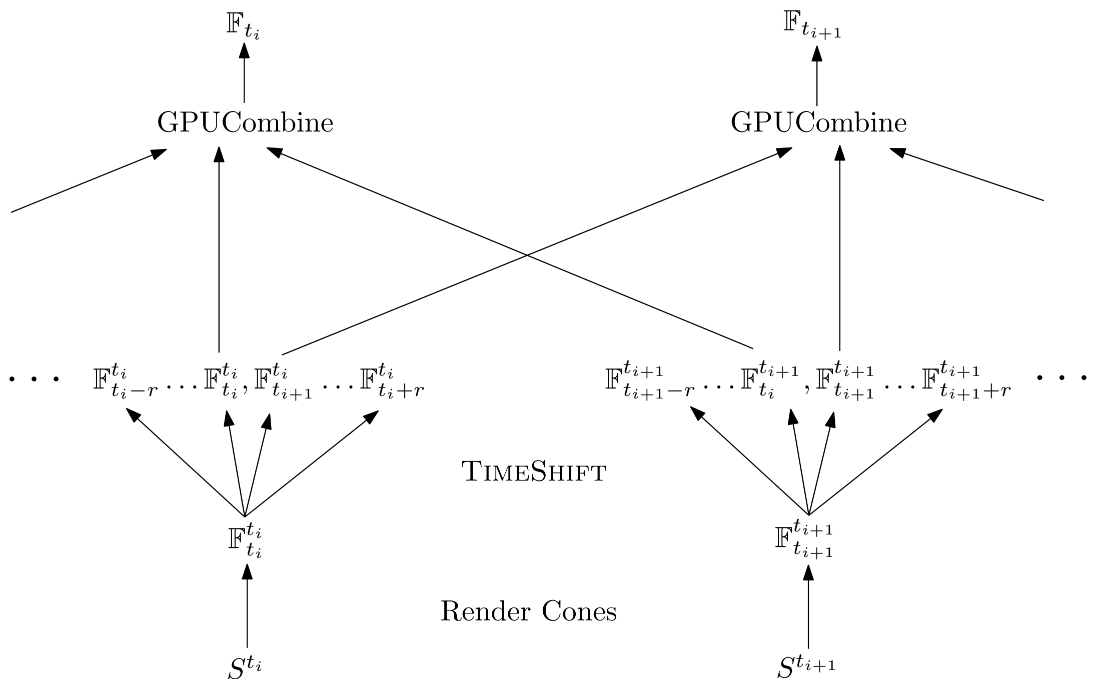
Rendering $O(\Delta)$ hyperboloids per point. As should be apparent from the description above, there is redundancy in the computations performed for different $\mathbb{Q}_t$. Here, we look to avoid repeatedly computing each $\mathrm{TPVor}(S, \tau)$ Δ times. We are able to do this, and thus save computational complexity, with a trade-off of memory space.

To describe the algorithm, we first define additional notation. Let framebuffer $\mathbb{F}_\tau$ refer to the combination of $\mathbb{C}_\tau$ and $\mathbb{D}_\tau$ describing $\mathrm{TPVor}(S, \tau)$. We look to interpolate the queries $\mathbb{Q}_{t_0} \ldots \mathbb{Q}_{t_{N-1}}$. To interpolate $\mathbb{Q}_{t_i}$, we must generate $\mathbb{F}_{t_i - r}, \mathbb{F}_{t_i - r + \rho} \ldots \mathbb{F}_{t_i + r}$ and perform the interpolation on each time-slice. When we proceed to $\mathbb{Q}_{t_{i+1}}$, we must now generate $\mathbb{F}_{t_{i+1}-r}, \mathbb{F}_{t_{i+1}-r+\rho} \ldots \mathbb{F}_{t_{i+1}+r}$. In the simple case that $t_{i+1} - t_i = \rho$, this becomes $\mathbb{F}_{t_i-r+\rho} \ldots \mathbb{F}_{t_i+r}, \mathbb{F}_{t_{i+1}+r}$. From this it is clear that all of the framebuffers except for the last, $\mathbb{F}_{t_{i+1}+r}$, were calcualated for the previous query grid and thus can be reused in the interpolation and BUFFERANALYSIS step. We only need to run GPUVORONOI (S,$t_{i+1} + r$). Therefore, we must store Δ framebuffers: $\Delta - 1$ to save values from the previous query grid and one to generate the new TPVor. Doing this for $\mathbb{Q}_i$ for all $0 \leqslant i < N$ *in order* lets us generate each $\mathbb{F}_{t_i}$ only once. Through this method, each point p is rendered $O(\Delta)$ times: once for each $\mathrm{TPVor}(S, \tau)$ for $t_p - r \leqslant \tau \leqslant t_p + r$.

