THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

NORMAL ESTIMATION AND SURFACE RECONSTRUCTION OF LARGE POINT CLOUDS

By

AMIT N. MHATRE

A Thesis submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Master of Science

Degree Awarded:
Spring Semester, 2006

The members of the Committee approve the Thesis of Amit N. Mhatre defended on April 5, 2006.

Piyush Kumar
Professor Directing Thesis

Ashok Srinivasan
Committee Member

Xiuwen Liu
Committee Member

Approved:

Dr. David Whalley, Chair
Department of Computer Science

Professor Joseph Travis, Dean, College of Arts and Sciences

The Office of Graduate Studies has verified and approved the above named committee members.

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Piyush Kumar for guiding me throughout my thesis and encouraging me to do better research. I am also thankful to my committee members, Dr. Ashok Srinivasan and Dr. Xiuwen Liu for their guidance.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Estimating normals for 3D point clouds and reconstructing a surface interpolating the points are important problems in Computational Geometry and Computer Graphics. Massive point clouds and surfaces with sharp features are active areas of research for these problems. This thesis provides a fast and accurate algorithm for normal estimation and surface reconstruction which can handle large datasets as well as sharp edges and corners. We were successfully able to compute accurate normals for all the points on a cube including corners and edges and reconstruct the cube.

We use several techniques to make the implementation fast and external-memory efficient. The implementation uses multiple threads operating in parallel and hence performs faster on a multiprocessor system. We use a technique called fast projective clustering for fitting multiple planes through the neighborhood of a point. We use a sliding window type streaming algorithm that uses a dynamic data structure for nearest neighbor search. We also develop a simplification algorithm that handles large datasets with sharp edges and corners and enables us to render these datasets using in-core rendering softwares.

CHAPTER 1

Introduction

Estimating normals for point clouds is a well-studied problem in computational geometry. A point cloud can be obtained from an object by using a scanner. A typical scanner consists of a laser/light beam emitter and receiver. The position of the point in 3D can be computed by the positions of the emitter and the receiver. Normal Estimation has many applications in computational geometry such as surface reconstruction and rendering.

The current implementations can compute normals fairly accurately for smooth surfaces. They work pretty well even when there is some noise. Some of the algorithms that process point clouds are [1, 2, 3, 4, 5, 6, 7, 8, 9]. However current implementations do not handle sharp edges and corners reliably. Computing normals for sharp edges and corners and reconstructing the edges and corners is an active area of research and one of the main focuses of the thesis. In this thesis we study how normal estimation can be done for sharp edges by fitting more than one surface at those points such as intersecting planes [10, 11]. We then use the computed normals to perform surface reconstruction.

With the development in scanner technology, very large point clouds can be created within a short period of time. Hence its necessary to be able to operate on large datasets in a reasonable amount of time without compromising on accuracy. Its hard to fit large datasets in memory. Hence all computations should be external-memory efficient. The second main focus of the thesis to handle large datasets efficiently.

The rest of the thesis is divided as follows. Chapter 2 explains our techniques for computing normals for a given dataset. The basic approach is simple. We look at every

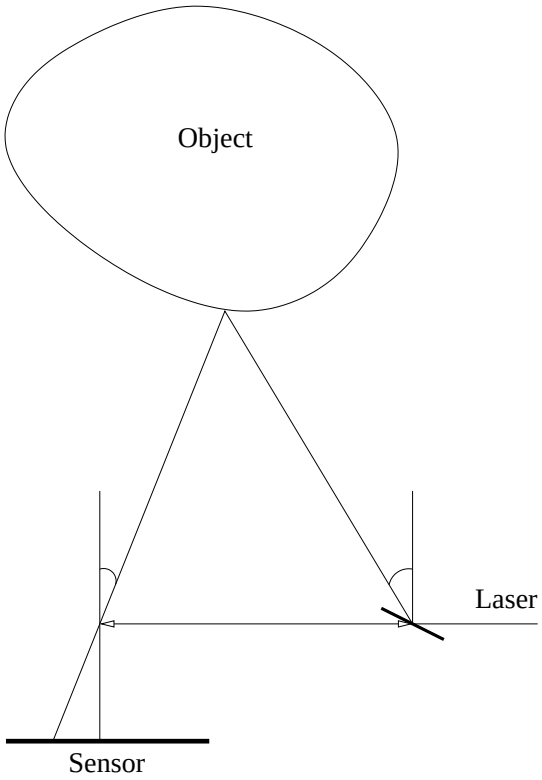


Figure 1.1: Scanner Arrangement

point and it neighborhood. We then try to fit various functions through the neighborhood. We perform various checks to make sure that the fit is valid and makes sense. Once we successfully fit a function, the normal at that point can be computed depending upon the type of function fitted. The functions we consider are one plane, two planes and three planes. For one plane, the normal is simply the normal to the plane. For two and three planes we may need to take an average of the normals of the planes depending upon the location of the point from the planes. If none of the functions fit the neighborhood, we declare a no-fit for the point.

Chapter 3 explains our techniques for surface reconstruction for the dataset. Once we compute the normal for a point, we project all the neighbors on a plane and perform local 2D delaunay triangulation. For two planes and three planes, we need to open the planes onto one single plane before we can perform the triangulation. Once we triangulate the entire

point-set like this, we can render the surface.

Chapter 4 studies the normal estimation and surface reconstruction for various points on a cube in more details. We were able to compute accurate normals at all the points on the cube including edges and corners. Like mentioned before, handling sharp edges and corners was one of the main focuses of the thesis. We also worked on a cube with two opposite faces missing and computed the normals and reconstructed it. Finally, we operated on a randomly sampled cube whose edges were not sampled. Using our algorithm, we were able to reconstruct the entire cube using the edges.

Like mentioned before, one of the main focuses of the thesis is efficiency. We use several techniques to make our program fast and memory-efficient. They are studied in Chapter 5. The implementation supports multi-threading and hence works faster on a multiprocessor system. Also we use a sliding window type approach to make sure that only a fixed number of points are in memory at a given instant and we work on only this window of points. The size of the window depend upon the memory available. We can also do normal-copying which is basically copying the normal at a point to its neighbors if they fit a plane. This saves the time required for computing the normals at the neighboring points.

In-core rendering softwares cannot render very large datasets. Hence we need to simplify a very large dataset before reconstructing it. Chapter 6 discusses a method for simplifying large datasets. It basically deletes points which are close by and fit a single plane and have the same normal. It tries to make sure that the resultant sampling is uniform.

Our results are provided in Chapter 7. We do better than the current implementations in both speed and accuracy.

Our sliding window type approach requires a dynamic nearest neighbor search data structure. Appendix A presents our implementation of a data structure for fast dynamic nearest neighbor data structure which allows insertions and deletions.

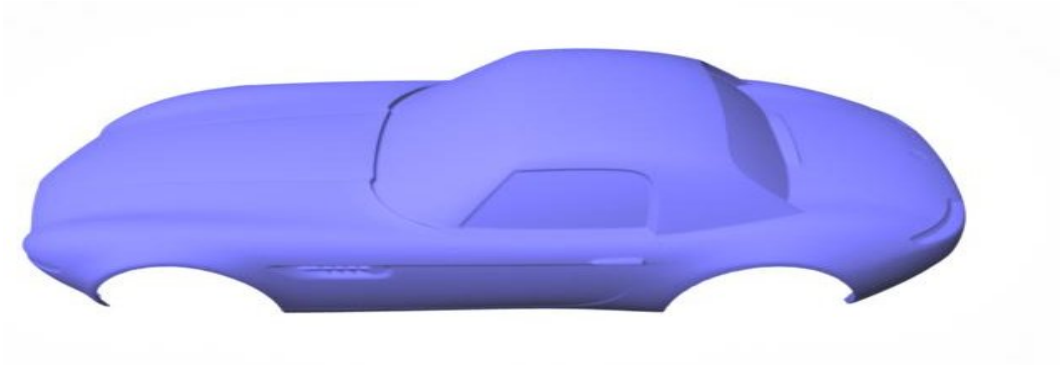
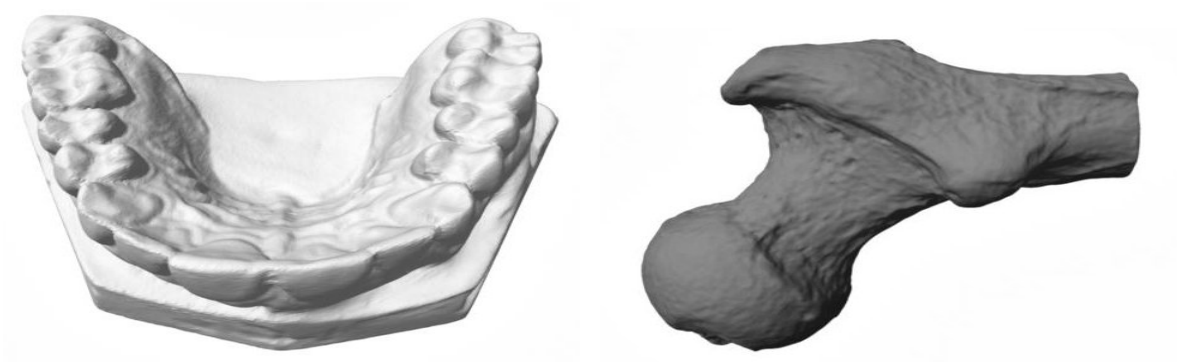


