4-2-2001

Management, retrieval, and visualization of spatial data from airborne light detection and ranging system (LIDAR) survey

Zheng Cui
Florida International University

DOI: 10.25148/etd.FI14061561

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

MANAGEMENT, RETRIEVAL, AND VISUALIZATION OF SPATIAL DATA FROM AIRBORNE LIGHT DETECTION AND RANGING SYSTEM (LIDAR) SURVEY

A thesis submitted in partial fulfillment of the

requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Zheng Cui

2001

To: Dean Arthur W. Herriott
College of Arts and Sciences

This thesis, written by Zheng Cui, and entitled Management, Retrieval, and Visualization of Spatial Data from Airborne Light Detection and Ranging System (LIDAR) Survey, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this thesis and recommend that it be approved.

Keqi Zhang

Wei Sun

Shu-Ching Chen, Major Professor

Date of Defense: April 2, 2001

The thesis of Zheng Cui is approved.

Dean Arthur W. Herriott
College of Arts and Sciences

Dean Douglas Wartzok
Graduate School

Florida International University, 2001

DEDICATION

I dedicate this thesis to my parents. Without their understanding, support, and all of love, the completion of this work would not have been possible.

ACKNOWLEDGMENTS

I wish to thank Dr. Shu-Ching Chen, my major professor, who gives me not only the proper academic guide but also much personal help through my master studies. From whom, I learned how to make a proper attitude at the research, how to explore my potential and how to fulfill the goals I have made. No exact words can express my appreciation to Dr. Chen. I wish to thank Dr. Wei Sun for taking the time reviewing my thesis and for his helpful comments. I would like to thank Dr. Keqi Zhang for the efforts to evaluate my thesis and for providing the excellent opportunity to work in the International Hurricane Center.

I am grateful to my wife, Yue, who gives me her entire support, dedication, and love. I personally thank Mr. Lixin Huang for providing me with very helpful comments. Also, I would like to take this opportunity to thank cordially everyone who supported me throughout the years at Florida International University.

ABSTRACT OF THE THESIS

MANAGEMENT, RETRIEVAL, AND

VISUALIZATION OF SPATIAL DATA FROM

AIRBORNE LIGHT DETECTION AND RANGING

SYSTEM (LIDAR) SURVEY

by

Zheng Cui

Florida International University, 2001

Miami, Florida

Professor Shu-Ching Chen, Major Professor

The primary purpose of this research was to develop new methodologies to process and analyze large amount of topographic data from airborne LIDAR (light detection and ranging) survey.

This research developed a suite of algorithms to resample dense clouds of point data from LIDAR survey, cut the large data set into smaller tiles, and filtered data to remove points from non-ground surface features such as vegetations, buildings, and vehicles. These algorithms were implemented on the PC platform using C++. The test results showed that the developed application software package based on these algorithms worked well. This application software package provided users an efficient way to retrieve, analyze, and display large volumes of LIDAR survey data, and to extract topographic information.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1.1 Introduction of Airborne Laser Mapping

1.1.1 What is Airborne LIDAR

Light is scattered and attenuated by molecules, aerosols (dust), and cloud (water or ice) particles in the atmosphere. The sky can be clear and blue or hazy and white. Red sunsets are a beautiful manifestation of the scattering and attenuation of sunlight. Clouds can appear white, gray, or dark depending on conditions. The rainbow and ice-particle displays like sundogs and light pillars are less frequent. Light scattering and attenuation can be used to investigate the atmosphere using a remote-sensing instrument called a LIDAR (Light Detection and Ranging System).

A LIDAR system uses laser pulses to measure atmospheric constituents such as aerosol particles, ice crystals, water vapor, or trace gases (e.g. ozone). Profiles of these atmospheric components as a function of altitude or location are necessary for weather forecasting, climate modeling, and environmental monitoring.

Light Detection and Ranging System (LIDAR) uses a laser beam to bounce between the aircraft and the ground, assessing distance from the camera to the ground at set points. The laser beams pierce vegetation, allowing measurement to be conducted during previously prohibitive times of the year when vegetation is thick. However, besides possibly expanding the flying season in some areas, the accuracy of ground measurement increases tremendously with the use of this new technology. LIDAR will help to support many of the topographic mapping needs of our diverse client base.

LIDAR technology allows year-round aerial mapping, and unlike traditional requirements of perfect weather, an aircraft equipped with LIDAR can operate in less than ideal weather conditions including night flights. This can have a significant effect on turnaround time for a project. Less than perfect weather grounds traditional aerial photography crews because of the need of strong overhead sunshine.

1.1.2 Airborne laser mapping

Airborne laser mapping is an emerging technology in the field of remote sensing that is capable of rapidly generating high-density, geo-referenced digital elevation data with an accuracy equivalent to traditional land surveys but significantly faster than traditional airborne surveys.

Airborne laser mapping offers lower field operation costs and post-processing costs compared to traditional survey methods. Point for point, the cost to produce the data is significantly less than other forms of traditional topographic data collection making it an attractive technology for a variety of survey applications and data end-users requiring low cost, high-density, high accuracy geo-referenced digital elevation data.

Airborne laser mapping use a combination of three mature technologies: Light Detection and Ranging (LIDAR), highly accurate inertial reference systems (IRS) and the global positioning satellite system (GPS). By integrating these subsystems into a single instrument mounted in a small airplane or helicopter, it is possible to rapidly produce accurate digital topographic maps of the terrain beneath the flight path of the aircraft.

The absolute accuracy of the elevation data is 15 cm; relative accuracy can be less than 5 cm. Absolute accuracy of the XY data is dependent on operating parameters such as flight altitude but is usually 10's of cm to 1 m.

The elevation data is generated at 1000s of points per second, resulting in elevation point densities far greater than traditional ground survey methods. One hour of data collection can result in over 10,000,000 individually geo-referenced elevation points. With these high sampling rates, it is possible to rapidly complete a large topographic survey and still generate DTMs with grid spacing of 1 m or less.

The technology allows for extremely rapid rates of topographic data collection. With current commercial systems it is possible to survey one thousand square kilometers in less than 12 hours and have the geo-referenced DTM data available within 24 hours of the flight. A 500-kilometer linear corridor, such as a section of coastline or a transmission line corridor, can be surveyed in the course of a morning, with results available the next day.

Airborne laser mapping instruments are active sensor systems, as opposed to passive imagery such as cameras. Consequently, they offer advantages and unique capabilities when compared to traditional photogrammetry. For example, airborne laser mapping systems can penetrate forest canopy to map the floor beneath the treetops, accurately map the sag of electrical power lines between transmission towers or provide accurate elevation data in areas of low relief and contrast such as beaches.

Airborne laser mapping is a non-intrusive method of obtaining detailed and accurate elevation information. It can be used in situations where ground access is limited, prohibited or risky to field crews.

Commercial airborne laser mapping systems are now available from several instrument manufacturers while various survey companies have designed and built custom systems. Similar to aerial cameras, the instruments can be installed in small

single or twin-engine planes or helicopters. Since the instruments are less sensitive to environmental conditions such as weather, sun angle or leaf on/off conditions, the envelope for survey operations is increased. In addition, airborne laser mapping can be conducted at night with no degradation in performance.

A number of service providers are operating these instruments around the world, either for dedicated survey needs or for hire on a project basis. Some organizations are starting to survey areas on speculation and then offering the laser-generated data sets for resale similar to the satellite data market.

While the core technologies for airborne laser mapping have been in development for the past 25 years, the commercial market for these instruments has only developed significantly within the last five years. This commercial development has been driven by the availability of rugged, low-cost solutions for each of the core subsystems and the growing demand for cheap, accurate, timely, digital elevation data.

In operation, a pulsed laser rangefinder mounted in the aircraft accurately measures the distance to the ground by recording the time it takes a laser pulse to reflect back to the aircraft from the ground or from objects such as buildings, trees or power lines. Since the speed of light is known, the elapsed time is converted to an accurate distance or slant range. Some instruments record multiple returns from a single laser pulse to capture a vertical profile along the slant range. A scanning or rotating mirror is used to provide coverage across the path of the aircraft with swath widths dependent on scan angle and operating altitude. Simultaneously the IRS subsystem records the roll, pitch and heading of the aircraft to determine its orientation in space, while the GPS subsystem provides the precise location of the aircraft through a differential kinematic

solution. During post-processing the IRS orientation and GPS position solutions are combined with the laser slant ranges to calculate accurate XYZ coordinates for each laser return.

The technology does not provide a real-time solution; it requires additional post-processing after the field operations and data collection are completed to generate the final XYZ data points. Post-processing is based on proprietary software developed by each instrument manufacturer but has significantly faster turn-around times than conventional survey techniques, on the order of 10's of hours compared to 10's days for traditional methods.

In addition to directly generating digital XYZ data points, post-processing software modules for the automatic analysis and classification of various features are being developed. Software modules already exist for such activities as vegetation classification and removal while other modules are being developed for automatic feature extraction, building recognition or automatic power wire detection and modeling.

1.2 Application of Airborne laser mapping

Depending on the application, airborne laser mapping technology is either a complementary or a competitive technology when compared to existing survey methods. For many survey applications airborne laser technology is currently deployed in conjunction with other more traditional sensors including standard aerial cameras, digital cameras, multi-spectral scanners or thermal imagers. However, in certain applications, such as forestry or coastal engineering, it offers capabilities not achievable with any other technology.

The most active application areas are:

1. DTM Generation for a Variety of GIS/Mapping Related Products.

Airborne laser mapping is a rapid, cost-effective source of high-accuracy, high-density elevation data for many conventional topographic mapping applications. Comparing with conventional survey methods, this technology has large area topographic surveys to be accomplished significantly faster and at a lower cost.

2. Forestry.

Airborne laser mapping in the forestry industry was one of the first utilizations for the commercial purpose. Accurate information on the terrain and topography beneath the tree canopy is very helpful to both the forestry industry and natural resource managers. Accurate information on tree heights and densities is also extremely important information that is hard to acquire by using traditional techniques. Airborne laser technology, different from radar or satellite imaging, can simultaneously map the ground beneath the tree canopy as well as the tree heights. Post-processing of the data allows the individual laser returns to be analyzed and classified as vegetation or ground returns allowing DTMs of the bare ground to be generated or accurate representative tree heights to be calculated. Therefore, airborne laser mapping is a very effective technique for forestry industry when compared to photogrammetry or extensive ground surveys.

3. Coastal Engineering.

This is another area where airborne laser technology provides state-of-the-art type performance with significant advantages over other survey techniques. Since conventional photogrammetry is difficult to use in areas of limited contrast, such as beaches and coastal zones, an active sensing technique such as airborne laser mapping

provides the ability to complete surveys that would be too expensive to utilize other methods. Furthermore, highly dynamic environments such as coastal zones often need constant updating of baseline survey data. Airborne laser mapping provides a cost-effective method to do this on a routine basis. It is also used for mapping and monitoring of shore belts, dunes, dikes and coastal forests.

4. Corridor or Right-of-Way Mapping.

Airborne laser mapping allows rapid, cost-effective, accurate mapping of linear corridors such as power utility right-of-ways, gas pipelines, or highways. A major market is mapping power line corridors to allow for proper modeling of conductor catenary curves, sag, ground clearance, encroachment and accurate determination of tower locations. For example the use of data acquired through airborne laser surveys can be combined with simultaneous measurements of air and conductor temperature and load currents to establish admissible increases in load-carrying capacity of power lines.

5. Construction.

Timely and accurate digital, geo-referenced elevation data is useful in a variety of construction and engineering activities. Examples include highway corridors, open-pit mines or daily surveying of large construction sites.

6. Flood Plain Mapping.

Accurate and updated modeling of flood plains is critical both for disaster planning and insurance purposes. Airborne laser mapping offers a cost-effective method of acquiring the topographic data required as input for various flood plain modeling programs.

7. Urban Modeling.

Accurate digital models of urban environments are required for a variety of applications including telecommunications, wireless communications, law enforcement and disaster planning. An active remote sensing system such as a laser offers the ability to accurately map urban environments without shadowing.

8. Disaster Response and Damage Assessment.

Major natural disasters such as hurricanes or earthquakes stress an emergency response organization's abilities to plan and respond. Airborne laser mapping allows timely, accurate survey data to be rapidly incorporated directly in to on going disaster management efforts and allows rapid post-disaster damage assessments. It is particularly useful in areas prone to major topographic changes during natural disasters; areas such as beaches, river estuaries or flood plains.

