

---

# Smart Nearest Neighbors

*Release 0.00*

David Doria and Wanlin Zhu

July 21, 2011

Rensselaer Polytechnic Institute, Troy NY and School of Psychiatry, University of New South Wales

## Abstract

This document presents an implementation of two algorithms, Voronoi Neighbors and Binary Space Partition (BSP) Neighbors. These algorithms find neighbors of a point in a point set that are somehow “better” than a “K nearest neighbors” or a “all neighbors within a radius” query. This type of nearest neighbor query is more computationally expensive, but results in set of neighbors with more desirable properties. The BSP Neighbors search ensures that there is less local duplication, while the Voronoi Neighbors search ensures that the spatial arrangement of the neighbors is as uniform as possible.

These algorithms are explained in “Point Primitives for Interactive Modeling and Processing of 3D Geometry”.

The code is available here: <https://github.com/daviddoria/SmartNearestNeighbors>

Latest version available at the [Insight Journal](#) [<http://hdl.handle.net/10380/3306>]  
Distributed under [Creative Commons Attribution License](#)

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Voronoi Neighbors</b>	<b>2</b>
2.1	Implementation Details . . . . .	3
2.2	Code Snippet . . . . .	3
<b>3</b>	<b>BSP Neighbors</b>	<b>4</b>
3.1	Code Snippet . . . . .	6

---

## 1 Introduction

Defining local neighborhoods of a point is a very important step in computing local surface properties of point clouds (point sampled surfaces). Typical nearest neighbor searches (K-Nearest and PointsWithinRadius) suffer from the necessity of setting a parameter ( $K$  or the radius) to which the resulting neighborhood is very sensitive. For example, in Figure 1 we show how with a bad choice of  $K$ , the resulting neighbors can be a very poor representation of the neighborhood of a point. This is not a particularly contrived case, as typically sample density varies significantly over the surface described by a point cloud.

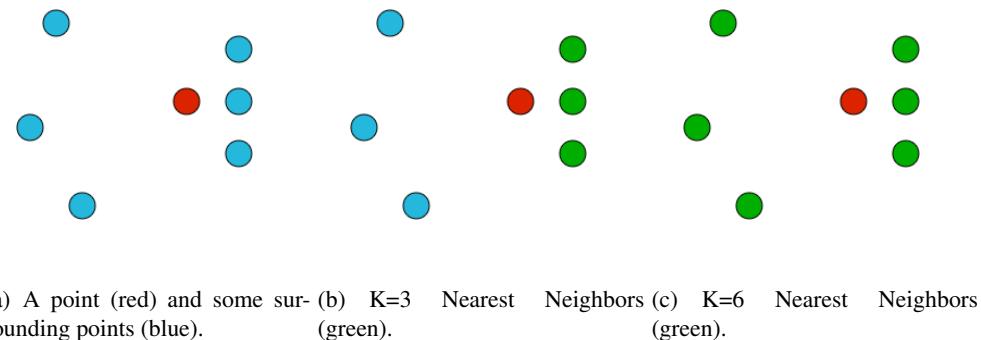


Figure 1: A demonstration of the sensitivity of KNearestNeighbors to the parameter  $K$ .

In this article we present implementations of two algorithms, Voronoi Neighbors and Binary Space Partition (BSP) Neighbors. These nearest neighbor queries are more computationally expensive than the naive versions, but result in sets of neighbors with more desirable properties. These implementations are based on the algorithms described in [1].

Figure 2 compares the two algorithms we will describe to a standard K-Nearest neighbors algorithm.

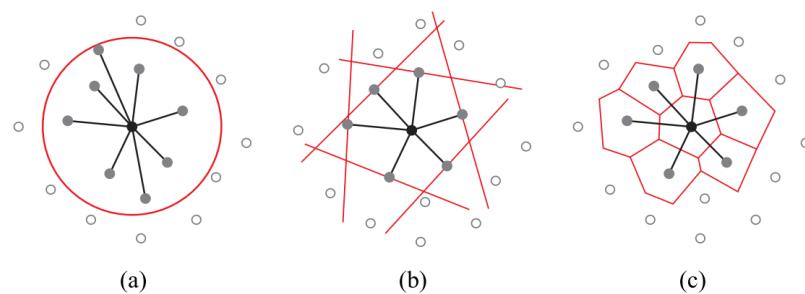


Figure 2: A comparison of (a) K-nearest neighbors, (b) BSP-neighbors, (c) Voronoi neighbors. (Taken from [1].)