Figure 1.2: Automobile Industry



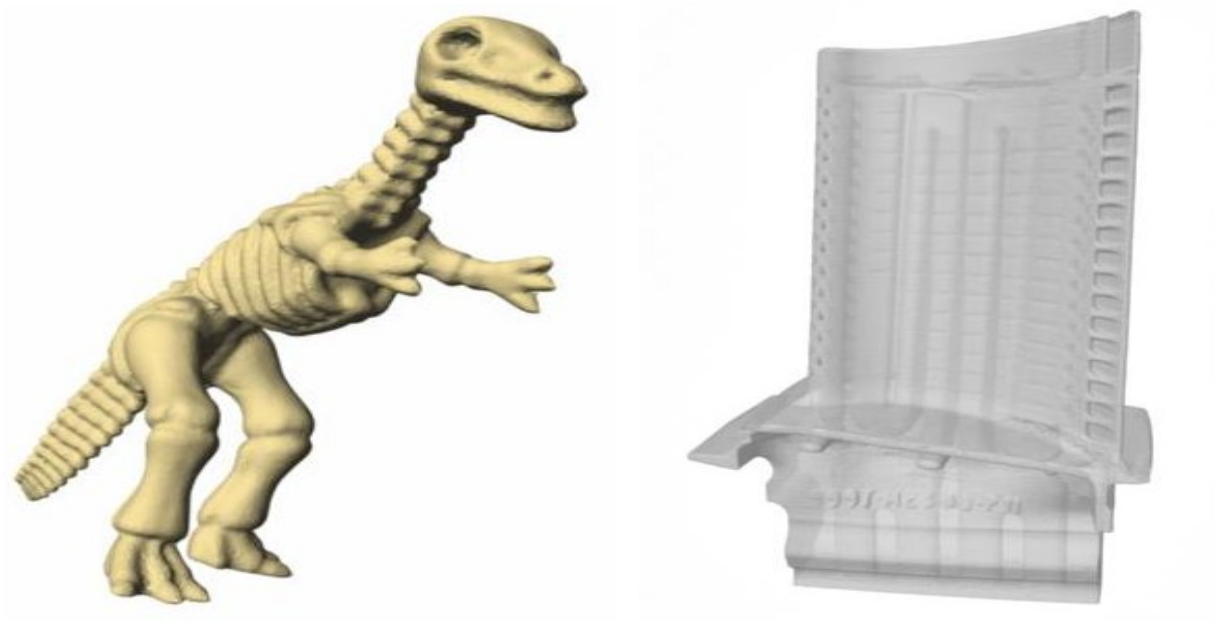
(a) Dental Industry

(b) Medical Industry

Figure 1.3: Applications in Medical and Dental Industry

1.1 Applications

Normal estimation and surface reconstruction have several important applications. Some of the important ones are Aircraft industry, Auto Industry (Figure 1.2) , Turbine Machinery (Figure 1.4(b)), Clothing, Space Exploration, GIS, Art and Archaeology (Figure 1.4(a)), Medical Industry (Figure 1.3(b)) and Dental Industry (Figure 1.3(a)). They are also used in Graphics, Movie making and Gaming.



(a) Archaeology (b) Turbine

Figure 1.4: Applications in Archaeology and Turbine Machinery

CHAPTER 2

Normal Estimation

Normal estimation is a very important problem in Computational Geometry. It is a well-studied problem and the previous implementations can accurately estimate the normals for a point cloud obtained from a smooth surface. However the previous implementations cannot compute the normals for sharp edges. Also they have an upper limit on the cloud size they are able to process. In this chapter we study a technique for fast, accurate normal computation of a point cloud which may contain sharp edges. Our implementation is fast and accurate. It is also memory-efficient. We have been successfully able to operate on datasets containing 200 million points.

2.1 Problem Statement

Suppose we are given a 3D point-set bound by assumptions made in Section 2.2. The normal estimation problem basically requires us to find the normals for all points in 3D. The normals should be close in accuracy to the actual normals for the points on the surface from which the points were scanned.

2.2 Assumptions

The algorithm makes the following assumptions.

Let $\mathcal{M}$ be a manifold and $\mathcal{S}$ be the sampling of $\mathcal{M}$ given to us by the 3d scanner.

1. All the coordinates are integers (after proper scaling).

2. The sampling is uniform with boundaries with undersampling at places.

3. The sampling has no noise or negligible noise.

4. We assume that we know K, a large constant, such that for every point p on $\mathcal{M}$, The smallest ball $\mathcal{B}_p^K$ containing K-nearest neighbors of p (centered at p) in $\mathcal{S}$ has only one component of the manifold inside.

5. The restricted Voronoi neighbors of nn_p on $\mathcal{M}$ are contained in $\mathcal{B}_p^K$ (here nn_p denotes the nearest neighbor of p in $\mathcal{S}$).

6. The K sample points inside $\mathcal{B}_p^K$ are drawn randomly independently and identically distributed according to some unknown probability density function.

2.3 Algorithm

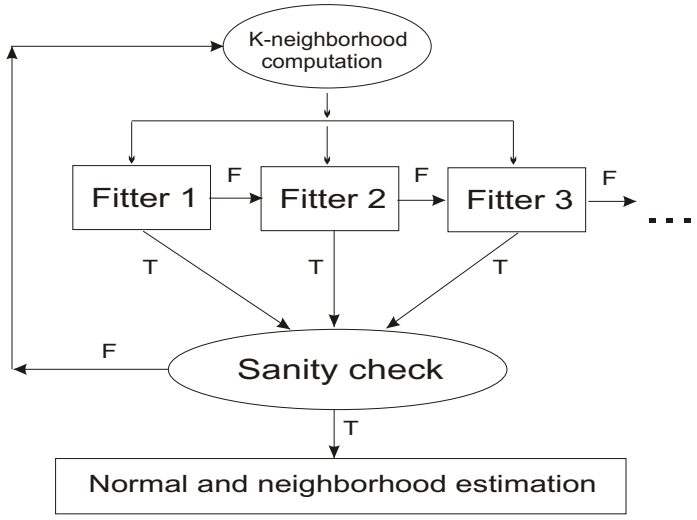


Figure 2.1: Flowchart of normal estimation algorithm

The basic algorithm is shown in Figure 2.1.

1. For every point, generate a set of K nearest neighbors. Use the dynamic nearest neighbor data structure mentioned before for this purpose. Assume, w.l.o.g that our current point p is $(0, 0)$ and all K points are contained in the unit sphere.

2. Check if a function in H fits the neighborhood.

3. Check if the fit makes sense by performing sanity-check. If sanity check fails go to the next function or increase the neighborhood.

4. If sanity-check is successful, compute the normal.

5. If no function in H provides a valid fit, return a no-fit.

For our experiments we used H as the set of functions consisting of one plane, two planes and three planes. We look at each of these fits separately

2.4 One plane

One plane is the simplest shape than can be fitted for a point neighborhood. It can fit most of the points in a dataset. The only places where it may fail is sharp corners or highly curved regions. Also if the points are under-sampled or there is lot of noise, one plane may fail. But since most of our samples were well sampled, we observed that one plane can fit the neighborhood of most of the points in the sample.

2.4.1 2D equivalent case

To understand one plane fitting, lets have a look at the 2D case. Suppose we are given some points on a plane and we want to find the best fitting line. We can do this by computing the covariance matrix of the points. The covariance matrix will have two eigenvectors and eigenvalues which will determine the ellipse representing the distribution of the points. If the points lie along a line, the ellipse will be very thin. As a result, one of the eigenvalue will be very big compared to the other. The eigenvector corresponding to the larger eigenvalue will give the direction of the line and the other eigenvector will be normal to this line in the plane. See Figure 2.2. The larger eigenvector essentially minimizes the sum of the squares of the distances of all the points to the line. This sum can be used as a metric to determine the validity of the fit. Also we can check if the ratio of the larger eigenvalue to the smaller eigenvalue is high. Thus computing the eigenvectors and eigenvalues can help us determine if the points fit a single line.

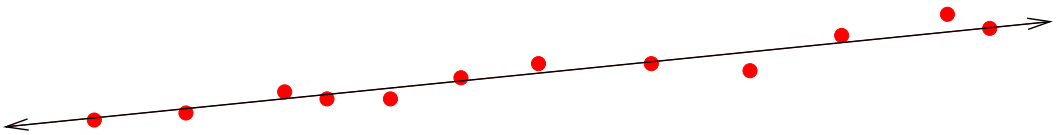
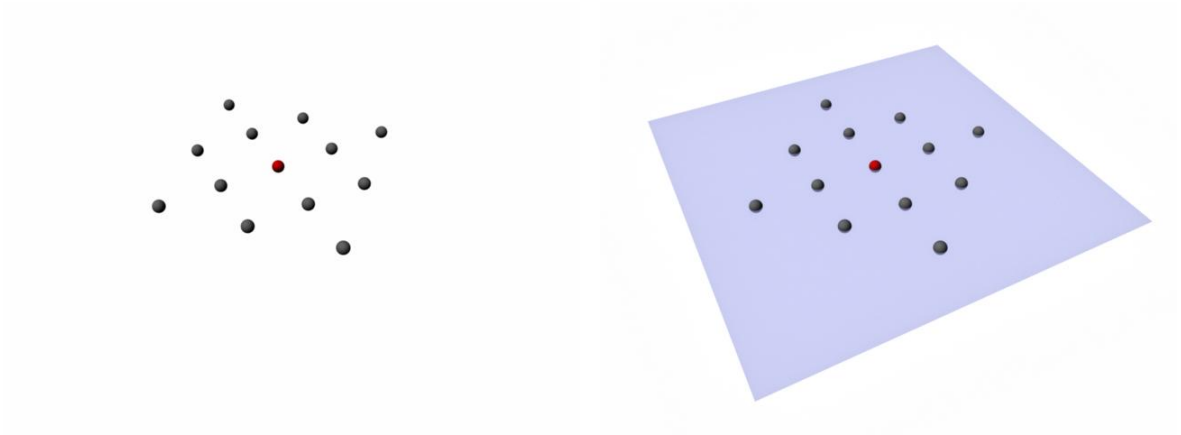


Figure 2.2: Larger eigenvector of the covariance matrix of 2D points that are roughly on a line



(a) A point and its neighborhood (b) Fitting one plane through the neighborhood

Figure 2.3: Fitting one plane

2.4.2 Fitting plane in 3D

We can extend this idea in 3D. In 3D the covariance matrix has three eigenvectors, which determine the ellipsoid representing the distribution of the points in 3D. If the points roughly lie on a plane, the ellipsoid will be very flat. Hence, the smallest eigenvalue will be pretty small compared to the other two eigenvalues. The eigenvector corresponding to the smallest eigenvalue will be along the normal to plane. Once again the plane minimizes the sum of the squares of the distances of the points to the plane. Thus the eigenvalues can help us determine if the points fit one plane. Figure 2.3 shows a point whose neighborhood fits one plane.