9. Wetlands and Other Restricted Access Areas.

Many environmentally sensitive areas such as wetlands offer limited ground access and due to vegetation cover are difficult to asses with traditional photogrammetry. Airborne laser mapping offers the capability to survey these areas. The technology can also be deployed to survey toxic waste sites or industrial waste dumps.

Since airborne laser mapping is a relatively new technology, applications are still being identified and developed as end-users start to work with the data. There are on going efforts to identify areas where this technology allows value-added products to be generated or where it offers significant cost reduction over traditional methods.

1.3 ALTM

The Airborne Laser Terrain Mapper (ALTM) is an airborne sensor that uses Airborne Laser Mapping technology to collect thousands of spot elevations per second as the aircraft flies over a land surface. Two Global positioning System (GPS) receivers are used to locate the aircraft with accuracy better than 1 meter. One receiver is installed in the aircraft, while the other is situated at a known ground location. The ground receiver identifies and corrects errors in the aircraft's position. A high accuracy laser rangefinder scans beneath the aircraft to produce a wide swath over which the distance from the aircraft to the ground is measured. The laser angles are also measured and corrections are applied to eliminate motions of the aircraft. Once gathered, the angles and distance determine the position of points on the Earth's surface. The ALTM provides data similar to that of a conventional ground surveying technology but at a much faster speed and with both day and night operations.

About two-thirds of all Airborne Laser Terrain Mappers in use worldwide are Optech ALTMs. They have been designing specialized laser ranging systems for more than twenty years, often for airborne platforms.

The Florida International University (FIU) International Hurricane Center and the University of Florida (UF) Geomatics program have recently purchased an Optech model 1210 ALTM system, at a cost exceeding one million dollars. The system is mounted in a Cessna 337 twin-engine light aircraft owned jointly by FIU and UF.

The Optech 1210 ALTM utilizes a 10 kHz pulsed laser range finder (LIDAR) which returns vertical ranges to the ground on a swath beneath the flight path. When

combined with advanced inertial navigation and kinematic GPS positioning, this system can return absolute elevations of the ground surface accurate to 6 inches (15cm). For a typical aircraft deployment (120 miles per hour ground speed, 3000 foot altitude), we are able to map a 2000-foot-wide, over 500-mile-long swath of ground surface elevations spaced 5 feet apart in just a few hours and at a fraction of the cost of conventional surveying.

2.1 OPTECH System

Recent advances in microcomputers, laser ranging technology (LIDAR) and Global Position System (GPS) have resulted in the development of a compact and lightweight airborne laser terrain mapping system (ALTM) that can inexpensively acquire topographic data of unprecedented detail and accuracy. The Florida International University (FIU) International Hurricane Center and the University of Florida (UF) Geomatics program have recently purchased an OPTECH Model 1210 ALTM system (Figure 2.1.1) at a cost exceeding one million dollars. The system is mounted in a Cessna 337 twin-engine light aircraft owned jointly by FIU and UF.

Figure. 2.1.1 Optech Model 1210 ALTM system

Figure. 2.1.2 Cessna 337 twin-engine light aircraft owned jointly by FIU and UF

2.2 Data acquisition

For testing of our method in this study, the major LIDAR data was collected in eastern Broward County over 4 days in December 1999 to March 2000. Over 240 km^2 of the county were surveyed with an average point spacing of 2.5 meters. The survey consisted of 25 N-S trending 600-m-wide swaths spaced every 500 m and 2 E-W trending cross lines (Figure 2.2.1). Data was measured from elevations ranging from 700 - 1200 m. Over 140 million irregularly spaced ground surface elevations were measured. Ground control was provided by two Ashtech Z-12 GPS receivers positioned over National Geodetic Survey (NGS) benchmarks.

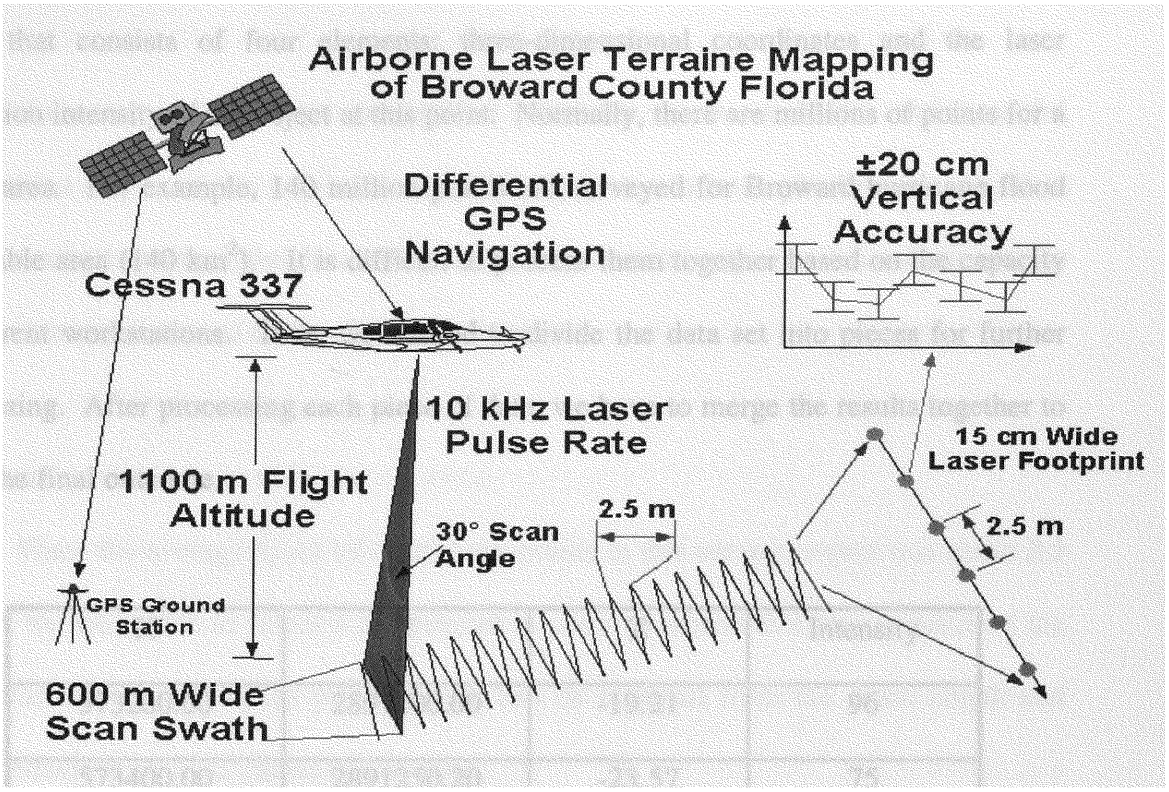

Figure. 2.2.1 LIDAR data collected in Eastern Broward County

2.3 Data storage and processing

After each flight, LIDAR and GPS data are downloaded to a computer and processed by proprietary Optech software to produce UTM X, Y coordinates and ellipsoidal heights of each laser return. Positional accuracy was improved by calculating a precise aircraft trajectory using the KARS software provided by Dr. Gerry Mader of NGS (Mader, 1986; 1992). Elevations were converted from GPS ellipsoidal heights to NAVD88 orthometric heights with the NGS GEOID99 model. Data from overlapping swaths were checked for internal consistency, combined and subdivided into over 300 1-km^2 tiles. Each tile was then gridded using the nearest neighboring interpolation to produce 2m resolution DEMs.

ALTM data are stored in ASCII files (Table 2.1), each line in the file represents a point that consists of four elements: three-dimensional coordinates and the laser reflection intensity of the object at this point. Normally, there are millions of points for a study area. For example, 140 million points are surveyed for Broward hurricane flood venerable area (140 km^2). It is difficult to process them together based on the capacity of current workstations. Thus, we needed to divide the data set into pieces for further processing. After processing each piece of data, we have to merge the results together to have the final outcome.

X	Y	Z	Intensity
573200.00	2891200.00	-19.21	96
573400.00	2891250.20	-23.57	75
573600.28	2891200.00	-23.02	72
573500.25	2891050.96	-23.40	13
573300.52	2891050.88	-19.57	119

Table 2.1 Format of ALTM data in ASCII file

Since there were huge amounts of data that contained useful and unuseful information for the user's requirement, our first job was to resample the data. In other words, we had to shrink the data and get some representative points in a certain area or retrieve a certain amount of data from a certain area.

3.1 Sparse method

Since the average space of contiguous points in this survey project was about 2.5 meters, we were able to select a representative point to represent a certain piece of the area; The size of the certain piece of area can be acquired from the given length and width, which is what we called sparse data.

For acquiring sparse data, we first have to acquire the boundary of the survey area from the data points file, then split the original data into grids. The size of the grid can be decided according to the terrain character of the surveyed area. If there is no significant difference of terrain character in the surveyed area, the size of grid can be large; otherwise, it can be relatively small.

Specifically, we can process the original data into sparse data by following steps below:

1) Scan all the data in the source file, acquire the amount of the data points in the file, which is decided by how many lines are in the file; each line in the file represents a point.

2) Get the boundary of the survey area by computing the minimum and maximum values of both X and Y among the data, and create a rectangular boundary.
3) Split the whole area into grid with given width and length inside the rectangular boundary. While each point belongs to one grid.
4) Scan each data point, and check which grid it belongs to, choose the one with the lowest elevation (Z value) as the representative point of each grid, and add them into an array.
5) Output the representative point array.

The method is illustrated in the Figure 3.1.1. We can split the whole area into grids. The coordinate of the left-bottom point is made by the minimum X, Y values; the coordinate of the right-top point is made by the maximum X, Y values. The direction of X is horizontal in the figure, and the direction of Y is vertical. We split the area in a number of columns along the X direction, and a number of rows along the Y direction. Thus, each grid in the figure can be indexed according to its index of column and row. The index of a grid can be counted as follows:

Index of Grid = IndexY * Columns + IndexX

IndexY refers to the index of Y's direction; IndexX refers to the index of X's direction. By using the index of the grid and the X and Y directions, we can collect all the data points in each grid in a point array. The index of the array is the index of the grid, and the one with the lowest elevation to be the representative point of that grid should be chosen.

The purpose of terrain mapping in this project is to acquire the data that can represent the real terrain character. Since those points that represent the building,

vegetation, vehicles, etc., can be scanned into the data file, so we have to choose the point that approaches the real terrain character when the representative point in each of grid is retrieved. In this sense, we can choose the point with the lowest elevation as the representative point of the grid. That is based on the reasonable selection of the grid size because the size of grid should fit a certain area without significant variation in the terrain character. Accordingly, we can use a point to represent that particular area.

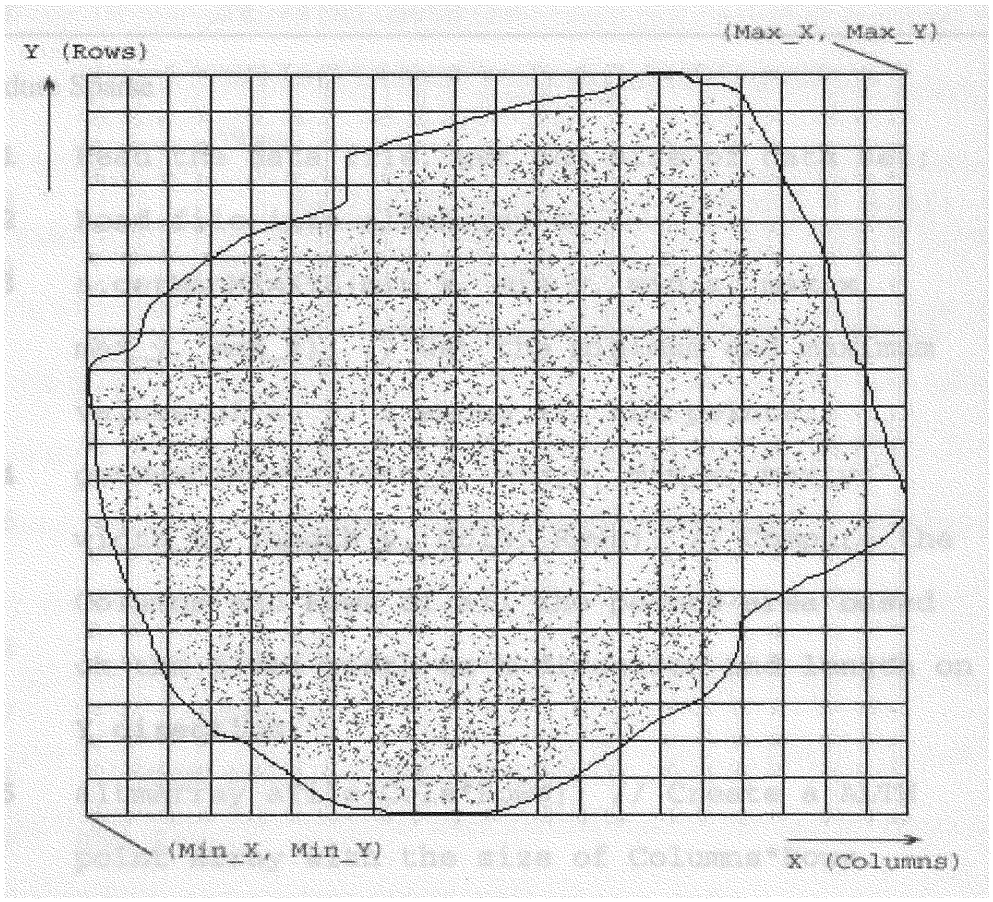

Figure 3.1.1 Method for Sparse data

Selection of the grid size is very important; it will determine the discrepancy between the sparse data and the real terrain character. After selecting a reasonable grid size, we generally can choose the point with the lowest elevation to be the representative point because it is the most likely one to approach the real terrain character.

