Hardware-Assisted Natural Neighbor Interpolation

Quanfu Fan* Alon Efrat† Vladlen Koltun‡ Shankar Krishnan§
Suresh Venkatasubramanian¶

Abstract

Natural neighbor interpolation is a weighted average interpolation method that is based on Voronoi tessellation. In this paper, we present and implement an algorithm for performing natural neighbor interpolation using graphics hardware. Unlike traditional software-based approaches that process one query at a time, we develop a scheme that computes the entire scalar field induced by natural neighbor interpolation, at which point a query is a trivial array lookup, and range queries over the field are easy to perform.

Our approach is faster than the best known software implementations and makes use of general purpose stream programming capabilities of current graphics cards. We also present a simple scheme that requires no advanced graphics capabilities and can process natural neighbor queries faster than existing software-based approaches.

Finally, recognizing the limitation incurred by the bounded size of graphics frame buffers, we propose a *subdivision* approach that allows performing queries locally in a subdivision of the input domain. This approach can reduce to a negligibly small degree ($< 1\%$) the loss of precision caused by the naive scaling method while still processing queries faster than the software-based approaches when the number of sites is large.

1 Introduction

A problem often encountered in many applications is that of reconstructing continuous surfaces from sampled data points. Many interpolation methods have been developed for this problem, including minimum curvature splines, Kriging, bilinear and Bezier patches being among the methods that have been used [24]. Natural neighbor interpolation *(NNI)*, also called *area stealing interpolation*, is one of the most popular methods and has been widely used in geophysical modeling, surface reconstruction, and even computational solid/fluid mechanics [1, 5, 16, 21].

First introduced by Sibson [19, 20], NNI is a weighted average method that constructs the interpolant by using *natural neighbor coordinates* based on the Voronoi tessellation of a set of sites. Besides being a fairly simple method to describe, NNI yields an interpolant that is C^0 everywhere and C^1 everywhere except at the sample points. It is also *nonparametric*, which is attractive in many settings as it thus assumes no special properties of the data. However, since NNI involves tessellating the Voronoi diagram, it suffers from the disadvantage of being computationally costly, especially when the number of sites is large.

Related work While extensive research has been conducted on the subject of NNI [3, 16, 20, 23, 26, 24], there are only a few implementations publicly available for general use [7, 14, 15, 25]. Among them, the "nngridr" package [25] developed by Watson received recognition for the method that computes all the Voronoi polygons required for one interpolation operation using a single combined calculation. NatGrid [7], a software package by NCAR, is based on the work of Watson and is not available for free download. Owen [14] implemented both 2D and 3D NNI in a package that was not released to the public. Recently, Sakov [15], implemented NNI based on Shewchuk's `Triangle` package [18] for computing the underlying Delaunay triangulation.

The powerful computational ability of modern graphics cards has led to their use as a fast streaming coprocessor, solving many problems outside the realm of computer graphics. Notable examples include path planning, collision detection, particle systems, and physical simulation. Starting with the work of Hoff *et al.* [10] on the computation of Voronoi diagrams, there are now fast hardware-assisted algorithms for problems in data analysis, geometric optimization, and solid modeling, among others [2, 10, 11, 12, 9, 13].

Our work In this paper, we extend the paradigm of hardware-assisted computation to the area of natural neighbor interpolation. Our main contributions are as follows.

- We present a simple approach that can answer

*Department of Computer Science, University of Arizona; *quanfu@cs.arizona.edu*

†Department of Computer Science, University of Arizona; *alon@cs.arizona.edu*

‡Computer Science Division, University of California, Berkeley; *vladlen@cs.berkeley.edu*

§AT&T Labs – Research; *krishnas@research.att.com*

¶AT&T Labs – Research; *suresh@research.att.com*

natural neighbor interpolation queries in one rendering pass with a single readback. (See Section 3 for definitions.) This scheme performs better than current software implementations and uses only standard features of modern graphics cards.

- We present a more general scheme, based on the latest generation of graphics architectures, that can compute natural neighbor interpolation queries simultaneously for each (discrete) point within the convex hull of the input point set. This approach also leads to a scheme for performing range queries on the natural neighbor interpolation function; an example would be the computation of the average or maximum value in a given range.

- We present a subdivision approach to reduce the loss of precision caused by the scaling that is usually required when the inputs are widely spread over a large domain.

- We present a detailed experimental study of the above approaches, comparing them to existing software implementations.

The paper is organized as follows. Section 2 introduces natural neighbor interpolation and Section 3 briefly reviews hardware-assisted computation of Voronoi diagrams. We present a simple overlay-based NNI interpolation scheme in Section 4 and demonstrate a general method for computing NNI over an entire range in Section 5. Section 7 presents a detailed experimental study and future directions are discussed in Section 8.