We first scale the neighborhood such that it lies within the unit circle with the current

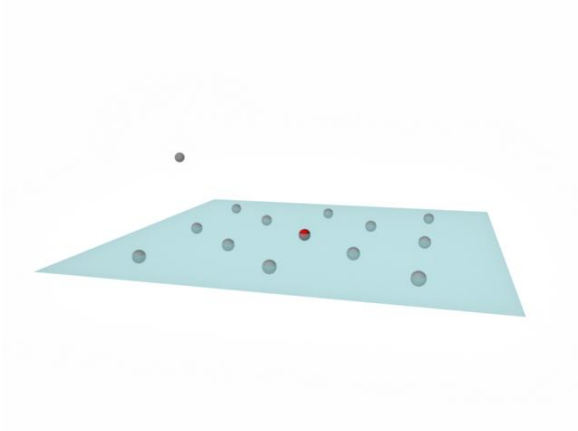


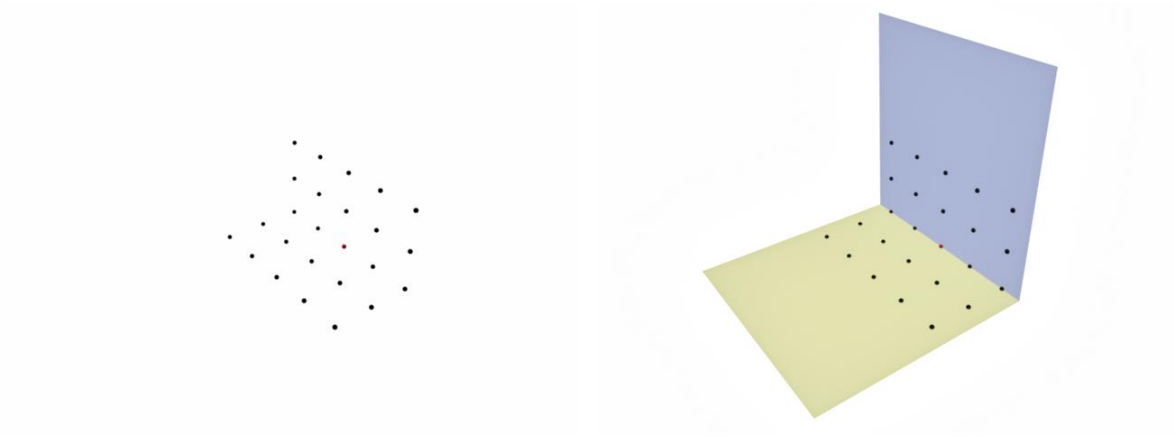
Figure 2.4: Bad fit since one point lies away from the fitted plane

point at the origin. We then compute the covariance matrix and look at its eigenvalues. We declare the fit to be valid if the ratio of smallest to the other two eigenvalues is smaller than the thickness ratio.

It is not enough to just check the ratio of eigenvalues though. For example we may have a case as shown in Figure 2.4. In this case, one point belongs to some other plane but since most of the points lie on one plane, the ratio of eigenvalues might still be small but the normal has small error in it. It also causes problems later on during reconstruction. There are various ways to handle this case.

1. Identify such points as outliers and repeat the fit by ignoring them.
2. Increase the neighborhood so that we can get a better fit. In this case, increased neighborhood may lead to a two plane fitting.
3. Declare a no-fit and try the next function.

Once we determine a one-plane fitting to be valid, the normal is given by the eigenvector corresponding to the smallest eigenvalue. We use Lapack [12] for eigenvalue and eigenvector computation.

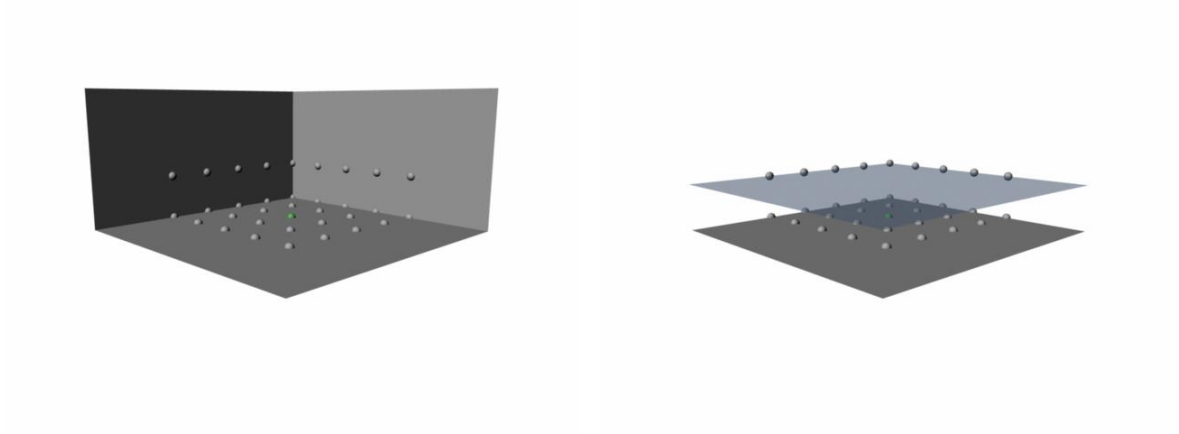


(a) A point and its neighborhood (b) Fitting two planes through the neighborhood

Figure 2.5: Fitting two planes

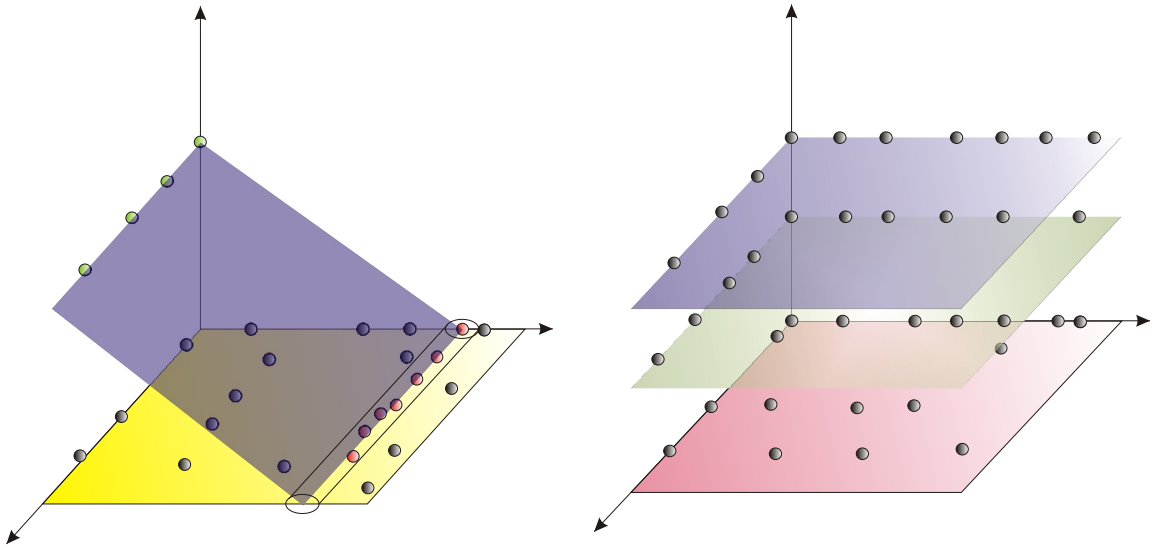
For example the distance of any point in the neighborhood from the fitting function should be less than thickness parameter ρ. If any point doesn't satisfy this condition, we declare that function as an invalid fit and try the next function in H. Even if a fit appears valid, it can lead to wrong normal estimation. For example in a two-plane fit, one of the planes can have only collinear points on it and hence it can be randomly oriented about the line passing through those points or the fitting can consist of two or more parallel planes. We identify such cases by performing a sanity-check on two-plane and three-plane fittings. Sanity-check involves determining whether every plane is well-defined by having at least three exclusive non-collinear points on it and also checking if the line of intersection of the planes is not too far away from the points. If sanity-check fails, we increase the neighborhood to obtain a better fit.

Finally we compute the normal from the fittings. This is trivial for one plane. For multiple planes, we take an average of the oriented normals of the planes close to the point under consideration.



(a) Correct fit (b) Wrong fit

Figure 2.6: Wrong parallel planes fit



(a) Bad two plane fit (b) Three parallel plane fit

Figure 2.7: Wrong fits

2.5 Two Planes

If one-plane fails to fit the neighborhood, we try to fit two planes. We use projective clustering to fit two planes.

2.5.1 Projective clustering

We use Projective clustering [13] to fit multiple planes on a neighborhood. In all our experiments, the approximation factor that we used for projective clustering was 2. In other words, we compute a 2-approximate projective clustering solution. It seemed that 2-approximation is a right balance between speed and accuracy. Better approximations lead to larger running times whereas coarser approximation factors increase the error significantly.