3.1.1 Algorithm of sparse data

The Pseudo-code of Sparse method is indicated below:

```
Procedure Sparse
01   Read the data file, get the size of data set;
02   Read file into altmArray a;
03   a.getMinMaxXYZ(min_x, min_y, min_z, max_x,
     max_y, max_z); // Get the minimum and maximum
     values of x, y, z among all the points;
04   getNumOfRowCol(min_x, min_y, max_x, max_y,
     width_x, length_y, Cols, Rows); // Compute the
     Columns and Rows of all the points area based
     on the given width on X direction and length on
     Y direction;
05   altmArray aTile[Cols*Rows]; // Create a ALTM
     point array with the size of Columns*Rows.
06   For (k=0; k<size; k++) // Go through each point
     of the Array
07       lIndexX = (a[k].getX() - min_x)/width_x; //
```

```
        Count the X Index of each point based on
        the Columns and Rows Coordinate;
08      lIndexY = (a[k].getY() - min_y)/length_y;
        // Count the Y Index of each point based on
        the Columns and Rows Coordinate;
09      If aTile[lIndexY*cols+lIndexX] == NULL Then
10          aTile[lIndexY*cols+lIndexX] = a[k];
11      Else
12          If (a[k].getZ()<
            aTile[lIndexY*cols+lIndexX].getZ())
            Then
13              aTile[lIndexY*cols+lIndexX] = a[k];
14          Else If ((a[k].getZ()==
            aTile[lIndexY*cols+lIndexX].getZ()) &&
            (a[k].getX() <
            aTile[lIndexY*cols+lIndexX].getX()))
15              aTile[lIndexY*cols+lIndexX] = a[k];

16          End if;
17          End if;
18      End if;
19  End for;
20  Output the aTile[Cols*Rows] to file.
```

Line 01 is used to read the data from the file, after we have scanned the whole file, we can acquire the amount of the data points in the file, which is decided by how many lines in the file, because each line in the file represents a point.

After getting the number of the point set, through Line 02, we can create an altmArray with the number, and load all the data points in the file into the altmArray.

Next, we need to compute the minimum and maximum value of X, Y, which are used to determine the boundary of the data points in the file. Line 03 is functioning for it.

In Line 04, we compute the number of columns and rows of all the point areas through the given parameter -- width and length. Width is used on the X direction, and length is used on the Y direction. Accordingly, columns can be computed based on the width, because it reflects how many strips can be divided on the direction of X; on the other hand, rows can be computed based on the length, because it reflects how many strips can be divided on the direction of Y. After that, we can split the whole area into tiles. Each point must fall into one tile.

Based on the number of columns and rows, we can create an array which points to altmArray with the size of columns $\times$ rows for collecting data points in each tile. Actually, all the points in each tile can be stored in one element of the array, because each element of the array points to an altmArray which is used to store a bunch of data points. The index of the array represents each tile, because we can use the unique index (columns, rows) to represent a tile.

Lines 06 to 19 are functioning as a main procedure for acquiring the sparse data. We can scan each point; first, we can count the column and row indexes of each point, in another words, we can decide which tile the point belongs to (Line 07 and 08 perform

this job). For computing the index of a tile in the aTile array, we can use the index of the column from Line 08 multiplied by the number of column, plus the index of the row from Line 07. If the corresponding element in aTile array is null, we just store the point in it; if it has the data point already, we can compare the Z value of the new point with the point stored in the aTile array. This process is used for acquiring the point with lower elevation. If the Z value is the same, continue to compare the X value, and get the point with the smaller X value. That can guarantee the unique result of sparse data. Lines 09 ~ 18 will perform this job.

After scanning the whole data set one time, we can get the sparse data in the aTile array. Each element in the array represents the data in each tile with the lowest elevation point. Finally, we can output the aTile array into a file (Line 20).

3.1.2 Time Complexity

In the Procedure Sparse, we read the source file, loaded all the data into the altmArray, and computed the mininum and maximum X, Y value. These three steps have the same time complexity, because they all have to scan the whole data set one time. Hence, the time complexity for these steps is O(N), if there are N points in the whole data set.

After that, we scanned the whole data set once to get the representative point of each grid. The time complexity for this step is O(N). Finally, we output the link list for each tile into a file. The time complexity for output is also O(N). Thus, we know the time complexity for the Procedure Sparse is O(N).

3.1.3 Result test and analysis for sparse

We chose the original source data from the survey in eastern Broward County. They are stored in a text file with the capacity of 8,568KB. There are 213,974 points in the file (Figure 3.1.2 Original data for sparse). We chose the size of the tile with 5m in width and 5m in length. So the square of the tile is $25m^2$. Since the average space of contiguous points in this survey project is about 2.5 meters, there are no more than three points along any side of the tile. Consequently, there are no more than nine points in the tile. This size is reasonable for processing sparse data, because according to the terrain character in Florida, there is no significant difference in a relatively small area. A $25m^2$ area is small enough and will not compromise the terrain character. Based on this width and length, we can split the area into 22,000 tiles. After processing the sparse procedure, there are only 18,839 points left. We can infer that there are some tiles without any point in them, so the number of sparse points is less than the number of tiles. The sparse result shows us (Figure 3.1.3 Sparse result) that it keeps the terrain character, but the data capacity shrinks more than ten times.

In the image (Figure 3.1.2) of the original data and sparse data, the color of point reflects the elevation of the points. After the sparse procedure, the shape of a certain area with a different elevation remains unchanged. That is what we want, to keep the terrain character for further processing.

The elapsed time on this original data file for the main function of sparse procedure on an Intel Pentium III 933MHZ CPU and 256M RAM is 4.718 seconds including the time for reading and writing the file.

Since loading the data into a point array needs to read file twice, the first time is for counting the amount of data points, the second time is for importing the data into array. Moreover writing the sparse result into a file, the total processing time includes reading and writing the file on the same computer. This time is concerned with the other performance of the computer, such as the hard drive.

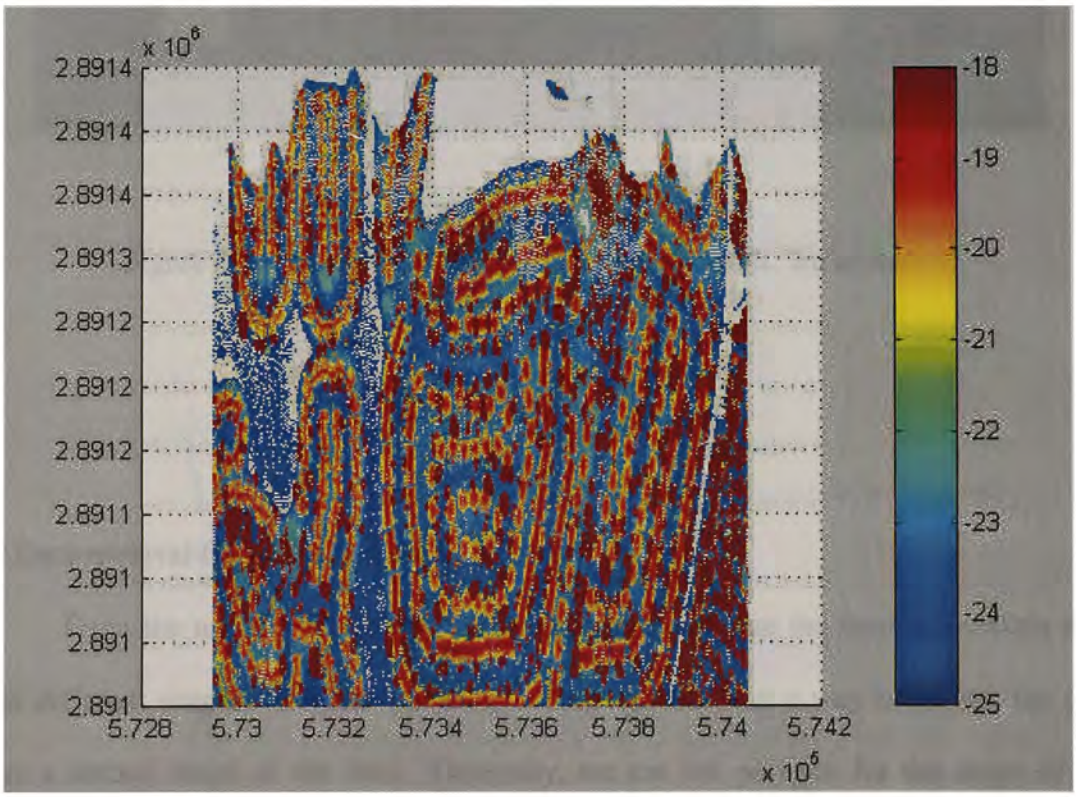

Figure 3.1.2 Sparse Original data (213,974 points, 8,568KB ASCII file)

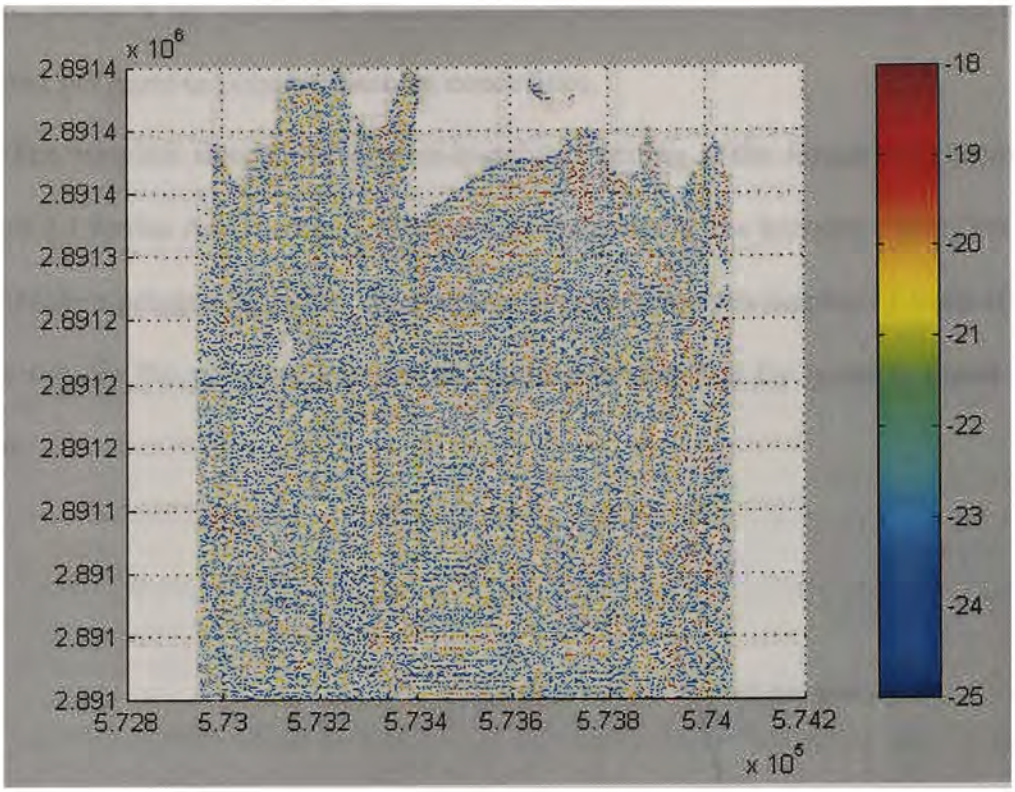

Figure 3.1.2 Sparse Result (Grid size: 5m in width, 5m in length)
(18,839 points, 773KB ASCII file)

3.2 Data retrieval from a certain shape of area

From the users' perspective, they may want to acquire the data in a certain area with different shapes. For this purpose, we have to find out a way to extract the data from a certain shape of the area. Generally, we use the polygon for the shape of the selection. Accordingly, we have to find out an algorithm to extract the data from the polygon. There are many algorithms for testing whether or not a point falls within a

polygon. Many of the algorithms utilize area computations, and many others work only for convex polygons or polygons without concavities.

