Group_9 Assignment 2

3D Computing Part B: Surface Modeling

Karim Daw, Shervin Azadi, Daniel Beran

Abstract:
The objective of this assignment was to investigate different methodologies in surface modeling. The investigation started off with specifically exploring quadrangular and triangular surface meshes, and as a result, estimating the normals of all the vertices's of said meshes. After which we used a different methodology for estimating these normal vectors by using Principal Component Analysis which involved using the given point cloud as a direct input for finding the presumed normal vectors without generating a mesh surface to begin with.

Introduction:
The goal of this two fold investigation would be to create a better understanding of the advantages and disadvantages of either methods in modeling surfaces and estimating the normal vectors. All illustrations and pseudo code will be presented in this paper as a set of figures.

Problem Statement:
The challenge of this assignment boils down to finding the best methodology for estimating the normals of all these points with the goal of accurately reconstructing the point cloud into a correct a semantic model using said normals. Thereby we create a more accurate categorization of what is a roof,wall,tree etc. Henceforth by investigating different methodologies, e.g quadrangulation, or PCA methods, we can conclude which method is best for this reconstruction problem.

Methodology:
The main limitation with the given LIDAR point cloud is the fact that the most reliable information we can use are the points equivalent to the 'top surface' such as building roofs and unobstructed ground planes.

Step1. Finding Top Layer of voxel centroids:

By using the voxelated point cloud of the previous assignment, we created an algorithm that extracts the 'top surface' of these voxels using a posgres data base and SQL queries. (figure 1)

```
#Input: our previous data_model which consists of all the centroids of the voxels of the rasterized point cloud
#Output: x,y,z position of the 'top most layer' of voxels
#Implemented in python using psycog2 and SQL

cur.execute('CREATE TABLE TopVoxels (u int, v int, w int);')
cur.execute('INSERT INTO TopVoxels (SELECT i,j, max(k) from centroid_of_voxels GROUP BY i,j);')

cur.execute('SELECT DISTINCT * FROM TopVoxels;')
data_model_list = cur.fetchall()
```

(figure 1)

Step2. Quadrangulation with 'Top Voxels':

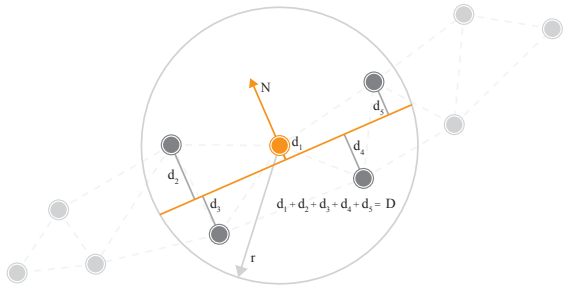
With the 'top layer' voxels, we used their centroid as the vertices of our quad mesh (figure 2). The rg,Mesh class in the rhino SDK requires a set of points that are both ordered and lacking in any 'gaps'. Therefore we had to implement auxiliary scripts that would remedy these issues before the mesh can be successfully created.

```
#Input: x,y,z positions of the 'top most layer' of voxels
#Output: 3d Quadrangulated Mesh
#Implemented in ironpython GH rhino & Vex Houdini

point3d = sorted(original_points , key=lambda k: [k[0], k[1], k[2]])

v = max_x - min_x + 1
u = max_y - min_y + 1
w = max_z - min_z + 1
point3d[i] = rg.Point3d(projected_point3d[i][0], projected_point3d[i][1], zsum/n)

Mesh = rg.Mesh()
for k in range (len(point3d)):
    Mesh.Vertices.Add(point3d[k])

for k in range(int((u-1)*(v-1))):
    v0 = int(k/(u-1))
    n0 = int(v0 * u + k - v0 * (u-1))
    n1 = int(n0 + 1)
    n2 = int(n1 + u)
    n3 = int(n0 + u)
    Mesh.Faces.AddFace(n0,n1,n2,n3)
```

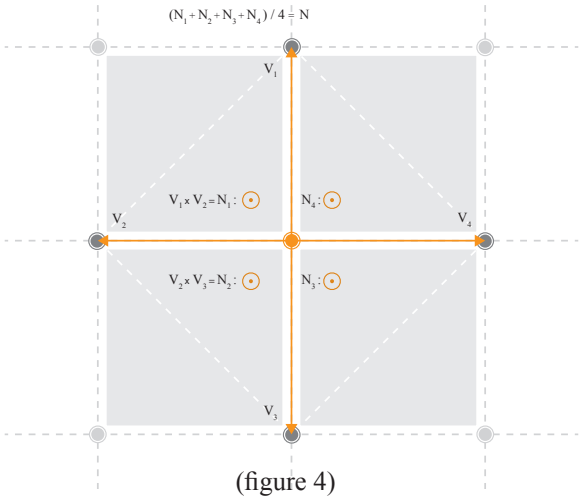
(figure 2)

Step3. Normal estimation of Quadrangular Surface

The problem with calculating the normal vectors for a quadrangular mesh faces lies in the fact that there is no guarantee in this model consisting of planar mesh faces. Therefore we approached this issue in two ways: firstly, calculating a 'best fit' plane through a set of neighboring vertices determined by a radius or 'threshold'(figure 3). Secondly, given that each vertex shares four mesh faces, we can compute the cross product between the vectors that are sharing an edge on each of these mesh faces, eg. V1 and V2. By averaging these 4 vectors we can have an estimation of the normal in this vertex (figure 4).