2.5.2 Bad fits

However even if the fitting appears good it may have some problem in it. For example it may fit two parallel planes. This is highly possible for a uniformly sampled dataset. For example see Figure 2.6. Another possible problem happens when the one of the planes is not well-defined. This can happen when the plane has only collinear points on it (again highly possible in a uniformly sampled dataset). In this case the plane can be randomly aligned about the line passing through those points. Thus even though two planes may seem to fit all the points, the fitting is incorrect. Another rare but possible case is shown in Figure 2.7(a). Here even though the points on both planes appear to determine a plane, if we ignore the points on the intersection line, one of the planes is no longer well-defined. To avoid these wrong fits, we perform a sanity-check on the fit.

2.5.3 Sanity Check

The sanity-check algorithm works as follows

1. First determine the line of intersection of the two planes. Then find the distance of the points from this line. If the distance is large, the fitting is like parallel planes and its declared to be failed.

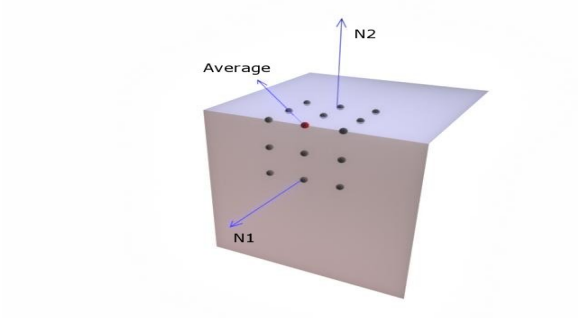


Figure 2.8: Normal computation for two plane fitting

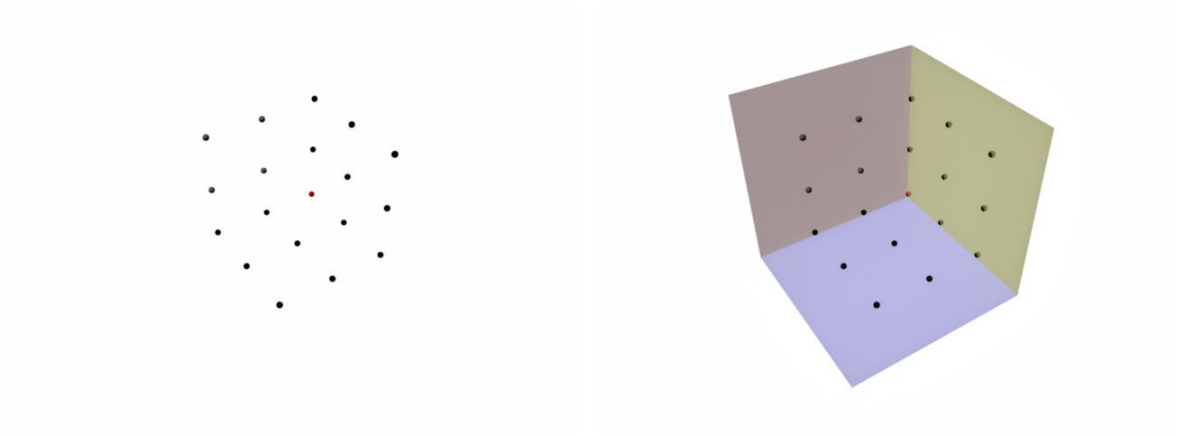
2. If the planes are not parallel, divide the points among the two planes and ignore the points common to both planes. Now look at the points on each plane and see if they can determine a plane. For this we check if the plane has at least three non-linear exclusive points. If any plane doesn't satisfy this, that plane is ignored. If both planes don't satify, the fitting is declared to be failed.

2.5.4 Normal Computation

If the sanity-check returns only one plane, the normal is determined by that plane. If both the planes are valid, then the average depends upon the location of the current point P. If P lies on one of the two planes, the normal at P is the normal of the plane. But if P lies on the intersection line, the normal is an average of the normals of the two planes. We have to be careful while averaging since two lines have two averages perpendicular to each other. We take the average of the oriented normals as shown in Figure 2.8. The location of the point can be determined by the distance of the point to both the planes and this information can be used to determined whether the point is close to one or both the planes.

2.6 Three planes

The three plane fitting is similar to the two plane fitting. It is again obtained by projective clustering. Figure 2.6(a) shows a sample three plane fitting.



(a) A point and its neighborhood (b) Fitting three planes

Figure 2.9: Fitting three planes

2.6.1 Sanity Check

Three planes has similar problems as two plane fitting. For example, two of the three planes may be parallel or all three may be parallel. Also one or more planes may not be well-defined by having only collinear exclusive points on it.

2.6.2 Normal Computation

Again, the normal is determined by computing the distance of the current point P from all the three planes and taking an average of the oriented normals of the planes that are close to the point as shown in Figure 2.10

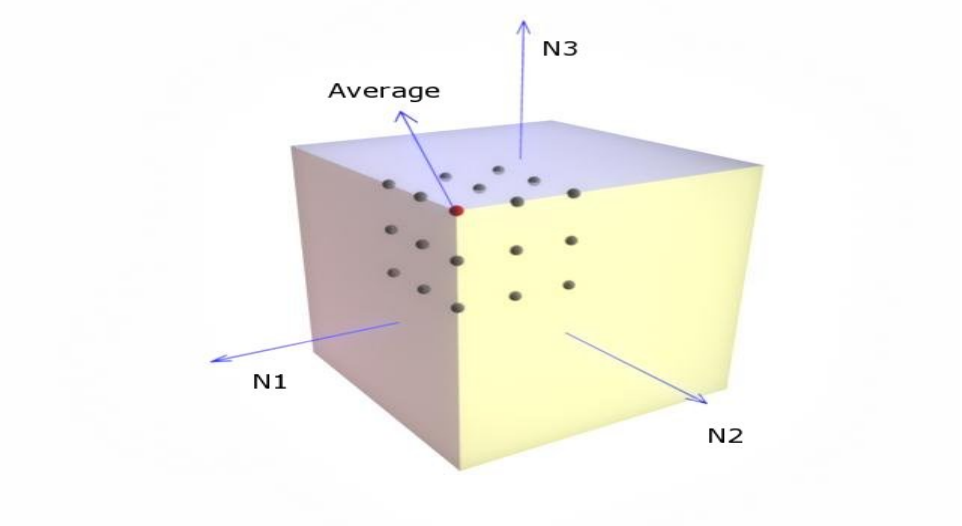


Figure 2.10: Normal computation for three plane fitting

CHAPTER 3

Surface Reconstruction

One of the main practical applications of projective clustering and normal estimation is surface reconstruction. Reconstruction is defined as creating back the surface from the points obtained by scanning the surface. For example in Figure 3, we try to obtain the reconstructed figure on the right from the point-set on the left. For reconstruction, we need to compute the triangles incident on each point. We perform a 2D local Delaunay triangulation on neighborhood of every point and select the triangles incident on the current point p. Doing this for all the points give us a triangulation of the entire dataset.

3.1 Delaunay Triangulation

Delaunay triangulation is a triangulation of a given point-set in which the circumcircle of any three triangle is empty, that is, it does not contain points inside it. Figure 3.2(a) shows a sample Delaunay triangulation We compute the 2D Delaunay triangulation as follows. Again assume w.l.o.g that the current point p is at the origin and its neighborhood is in the unit circle on the X-Y plane. We observe that the Delaunay triangles incident on p are dual to the Voronoi cell 3.2(b) of p. Also, the Voronoi cell of the origin is the convex hull of the inversion. Hence we first compute the dual of the neighborhood. For this, we map every point p_i in the neighborhood to $-p_i/||p_i||^2$. Now the points on the convex hull give us the edges incident on p in the Delaunay triangulation.

Thus we have reduced our problem to finding the convex hull of the dual of the neighborhood. Currently we use [14] for convex hull computation. It first sorts the points in x-coordinate using STL and then computes upper and lower hulls. The complexity of this

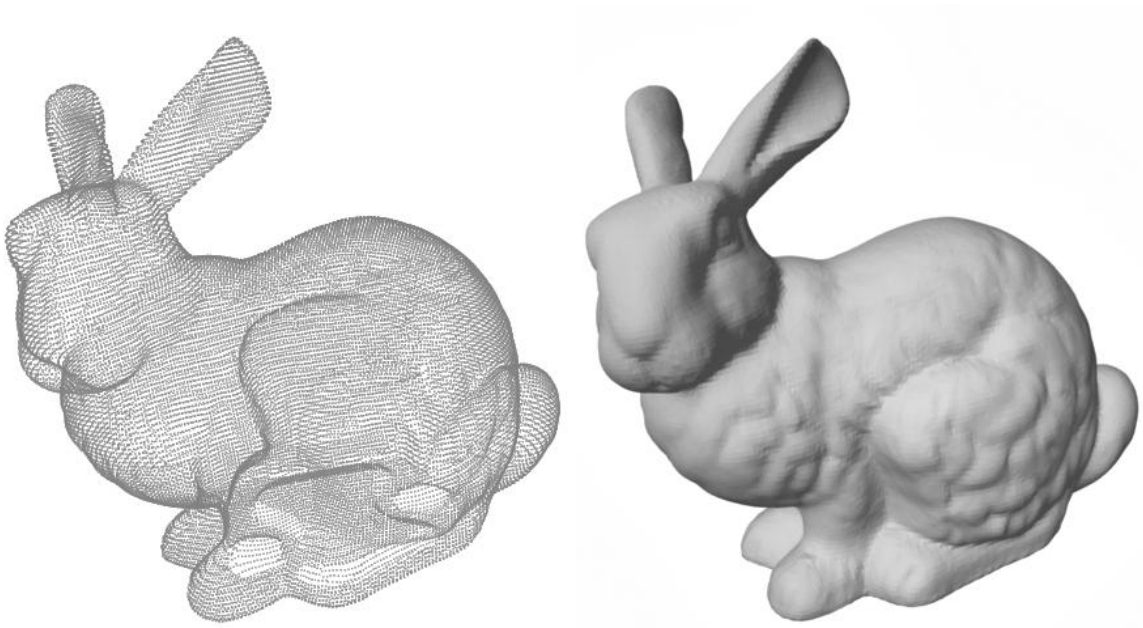


Figure 3.1: Surface Reconstruction

algorithm is $\mathcal{O}(K)$.