## 2 Voronoi Neighbors

The Voronoi neighbors of a point are all points whose Voronoi cell borders the Voronoi cell of the query point. This set of neighbors is spatially as uniform as possible. Figure 3 shows the outputs of the included

Demo2D.cpp.

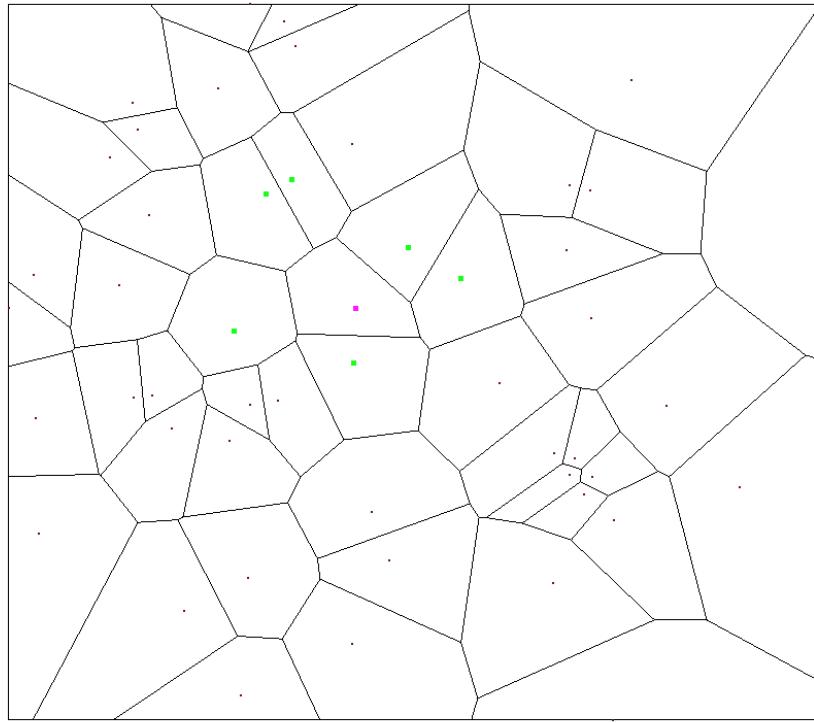


Figure 3: The Voronoi diagram of a point set and the Voronoi neighbors (green) of a query point (pink).

All of the Voronoi computations are performed using `itkVoronoiDiagram2DGenerator`. We have simply provided a wrapper which takes a point set and returns the neighboring points. Note that this method currently only works with 2D point clouds, as the ITK implementation of Voronoi diagrams only works on 2D point clouds. One very strong advantage of this method of finding nearest neighbors is that there are absolutely no parameters to specify.

## 2.1 Implementation Details

Internally, `itkVoronoiDiagram2DGenerator` sorts the seeds (points) that are provided. This procedure changes the IDs of the points, so any information (IDs of neighboring Voronoi cells, etc) further along in the algorithm will no longer correspond to the IDs of the input points. To remedy this, we sort the input points before passing them to the Voronoi generator using the same sorting algorithm that it uses internally. We perform this sort as a parallel sort with the point IDs so that we produce a map from the new IDs to the old IDs that can be used to determine the input point corresponding to the ID of the seed at the output of the Voronoi diagram generation algorithm.

## 2.2 Code Snippet

```
// Find the Voronoi Neighbors of point 0
unsigned int queryPointId = 0;
```

---

```

// Create the input point set
vtkSmartPointer<vtkPoints> points2D = vtkSmartPointer<vtkPoints>::New();
... Fill points2D with points on the Z=0 (XY) plane ...

// Create an object in which to store the result
vtkSmartPointer<vtkPoints> neighbors = vtkSmartPointer<vtkPoints>::New();

// Compute the Voronoi neighbors.
VoronoiNeighbors(points2D, queryPointId, neighbors);

```

### 3 BSP Neighbors

Each nearest neighbor point defines a halfspace. The BSP neighbors are a subset of the kNearestNeighbors which are in the intersection of all of the halfspaces induced by the kNearestNeighbor points.