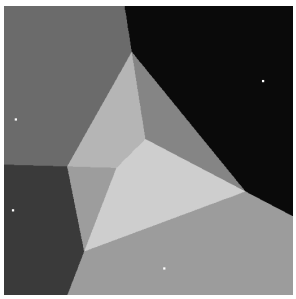


Figure 1: Area stolen after a query point is inserted into a Voronoi diagram.

2 Natural Neighbor Interpolation

We are given a set $S = \{s_1, \ldots, s_n\}$ of *sites*, and a weight function $f : S \to \mathbb{R}$. For a point q, the *interpolated value* at q is estimated as $f(q) \triangleq \sum_i w_i(q) f(s_i)$, where $w_i(q)$ is the weight of s_i. This method is called *weighted average interpolation*. Different interpolation methods differ by the way the weights are determined.

NNI relies on the construction of Voronoi diagrams. Here w_i is defined as follows:

$$w_i = \frac{\text{Area}(\mathcal{V}_{S\cup\{q\}}(q) \bigcap \mathcal{V}_S(s_i))}{\text{Area}(\mathcal{V}_{S\cup\{q\}}(q))}. \tag{2.1}$$

Here $\mathcal{V}_S(s_i)$ is the Voronoi cell of s_i in $Vor(S)$ (the Voronoi diagram of S) and $\mathcal{V}_{S\cup\{q\}}(q)$ is the Voronoi cell of q in $Vor(S \cup \{q\})$. (See Figure 1.) We refer to the set of points $\{s_i \mid \mathcal{V}_{S\cup\{q\}}(q) \cap \mathcal{V}(s_i) \neq \emptyset\}$ as the *natural neighbors* of q. In the sequel, when we refer to $f(q)$, it should be understood that the weight vector w_i associated with f is defined as in Equation 2.1.

In our problem, the area of the Voronoi cell is approximated by the number of pixels in the Voronoi cell. Let $N(\mathcal{V}_{S\cup\{q\}}(q))$ be the number of pixels in the Voronoi cell $\mathcal{V}_{S\cup\{q\}}(q)$. Equation 2.1 can be closely approximated by

$$w_i \approx \frac{N(\mathcal{V}_{S\cup\{q\}}(q) \bigcap \mathcal{V}_S(s_i))}{N(\mathcal{V}_{S\cup\{q\}}(q))} \tag{2.2}$$

3 Preliminaries

3.1 The Graphics Pipeline. The graphics pipeline is often used as a rendering (or "drawing") engine to facilitate interactive display of complex three-dimensional geometry. The input to the pipeline is a set of geometric primitives and images, which are *transformed* and *rasterized* at various stages of the pipeline to produce a stream of *fragments* that is "drawn" on a two-dimensional grid of pixels known as the *frame buffer*. The frame buffer is a collection of several individual dedicated buffers (color, stencil, depth buffers, etc.). The user interacts with the pipeline via a standardized software interface (such as OpenGL or DirectX) that is designed to mimic the graphics subsystem. For more details, the reader may refer to the OpenGL programming guide [27].

3.2 Computing Voronoi Diagrams and Overlays. It is well known that the Voronoi diagram of a set of points in the plane can be computed using the graphics pipeline [10]. Each point is assigned a

specific color, and at the end of the calculation, each Voronoi region is represented as an area of the appropriate color in the color buffer. For the Euclidean metric, this is achieved by drawing right angled cones with their apex at each point, as seen from below. The paper by Hoff *et al.* [10] has further details on this procedure; since cones are drawn by approximating their surface with triangles, an appropriate number of triangles must be drawn to ensure that the error incurred is less than one pixel.

We also need the depth field that the Voronoi diagram induces. At each pixel, the depth buffer contains the height of the lowest cone. Since the cones are right angled, this is also the distance of the nearest Voronoi site to the pixel.

4 A Simple Overlay-Based Method.

In this section, we present a simple algorithm for natural neighbor interpolation of a function $f(\cdot)$. The input includes a set $S = \{s_1, \ldots, s_n\}$ of n sites, each site with its functional value $f(s_i)$, and a set $Q = \{q_1, \ldots, q_m\}$ of m query points. The algorithm outputs the interpolated value $f(q_i)$ for each q_i.

Our algorithm is based on the following *overlay* method:

Algorithm 1 Overlay Method for NNI queries

Compute *Vor*(S)
for Each query $q \in Q$ **do**
  Compute $Vor(S \cup \{q\})$
  Compute the overlay of $Vor(S)$ and $\mathcal{V}(q)$ (in $Vor(S \cup \{q\})$)
  Determine the weights of the natural neighbors of q from the overlay, and return $f(q)$
end for

We remark in passing that the running time of this algorithm for a single query q, using standard algorithms for Voronoi diagram computation and line segment intersection, is $O((n+k)\log n)$ in a traditional RAM model, where k is the number of natural neighbors of q.