We perform local Delaunay triangulation on a neighborhood and output the triangles incident on each point. Once we do this for all the points, we get a triangulation for the entire point-set. Again we study each fitting separately

3.2 One Plane

The triangulation for one plane is very simple. Since the neighborhood already fits a plane, all we need to do is project the points on the plane and use above algorithm.

3.3 Two Planes

For two planes, we open the planes onto one single plane. Basically we rotate one of the planes about the line of intersection till it becomes parallel to the other plane. This can be seen more clearly from Figure 3.6. The nine figures show how the process works. Figure 3.6(a)

(a) Delaunay Triangulation (b) Voronoi Cell

Figure 3.2: Delaunay Triangulation and Voronoi Cell

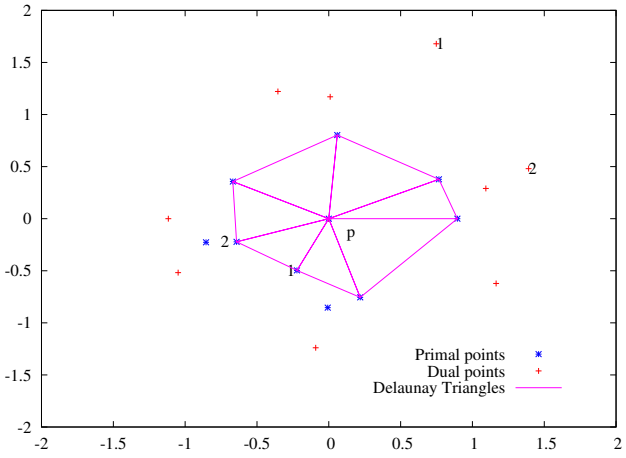


Figure 3.3: Primal-Dual

shows the neighborhood with current point p in the center. Figure 3.6(b) shows the two fitted planes. We then rotate one of the planes till the planes become parallel as in Figure 3.6(d). Figure 3.6(e) shows the Voronoi cell of point p. Figure 3.6(e) shows the triangles in the Delaunay triangulation incident on p.

3.4 Three Planes

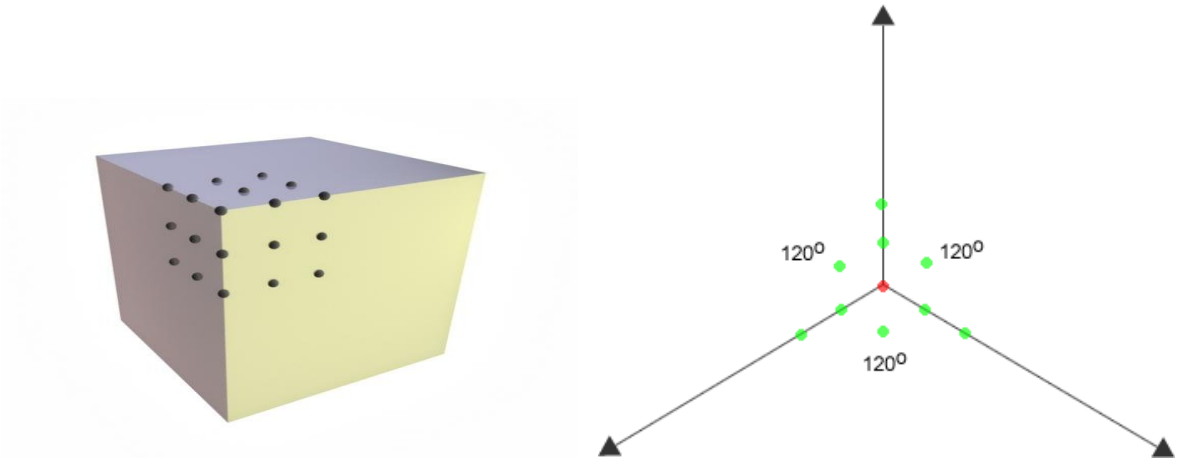
For three planes, its slightly more complicated to open all planes to one single plane and maintain the relative distances. One solution is to project each plane to the plane perpendicular to the normal at the point of intersection of the three planes. This doesnt maintain the relative distances very accurately. Hence we later replaced this with a slightly better algorithm

3.4.1 Three Plane Opening

Each pair of planes gives a line of intersection. We first calculate the angles between each pair of intersection lines. We then stretch each angle in the same ratio such that the sum of the three angles is 360 degrees. The points in each quadrant is scaled too. To find the angle between two planes, we first determine all the points that are on the third plane and determine which all quadrants they occupy out of the four quadrants obtained by the intersection of the two planes. If they occupy only one quadrant, the angle is the angle for that particular quadrant. If they occupy two consecutive quadrants, the angle is 180. If they occupy three quadrants, the angles are summed accordingly. Its impossible to obtain points in all four quadrants for a noise-free dataset under our assumptions.

We consider two examples to understand this better. First one is the corner of a perfect cube. In this case all the three angles will be 90 degrees. The algorithm will stretch each of them to 120 degrees. Then we can obtain the triangulation. See figure 3.4

Another example is when you have three planes having a concave intersection as shown in Figure 3.5. In this case one angle is 270 degrees and the other two are 90 degrees. On



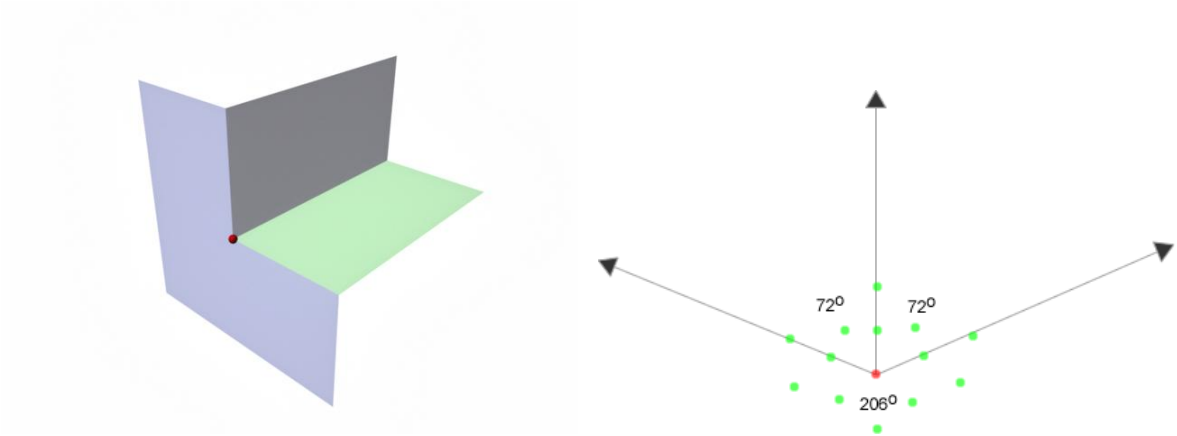
(a) (b)

Figure 3.4: Opening convex three planes

opening, the three angles become 206, 72, 72 degrees respectively.

3.5 Sample Reconstructions

Some of the sample reconstructions created by our program can be seen in Figures 3.7, 3.8, 3.9, and 3.10.



(a) (b)

Figure 3.5: Opening concave three planes

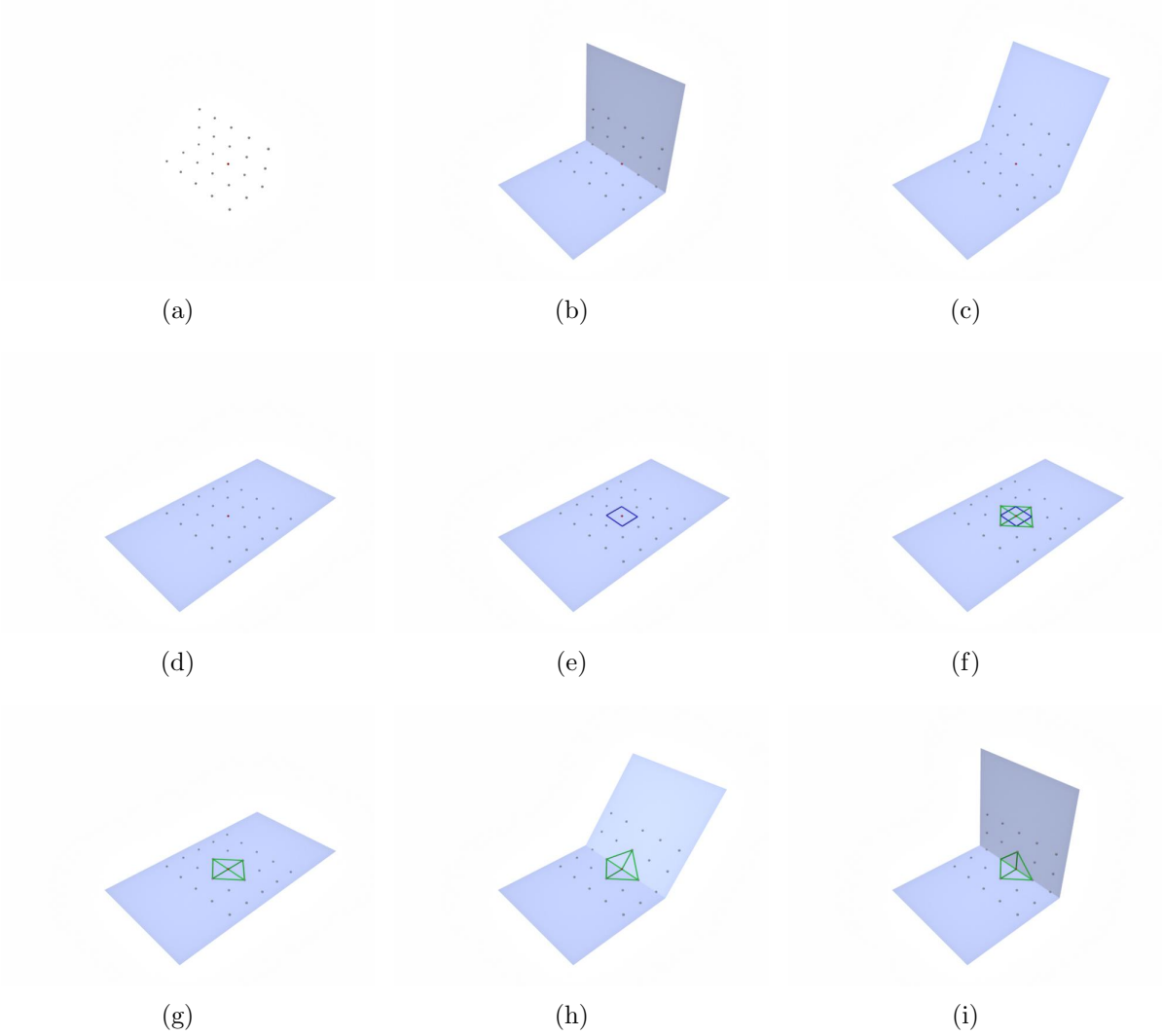
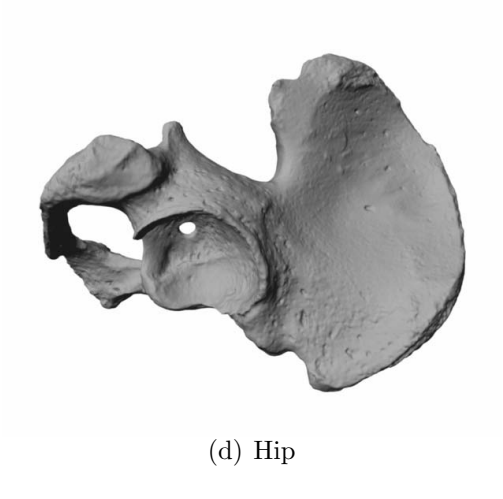