Rendering $O(1)$ hyperboloids per point. We can further decrease the number of times each point is used by drawing a cone for each point only once and then repeatedly modifying the framebuffer. We can only do this if the data is in discrete time slices; we define S^τ to be the set of data from one time slice $\{p \in S | t_p = \tau\}$. We define $\mathbb{F}^{\tau_1}_{\tau_2}$ to be the framebuffer containing the Voronoi diagram for S^{τ_1} from the perspective of time τ_2, ie. $\mathrm{TPVor}(S^{\tau_1}, \tau_2)$. Again $\mathbb{C}^{\tau_1}_{\tau_2}$ and $\mathbb{D}^{\tau_1}_{\tau_2}$ are the color and depth buffer of $\mathbb{F}^{\tau_1}_{\tau_2}$.

Our goal is to generate $\mathbb{F}_{t_i}$ by only rendering $\mathbb{F}^{\tau}_{\tau}$ for all τ once, and thus rendering a hyperboloid for each point only once. An overview of the algorithm can be seen in Figure 3.5. We note that when we are generating $\mathbb{F}^{\tau}_{\tau}$, we need only draw a cone for each point $p \in S^\tau$ as we did for 2D data. Given that we have $\mathbb{F}^{\tau}_{\tau}$ for all τ we can generate $\mathbb{F}_{t_i}$ through a two step process. First, we need to be able to convert $\mathbb{F}^{\tau}_{\tau}$ to $\mathbb{F}^{\tau}_{t_i}$ for each τ. We call this function TIMESHIFT($\mathbb{F}^{\tau}_{\tau}, \pi, \tau, t_i$). This gives us the Voronoi diagram for each slice of data as viewed from time t_i. We must then combine these framebuffers, specifically $\mathbb{F}^{t_i-r}_{t_i} \ldots \mathbb{F}^{t_i+r}_{t_i}$, to create $\mathbb{F}_{t_i}$, as though we had taken the lower envelope of the hyperboloids drawn on each respective framebuffer. We only consider this subset of framebuffers because only the data in $S^{t_i-r} \ldots S^{t_i+r}$ has a region of influence that reaches t_i. We call this algorithm GPUCOMBINE.

In performing TIMESHIFT($\mathbb{F}^{\tau}_{\tau}, \pi, \tau, t_i$) we look to find the distance $\|\pi' \phi(\pi', S^\tau)\|$ given



$\mathbb{F}_{t_i}$
$\mathbb{F}_{t_{i+1}}$
GPUCombine
GPUCombine
$\cdots$ $\mathbb{F}^{t_i}_{t_i-r} \ldots \mathbb{F}^{t_i}_{t_i}, \mathbb{F}^{t_i}_{t_{i+1}} \ldots \mathbb{F}^{t_i}_{t_i+r}$
$\mathbb{F}^{t_{i+1}}_{t_{i+1}-r} \ldots \mathbb{F}^{t_{i+1}}_{t_i}, \mathbb{F}^{t_{i+1}}_{t_{i+1}} \ldots \mathbb{F}^{t_{i+1}}_{t_{i+1}+r}$ $\cdots$
TimeShift
$\mathbb{F}^{t_i}_{t_i}$
$\mathbb{F}^{t_{i+1}}_{t_{i+1}}$
Render Cones
S^{t_i}
$S^{t_{i+1}}$

FIGURE 3.5: The process of creating $\mathbb{F}_{t_i}$ from sets of data S^{t_i} rendering each point only once.

that $\pi' = (x_\pi, y_\pi, t_i)$. We can calculate $\|\pi'\phi(\pi', S_\tau)\| = \sqrt{(\mathbb{D}^\tau_\tau[\pi])^2 + (\tau - t_i)^2}$ since $\mathbb{D}^\tau_\tau[\pi] = \sqrt{(x_\pi - x_{\phi(\pi,S^\tau)})^2 + (y_\pi - y_{\phi(\pi,S^\tau)})^2}$. This is because the nearest neighbor from S^τ for any given pixel does not change, but the distance to it now includes the distance in time; see Figure 3.6. As such we can perform TIMESHIFT$(\mathbb{F}^\tau_\tau, \pi, \tau, t_i)$ for each pixel $\pi \in \mathbb{F}^\tau_\tau$ for $t_i - r \leqslant \tau \leqslant t_i + r$.