The simplest algorithm for point-in-polygon testing is the Jordan Arc Theorem (Figure 3.2.1 Jordan Arc Theorem). This simply states that a line between a point known to be outside a polygon will cross the polygon boundary an even number of times if the point is outside the polygon, and an odd number of times if the point is inside the polygon.

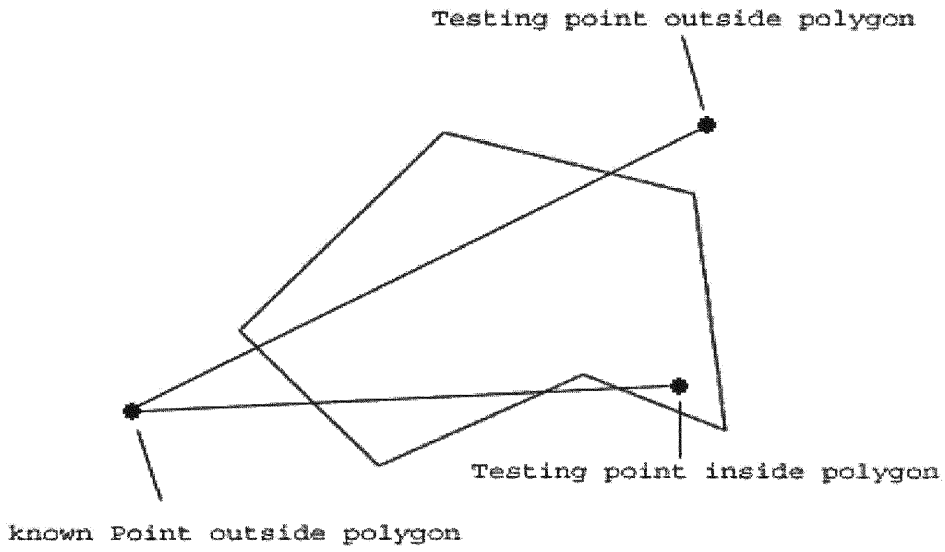

Figure 3.2.1 Jordan Arc Theorem

This theorem provides solutions in all cases except when the lines either touch a vertex or run parallel to an edge. The parallel problem is often significant, because the

outside point can simply be chosen as vertically above or below any given test point, and because many map lines run parallel to the axes.

Thus, in our cases, we can process the polygon retrieval according to the following steps:

1) Scan the vertex file, and acquire the number of vertexes in the file, which is decided by how many lines are in the file, because each of the lines in the file represents a vertex.
2) Scan the whole data file, and acquire the amount of the data points in the file, which is decided by how many lines are in the file, because each of line in the file represents a point.
3) Get the boundary of the survey area by computing the minimum and maximum X, Y value among all the data. Create a reference point outside the boundary.
4) Check each point in the data file, and state whether or not it is inside the polygon created by the vertex loaded in Step 1 by using the Jordan Arc Theorem. If it falls into the polygon, add it into a link list.
5) Output the link list to a file.

3.2.1 Algorithm of data retrieval from polygon

The Pseudo-code of retrieving data from a polygon is indicated below:

```
Procedure altmPolygonFilter (char *inputFileName,
char *outputFileName, char *vertexFileName)
```

```
01    Read vertex from file vertexFileName;
02    Extract vertex into altmArray vertex;
03    Call procedure altmInPolygon (vertex,
      inputFileName, outputFileName);
```

This procedure is to load the vertex of a polygon in a file into an altmArray. The sequence of the vertex of a polygon should be stored in the order along the edge of the polygon. That can guarantee the shape of the polygon. Then call the procedure altmInPolygon to retrieve the data point in the polygon.

```
Procedure altmInPolygon (altmArray& vertex, char
*inputFileName, char *outputFileName)
   01    Read data from file inputFileName;
   02    Extract data points into altmArray a;
   03    Compute min_x, min_y, max_x, max_y; // Get the
         minimum and maximum value of x, y.
   04    outside_x = min_x -1, outside_y = min_y -1; //
         Get the point out side of any polygon in the
         survey area.
   05    std::list<altmPoint> lst; // Create a list
         that stores altmPoint for gathering the points
         in the given polygon area.
   06    For All points in the altmArray a // Check
```

```
      whether or not each point falls in the polygon
07        Bool bInside = altmCheckInPolygon (vertex,
          a[i], outside_x, outside_y); //Call the
          procedure altmCheckInPolygon(vertex, a[i],
          outside_x, outside_y) to test.
08        If bInside is TRUE Then
09           Add the point to the lst;
10        End if;
11    End for;
12    Output the lst to the file outputFileName;
```

This procedure is functioning as retrieving the data points in the polygon. First, we import the data points into the altmArray *a* (Line 01 ~ 02). After that, we compute the minimum and maximum value of X and Y. Then we can lower the minimum X and Y value for creating the point out of the polygon (Line 04). Based on the Jordan Arc Theorem, we know we have to find a point outside the point for testing. Consequently, we can guarantee the point is out of the polygon by lowering the minimum X, Y value.

From Lines 06 to 11, we can test whether or not each point falls into the polygon by calling the procedure altmCheckInPolygon, and add the points inside the polygon in a link list. After the loop, we can collect all the points in the polygon in the link list. Finally, we can output the link list into a file (Line 12).

```
Procedure altmCheckInPolygon (altmArray& vertex,
```

```
altmPoint& point, double dOutside_x, double
dOutside_y)
   01    Check whether or not the testing point is the
         vertex;
   02    If the testing point is one of the vertex Then
   03       Return TRUE;
   04    End if;
   05    For (i=0; i<vertex.getSize(); i++) // For all
         vertex points
   06       If i < vertex.getSize()-1 Then
   07          Bool bIntersec =
               altmIntersection(vertex[i], vertex[i+1],
               point,  dOutside_x, dOutside_y); // Call
               procedure altmIntersection to check
               whether or not the line created by the
               testing point and outside-polygon point
               intersects with edge of polygon.
   08       Else // the last edge is created by the
            last point and the first point in the
            vertex array.
   09          Bool bIntersec =
               altmIntersection(vertex[i], vertex[0],
               point,    dOutside_x, dOutside_y);
   10       End if;
   11       If bIntersec is TRUE Then
   12          count++;
```

```
13      End if;
14    End for;
15    If count is an even number Then
16      Return FALSE;
17    Else
18      Return TRUE;
19    End if;
```

This procedure is to check whether or not the point falls into the polygon. The first step is to check whether or not the point is one of the vertexes. Lines 01 ~ 04 perform this job. If the point is the vertex, return TRUE, otherwise go on checking. Lines 05 to 14 are used to count the number of intersections between the line from the point outside the polygon to the testing point and each edge of the polygon. The procedure altmIntersection will be called, it is for testing whether or not the two lines intersect. Finally, we can tell whether or not the point falls into the polygon by checking the number of the intersection is even or odd. If it is an even number, return FALSE, that means the testing point is outside the polygon; if it is an odd number, return TRUE, that means it is inside the polygon. Lines 15 ~ 19 perform this job.

```
Procedure altmIntersection (altmPoint& vertex1,
altmPoint& vertex2, altmPoint& point, double
dOutside_x, double dOutside_y)
* Line1 is created by testing point and outside-
```

```
polygon point, Line2 is the edge of polygon created by
the continuous vertex;
   01    If Line1 is not parallel with Line2 Then
   02       Compute the intersection of Line1 and Line2;
   03       If the intersection is between the two end
            points on both Line1 and Line2, Then
   04           Return TRUE;
   05       Else
   06           Return FALSE;
   07       End if;
   08    Else
   09       Return FALSE;
   10    End if;
```

This procedure is used to check the two lines that are made by four points. One is made by the point outside the polygon and the testing point, the other is made by the two vertexes, in other words, it is one edge of the polygon.

First, we have to check whether or not the two lines are parallel. If they are parallel, there is no intersection, return FALSE; if not, continue to check whether or not they have an intersection. If they have, return TRUE, otherwise, return FALSE.

3.2.2 Result test and analysis for polygon retrieval

We tested our method through the original data in sparse part (Figure 3.1.1). We chose the vertex indicated below:

	X	Y	Z	Intensity
1	573200.00	2891200.00	-19.21	96
2	573400.00	2891250.20	-23.57	75
3	573600.28	2891200.00	-23.02	72
4	573500.25	2891050.96	-23.40	13
5	573300.52	2891050.88	-19.57	119

Table 3.2.1 Vertex sample 1 for polygon retrieval

There are 5 vertexes with the order of 1 to 5. They could be selected from the original data file. But it does not matter whether or not the point is chosen exactly from the original data file, as long as the point we choose is inside the boundary of the survey area, in another words, as long as the X, Y value of the points are inside the boundary. Thus, we have to guarantee the points fall into the survey area and can make a polygon in the order they are stored. Since with the same set of points, we can create different shapes of the polygon, so the sequence of the vertex is very important, that will decide the shape of the polygon.

There are 213,974 points in the original data file with 8,568KB. According to the polygon we created, we can retrieve the polygon with 25,198 points with 1,034KB (Figure 3.2.2).

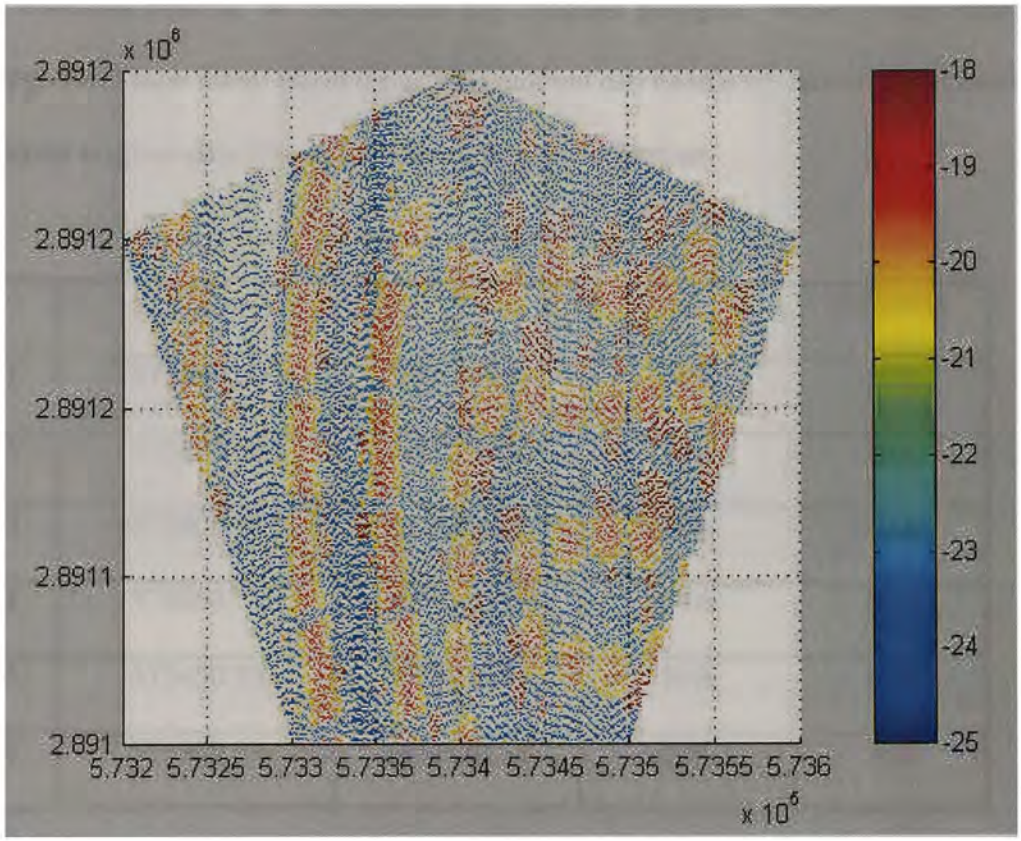

Figure 3.2.2 Polygon Result of Sample 1 (25,198 points, 1,034KB ASCII file)

The Figure 3.2.2 illustrates we can retrieve the data completely. It shows the shape of the polygon we created. It matches perfectly the vertex we selected.