Figure 3.6: Delaunay triangulation on a two-plane fitting

(a) Isis

(b) Rocker

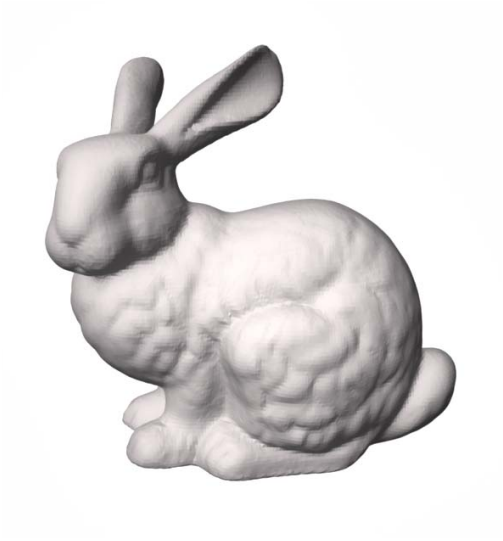
(c) Igea



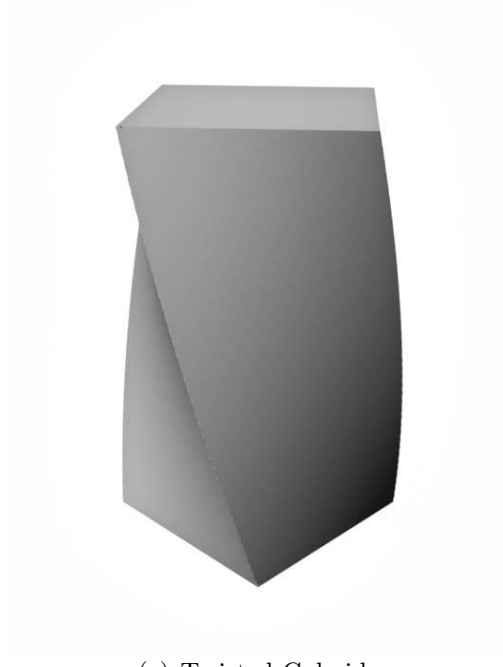
(d) Hip

Figure 3.7: Reconstruction results for some sample data sets.

(a) Hand



(b) Bunny



(c) Twisted Cuboid

(d) Face

Figure 3.8: Reconstruction results for some sample data sets.

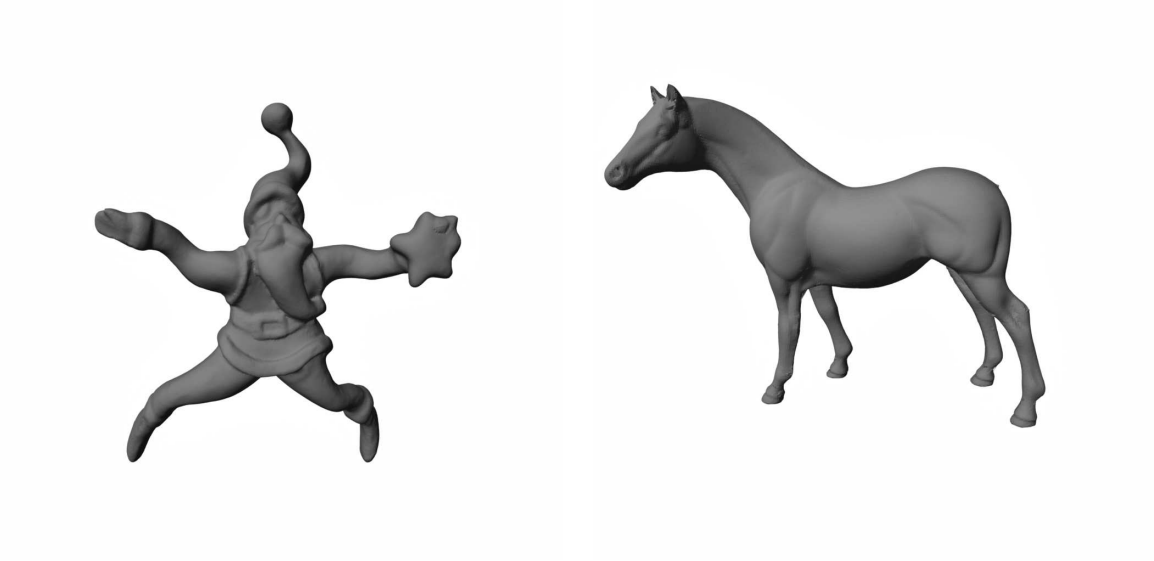
(a) Ball Joint

(b) Club

(c) Dinosaur

(d) Female

Figure 3.9: Reconstruction results for some sample data sets.



(a) Santa (b) Horse

Figure 3.10: Reconstruction results for some sample data sets.

CHAPTER 4

Cube Reconstruction

One of the biggest achievement of the thesis was the successful reconstruction of a cube. We operated on both uniformly sampled cube (perfect cube) and randomly sampled cube (random cube) and were able to obtain good results in both cases. In fact the perfect cube has an average error of 0.079 degrees for normal estimation as seen in Table 7.1. Also there were zero points which were not fitted. On the other hand NormFet [15] could not find the normal for 1028 points out of 5402 and even in the points in which it could find the normal, it had an average error of 7.943. On reconstruction, both the perfect cube and the random cube looked perfect. In the following sections, we will study the normal estimation and reconstruction for various points on these cubes in detail.

4.1 Perfect Cube

The perfect cube was created with equal points on all face, uniformly sampled. The total dataset size was 5402 points. When dealing with a perfect cube, there are basically three main fittings.

4.1.1 One plane

Majority of the points fit one plane. These points are the ones on the face away from any edge. Since they are away from the edge, any neighborhood will consist of perfectly planar points. Hence it can fit one plane easily. The normal in this case is just the normal to the plane and the reconstruction can be done by triangulating as seen in the reconstruction chapter.

4.1.2 Two planes

When you move towards the edges (but away from corners), we are more likely to fit two planes. It depends on the neighborhood, but typically points upto distance 4-5 points from the edge fit two planes. The fitting is done using projective clustering. In most case, if enough points are sampled on both planes, the projective clustering algorithm returns the two planes correctly. There might be some cases when it returns two parallel planes or one plane thats not defined well. In this case sanity-check filters out the bad planes. The normal for the edges is the average of the oriented normals to the two perpendicular planes. At all points away from the edge, the normal is the normal of the plane on which the point lies. The reconstruction is done by opening the two planes and triangulating as seen in the reconstruction chapter.

4.1.3 Three planes

The points close to the corners fit three perpendicular planes. Once again bad fits are filtered by sanity-check. The normal on the corners is the average of the oriented normals of the three planes. The normal on the edge close to the corner is the average of the oriented normals of the two planes creating the edge. The normal on any face is simply the normal of that face. Reconstruction is done by opening the planes as explained in the reconstruction chapter.

Figure 4.2 shows the perfect cube with points colored according to the function they fit. Blue points fit one plane, green points fit two planes and red points fit three planes. The output is generated using a splat viewer. The splat is perpendicular to the direction of the normal.

4.2 Random Cube

The random cube performs the same as the perfect cube for one plane fittings. However the difference happens at the edges and corners. The edges and corners of the random cube are not sampled. However our algorithm of opening the two planes and performing triangulation

Figure 4.1: Renderings of reconstructions generated by our implementation of open and closed cubes.