To ease the explanation, we make some definitions:

- $x$  a test point, any point in the space
- $n_i$  is the  $i^{th}$  nearest neighbor point
- $q$  is the query point (the point for which the neighbors are being computed)

We can then define some vectors:

- $v_{n_i \rightarrow q} = q - n_i$  is the vector from the  $i^{th}$  nearest neighbor to the query point
- $v_{n_i \rightarrow x} = x - n_i$  is the vector from the  $i^{th}$  nearest neighbor to a point  $x$

Each kNeighbor defines a halfspace as:

$$v_{n_i \rightarrow q} \cdot v_{n_i \rightarrow x} \geq 0 \quad (1)$$

The collection of points  $x$  that satisfy this equation is exactly the positive halfspace defined by the  $i^{th}$  neighbor. This equation simply says that the angle between  $v_x$  and  $v_{n_i}$  must be less than 90 degrees. This is shown graphically in Figure 4.

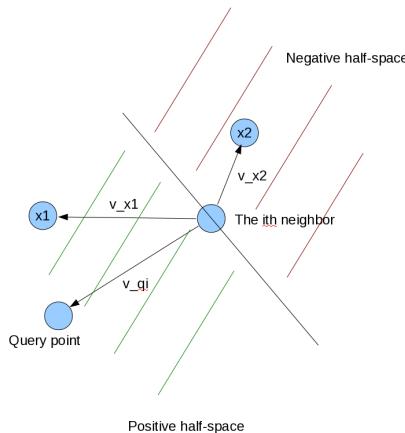


Figure 4: An example halfspace.

Each nearest neighbor point defines a halfspace like this. The BSP neighbors are a subset of the KNearestNeighbors which are in the intersection of all of the positive halfspaces induced by the kNearestNeighbor points.

Figure 5 shows an example 2D point set, a query point, and its neighbors. This demo was done in 2D only for clarity of the figure. The method also works in 3D (we have provided Demo3D.cpp to demonstrate this).

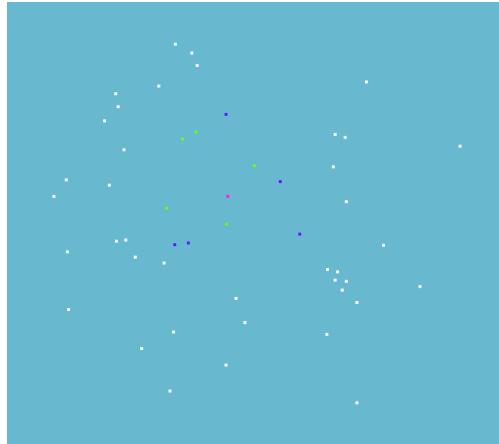


Figure 5: An input point set (white), the query point (pink), BSP Neighbors (green), and KNearestNeighbors that are not included in BSP Neighbors (purple).

The qualitative effect is that the set of BSP Neighbors has less local duplication, so that each neighbor contributes more information to further computations.

Since this search starts with the K-Nearest neighbors,  $K$  must still be specified. However, as long as  $K$  is chosen to be large enough, the complete set of BSP neighbors will be found, making this method very insensitive to the parameter as desired.

### 3.1 Code Snippet

```
// Find the BSP Neighbors of the point with ID = 0
unsigned int queryPointId = 0;

// Create the input point set
vtkSmartPointer<vtkPoints> points = vtkSmartPointer<vtkPoints>::New();
... Fill points2D with points (2D or 3D) ...

// Create an object in which to store the result
vtkSmartPointer<vtkPoints> bspNeighborPoints = vtkSmartPointer<vtkPoints>::New();

// Compute the BSP neighbors
BSPNeighbors(reader->GetOutput()->GetPoints(), queryPointId, bspNeighborPoints);
```

## References

[1] Pauly, Mark, *Point Primitives for Interactive Modeling and Processing of 3D Geometry*. PhD Thesis, 2003 [1](#), [2](#)