The elapsed time on this original data file for retrieval procedure on an Intel Pentium III 933MHZ CPU and 256M RAM is 5.109 seconds including the time for reading and writing file.

array. Plus, writing the processing result into a file, the total processing time is involved with other performances of the computer, such as memory and hard drive.

With this method, we can process any irregular polygon. Whatever the shape of the polygon and how many edges it has, this method can handle it. Below is another test on the same original data file with different polygon selection.

	X	Y	Z	Intensity
1	573200.00	2891200.00	N/A	N/A
2	573400.00	2891250.20	N/A	N/A
3	573800.28	2891180.00	N/A	N/A
4	573650.25	2891120.96	N/A	N/A
5	573450.25	2891220.96	N/A	N/A
6	573300.52	2891150.88	N/A	N/A

Table 3.2.2 Vertex sample 2 for polygon retrieval

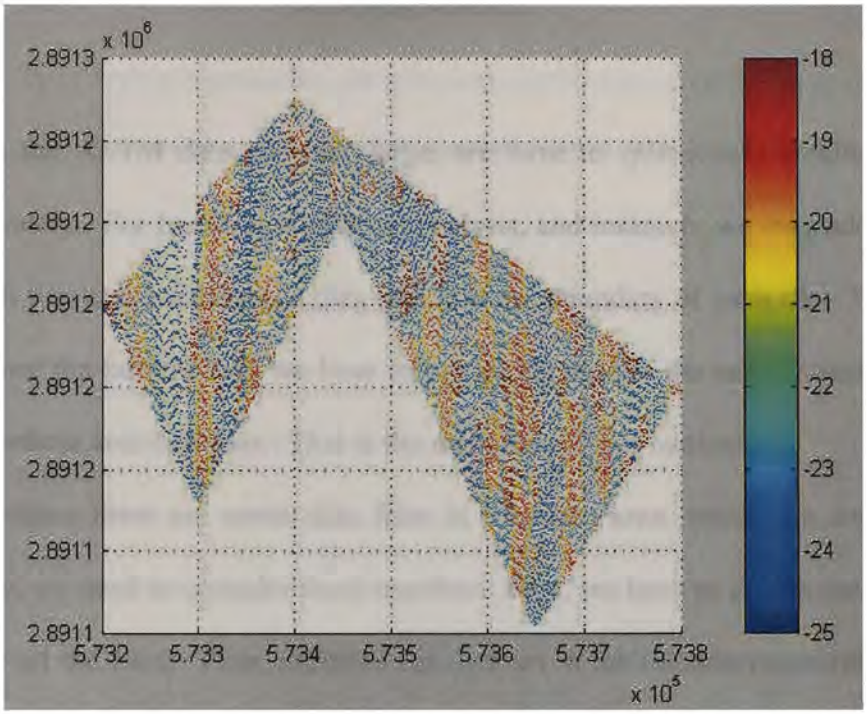

Figure 3.2.3 Polygon Result of Sample 2 (13,357 points, 548 KB ASCII file)

3.3 Discussion

In this chapter, methods of making sparse data and retrieving data from any shape of a polygon area are presented to facilitate the shrinking of huge amounts of ALTM data and retrieval of data in specific area. The test results based on the LIDAR data collected in eastern Broward County show that the proposed methods are effective and promising in resampling the terrain data.

Since our current work is focused on initial processing of terrain data for further research, we have not proposed more complicated models for resampling data. There are still some aspects of resampling data, such as adding some other parameters (Intensity) to analyze and process the original data according to different terrain characters.

Since the ALTM data are very large, we have to split it into smaller tiles to process separately. For further comparison, analysis, and research, we may add a buffer zone for each tile, and collect more data into it at the boundary of each tile. Whatever, with or without the buffer zone, we have to put all the data in the same coordinate and partition the whole area into tiles. That is the main idea of our method.

Sometimes there are some data files in a certain area, which are overlapped. Consequently, we need to consider them together. First, we have to decide the common boundary for all the files. Then, partition the data set in the boundary according to the given size of the tile we want to partition. Finally, we store the data for each tile in different files.

For processing multiple overlapped data files, we have to get the common boundary for all the data files. First, we can get the minimum and maximum X, Y value in each file, then compare with each other, finally we can get the minimum and maximum X, Y value in all the data files. Based on these minimum and maximum values, we can draw the common boundary of multiple overlapped data files, and put them in the same coordinate.

After putting multiple files in one coordinate, we can partition all the data into pieces of tiles based on the given length and width of the tile. The next step is to collect data that belongs to each tile. For the further analysis and research on the data, we need to collect some data points geographically outside the partitioned tile. It will be helpful for the analysis of any adjacent tiles. Consequently, we are going to add a buffer for each

tile. In other words, we collect the data points in the buffer that belong to adjacent tiles into each tile, and output together as one tile data for the final result. Accordingly, we have to check whether or not each point falls into the buffer of any possible tile.

4.1 Single source tile without buffer

The purpose of the tile method is to split huge data files into small files. That will be very helpful for further processing and storage.

For splitting the original data file, first of all, we have to acquire the boundary of the survey area from the original file. Then, we have to split the original data into a tile with given width and length. Each point belongs to one tile. The size of the grid can be decided according to how large we want each piece of tile. How to choose the size of tile depends on the use, ability and efficiency of further processing and storage. It will be distinct due to different cases. The ability and efficiency for further processing and storage should be considered, and sometimes they may be easily ignored. For example, to choose small size of tile would create too many small tile files, and would not be efficient for processing and storage.

Specifically, we can split the original data file into separate small files through steps below:

1) Scan all the data in the original data file, acquire the amount of the data points in the file, which is decided by how many lines are in the file, because each line in the file represents a point.
2) Get the boundary of the survey area by computing the minimum and maximum X, Y value among all the data. Create a rectangular boundary.

3) Inside the rectangular boundary, split the whole area into a tile with the given width and length. Each point belongs to one tile.

4) Scan each data point in the original data file, check which tile it belongs to, and add it to the corresponding tile point list.

5) Output each tile point list into separate files.

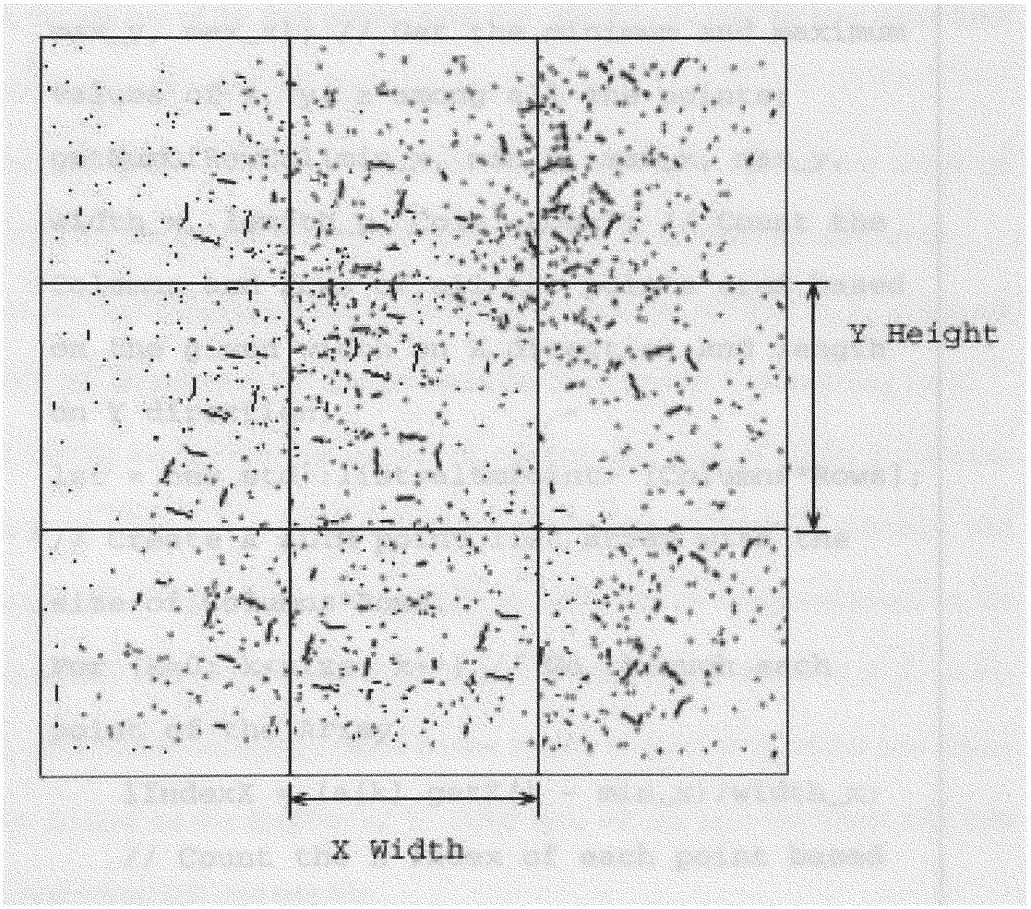

Figure 4.1.1 Method for Tile data

4.1.1 Algorithm of single source tile without buffer

The pseudo-code for the tile Program (Single source file, no buffer) is as below:

```
Procedure Tile_Single_File_Without_Buffer
  01   Read the data file, get the size of data set;
  02   Read file into altmArray a;
  03   a.getMinMaxXYZ(min_x, min_y, min_z, max_x,
       max_y, max_z); // Get the minimum and maximum
       values of x, y, z among all the points;
  04   getNumOfRowCol(min_x, min_y, max_x, max_y,
       width_x, length_y, Cols, Rows); // Count the
       Columns and Rows of all the points area based
       on the given width on X direction and length
       on Y direction;
  05   lst = new std::list<altmPoint> [Columns*Rows];
       // Create a ALTM point list array with the
       size of Columns*Rows.
  06   For (k=0; k<size; k++) // Go through each
       point of the Array
  07       lIndexX = (a[k].getX() - min_x)/width_x;
           // Count the X Index of each point based
           on the Columns and Rows Coordinate.
  08       lIndexY = (a[k].getY() - min_y)/length_y;
           // Count the Y Index of each point based
```

```
            on the Columns and Rows Coordinate.
09          lst[lIndexY*cols+lIndexX].push_back(a[k]);
            //Add the point to the list array
            according to which tile it belongs.
10      End for;
11      Output each list in the list array into file;
```

Line 01 is used to read the data from the file. After scanning the whole file, we can acquire the amount of the data points in the file, which is decided by how many lines in the file, because each line in the file represents a point.

After getting the number of the point set through Line 02, we can create an altmArray with the number and load all the data points in the file into the altmArray.

Next, we need to compute the minimum and maximum value of X, Y, which are used to decide the boundary of the data points in the file. Line 03 is functioning for it.

In Line 04, we compute the number of columns and rows of all the point areas through the given parameter -- width and length. Width is used on the X direction, and length is used on the Y direction. Accordingly, columns can be computed based on the width, because they reflect how many strips can be divided on the direction of X; on the other hand, rows can be computed based on the length, because they reflect how many strips can be divided on the direction of Y. After that, we can split the whole area into tiles. Each point must fall into one tile.

Based on the number of columns and rows, we can create an array which points to altmArray with the size of columns × rows for collecting data points in each tile.

Actually, all the points in each tile can be stored in one element of the array, because each element of the array points to an altmArray which is used to store a set of data points. The index of the array represents each tile, because we can use the unique index (columns, rows) to represent a tile.

Lines 06 to Line 10 are functioning for collecting the tile data. We can scan each point; first, we can count the column and row indexes of each point, in another words, we can determine to which tile the point belongs (Lines 07 and 08 perform this job). For computing the index of a tile in the aTile array, we can use the index of the column got from Line 08 multiplied the number of columns plus the index of the row got from Line 07. Then, add the point to the end of the point list in the list array. In this way, we can collect all the data points into the list array according to which tile it belongs.