4.1 Implementation. The first step is easy to implement (see Section 3). It yields two two-dimensional arrays. One contains at each pixel a color corresponding to the site whose Voronoi cell that pixel lies in. The other contains the distance of the pixel from the corresponding site.

When we now process a query point q by drawing its corresponding cone, an appropriate depth test will ensure that pixels of this cone *will only be drawn where this cone is lower than all other cones*, implying that for all such pixels, the query point is the closest site.

If we clear the color buffer before this rendering step, it will now contain all (and only) the pixels in the Voronoi cell of q. If we now read back the color buffer, and access the color (in the earlier color buffer) of all drawn pixels (i.e., all pixels in $\mathcal{V}_{S\cup\{q\}}(q)$), the number of pixels of each color represents the stolen area from the corresponding site, and the total number of pixels is a measure of the area of $\mathcal{V}_{S\cup\{q\}}(q)$. Given these quantities, $f(q)$ can now be computed.

This is a straightforward implementation of the above algorithm, and as we shall see in Section 7, performs reasonably well in comparison with standard software-based approaches. However, it suffers from two drawbacks. The number of generated fragments is very large (see below for details) and to perform a readback for each query is very expensive. In what follows, we address these two issues.

4.2 Clustering queries. In the graphics pipeline, each geometric primitive is rasterized into fragments. A sequence of per-fragment operations such as depth tests and stencil tests are subsequently performed on all fragments after rasterization and only those fragments that pass all the tests update their corresponding pixels in the frame buffer. Generally speaking, the larger the area of the primitive, the more fragments generated and thus the more expensive the rendering.

Another issue peculiar to hardware-assisted algorithms is the issue of *read backs*: extracting data from the frame buffer into main memory. This is typically an expensive operation (graphics cards use a slow bus for data transfer in this direction), and is costly both in terms of its fixed cost as well as the bandwidth available for transfer. It also stalls the pipeline for future rendering passes.

The first observation that helps minimize the number of rendered fragments is that the Voronoi cones only need to be tall enough to detect every Voronoi vertex. (As opposed to cones that are as large as the whole frame buffer.) It is easy to see that the cone height needs only be as large as the radius r^* of the largest empty circle for the algorithm to detect all Voronoi cells inside the convex hull of the points.

The second observation makes use of *logical* hardware functions. In the above algorithm, all the information we need for a particular query q is a bit that determines for each pixel whether it is in $\mathcal{V}_{S\cup\{q\}}(q)$ or not. Since color buffers have 32 bits (eight each for

red, green, blue, and the alpha channel), we can process thirty two queries simultaneously using a hardware operation that performs a bitwise OR of the colors being rendered at a pixel. It is then a straightforward exercise to read these pixels back and determine the stolen area costs for each of the queries.

The idea of query clustering is to batch nearby queries together and retrieve only a small portion of the frame buffer that suffices to compute the interpolation function correctly for these queries. By doing so, we avoid reading back the whole frame buffer, which is expensive. As described below, with query clustering, we only need to retrieve the *accumulative stolen area* of a region for answering queries in that region, thus greatly reducing the readback time.

We subdivide the frame buffer window into a uniform grid. Define the *accumulative stolen area* (ASA) of a grid cell as the union of the stolen areas of all points in the cell. Such a region contains all the information needed to process natural neighbor interpolation queries in the cell. The ASA of a cell is equivalent to the Voronoi region of the cell, which can be efficiently constructed by rendering for each vertex of the cell a quarter cone and for each edge a half "tent", a rectangle growing upward in an angle of 45 degree from that edge [10]. The ASA is then computed by reading the frame buffer back into main memory.

Combining queries that lie in this cell using blending completes the process. As we shall see in section 7, this optimization saves a significant amount of time in query processing.

5 A General Approach To Compute f()

The algorithm of the previous section employs a simple hardware-based overlay method to answer a single natural neighbor query. In this section, we present a more sophisticated scheme based on certain advanced features of current graphics cards that can compute the entire natural neighbor function over the two dimensional grid. The key ingredient is a pseudo-streaming algorithm to compute the interpolant at a query point, which is based on the *compound signed decomposition* technique for natural neighbor interpolation proposed by Watson [23].

We briefly review Watson's algorithm. The basic operation in computing the natural neighbor weights is to compute areas of convex polygons that are stolen from the original Voronoi diagram. Instead of triangulating the polygon to compute the area, Watson's method generates a triangle which contains the polygon and each of its edges contains some edge of the polygon. Further the portions of the bounding triangle that are outside the polygon are triangles themselves. This allows us to compute the area of the polygon as a signed expression involving the areas of the above triangles. This is called the compound signed decomposition.