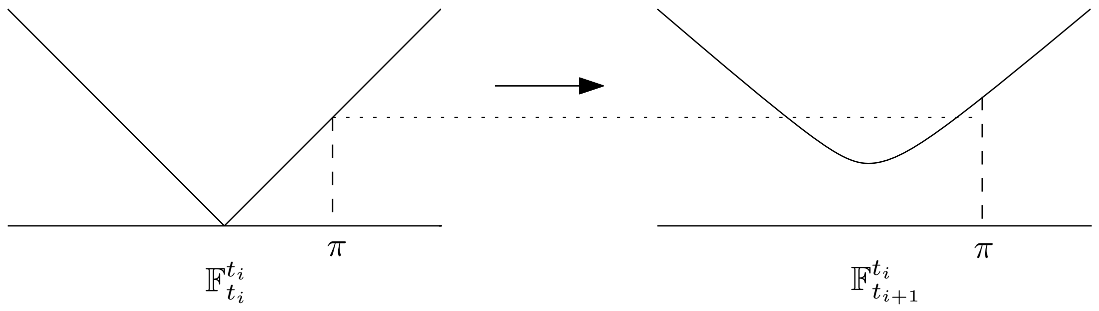


FIGURE 3.6: TIMESHIFT: Modifying the distance in the framebuffer for each pixel to reflect an offset in time.

Algorithm 4 GPU$\textsc{Combine}(\mathfrak{F}_{t_i})$

```
for all $\pi \in \mathbb{F}_{t_i}$ do
  $\mathbb{C}_{t_i}[\pi] \leftarrow 0, \mathbb{D}_{t_i}[\pi] \leftarrow \infty$
for all $\pi \in \mathbb{F}_{t_i}$ do
  for all $\mathbb{F}_\tau^\tau \in \mathfrak{F}_{t_i}$ do
    if $\textsc{TimeShift}(\mathbb{F}_\tau^\tau, \pi, \tau, t_i) < \mathbb{D}_{t_i}[\pi]$ then
      $\mathbb{D}_{t_i}[\pi] \leftarrow \textsc{TimeShift}(\mathbb{F}_\tau^\tau, \pi, \tau, t_i)$
      $\mathbb{C}_{t_i}[\pi] \leftarrow \mathbb{C}_\tau^\tau[\pi]$
return $\mathbb{F}_{t_i}$
```

Given that we can calculate TimeShift for all pixels in $\mathbb{F}_{t_i-r}^{t_i-r} \dots \mathbb{F}_{t_i+r}^{t_i+r}$ we can now perform GPUCombine$(\mathbb{F}_{t_i-r}^{t_i-r} \dots \mathbb{F}_{t_i+r}^{t_i+r})$ with the goal of producing $\mathbb{F}_{t_i}$. In combining this set of framebuffers we would like to compare corresponding pixels in each framebuffer and keep the one that has the closest nearest neighbor as shown in the depth buffer. Mathematically we calculate $\mathbb{F}_{t_i}$ as shown below:

$$\mathbb{D}_{t_i}[\pi] = \min_{t_i - r \leqslant \tau \leqslant t_i + r} \textsc{TimeShift}(\mathbb{F}_\tau^\tau, \pi, \tau, t_i)$$

$$\tau_\pi = \operatorname*{argmin}_{t_i - r \leqslant \tau \leqslant t_i + r} \textsc{TimeShift}(\mathbb{F}_\tau^\tau, \pi, \tau, t_i)$$

$$\mathbb{C}_{t_i}[\pi] = \mathbb{C}_{\tau_\pi}^{\tau_\pi}[\pi]$$

The pseudocode to perform this calculation can be found in Algorithm 4. However, in implementation we would perform almost this exact computation on each pixel in parallel with CUDA.

Now that we have described the process of creating $\mathbb{F}_{t_i}$, we must now step back and look at how this effects the process of doing the entire interpolation. Because we must keep a framebuffer for each time slice of data, the amount of data stored on the GPU at a time increases to now 2Δ. As in the previous $O(\Delta)$ algorithm, we must store Δ buffers $\mathbb{F}_t$. However, we now must store an additional $O(\Delta)$ buffers for $\mathbb{F}_\tau^\tau$.