After scanning the whole data set one time, we can collect all the data in the list array. Each element in the array represents a list of data in each tile. Finally, we can output the list array into a separate file (Line 11). When outputting the data into files, we have to use different file names to identify each tile. Thus, we use some parameters to create the file name, including the X, Y coordinate of the left-bottom point of each tile, the width and length of tile. That can guarantee each tile file has a unique and clear file name. For example, the coordinates of the left-bottom point are 573000, 2891000, and the width and length of the tile are 30 and 50, then the output tile file name should be "573000_2891000_30_50."

4.1.2 Result test and analysis for single source tile without buffer

We still chose the original source data from the survey in eastern Broward County. They are stored in a text file with the capacity of 8,568KB. There are 213,974 points in the file (Figure 4.1.2). We chose the size of the tile with 400m in width and 250m in length. It splits the survey area into three strips along the X direction and two strips along Y direction, so that makes six tiles. Finally it will be stored in six files.

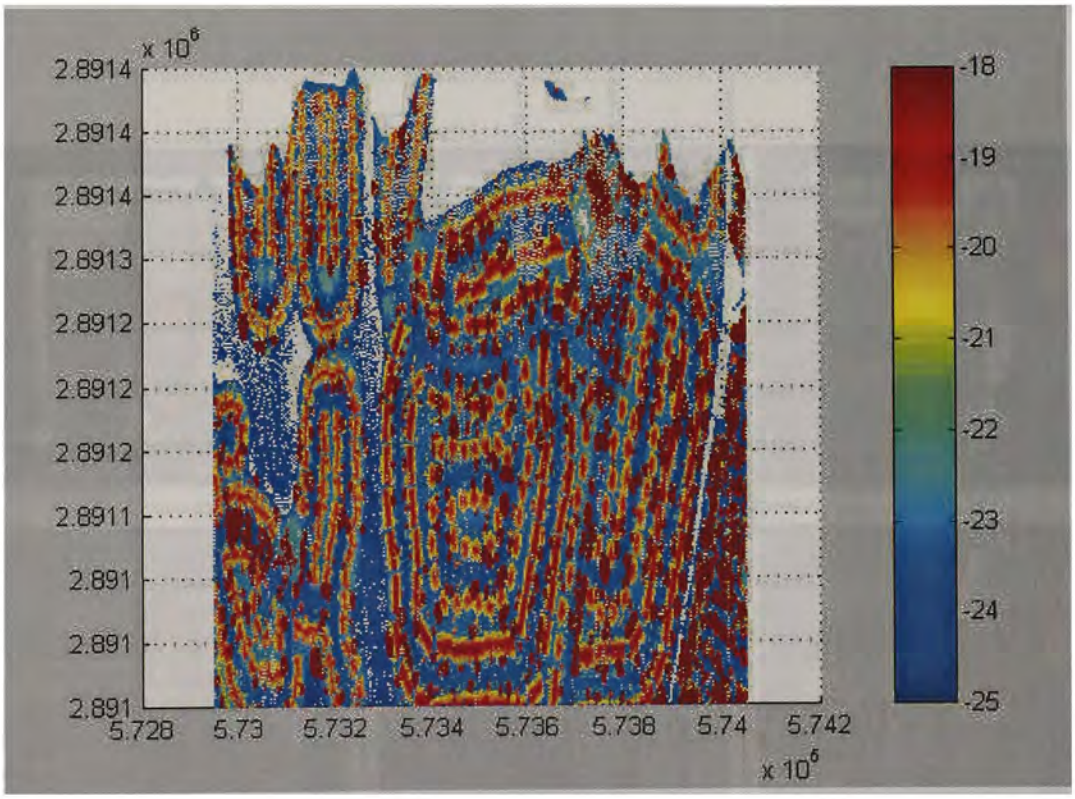

Figure 4.1.2 Tile Original data (213,974 points, 8,568KB ASCII file)

We can compare the figure of original data with the figures of output tiles. It shows us, we separate the source data into six parts without loss of any data. The boundary of each tile matches well.

The elapsed time on this original data file for the main function of the tiling procedure on an Intel Pentium III 933MHZ CPU and 256M RAM is 12.906 seconds including the time for reading and writing the file. This time is involved with other performances of the computer, such as memory and hard drive.

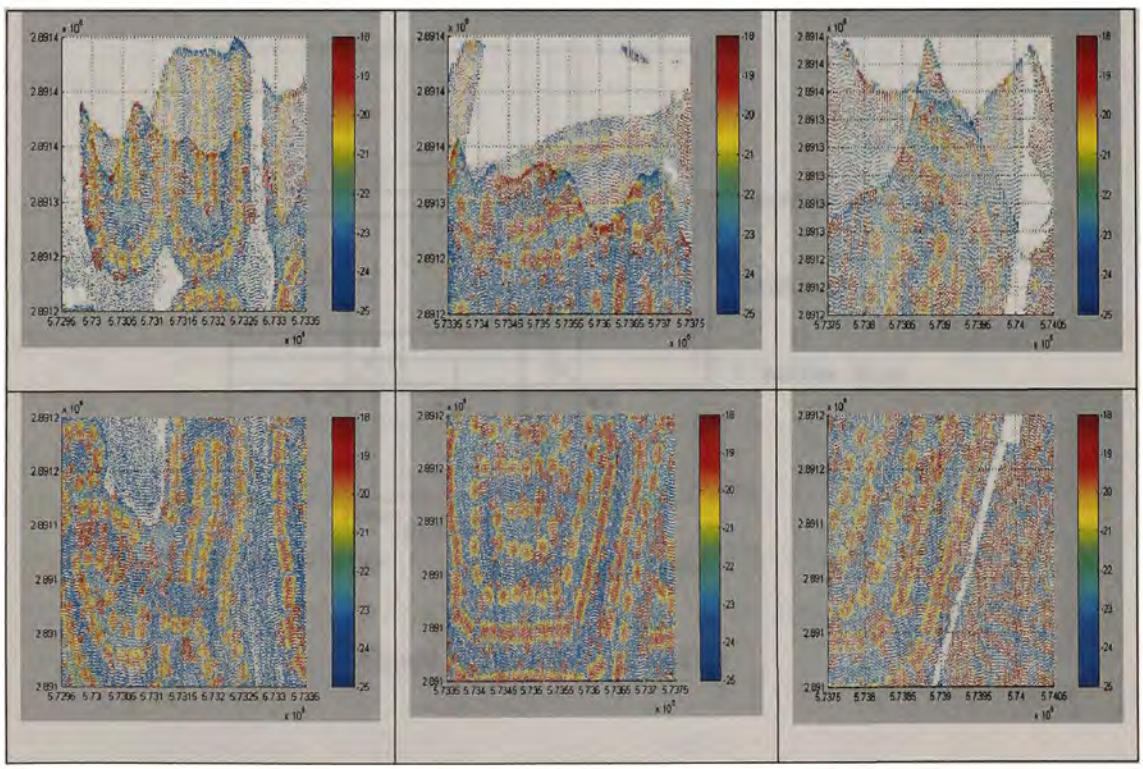

Figure 4.1.3 Tile result of single source without buffer

4.2 Single source tile with buffer

We break down the huge original data into pieces of tile and store them in the separate files. Thus, there are many data points that belong to one of the tiles in each file. For comparison and analysis of the boundary points in each tile, we need to make a buffer around each tile. That can put more points that belong to an adjacent tile into each file. That will make some redundant points in each file. From the figure, we can see it makes the adjacent tiles overlap. Figure 4.2.1 illustrates the overlapping of the adjacent tiles with a buffer. We can adjust our result through comparing the buffer area of each tile.

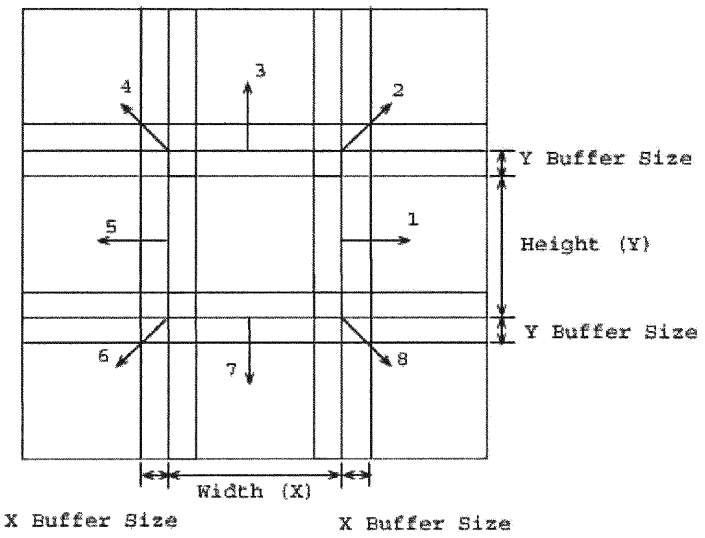

Figure 4.2.1 Tile with Buffer.

Since we will collect the data points from the adjacent tiles, we have to modify our algorithm a little bit. For the previous method, we just collect the data from where it belongs. So far, we have to collect those data that fall into the buffer of each tile into each tile file. In other words, each set of data may be stored in more than one file. For each point, it can possibly fall into eight adjacent tiles (Figure 4.2.1 Tile with Buffer). So what we need to do is to check whether or not each point falls into the buffer of eight possible adjacent tiles. If it does, put it into the tile file whose buffer it falls into.

4.2.1 Algorithm of tile single source tile with buffer

Thus, we modify the algorithm as indicated in the next procedure. The additional entry is at line 10.

The pseudo-code for the single source tile with buffer is indicated in the following procedure:

```
Procedure Tile_Single_File_With_Buffer
  01   Read the data file, get the size of data set;
  02   Read file into altmArray a;
  03   a.getMinMaxXYZ(min_x, min_y, min_z, max_x,
       max_y, max_z); // Get the minimum and maximum
       values of x, y, z among all the points;
  04   getNumOfRowCol(min_x, min_y, max_x, max_y,
       width_x, length_y, Cols, Rows); // Count the
       Columns and Rows of all the point areas based
       on the given width on X direction and length on
```

```
    Y direction;
05  lst = new std::list<altmPoint> [Columns*Rows];
    // Create a ALTM point list array with the size
    of Columns*Rows.
06  For (k=0; k<size; k++) // Go through each point
    of the Array
07      lIndexX = (a[k].getX() - min_x)/width_x; //
        Count the X Index of each point based on the
        Columns and Rows Coordinate.
08      lIndexY = (a[k].getY() - min_y)/length_y; //
        Count the Y Index of each point based on the
        Columns and Rows Coordinate.
09      lst[lIndexY*cols+lIndexX].push_back(a[k]);
        //Add the point to the list array according
        to which tile it belongs to.
        **********************
10      Procedure Buffer_Retrieval;
        **********************
11  End for;
12  Output each list in the list array into file;
```