Let us assume that we have the Delaunay triangulation of the original point set and the triangle vertices are in a canonical (anticlockwise, for example) order. Further, let us assume that for any query point q, we can find all the Delaunay circles that contain q. Watson's algorithm takes a set of points P and a query point q as inputs and outputs the natural neighbor coordinates of q.

Algorithm 2 Watson's NNI Algorithm

for each Delaunay triangle $t = (p_0, p_1, p_2)$ whose circumcircle C^t contains query q **do**
  Let cc be the center of C^t
  for each $i \in \{0, 1, 2\}$ **do**
    Let cc_i=circumcenter(q, $p_{(i+1) \bmod 2}$, $p_{(i+2) \bmod 2}$)
  end for
  for $i = 0$ to 2 **do** {**/* for each vertex of t */**}
    /* Compute area of two cc with t_c */
    Let $A_i^t = 0.5 \cdot \mathrm{Det}(cc_{(i+1) \bmod 2}, cc_{(i+2) \bmod 2}, cc)$
    $A_i = A_i + A_i^t$
  end for
end for
Normalize all A_i to compute q's natural neighbor coordinates

Running time Both Watson's implementation of this algorithm and the implementation developed by Sakov use a brute force $O(n)$ time procedure to determine which Delaunay circles contain a given point. In general, if we were to construct the arrangement of Delaunay circles, we could achieve a query time of $O(\log n)$ (via point location) at the cost of quadratic space complexity.

However, since the point location is performed on an arrangement of circles, we can invoke the following result:

THEOREM 5.1. (SHARIR [17], THEOREM 4.3) *Given a collection $\mathcal{D}$ of n discs in the plane, we can preprocess it in randomized expected time $O(n \log^2 n)$ into a data structure of expected size $O(n \log n)$, such that for any query point x, the k discs of $\mathcal{D}$*

containing x can be reported in time $O((k+1)\log n)$.

It is easy to see by planarity arguments that the number of natural neighbors of a query q is $\Theta(k)$, and hence the above query time suffices to determine all natural neighbors of q as well.

A Streaming View Watson's algorithm provides an interesting decomposition scheme for computing the contribution of each vertex. Instead of overlaying two Voronoi diagrams (with and without the query point) and computing areas of irregular convex polygons, we can perform simpler triangle area computations. Another observation is that the algorithm separately maintains the contribution from each of q's natural neighbors (potentially requiring linear space) before computing the interpolated value.

Observe that the interpolated value at q is

$$f_q = \frac{\sum_i f_i A_i}{\sum_i A_i} = \frac{\sum_i f_i \sum_{q \in C^t} A_i^t}{\sum_i \sum_{q \in C^t} A_i^t}, \tag{5.3}$$

where f_i is the value at input site i. We can compute the interpolated value by maintaining the numerator and denominator, i.e., $\sum_i \sum_{q \in C^t} f_i A_i^t$ and $\sum_i \sum_{q \in C^t} A_i^t$.

With this modification, Watson's method can be rephrased as a *streaming algorithm*, where the elements of the data stream are the Delaunay circles of the input point set. Algorithm 3 then computes the natural neighbor interpolant in a streaming fashion, using only constant-sized temporary storage.

5.1 Computing the Scalar Field using Graphics Hardware. Streaming algorithms are closely related to algorithms on graphics cards. The latest generation of graphics cards provide the user with the ability to write C-like programs called *fragment programs* that are executed by each fragment. These programs are stateless functions - they merely operate on and modify the state of the current fragment (for example, its color); carrying the state between fragment program invocation is not allowed. The only way to maintain state between fragments is through temporary storage in *textures*. Thus these programs are (severely restricted forms of) multi-pass streaming algorithms.

Another feature of current hardware is the ability to perform full 32-bit signed floating point operations, thus maintaining precision. This is important for our computation since the intermediate values that are accumulated can get fairly large.

Algorithm 3 Streaming algorithm for natural neighbor interpolation

```
num = 0, den = 0
for each Delaunay triangle t (with vertices p_0, p_1 and p_2) do
  Let (t_c, t_r) be the circumcenter and circumradius of t
  if || q - t_c ||_2 < t_r then {q is inside circumcircle of t}
    for i = 0 to 2 do
      cc_i = circumcenter(q, p_{(i+1) mod 2}, p_{(i+2) mod 2})
    end for
    for i = 0 to 2 do {/* each vertex of t */}
      /* Compute area of two circumcenters with t_c */
      num = num + 0.5 f_i · Det(cc_{(i+1) mod 2}, cc_{(i+2) mod 2}, t_c)
      den = den + 0.5 · Det(cc_{(i+1) mod 2}, cc_{(i+2) mod 2}, t_c)
    end for
  end if
end for
f_q = num/den /* f_q is the natural neighbor interpolant at q */
```

For brevity, we will not go into the details of how such programs are implemented on a graphics card. Current developments in graphics programming have led to the development of a C-like language called Cg [8] for programming these cards, and there are higher level constructs that allow for fully general purpose stream programming [6]. In the sequel we will thus describe our algorithm merely as a high level stream algorithm.