(figure 3)



(figure 4)

Step4. Delaunay Triangulation with LIDAR pointcloud

Given a C# code for implementing Delaunay triangulation surface, we converted the code into python to better understand the algorithm and to implement it into the same work flow of finding the normals of the vertices. The calculation for the normal vector of TIN was done with the 2nd algorithm in Step3 (figure 4).

Step5. Estimate normal vector using PCA method

a. Neighborhood

The first step was establishing the neighborhoods with two strategies, KNN and FDN. (figure 5)

```
#Input: Lider Point cloud as CSV file and the 'radius' for the FDN method and K-value for the KNN method
#Output: A local neighborhood as a pointcloud
#Vex SDK for pcopen method: http://www.sidefx.com/docs/houdini/vex/functions/pcopen.html
int Neigh = pcopen(0, "P", @P, radius, K);   #@means calling the specific attribute in this case 'P' or position
i@NeighCount = pcnumfound(Neigh);
```

(figure 5)

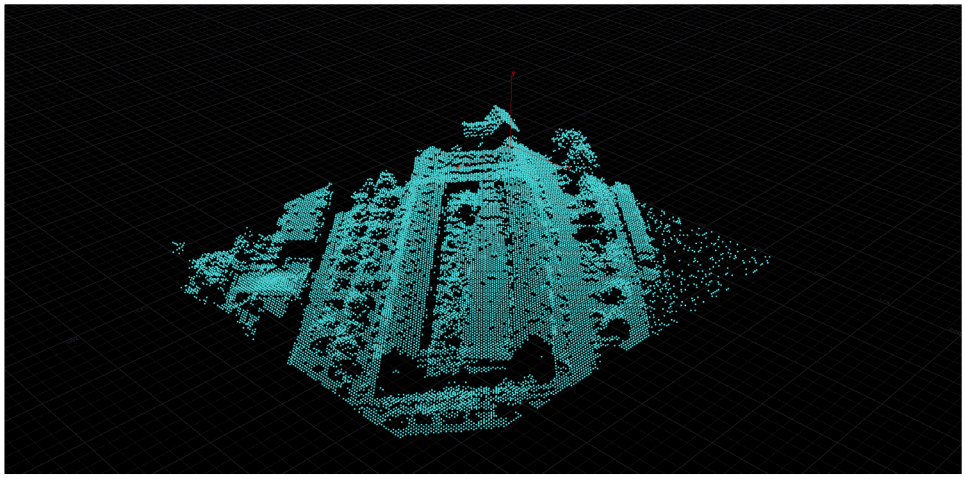
b. Covariance Matrix and Normal computing

After establishing the KNN and FDN neighborhoods, we computed the normal vectors using the cross product between the 'x row' of the covariance matrix and the 'z row'(Y is equivalent to the vertical direction in Houdini) which gave us the normal vector. (figure 6)

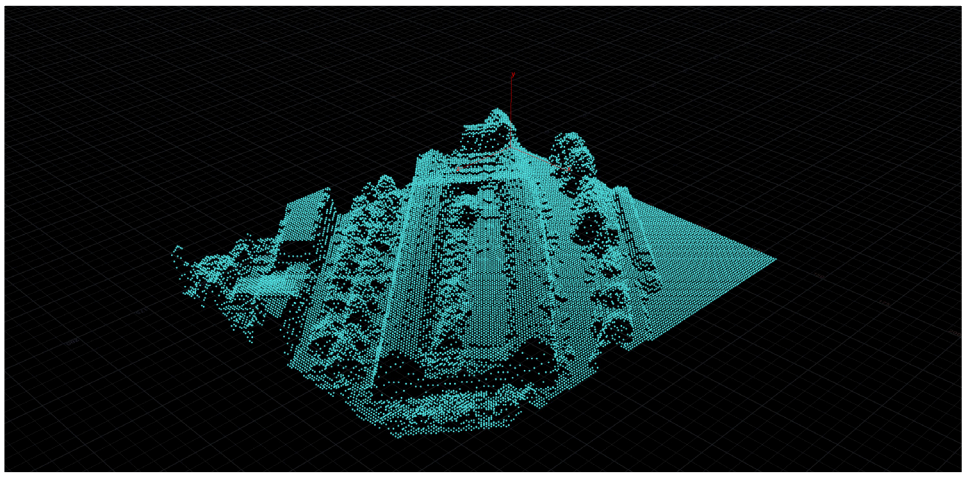
```
#Input: An array of vectors from centroid of neighborhood to neighbors and number of neighbors
#Output: Normal Vectors
#below is a loop that computes this algorithm for every point
vector TransMultAvrg(vector VecArr[], NeighCount)
{
    matrix3 CovM;
    vector vtemp[];
    vector mintemp;
    for (int k=0; k<NeighCount; k++)
    {
        vtemp[0] += set(VecArr[k].x * VecArr[k].x, VecArr[k].x * VecArr[k].y, VecArr[k].x * VecArr[k].z);
        vtemp[1] += set(VecArr[k].y * VecArr[k].x, VecArr[k].y * VecArr[k].y, VecArr[k].y * VecArr[k].z);
        vtemp[2] += set(VecArr[k].z * VecArr[k].x, VecArr[k].z * VecArr[k].y, VecArr[k].z * VecArr[k].z);
    }
    CovM = set(vtemp[0],vtemp[1],vtemp[2]);
    CovM = CovM / (NeighCount - 1);                  #calculating the covariance matrix
    vector real;
    vector imaginary;
    int i;
    eigenvalues(i, CovM, real, imaginary);           #calculating the eigen values

    vector cross = cross(vtemp[2],vtemp[0]);         #calculating the cross product of zvector and xvector


   return cross;
}
```