To interpolate $\mathbb{G}_{t_i}$ we must calculate $\mathbb{F}_{t_i-r} \dots \mathbb{F}_{t_i+r}$. Of course $\mathbb{F}_{t_i-r} \dots \mathbb{F}_{t_i+r-\rho}$ are already being stored on the GPU but we must now generate $\mathbb{F}_{t_i+r}$. To do this we must have $\mathbb{F}_{t_i}^{t_i} \dots \mathbb{F}_{t_i+2r}^{t_i+2r}$. Again, we note that $\mathbb{F}_{t_i}^{t_i} \dots \mathbb{F}_{t_i+2r-\rho}^{t_i+2r-\rho}$ were calculated for previous steps. Thus, we can keep these saved in GPU memory and only generate $\mathbb{F}_{t_i+2r}^{t_i+2r}$ by drawing a cone for each point in S^{t_i+2r}. We can now generate $\mathbb{F}_{t_i+r}$ with GPUCOMBINE. Therefore, each point is only rendered as a cone on the GPU once in its drawing of $\mathbb{F}_{\tau}^{\tau}$ and the GPU must store 2Δ framebuffers.

3.5 Implementation and Experimentation

We now discuss the details of the implementation and testing of the above algorithms. The algorithm was implemented in C++ and uses OpenGL for rendering on the GPU. All experiments were run on the same hardware as was previously described.

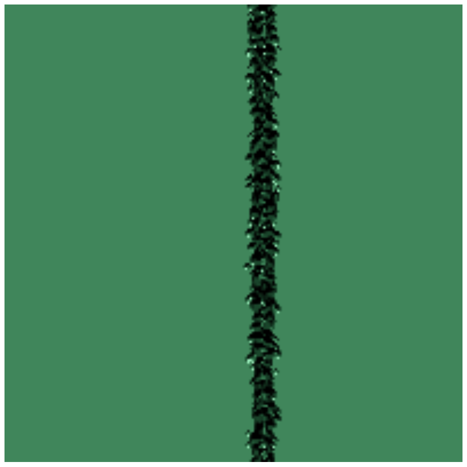
Reducing communication costs. As explained in Section 2.4 and the algorithms described above, we use NVIDIA's CUDA architecture to implement much of the algorithm and thus minimize the transfer of data between GPU and CPU memory. To handle large data we again use the Templated Portable I/O Environment (TPIE) [5] library to implement the recursive algorithm that performs this binning. Additionally, to avoid extra reading of input points during rendering that fall outside the radius in time from our queries, we sort each tile's points in time using TPIE's external-memory sorting algorithm. While this is slightly overkill and could be done more efficiently (saving one disk read and write) with a custom binning algorithm, this process was not the bottleneck in our algorithm and thus was not addressed in the current version of the code.

Data sets. We ran our algorithms on both artificial as well as natural data sets. To compare the efficiency of our algorithms, we ran the implementations on two data sets of random points over a 10 year period. For each year we generated either 10 million or 50 million random data points in a 40 km^2 region. The data set with 10 million points per year took up 2.1 gigabytes on disk and the data set with 50 million points per year took up 11 gigabytes on disk. To test that the algorithm worked as expected, we generated a set of data, random in the xy plane, showing a 20 meter high wall moving across a plane over 110 year period. For each year we generated 20,000 points, totaling to 41 megabytes on disk. Last, we tested our algorithm on data from the coast of North Carolina taken over nine years. Data ranges from 100,000 points per year to nearly 3 million points per year and comes to a total of just under 20 million input points. This data set took up 382 megabytes on disk.

Parameter choices. Within our experiments there are numerous parameters that can be adjusted to shift our algorithm's speed and quality trade-offs. Many of these parameters are the same as the prevoiusly described experiments. For all experiments we set the scaling parameter to $s = 5$, $r_q = s\rho B/2$, $B = 5$, and $r_s = 5$ meters. As before, the speed of the algorithm is highly dependent on the number of triangles rendered. For these tests the number of triangles rendered greatly increased both because of parameters chosen and because we now have to render a hyperboloid and not just a cone. For these tests we set $k = 50$ and rendered 5 triangle strips. Therefore each hyperboloid was composed of 500 triangles. Parameters were chosen to offer sufficiently high quality of output and test the rigor of the implementation.