The algorithm proceeds by drawing each Delaunay circle and updating the interpolant value at all points that lie inside the circle. When all the circles are drawn, the value at each pixel is its natural neighbor interpolant. This produces the scalar field. Let each point i in the input point set contain its position (x_i, y_i) and value f_i.

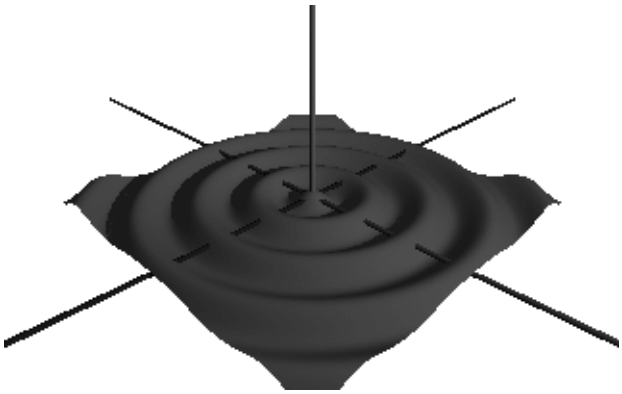
Below in Figure 2, we show an example output of the algorithm when presented with data points sampled randomly from the unit square, with weight values defined by the function

$$f(x,y) = 0.5 + 0.5\cos(20\sqrt{(x-0.5)^2 + (y-0.5)^2}) \tag{5.4}$$

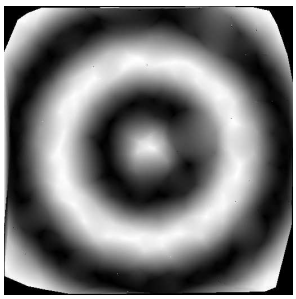
which is radially symmetric around the point $(0.5, 0.5)$ and has range $[0, 1]$.

Algorithm 4 Hardware-assisted algorithm for Natural Neighbor Interpolation

Precompute Delaunay triangulation (and corresponding Delaunay circles) of the original point set.
Set initial scalar field everywhere to zero and bind to floating-point texture FP.
Encode input point set $p_i = (x_i, y_i, f_i), i = 1 \ldots n$ as color values and store in texture T.
Draw Delaunay circle C passing through p_l, p_m and p_n with color value (l, m, n) that index T.
for each Delaunay circle C **do**
  Draw C directly onto FP.
  Fragments generated by C execute the fragment program in Algorithm 3 and update their num, den and num/den values in red, green and blue color channels of FP respectively.
end for
When all the circles are drawn, the blue channel has the interpolated value f_q.



(a) A plot of the function



(b) The interpolated values (grayscale: 0 (dark) to 1(white))

Figure 2: The scalar field generated by our algorithm

We do not render points outside the convex hull of the input points, firstly because Watson's algorithm is not valid in this region, and secondly because NNI interpolation itself does not make sense outside the convex hull of the input. As a simple preprocessing step, we compute the convex hull of the input points and use a *stencil mask* to disable rendering outside it.

6 Interpolation Over Large Domain

The main problem of using a buffer-based approach is the loss of accuracy involved. The locations of the input sites have to be rounded to the next pixel of the buffer. This lost of accuracy tends to become more significant as the number of input sites increases, since the average distance between the sites and its distance to the query points decreases. When the number of input sites is large and the loss of accuracy is not tolerable, it is recommended to break the problem into several sub-problems using the method we propose in the current section.

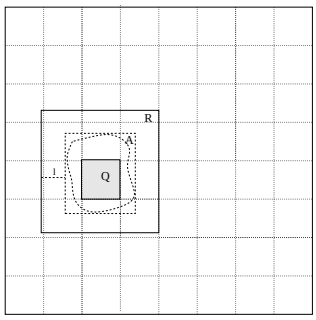


Figure 3: A uniform grid applied over the input domain. A is the axis-aligned bounding rectangle of the *accumulative stolen area* of a subdivision Q and R is a rectangle whose points have distance at most ℓ from ∂A.