The pseudo-code for Procedure Buffer_Retrieval is as followings:

```
Procedure Buffer_Retrieval
01    //Direction 1
      //Move X coordinate (+buffer_size), keep Y
unchanged
      lIndexXnew = (a[k].getX() + buffer_size -
      min_x)/width_x;
      If ((lIndexXnew != lIndexX) && (lIndexXnew <
      cols)) Then
        lst[lIndexY*cols+lIndexXnew].push_back(a[k]);
      End if;
02    //Direction 2
      //Move X coordinate (+buffer_size), move Y
      coordinate (+buffer_size)
      lIndexXnew = (a[k].getX() + buffer_size -
      min_x)/width_x;
      lIndexYnew = (a[k].getY() + buffer_size -
      min_y)/length_y;
      If ((lIndexXnew != lIndexX) && (lIndexYnew !=
lIndexY)
      && (lIndexXnew < cols) && (lIndexYnew < rows))
Then
        lst[lIndexYnew*cols+lIndexXnew].push_back(a[k]
      );
      End if;
03    //Direction 3
```

```
        //Move Y coordinate (+buffer_size), keep X
unchanged
        lIndexYnew = (a[k].getY() + buffer_size -
        min_y)/length_y;
        If ((lIndexYnew != lIndexY) && (lIndexYnew <
        rows)) Then
           lst[lIndexYnew*cols+lIndexX].push_back(a[k]);
        End if;
   04   //Direction 4
        //Move X coordinate (-buffer_size), move Y
        coordinate (+buffer_size)
        If (a[k].getX() >= min_x + buffer_size) Then
           lIndexXnew = (a[k].getX() - buffer_size -
           min_x)/width_x;
           lIndexYnew = (a[k].getY() + buffer_size -
           min_y)/length_y;
           If ((lIndexXnew != lIndexX) && (lIndexYnew !=
           lIndexY)
           && (lIndexXnew < cols) && (lIndexYnew < rows))
Then
               lst[lIndexYnew*cols+lIndexXnew].push_back(
           a[k]);End if;
        End if;
   05   //Direction 5
        //Move X coordinate (-buffer_size), keep Y
unchanged
```

```
        If (a[k].getX()>= min_x + buffer_size) Then
           lIndexXnew = (a[k].getX() - buffer_size -
           min_x)/width_x;
           If ((lIndexXnew != lIndexX)) Then
               lst[lIndexY*cols+lIndexXnew].push_back(a[k
           ]);
           End if;
        End if;
  06    //Direction 6
        //Move X coordinate (-buffer_size), move Y
        coordinate (-buffer_size)
        If ((a[k].getX() >= min_x + buffer_size) &&
        (a[k].getY() >= min_y + buffer_size)) Then
           lIndexXnew = (a[k].getX() - buffer_size -
           min_x)/width_x;
           lIndexYnew = (a[k].getY() - buffer_size -
           min_y)/length_y;
           If ((lIndexXnew != lIndexX) && (lIndexYnew !=
           lIndexY) Then
           && (lIndexXnew <= cols) && (lIndexYnew <=
rows)
           && (lIndexXnew >= 0) && (lIndexYnew >= 0))
               lst[lIndexYnew*cols+lIndexXnew].push_back(
           a[k]);
           End if;
        End if;
```

```
07    //Direction 7
      //Move Y coordinate (-buffer_size), keep X
unchanged
      If (a[k].getY()>= min_y + buffer_size) Then
         lIndexYnew = (a[k].getY() - buffer_size -
         min_y)/length_y;
         If ((lIndexYnew != lIndexY)) Then
            lst[lIndexYnew*cols+lIndexX].push_back(a[k
         ]);
         End if;
      End if;
08    //Direction 8
      //Move X coordinate (+buffer_size), move Y
      coordinate (-buffer_size)
      If (a[k].getY() >= min_y + buffer_size) Then
         lIndexXnew = (a[k].getX() + buffer_size -
         min_x)/width_x;
         lIndexYnew = (a[k].getY() - buffer_size -
         min_y)/length_y;
         If ((lIndexXnew != lIndexX) && (lIndexYnew !=
lIndexY)
         && (lIndexXnew <= cols) && (lIndexYnew <=
rows)
         && (lIndexYnew >= 0)) Then
            lst[lIndexYnew*cols+lIndexXnew].push_back(
         a[k]);
```

```
            End if;
        End if;
```

In the Buffer_Retrieval Procedure, the main idea is to move the coordinate of the data point either by X direction or Y direction, or both, then check the new coordinates whether or not they fall into other tiles. If it falls into other tiles, put it into the point list of that tile. The distance of the coordinate move is up to the buffer size of X and Y. As indicated in Figure 4.2.1, each point has eight possible moving directions (Table 4.2.1).

Moving Direction	**X**(+0)	**X**(+Buffer_Size_X)	**X**(-Buffer_Size_X)
Y(+0)	N/A	1	5
Y(+Buffer_Size_Y)	3	2	4
Y(-Buffer_Size_Y)	7	8	6

Table. 4.2.1 Eight Possible Moving Directions

In the Table, there are three possible moves for both X and Y directions. The "+0" means to keep the corresponding coordinate unchanged. The "+Buffer_Size" or the "-Buffer_Size" refers to moving along or against the corresponding coordinate directions. The label of moving direction in Table 4.2.1 is according to the Figure 4.2.1.

After the coordinate moving, we can get a new set of coordinates for each point. Subsequently, we can compute the new index of the point through the process we did previously. Then we can compare the new index of X and Y with the old one. Since each tile can be decided by the unique pair of X and Y index, if either of them changes,

we can put it into another tile point list. After checking all eight directions, we can put each point into all possible tile lists. Through this Buffer_Retrieval Procedure, we make each output tile file bigger than without a buffer retrieval. How many points are added depends on the size of the buffer. The buffer size should not be too large, because that will make too much redundant storage.

4.2.2 Time Complexity

In the Procedure Tile_Single_File_With_Buffer, we read the source file, load all the data into the altmArray, and compute the mininum and maximum X, Y value. These three steps have the same time complexity, because they all have to scan the whole data set one time. Hence, the time complexity for these steps is O(N).

After that, we scan the whole data set once to collect data into the tile file. The time complexity for this step is O(N). Finally, we output the link list for each tile into a file. The time complexity for output is also O(N). Thus, we know the time complexity for the Procedure Tile_Single_File_With_Buffer is O(N).

4.2.3 Result test and analysis for single source data tile with buffer

We still use the data source file in the previous section -- the survey in eastern Broward County. It is stored in a text file with the capacity of 8,568KB. There are 213,974 points in the file (Figure 4.2.2). We still chose the size of the tile with 400m in width and 250m in length. It splits the survey area into six tiles. Also, we chose the buffer size with 20m in both X and Y directions.

213,974 points in the file (Figure 4.2.2). We still chose the size of the tile with 400m in width and 250m in length. It splits the survey area into six tiles. Also, we chose the buffer size with 20m in both X and Y directions.

The elapsed time on the main function of the tile procedure with buffer is supposed to be slightly longer than the elapsed time of the procedure without the buffer. The increased elapsed time is caused by the running time of the Buffer_Retrieval procedure and more data writing time. There are still six output files, but the size of each file increased. This is because there are some data collected into the file from the buffer of each tile.

The image of the result is illustrated in Figure 4.2.2 Single Source Tile with Buffer. We can see from the six images, each image overlaps with adjacent images. The result shows us that our method effectively split the survey area into tiles with buffers.

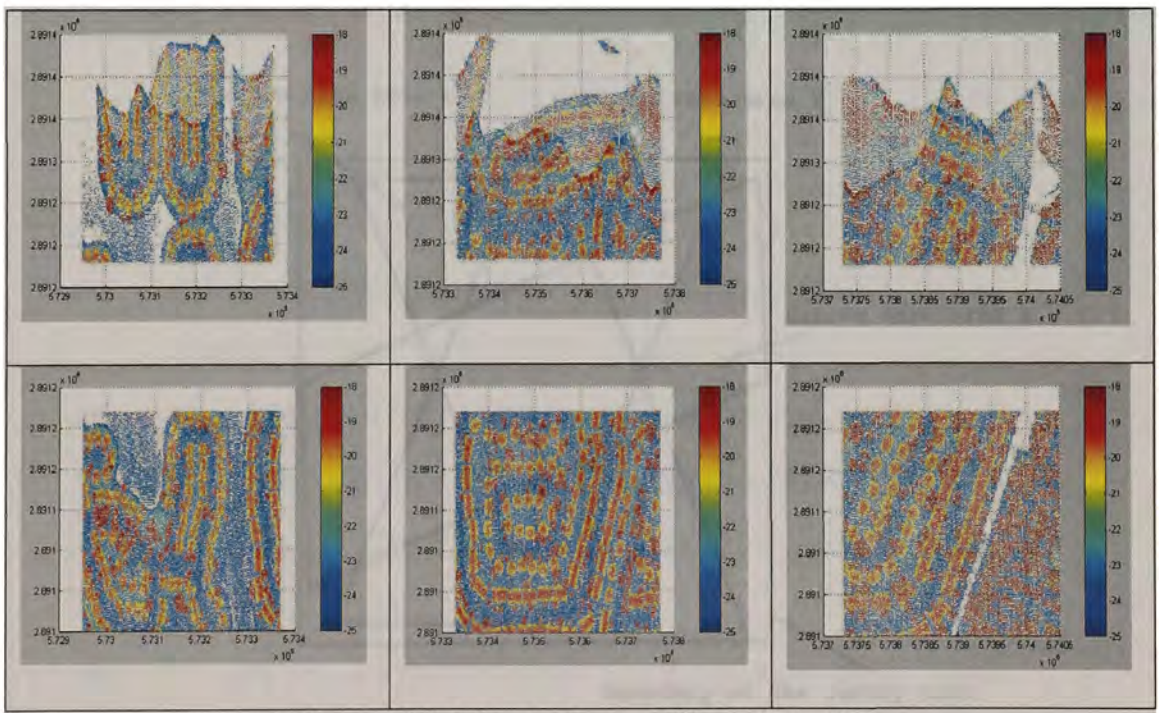

Figure. 4.2.2 Single Source Tile with Buffer

Since loading the data into a point array needs to read the file twice, the first time is for counting the amount of data points, the second time is for importing the data into an array. Furthermore, writing the sparse result into a file, the total elapsed time including reading and writing the file on the same computer is about 12.734 seconds. This time is involved with other performances of the computer, such as memory and hard drive.

4.3 Multiple source tiling with buffer

Since the flight will scan the area in different directions, it might collect the data from the survey area with overlaps (Figure 4.3.1 Overlaps in Multiple Source Files). We need to combine multiple source files for a certain area, because it will reflect the terrain character more exactly. Thus, when we split the source file into tiles, we have to combine the overlap files of a certain area first, and collect data into the tile separately.

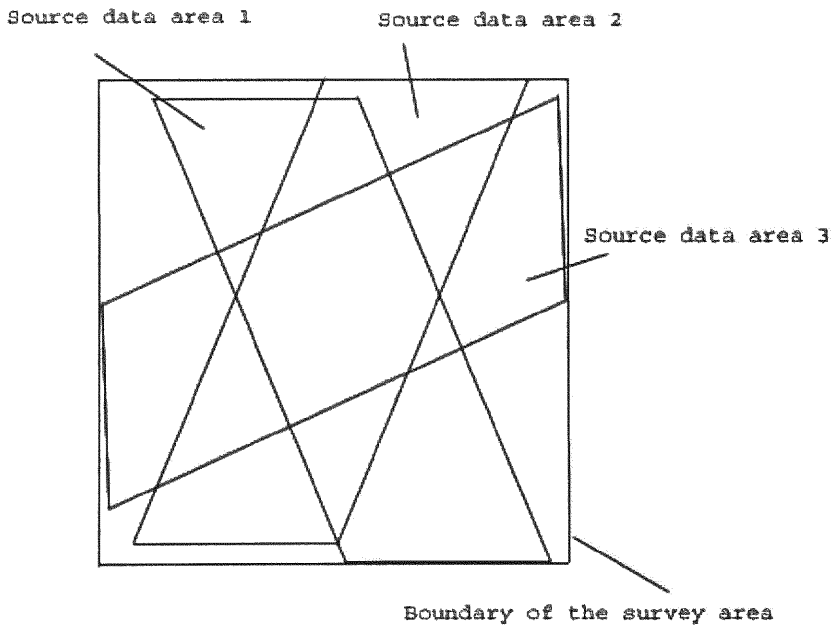

Figure. 4.3.1 Overlaps in Multiple Source Files

In Figure 4.3.1 Overlaps in Multiple Source Files, it illustrates the overlaps of multiple source data files. Actually, each source data file may cover any geographic shape of a surveyed area. Whatever the shape of each source data file covers, we can process the data retrieval through our method.

The main idea of tiling multiple source files is that, first, we have to put multiple source files into the same coordinate, get the boundary of all the data in all the source files; second, we can split them into tiles through the previous method; third, we scan all the data in each file, and collect them into corresponding tiles; and finally, we can output each tile into separate files.