Quality. In testing on both the moving wall data set as well as the NC coastal data set, we performed the NNI both during years there were data as well as between years of data. For the moving wall data we used a radius of influence in time of 3 years and for the NC coastal data set we used a radius of influence in time of 2 years. The interpolation performed as expected and the implementation gives accurate results. In the case of the wall data set we see it smoothly move across the plane, and for the NC coastal data set we can watch a sand dune shift over time. A snapshot of the results can be seen in Figure 3.7.

Efficiency. We additionally compared the efficiency of the $O(\Delta^2)$ and $O(\Delta)$ algorithms by running them both on the 10 million and 50 million data sets. In both cases we only ran over a coarse 200 by 200 grid at a 1 meter resolution. Interpolation was performed for 8 years with a radius in time of 2 years. The results can be found in Table 3.1. As can be seen in the table, the binning time took up between one thrid and one tenth the total running time of the algorithms. For both data sets we see that the use of the extra GPU memory makes the $O(\Delta)$ algorithm take one quarter of the time of the $O(\Delta^2)$ algo-


(a)

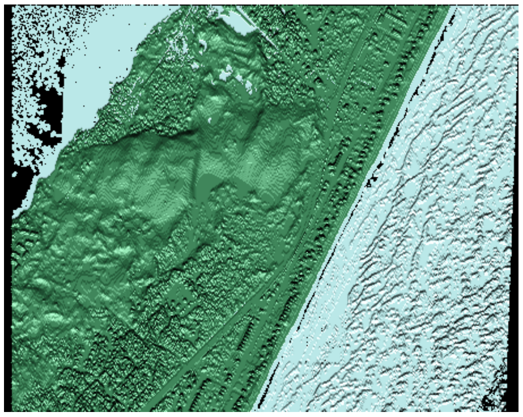

(b)

FIGURE 3.7: Interpolated heights using 3D NNI. (a) Shows the wall moving across the plane and (b) shows a sand dune on the NC coast. Both DEMs were interpolated between slices of data.

	$O(\Delta^2)$	$O(\Delta)$	$O(\Delta^2)$	$O(\Delta)$
Points/Year (10^6)	10		50	
Binning Time	2m 40s	3m 2s	15m 55s	16m 17s
GPUVORONOI (S)	18m 28s	4m 58s	2h 7m	33m 18s
Draw Query Cones	1.128s	1.02s	1.2s	0.98s
BUFFERANALYSIS	8m 36s	2m 11s	8m 23s	2m 21s
Write points	0.71s	0.01s	1.68s	1.63s
Total running time	29m 54s	10m 19s	2h 31m	52m 6s

Table 3.1: Algorithm speedup through saving more buffers in GPU memory

rithm for the GPUVORONOI and BUFFERANALYSIS steps. When summed with the other computational costs we see that the $O(\Delta)$ algorithm takes approximately one third of the time of the $O(\Delta^2)$ algorithm. Therefore, given the available GPU memory, the $O(\Delta)$ is advantageous over the $O(\Delta^2)$ algorithm.

Bibliography

[1] P. K. Agarwal, L. Arge, and A. Danner. From point cloud to grid DEM: A scalable approach. In Andreas Riedl, Wolfgang Kainz, and Gregory Elmes, editors, *Progress in Spatial Data Handling. 12th International Symposium on Spatial Data Handling*, pages 771–788. Springer-Verlag, 2006.

[2] Pankaj K. Agarwal, Lars Arge, and Ke Yi. I/O-efficient construction of constrained Delaunay triangulations. In *Proc. European Symposium on Algorithms*, pages 355–366, 2005.

[3] Pankaj K. Agarwal, Shankar Krishnan, Nabil H. Mustafa, and Suresh Venkatasubramanian. Streaming geometric optimization using graphics hardware. In *In Proc. 11th European Sympos. Algorithms, Lect. Notes Comput. Sci*, pages 544–555. Springer-Verlag, 2003.

[4] L. Arge. External-memory algorithms with applications in geographic information systems. In M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer, editors, *Algorithmic Foundations of GIS*, pages 213–254. Springer-Verlag, LNCS 1340, 1997.

[5] L. Arge, R. Barve, D. Hutchinson, O. Procopiuc, L. Toma, D. E. Vengroff, and R. Wickeremesinghe. *TPIE User Manual and Reference (edition 0.9.01b).* Duke University, 1999. The manual and software distribution are available on the web at `http://www.cs.duke.edu/TPIE/`.