LEMMA 6.1. *Let Q be a region of the input domain and let A be the accumulative stolen area of Q. Let $S_Q = \{s_i \in S | \mathcal{V}_S(s_i) \cap A \neq \emptyset\}$. Then, for any query point $q \in Q$, $S_q \subset S_Q$ where S_q is the set of natural neighbors of q.*

We omit the proof due to lack of space.

Below is an outline of the algorithm. Partition the input domain using a grid Γ and for each cell Q of Γ find a set of sites S_Q that suffice for any queries in Q. This is done as follows:

1. Compute $Vor_{S \cup Q}$ (i.e the *accumulative stolen area* of Q) and compute the bounding rectangle A of the *accumulative stolen area* of Q in this diagram, as describe in Section 4.2.

2. Read the depth buffer into the main memory, trace ∂A, and for each pixel $p \in \partial A$, find the distance to its nearest site in this diagram. This distance is the depth value of p in the depth buffer.

   Let ℓ denote the maximum value achieved. Generate the rectangle R, defined as all points in the plane whose distance from ∂A, under the L_∞ norm is at most ℓ (see Fig. 3).

3. Compute $S_Q = S \cap R$ by checking for each $s \in S$ if it is inside R. Clearly S_Q contains all sites whose Voronoi cell intersect ∂A, and the sites inside A. Thus, by Lemma 6.1 these are all the sites that might participate in computing the value of the interpolated function in Q.

Once R and S_Q are found, we can construct Vor_{S_Q} and answer queries in Q safely with Vor_{S_Q}. The process above is repeated until each cell is finished.

Our implementation picks the size of the grid cell of Γ as $1/2$ of the size of the graphics window. As we will see in section 7, this approach incurs a precision loss within a negligibly small degree while still processing queries faster than the software-based implementations when the number of sites is large.

7 Performance Analysis

Code platform In this section, we present an empirical study of our approach. Our algorithms were implemented in C/C++ and OpenGL 1.4. We used two platforms for testing the code. `LINUX` is 1 Ghz Pentium processor with 256 MB of memory and an Nvidia GeForce FX 5900 graphics card running Red Hat 9.0. `WINDOWS` is a Pentium M 1.4Ghz laptop running Windows XP and Cygwin 1.5, with an Nvidia GeForce Go FX 5650 graphics card. The laptop supports the `WGL` extensions required to run the scalar field calculation of Section 5. In all cases, we compiled the code using `g++ -O3`. We refer to the area-based query algorithm of Section 4 as **Area**, the streaming scalar field algorithm of Section 5 as **Stream** and the subdivision algorithm for large input domain of Section 6 as **Subdivision**. Note that **Stream** currently only runs on `WINDOWS`; this is due to the lack of appropriate driver support on Linux.

Reference implementations To demonstrate the performance of our algorithm, we choose two software-based implementations of natural neighbor interpolation for comparison. The first one is **nni**, Watson's implementation of NNI [22] (a standard code base) with slight modification to allow it to accept multiple queries. The second one is **nniT**, Sakov's [15] implementation of Watson's algorithm that makes use of Jonathan Shewchuk's `Triangle` package [18].

Test function The function we approximate is the radially symmetric $\cos(cr)$ (Eq. (5.4)) from Section 5. The input sites and query points were both randomly generated in a range of $[0, 1]^2$. Note that care must be taken to ensure that query points lie inside the convex hull of the input points.

Running time For all experiments, we used a rendering window size of 512×512. In Table 1 we compare the running times for **Area**, **nniT** and **nni** on `LINUX`. **nniT** is initially superior to both **Area** and **nni**, but as the number of sites increases, **Area** starts to dominate. It is worth noting that the algorithm **Area** implements is a trivial one and **nniT** is one of the best known software implementations to date. Thus the fact that **Area** outperforms **nniT** is significant.

Table 2 presents a breakdown of the individual elements of **Area**. It also demonstrates how clustering queries improves the performance of the algorithm.

Breakdown	Clustered Queries	Non-Clustered Queries
Preprocess	0.16	0.0
VoronoiDraw	0.12	0.12
QueryDraw	0.17	0.25
Readback	0.10	4.34
Counting	0.07	3.28
Total	0.62	7.99

Table 2: The breakdown of specific steps in **Area**. *Preprocess:* Time spent on computing readback windows; *VoronoiDraw:* Time spent on drawing Voronoi sites; *QueryDraw:* Time spent on drawing query points; *Readback:* Time spent on readback; *Counting:* Time spent on counting stolen pixels. Numbers reported for a run with 10000 input points and 10000 query points.

Table 3 compares **nniT** and **Stream** on `WINDOWS`. We drop **nni** from this comparison, as its running time is significantly worse even for a few queries. Since **Stream** computes the entire scalar field, we compare running times by making the same number of queries (512×512) to **nniT**.