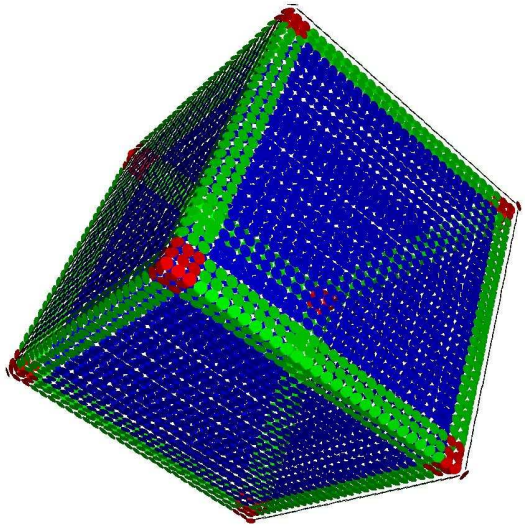


Figure 4.2: Color-coded cube

creates triangles with points on two different face. Hence in the final reconstruction there are no holes.

CHAPTER 5

Speed and Memory Issues

One of the main purpose of this thesis is to be able to perform efficient computation on very big datasets. The computation should be pretty fast but at the same time it should be accurate. Operating on very large dataset has memory issues. For example we can load all the points from a small or even medium sized dataset into memory. But there is no way we can load say 400 million points into memory unless you have very large memory. We did various modifications to handle these issues.

5.1 Speed

5.1.1 Threading

The program is designed to be multi-threaded. Hence it will perform faster on a multiprocessor machine using more than one threads. If there are N points and t threads, each thread operates on N/t points. For example if there are four threads, thread one operates one points 1,5,9 ..., thread 2 operates on points 2,6,10 ... and so on. If the points and randomly arranged, each thread should have roughly the same workload leading to good scaling for the time of operation. The operation of one thread is independent of other threads. We use locks to make sure, that multi-threading works correctly.

5.1.2 Normal Copying

If a lot of neighboring points lie on a plane, they have roughly the same normal. We use this face to avoid a lot of unnecessary computation. For example if the 25 neighbors of a points all lie on a plane, then we simply copy the normal computed for the point to some of the neighboring points. The number of points for which the normals are copied depends upon the distribution and accuracy desired. We can extend this idea to two planes and

three planes fitting too. We can simply determine which planes are close to the point whose normal we want to find and simply calculate the normal from that by averaging if necessary. However we need to be careful not to copy normals for the point which lie on the edge of the neighborhood since they may a fit different function on the other side.

5.1.3 2D Triangulation

Some of the previous approaches were slow because they did 3D triangulation. Our approach to fit shapes and project the points in 2D enables us to use 2D triangulation, which is faster than 3D triangulation.

5.1.4 External Memory Sorting

To sort points whenever we need it, we use a sorting algorithm based on size of dataset. If it is a small dataset, we can use STL sort. But if the dataset is very large, we use an out-of-core sorting algorithm implemented in STXXL [16]. Out-of-core sorting gives much better results than STL sort for datasets which dont fit in memory.

5.2 Memory

5.2.1 Sliding Window

Since it is hard to keep all the points in memory for a very large data, we use a sliding window approach [17]. See Figure 5.1. The width of the window depends on the available memory. For every window instance, we operate on the middle third of the window, which forms the operating window. The main window is wide enough to give the accurate nearest neighbors for every point in the operating window. After we are done processing the middle third, we delete the first third points and insert new points at the end. The points are initially sorted along x-axis and the entire point-set is aligned along x-axis. Thus by moving the window along x-axis, we make sure that the nearest neighbors are present in the main window The need to be able to insert and delete points led us to develop our own dynamic nearest neighbor data structure in place of using ANN [18].

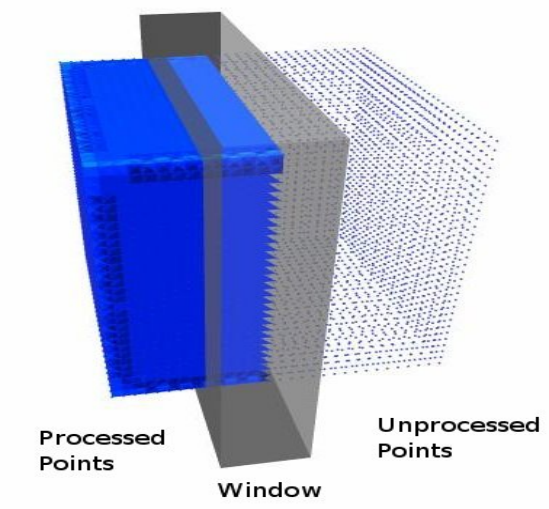


Figure 5.1: Sliding Window

5.2.2 Normals Storage

The normals computed, are stored in disk in a compressed format. Each normal occupies four bytes rather than twelve as required by the uncompressed normals. Each normal is compressed to 4 bytes using a dictionary of eight vectors and a differential to the closest vector in the dictionary. This gave us much better results than just discretizing the sphere. The mean error that compression causes is around .003 degrees. This reduces the time needed to access the disk while reading or writing the normals.

CHAPTER 6

Simplification

One of the focuses of the thesis is handling very large datasets. However even after successfully computing the normals and triangulating the point-set, it is hard to render the figure if the number of points is very high. We observed that on our system, Maya can render upto a million triangles which was much less than some of the large datasets we have. For example St. Matthews has around 200 million points. It is practically impossible to render it with off the shelf software. Hence we use a technique called simplification

The basic assumption for simplifying is that a large number of points in many parts of the dataset fit one single plane. Hence its redundant to store the triangles for all those points. Rather we can delete most of those points and leave only certain *representative* points. The basic requirements for this simplification are as follows.

1. The final number of points should be less than the maximum limit that can be rendered
2. The final point-set should be mostly uniform, else we are likely to get holes in the rendering
3. The deleted points should have the same normal as the point representing them, else the rendering wont be accurate

This forms the basis of our simplification algorithm.

6.1 Simplification Algorithm

The main idea behind the algorithm is to cluster together points which are close by and have the same normals. Suppose we want to obtain a k order simplification i.e the resultant

point-set should have roughly k^{th} fraction of the points of the original point-set. Currently we operate only on points which fit one plane. But since these form the majority of the points, the resultant simplification is roughly the same. The main steps followed by the algorithm are

1. For all points fitting one plane, find the distances between neighboring points in the Morton ordering.

2. We need to get rid of the $(1-k)^{th}$ fraction of the points. Hence we first sort the distances in increasing order and find the k^{th} smallest distance.

3. We then divide all the points into groups by clustering together all points which have the distances between neighboring points smaller than the k^{th} smallest distance.

4. We then further divide each group based on normal for each point in that group. The points with normals roughly the same are put in the same subgroup.

5. Each of this subgroup is then replaced by a point in that group. The resultant points gives the simplified points

6. Finally add the points fitting two and three planes to the simplified point-set

The actual order of simplification obtained is hard to determine since we operate only on points fitting one plane and also it depends on the the overall distribution.

6.2 Results

We tried our algorithm on a BMW dataset which had 1.2 million points. We obtained a 25% simplification and reduced to to 300 thousand points which allowed us to render it in Maya. As seen in Figure 6.1, the simplified dataset looks pretty accurate. We also tried our algorithm on St Matthews dataset which needed a simplification factor of 0.005 to be able to render.

Though the algorithm looks to work well, there is scope for improving it for speed and accuracy. We are currently working on this part.

Figure 6.1: Rendering of the simplified BMW dataset

CHAPTER 7

Results

We performed our experiments on a 4-CPU 852, 2.4GHz opteron processor system with 8GB RAM, SATA RAID 0, with 4 western digital raptor hard-drives(73GB) running 64 bit Fedora Linux. The experiments were performed with one thread and four threads to study the speed-up dues to threading. For studying the accuracy and speed of our experiments, the results were compared with NormFet [15]. The results can be seen in Table 7.1. All timings are in seconds and errors in degrees. All experiments use neighborhood sizes of 25,25,50 for 1,2,3 planes respectively. The thickness parameter was set to .2 for all the experiments except the cube ($\rho = .01$) and FACE ($\rho = .1$). The *FACE* data set consists of 20 three dimensional face meshes generated by a Cyberware scanner. The results presented for *FACE* are average error and total computation time for the 20 data sets.

As seen in the results table, our program does a better job than NormFet in both, speed and accuracy. We computed the average error in the normals and standard deviation given by our program as well as NormFet. Our program gives smaller error than NormFet in most of the cases. We also computed the number of points not fitted. In this case, we performed better than NormFet in datasets having a lot of sharp edges such as the cube. The reason is that, NormFet cannot handle sharp edges and declares them as no-fits. Also we perform much better than NormFet even on single processor. When we use four processors, we perform much faster. Actually NormFet needs the time for only normal computation whereas our time is the time needed for normal computation as well as surface reconstruction. Thus we are much faster than NormFet. Also, we obtain a speedup of around 3 times to 3.5 times when going from one processor to four processors. Its not possible to get a perfect speedup since some of the tasks are not parallelized. But if we ignore these times, we observe that

Table 7.1: Table of results.