[6] Alex Beutel, Thomas Mølhave, and Pankaj K. Agarwal. Natural neighbor interpolation based grid dem construction using a gpu. In *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '10, pages 172–181, New York, NY, USA, 2010. ACM.

[7] David Blythe. The direct3d 10 system. *ACM Trans. Graph.*, 25(3):724–734, 2006.

[8] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröoder. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 917–924, New York, NY, USA, 2003. ACM.

[9] Luc Buatois, Guillaume Caumon, and Bruno Levy. Concurrent number cruncher: a gpu implementation of a general sparse linear solver. *Int. J. Parallel Emerg. Distrib. Syst.*, 24(3):205–223, 2009.

[10] Andrew Danner, Thomas Mølhave, Ke Yi, Pankaj K. Agarwal, Lars Arge, and Helena Mitasova. TerraStream: from elevation data to watershed hierarchies. In *GIS '07: Proceedings of the 15th annual ACM international symposium on Advances in geographic information systems*, pages 1–8, New York, NY, USA, 2007. ACM.

[11] Herbert Edelsbrunner and Ernst P. Mücke. Three-dimensional alpha shapes. *ACM Trans. Graph.*, 13(1):43–72, 1994.

[12] Quanfu Fan, Alon Efrat, Vladlen Koltun, Shankar Krishnan, and Suresh Venkatasubramanian. Hardwareassisted natural neighbor interpolation. In *In: Proc. 7th Workshop on Algorithm Engineering and Experiments (ALENEX*, 2005.

[13] Tom G. Farr, Paul A. Rosen, Edward Caro, Robert Crippen, Riley Duren, Scott Hensley, Michael Kobrick, Mimi Paller, Ernesto Rodriguez, Ladislav Roth, David Seal, Scott Shaffer, Joanne Shimada, Jeffrey Umland, Marian Werner, Michael Oskin, Douglas Burbank, and Douglas Alsdorf. The shuttle radar topography mission. *Rev. Geophys.*, 45, 5 2007.

[14] Naga K. Govindaraju, Ming C. Lin, and Dinesh Manocha. Fast and reliable collision detection using graphics processors. In *SCG '05: Proceedings of the twenty-first annual symposium on Computational geometry*, pages 384–385, New York, NY, USA, 2005. ACM.

[15] Naga K. Govindaraju, Nikunj Raghuvanshi, and Dinesh Manocha. Fast and approximate stream mining of quantiles and frequencies using graphics processors. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 611–622, New York, NY, USA, 2005. ACM.

[16] GRASS Development Team. GRASS GIS homepage. http://www.baylor.edu/grass/.

[17] Kenneth E. Hoff, III, John Keyser, Ming Lin, Dinesh Manocha, and Tim Culver. Fast computation of generalized voronoi diagrams using graphics hardware. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 277–286, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[18] E. Kilgariff and R Fernando. *The GeForce 6 series GPU architecture*, pages 471–491. Addison Wesley Professional, 2005.

[19] Wei Li, Zhe Fan, Xiaoming Wei, and Arie Kaufman. Gpu-based flow simulation with complex boundaries. In Matt Pharr and Randima Fernando, editors, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*, pages 747–764. Addison-Wesley Professional, 2005.

[20] L. Mitas and H. Mitasova. Spatial interpolation. In P. Longley, M. F. Goodchild, D. J. Maguire, and D. W. Rhind, editors, *Geographical Information Systems: Principles, Techniques, Management and Applications*, volume 1, pages 481–492. Wiley, 1999.

[21] Helena Mitasova and Lubos Mitas. Interpolation by regularized spline with tension: I. theory and implementation. *Mathematical Geology*, 25:641–655, 1993.

[22] J. Montrym and H. Moreton. The geforce 6800. *Micro, IEEE*, 25(2):41 – 51, mar. 2005.

[23] NVIDIA. CUDA homepage. http://nvidia.com/cuda, 2010. 3.0.

[24] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[25] Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis. *OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL(R), Version 2 (5th Edition) (OpenGL)*. Addison-Wesley Professional, 2005.

[26] R. Sibson. A brief description of natural neighbour interpolation. In Vic Barnet, editor, *Interpreting Multivariate Data*, pages 21–36. John Wiley & Sons, Chichester, 1981.