#Input Sites	5000			10000			20000		
#Queries	**Area**	**nniT**	**nni**	**Area**	**nniT**	**nni**	**Area**	**nniT**	**nni**
100	0.25	0.06	0.26	0.29	0.13	0.58	0.43	0.26	-
1000	0.27	0.11	2.18	0.35	0.20	5.87	0.44	0.41	-
5000	0.35	0.32	11.09	0.40	0.53	27.57	0.52	0.97	-
10000	0.49	0.58	24.27	0.50	0.94	56.45	0.65	1.61	-
20000	0.69	1.14	44.04	0.71	1.84	112.36	0.86	3.10	-

Table 1: Total Running time (in seconds) for **Area**, **nniT**, and **nni** on `LINUX`. The number of input points is taken from {5000,10000,20000}, and the number of queries ranges from 100 to 20000.

#Input Sites	2000		40000		60000	
#Queries	**Subdivision**	**nniT**	**Subdivision**	**nniT**	**Subdivision**	**nniT**
20000	14.47	2.98	15.40	13.91	16.39	38.73
40000	26.69	5.56	27.15	27.33	27.49	69.29
60000	35.40	8.24	38.71	40.67	39.08	129.72

Table 4: Total Running time (in seconds) for **Subdivision** and **nniT** on `LINUX`. The input domain is 2048×2048. The number of input points is taken from {20000,40000,60000}, and the number of queries is taken from {20000, 40000, 60000}

#Inputs	**nniT**	**Stream**
5000	11.93	3.23
10000	13.71	5.6
20000	18.0	10.6

Table 3: Total running time (in seconds) for processing 512×512 queries on `WINDOWS`

Table 4 compares the running time for **Subdivision** and **nniT** on `LINUX` over a large domain of 2048×2048. When the number of input sites is large, **Subdivision** outperforms **nniT**.

Error analysis To determine the error incurred by our approach, we use **nniT** as the reference implementation, and measure the relative error of queries to **Area** and **Stream**. Because of the rapidly changing behavior of the function, and the inherent floating point errors incurred in calculating it, for values less than 0.01 we report absolute rather than relative differences.

Using a 512×512 window, **Stream** yields an average relative error of 2.6%, with a standard deviation of 0.25. The median relative error is 3.9%. For reported values of less than 0.01 in both the reference and **Stream**, the average absolute difference was 0.0004. A total of 53 observations were excluded from this calculation because they returned invalid values (values greater than 1.0, or NaN).

Performing a similar analysis for **Area**, the results were similar. The average relative error was 3%, with a standard deviation of 0.29. The average absolute error was 0.001. The median relative error was much smaller, at 0.4%. 28 values were rejected as invalid.

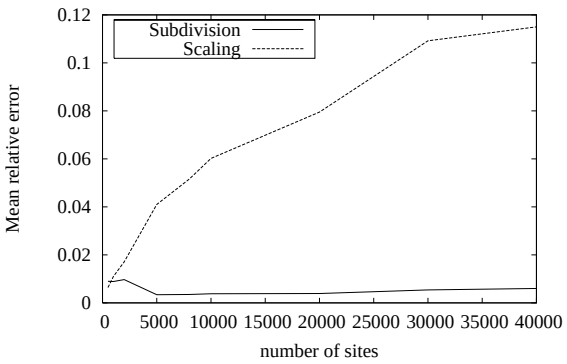


Figure 4: The loss of precision incurred by the *subdivision* method and the scaling method

Figure 4 shows how the average error varies as the number of sites increases for the **Subdivision** and the scaling method. The loss of precision incurred by **Subdivision** is always below 1% regardless of the number of sites, thus negligible. For comparison, the scaling method yields quite noticeable errors as the number of sites increases.

8 Discussion

The results presented in this paper demonstrate that the use of graphics hardware can speed up the

processing of natural neighbor interpolation queries. Since we can also compute the scalar field induced by the natural neighbor function, it is possible to do range searching over a domain completely in hardware.

One significant problem that comes up when we use graphics cards is the bounded size of frame buffers. The loss of precision could be fairly noticeable, when the scaling is required to place widely spread inputs over a large domain into the frame buffer window. We demonstrate that the subdivision approach can reduce the loss of precision to a negligible degree while still processing queries faster than the software-based implementations in some cases.

This paper also demonstrates the expressive power of fragment programs. As general purpose stream programs, their potential is only now being exploited, and it is likely that they can facilitate practical and efficient solution of many problems in computational geometry. This is a fruitful area for future exploration.

References

[1] Abramov, O. An evaluation of interpolation methods for mola data. In *Meeting of the American Geophysical Union (AGU)* (2001). Poster.