(figure 6)

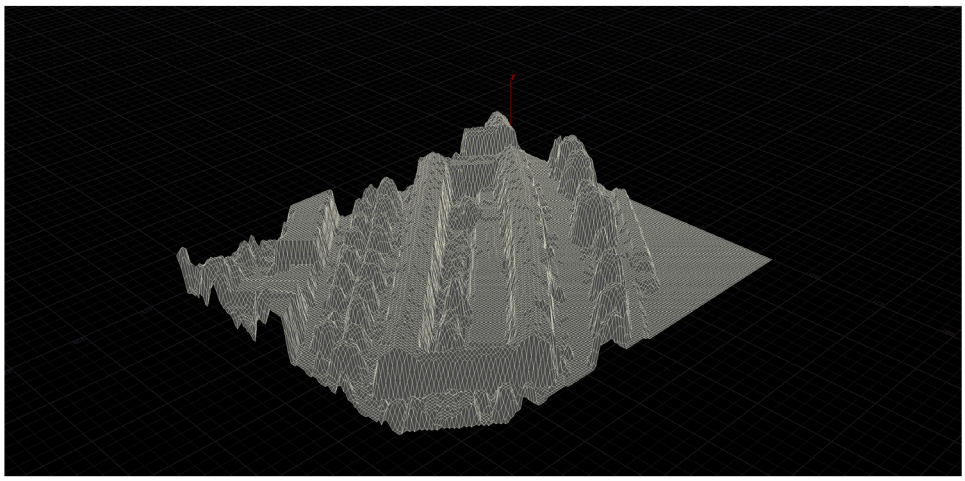
Appendix:



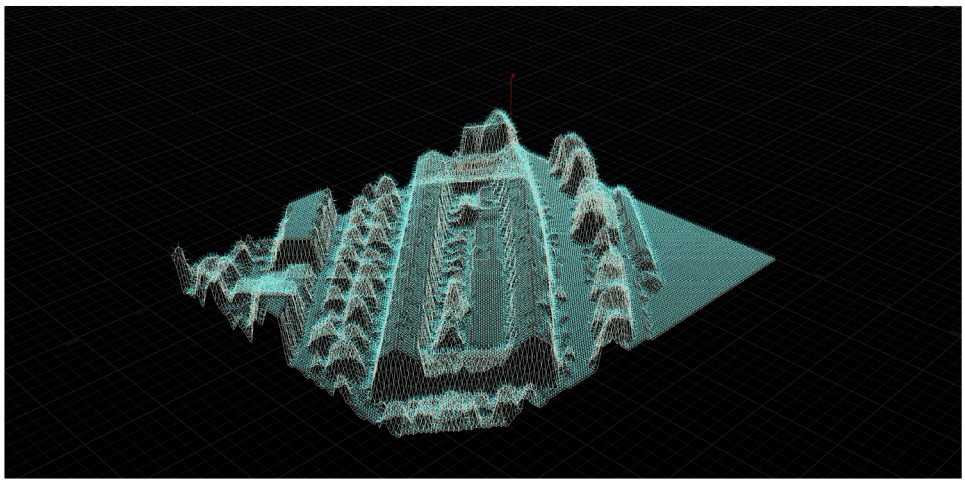
Raw input with faults such as gaps in point cloud and an-sorted points



Cleaned up point clouds where gaps are filled and points are sorted



Quadrangulated surface reconstruction with ‘topvoxels’



Normal Vector calculations exhibited in light blue