DataSet	Size	Our Implementation					NormFet			
		Time (1cpu)	Time (4cpu)	Mean Error	Std. Dev.	No Fits	Time (1cpu)	Mean Error	Std. Dev.	No Fits
Cube	5402	6.228	4.161	0.079	0.079	0	4.298	7.943	8.92	1028
Twisted	230402	127.768	38.796	0.682	3.538	0	150.144	13.379	23.009	0
Dragon	434857	271.806	92.950	3.098	6.075	171	467.780	4.298	6.435	176
Happy	542612	352.370	113.035	4.422	7.836	70	622.724	5.384	7.714	121
Bunny	35947	20.2	6.295	4.906	11.373	0	16.619	6.286	12.018	5
Hand	327323	174.454	52.714	2.495	3.602	5	1253.925	3.945	5.078	18
Rocker	40177	23.712	6.984	2.362	3.803	0	35.805	3.462	4.926	2
Isis	187644	121.300	35.581	2.013	3.432	0	246.902	2.657	3.897	5
Screw	27152	17.301	5.671	2.356	4.421	0	24.483	3.631	5.051	1
Hip	530168	348.954	101.973	2.652	3.790	5	567.599	3.398	4.597	44
Blade	882954	791.828	477.884	3.618	7.799	3019	3665.952	5.751	8.787	958
Dinosaur	56194	41.622	15.563	4.648	6.660	10	45.102	6.820	8.503	78
Santa	75781	43.826	12.733	2.162	3.037	0	69.070	3.757	4.764	1
Igea	134345	82.995	24.539	2.061	3.097	0	100.952	2.669	3.697	8
Club	209779	114.806	37.368	1.084	2.063	8	392.375	1.336	2.299	77
Female	309735	448.157	329.380	5.079	10.838	1843	270.120	5.026	8.847	3453
Horse	48485	26.181	8.045	2.396	4.927	0	35.107	3.957	5.872	13
BallJoint	137062	86	24.489	2.576	3.399	0	99.649	3.189	4.153	3
FACE	1336927	405.804	214.516	2.283	6.809	253	978.296	3.986	8.441	3903

we obtain a pretty good speedup. In the future, we plan to work on supercomputers with a huge number of processors. This will lead to very fast computation which is very important for very big datasets.

CHAPTER 8

Conclusions and Future Work

The thesis work provided a method to perform fast and memory-efficient normal estimation and surface reconstruction of point clouds which can be very big. Using multi-threading it may be possible to obtain a huge speedup. The reconstructions created by our program looked pretty good and accurate. Also we were successfully able to reconstruct a cube which required fitting two planes and three planes at certain points. There is much scope for improvements. Some of them are as follows.

1. Our program currently doesn't handle noise and doesn't remove outliers. Though a lot of current scanners give pretty good noise-free datasets, it would be useful to have a program that can remove points created due to noise.

2. We expect the user to provide the initial neighbor count. There is a way to avoid this. We can make the program gradually increase the value of the parameter by looking at the neighbors and determining if they are enough.

3. We also expect the user to provide the thickness parameter. There should be a way of determining this automatically depending on sampling density.

4. The current multi-threading just assigns equal points to all processors beforehand which may not be the best way of doing it. Another approach could be assigning a new point to a thread only after it gets free. But this has the extra overhead of a centralized table and the producer-consumer problem. But it may still lead to some speedup.

5. The external memory sorting relies on STXXL [16]. We need to implement our own external memory sorter so that we dont have to rely on other libraries.

6. The nearest neighbor data structure has a scope of improvement. Speeding up searches, insertions and deletions will speed up the program.

7. We still haven't figured out a way to open spheres and quadrics on a plane such that the relative distances are preserved. Hence we cannot use these functions for reconstruction. We need a way to figure this out.

8. When need to add SVM to the set of fitter H as the final alternative. A lot of complex surfaces that do not fit any of the current functions can be fitted by SVM.

APPENDIX A

Nearest Neighbor Search

An important sub-problem for normal estimation is nearest neighbor search. This is again a well-studied problem with some very fast implementations such as ANN [18].

A.1 Problem Statement

The problem is defined as follows. Given some points in 3D. Find the nearest neighbors for a point in 3D from the given points. The search should be accurate and fast. Also our normal estimation algorithm requires dynamic nearest neighbor search. This means we need to have a data structure which allows us to do insertions and deletions of 3D points dynamically. This is the main reason we cannot use ANN [18] which cannot perform dynamic insertions and deletions.

A.2 Nearest Neighbor Data Structure

We use a dynamic approximate nearest neighbor search data structure based on Chan's algorithm [19].The algorithm is easy to implement and has been used before [20] without guarantees. The basic algorithm is pretty simple. It keeps $d+1$ shifted quad-trees in d dimensions (see figure A.1 courtesy M. Connor). To compute $K-$nearest neighbors, we first compute $\mathcal{O}(K)$ closest points in the Morton ordering of each of the shifted quad-trees and then pick the best $K-$neighbors from this set.

We use a combination of deterministically and randomly shifted quad-trees stored inside dynamic red-black trees to produce $(1+\epsilon)$-approximate nearest neighbors. Our implementation is thread safe. In practice we found out that random rotations with random shifts

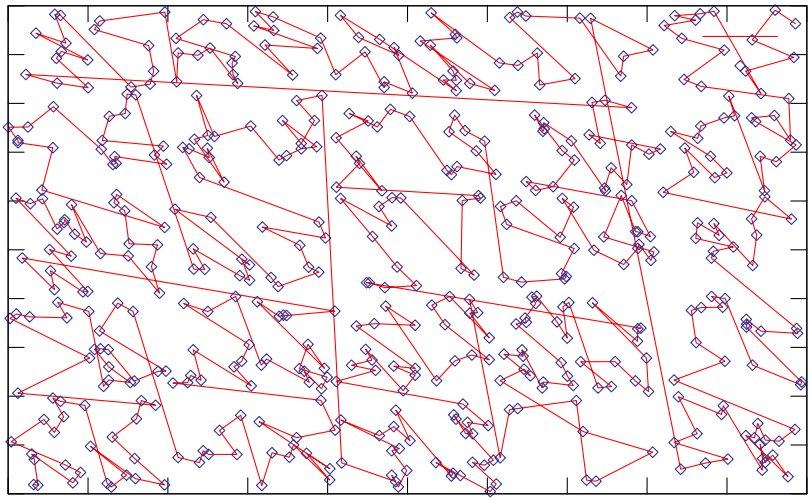


Figure A.1: One instance of our shifted quad-tree

do better on average than only random/deterministic shifts and hence we use both of these in our implementation. Our current implementation for 3D does three deterministic shifts and one random shift. We are still working on speeding up this data structure and plan to release it for public use in the future. Note that our current implementation assumes that points are integers.

REFERENCES

[1] F. Cazals and M. Pouget. Estimating differential quantities using polynomial fitting of osculating jets. *Comput. Aided Geom. Des.*, 22(2):121–146, 2005. 1

[2] Niloy J. Mitra and An Nguyen. Estimating surface normals in noisy point cloud data. In *SCG '03: Proceedings of the nineteenth annual symposium on Computational geometry*, pages 322–328, New York, NY, USA, 2003. ACM Press. 1

[3] Nina Amenta, Sunghee Choi, Tamal Dey, and Naveen Leekha. A simple algorithm for homeomorphic surface reconstruction. *International Journal of Computational Geometry and its Applications*, 12(1–2):125–1141, 2002. 1

[4] Nina Amenta and Marshall Bern. Surface reconstruction by Voronoi filtering. *Discrete and Computational Geometry*, 22:481–504, 1999. 1

[5] D. Levin. Mesh-independent surface interpolation. *Geometric modelling for scientific visualization*, pages 37–49, 2003. 1

[6] M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, D. Levin, and C. T. Silva. Point set surfaces. *IEEE Visualization 2001*, pages 21–28, October 2001. ISBN 0-7803-7200-x. 1

[7] Ravikrishna Kolluri. Provably good moving least squares. In *SODA '05: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1008–1017, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics. 1

[8] Jean-Daniel Boissonnat and F. Cazals. Smooth surface reconstruction via natural neighbour interpolation of distance functions. In *SCG '00: Proceedings of the sixteenth annual symposium on Computational geometry*, pages 223–232, New York, NY, USA, 2000. ACM Press. 1

[9] Mark Pauly, Richard Keiser, Leif P. Kobbelt, and Markus Gross. Shape modeling with point-sampled geometry. *ACM Trans. Graph.*, 22(3):641–650, 2003. 1

[10] Piyush Kumar and Amit Mhatre. Fast and accurate normal estimation of point clouds. *Submitted to Eurographics*, 2006. 1

[11] Amit Mhatre and Piyush Kumar. Projective clustering and its application to surface reconstruction. In *SCG '06: Proceedings of the twenty-second annual symposium on Computational geometry*, 2006. 1

[12] LAPACK. http://www.netlib.org/lapack/. 2.4.2

[13] P. Kumar and P. Kumar. Almost optimal solutions to k-clustering problems. Unpublished, submitted to cgta, Florida State University, 2005. 2.5.1

[14] A. M. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, 9(5):216–219, December 1979. 3.1

[15] T. K. Dey and J. Sun. Normal and feature estimations from noisy point clouds. Osucisrc-7/50-tr50, Ohio State University, 2005. 4, 7

[16] Roman Dementiev and Peter Sanders. Asynchronous parallel disk sorting. In *Proceedings of the 15th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA-03)*, pages 138–148, New York, 2003. ACM Press. 5.1.4, 5

[17] Renato Pajarola. Stream-processing points. In *Proceedings IEEE Visualization, 2005, Online.* Computer Society Press, 2005. 5.2.1

[18] D. Mount. ANN: Library for Approximate Nearest Neighbor Searching, 1998. `http://www.cs.umd.edu/~mount/ANN/`. 5.2.1, A, A.1

[19] Timothy M. Chan. Closest-point problems simplified on the RAM. In *Proceedings of the 13th Annual ACM-SIAM Symposium On Discrete Mathematics*, pages 472–473, New York, January 2002. ACM Press. A.2

[20] J. Shepherd, X. Zhu, and N. Megiddo. A fast indexing method for multidimensional nearest neighbor search, 1999. A.2

BIOGRAPHICAL SKETCH

Amit N. Mhatre

Amit Mhatre was born in Bombay, India in 1981. He did his B.Tech from the Indian Institute of Technology (IIT), Bombay graduating in 2003. He came to US in December 2003 to pursue M.S in Computer Science at FSU. He is currently working towards the completion of his M.S in Spring 2006.

Amit has been on the Dean's list and President's list for all his semesters at FSU. He is a member of the all-discipline honor society Phi Kappa Phi and the Computer Science honor society Upsilon Pi Epsilon.