[2] Agarwal, P., Krishnan, S., Mustafa, N., and Venkatasubramanian, S. Streaming geometric optimization using graphics hardware. In *11th European Symposium on Algorithms* (2003).

[3] Anton, F., Gold, M. C., and Mioc, D. Local coordinates and interpolation in a voronoi diagram for a set of points and line segments. In *The Voronoi Conference on Analytic Number Theory and Space Tillings* (1998), pp. 9–12.

[4] Arya, S., Mount, D. M., Netanyahu, N. S., Silverman, R., and Wu, A. An optimal algorithm for approximate nearest neighbor searching. In *Proc. 5th ACM-SIAM Sympos. Discrete Algorithms* (1994), pp. 573–582.

[5] Boissonnat, J.-D., and Cazals, F. Smooth surface reconstruction via natural neighbour interpolation of distance functions. In *Symposium on Computational Geometry* (2000), pp. 223–232.

[6] Buck, I., and Hanrahan, P. Data parallel computation on graphics hardware. In *Graphics Hardware* (2003).

[7] Clare, F. `http://ngwww.ucar.edu/ngdoc/ng/ngmath/natgrid/nnhome.html`.

[8] Fernando, R., and Kilgard, M. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley, 2003.

[9] Guesgen, H. W., Hertzberg, J., Lobb, R., and Mantler, A. Buffering fuzzy maps in gis. *Spatial Cognition and Computation (Special Issue on Vagueness, Uncertainty and Granularity) 3*, 2&3 (2003), 207–222.

[10] Hoff, K., Culver, T., Keyser, J., Lin, M., and Manocha, D. Fast computation of generalized voronoi diagrams using graphics hardware. *Proceedings of ACM SIGGRAPH 1999* (1999).

[11] Hoff, K., Culver, T., Keyser, J., Lin, M., and Manocha, D. Interactive motion planning using hardware-accelerated computation of generalized voronoi diagrams. In *Proc. IEEE International Conference on Robotics and Automation* (2000).

[12] Krishnan, S., Mustafa, N., and Venkatasubramanian, S. Hardware-assisted computation of depth contours. In *Proc. 13th ACM-SIAM Symp. on Discrete Algorithms* (January 2002), pp. 558–567.

[13] Mustafa, N., Krishnan, S., Varadarajan, G., and Venkatasubramanian, S. Dynamic simplification and visualization of large maps. *Intnl. Jornal of Geographic Information Systems* (2004, to appear).

[14] Owen, S. J. An implementation of natural neighbor interpolation in three dimensions. Master's thesis, Brigham Young University, 1992.

[15] Sakov, P. `http://www.marine.csiro.au/~sakov/`.

[16] Sambridge, M., Braun, J., and McQueen, H. Geophysical parameterization and interpolation of irregular data using natural neighbors. *Geophysical Journal International 122* (1995), 837–857.

[17] Sharir, M. On k-sets in arrangements of curves and surfaces. *Disc. Comput. Geom 6* (1991), 593–613.

[18] Shewchuk, J. R. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In *Applied Computational Geometry: Towards Geometric Engineering*, M. C. Lin and D. Manocha, Eds., vol. 1148 of *Lecture Notes in Computer Science*. Springer-Verlag, May 1996, pp. 203–222. From the First ACM Workshop on Applied Computational Geometry.

[19] Sibson, R. A vector identity for the Dirichlet tessellation. *Math. Proc. Camb. Phil. Soc 87* (1980), 151–155.

[20] Sibson, R. *Interpreting Multivariate Data*. John Wiley & Sons, 1981, ch. A brief description of natural neighbour interpolation, pp. 21–36.

[21] Sukumar, N. The natural elemend method in solid mechanics. *Intnl. Journal for Numerical Methods in Engg. 43*, 5 (1998), 839–888.

[22] Watson, D. `http://www.iamg.org/naturalneighbour.html`.

[23] Watson, D. F. Computing the n-dimensional Delaunay tesselation with application to Voronoi polytopes. *The Computer Journal 8*, 2 (1981), 167–172.

[24] Watson, D. F. *Contouring: a guide to the analysis*

and display of spatial data. Pergamon Press, 1992.

[25] WATSON, D. F. *nngridr: An implementation of natural neighbour implementation*, vol. 1 of *Natural Neighbour Series*. David Watson, 1994.

[26] WATSON, D. F., AND PHILLIP, G. M. Neighbour based interpolation. *Geobyte 2*, 2 (1987), 12–16.

[27] WOO, M., NEIDER, J., DAVIS, T., AND SHREINER, D. *OpenGL(R) Programming Guide: The official guide to learning OpenGL, Version 1,2,3*. Addison-Wesley, 1999.