4.3.1 Algorithm of multiple source tiling with buffer

We have to modify the pseudo-code of a single file with the buffer for multiple files.

```
Procedure Tile_Multiple_File_With_Buffer
  01   Read the source data file names from a file
       list;
  02   For all the file names in the file list;
  03       Get the minimum and maximum value of X, Y
           (min_x, min_y, max_x, max_y) in the source
           data file;
  04       If it is the first file in the list Then
```

```
05          min_x_all = min_x, min_y_all = min_y,
            max_x_all = max_x, max_y_all = max_y;
06      Else
07          If (min_x < min_x_all) min_x_all =
             min_x;
08          If (min_y < min_y_all) min_y_all =
    min_y;
09          If (max_x > max_x_all) max_x_all =
    max_x;
10          If (max_y > max_y_all) max_y_all =
    max_y;
11      End if;
12  End for;
13  getNumOfRowCol(min_x_all, min_y_all, max_x_all,
    max_y_all, width_x, height_y, Cols, Rows); //
    Count the Columns and Rows of all the points
    area based on the given width on X direction
    and height on Y direction;
14  lst = new std::list<altmPoint> [Columns*Rows];
    // Create a ALTM point list array with the size
    of Columns*Rows.
15  For all the file names in the file list;
16      Read the data file, get the size of data
    set;
17      Read file into altmArray a;
18      lst = new std::list<altmPoint> [Cols*Rows];
```

```
        // Create a ALTM point list array with the
        size of Columns*Rows.
19      lIndexX = (a[k].getX() -
        min_x_all)/width_x; // Count the X Index of
        each point based on the Columns and Rows
        Coordinate.
20      lIndexY = (a[k].getY() -
        min_y_all)/height_y; // Count the Y Index
        of each point based on the Columns and Rows
        Coordinate.
21      lst[lIndexY*cols+lIndexX].push_back(a[k]);
        //Add the point to the list array according
        to which tile it belongs to.
22      Call Procedure Buffer_Retrieval;
23      For all the list in the list array;
24          If the output tile file does not exist
            Then
25              Create a new tile file, and output
                the list into the file;
26          Else
27              Append the list into the existing
                tile file;
28          End if;
29      End for;
30  End for;
```

When we implement this method according to the pseudo-code, we can store a bunch of source data file names in a file. Subsequently, we can get each file according to this file name list. Since we have to know the minimum and maximum X, Y value among all the source data, we have to check each source file, get the minimum and maximum X, Y value, and compare with that of other files. Finally, we can get the minimum and maximum X, Y value among all the data in all source files. After getting this job done, we can setup a general coordinate for all the source files. The next step is to split the whole area that all the source files cover into columns and rows based on the minimum and maximum X, Y values, which we get in the first step. Actually, the minimum and maximum X, Y value give the boundary of the survey area in which all the source files are involved. After we split the data into tiles in this boundary based on the width in X direction and length in Y direction, we can collect data for each tile from the source files one by one.

The way we collect data into a separate tile is the same as before. When we scan each data point in the source files, first, we have to count the column and row indexes of each point, and we know to which tile it belongs; finally, we put it into the point list of that tile. If we need to embed the Buffer_Retrieval procedure for buffer points retrieval, we have to process the buffer retrieval during the data collecting. After scanning each source file, we can put the data into corresponding tile files.

However, there is some difference from the previous method on storing results into files. After we finish processing the first source file, we will create all the tile files. When we finish processing the rest of the source files, we have to append the result into each existing file rather than overwriting it. Thus, when we output the data list into the

tile file, we have to check whether or not it is the first time to output the data. If it is the first time to output, we create all the tile files and store the data into them; otherwise, append the data into files. Lines 23 to 29 in the pseudo-code show this function.

4.3.2 Result test and analysis for multiple source tiling

We still use the data source file in the previous section -- the survey in eastern Broward County. It is stored in a text file with the capacity of 8,568KB. There are 213,974 points in the file (Figure 4.1.2). We still chose the size of the tile with 400m in width and 250m in length. It splits the survey area into six tiles. Also, we chose the buffer size with 20m in both X and Y directions.

Before we test our method, we have to make some changes on our source data file. We must compare and prove the correctness of our method for further analysis. We can separate the source data file into two or more files randomly. That is for satisfying multiple data sources. Since they are separated randomly, we can get multiple source files that represent random geographic shape. That can prove our method stronger. The reason why we choose the same parameter as in the previous section is that we can compare the result of these two cases, because theoretically the tile results should be the same. Since there is no difference between the whole set of data, in this case, it is just stored in separate files. Thus, when we count the minimum and maximum X, Y value, we should get the same result. And when we use the same width, length and buffer size as the parameter, we are supposed to get the same tile split. Finally, when we collect data into corresponding tiles, we should also get the same result, because the points do not change, they will be collected into the tile as in the previous result.

Our test result does prove the correctness of this method. As the result we got six tile files. Then we can compare the data of these six files. First we can compare the size of the files and the amount of points in each file (Table 4.3.1 Contrast of the result of Single Source File Tile and Multiple Source File Tile).

Tile Files List		**Single Source**	**Multiple Source**
572950_2890950_400_250_20	File Size (KB)	2,080	2,080
	Number of Points	50709	50709
573350_2890950_400_250_20	File Size (KB)	2,384	2,384
	Number of Points	58123	58123
572950_2891200_400_250_20	File Size (KB)	1,583	1,583
	Number of Points	38583	38583
573350_2891200_400_250_20	File Size (KB)	1,630	1,630
	Number of Points	39721	39721
573750_2890950_400_250_20	File Size (KB)	1,496	1,496
	Number of Points	36465	36465
573750_2891200_400_250_20	File Size (KB)	1,134	1,134
	Number of Points	27648	27648

Table. 4.3.1 Contrast of the result of Single Source File Tile and Multiple Source File Tile

We know from the previous table, we must collect the right amount of points into the tile files. Furthermore, we have to contrast the data between these two sets of results in detail. We utilize the software called TextPad to get this job done. The version of this software is 4.1.01: 32-bit Edition (Copyright © 1992-1999 Helios Software Solutions). TextPad is designed to provide the power and functionality to satisfy the most demanding text editing requirements. The 32-bit edition can edit files up to the limits of virtual memory, and will work with MS Windows™ 9x, Windows NT and Windows 2000. We can use this software to open each tile file, and sort each file according to X, Y, Z, and Intensity value. Our rule is when X value is the same, sort by Y value; when Y value is the same, sort by Z value; when Z value is the same, sort by Intensity value. Since there are not any two points with the same X, Y, Z, and Intensity value completely, we can get a unique sort order of the file. Also, we can use the functionality of the software to compare two output tile files. It can tell whether or not two files are the same. Through this processing, we know our two sets of result are the same. That also proves our method's correctness. We can also see it from the image (Figure 4.3.1 Tile Files from Multiple Source Files) of tile files from multiple source files. It is the same as the result from a single source file.

The elapsed time on processing multiple source files is much different from a single source file, because between two tile data retrieve cycles, we have to output the data into files. Rather than in a single source file retrieval, we only need to process one cycle of data retrieval and then output the result into files. Thus, the elapsed time of a single source does not count the time of the output tile result into files. However, the main function of multiple source tile retrieve has to include the time of output result at

each retrieval cycle. That makes the elapsed time appear much longer than in the single source. But it counts the output procedure time in. Thus, the elapsed time on this test is 16.734 seconds including reading and writing files on a workstation with an Intel Pentium III 933MHZ CPU and 256M RAM.

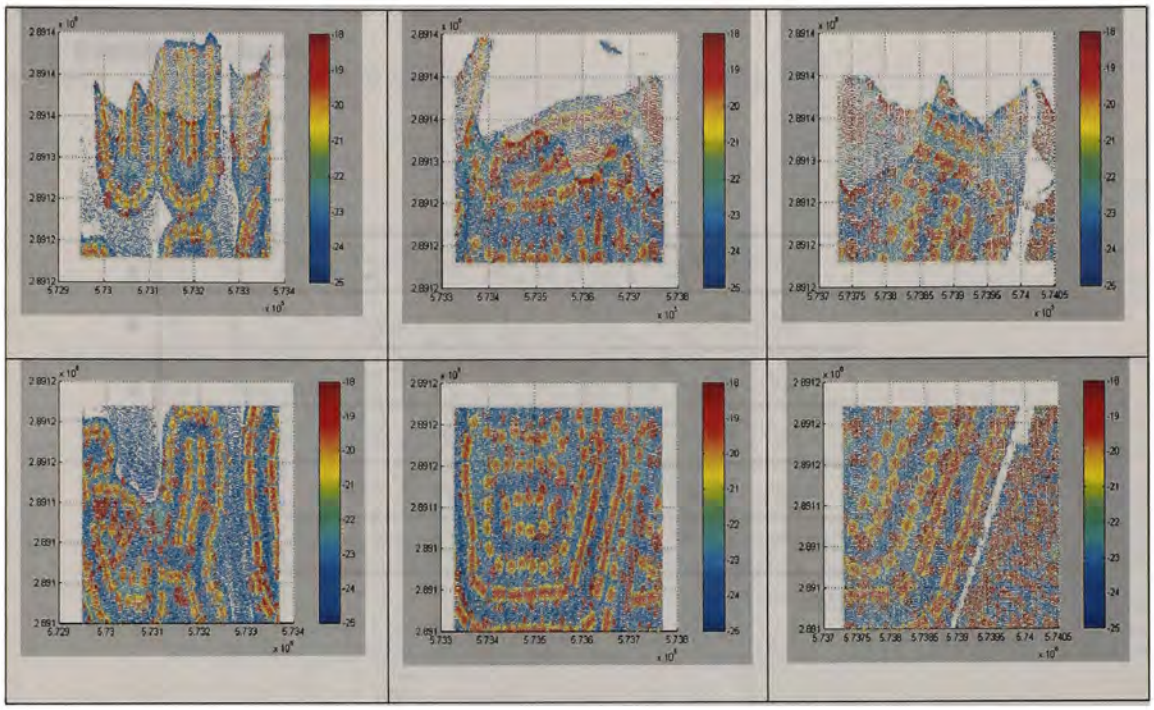

Figure. 4.3.1 Tile Files from Multiple Source Files

4.4 Sorted tiling method

The main idea of sorted tiling method is also to partition the survey into tiles, but the algorithm is different. It needs two steps to retrieve tile data. First, we partition the area along the Y direction. That can split the area into rows. After we retrieve each row of data, we can continue to the next step. The next step is to partition each row of data into strips along the X direction, in another words, it splits each row into columns. That makes the whole area a bunch of tiles. Since we need to partition the data along X or Y direction in this method, we have to sort the data along the corresponding direction, that is the reason why we called this method sorted tiling method.

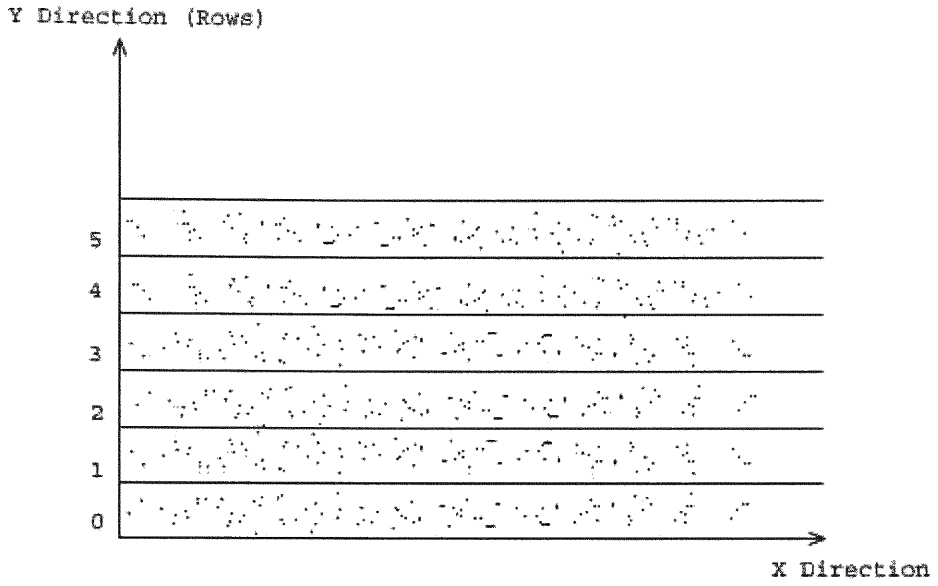

Figure. 4.4.1 Rows Partition of Sorted Tiling

We can see from the Figure 4.4.1 Rows Partition of Sorted Tiling, we sort the data set along Y direction. Then we can split it into rows according to the given length. We retrieve each row of data separately and proceed to the second procedure. In the Figure 4.4.2 Columns Partition of Sorted Tiling Method, it shows that after we sorted each row by X value, we partition each row along the X direction. That splits the row into columns. We can see from the figure that these two steps partition the area into tiles. Our algorithm is based on these two steps. If we consider the buffer size for the boundary, we just need to add some buffer size parameters on the given length and width when partitioning the data set into tiles. This is illustrated in the Figure Sorted Tiling

with Buffer. When we collect data for each tile, we just need to extend the boundary of row or columns with Y buffer size or X buffer size like the dotted line in the figure. If the buffer size is zero, that will be the case without the buffer.

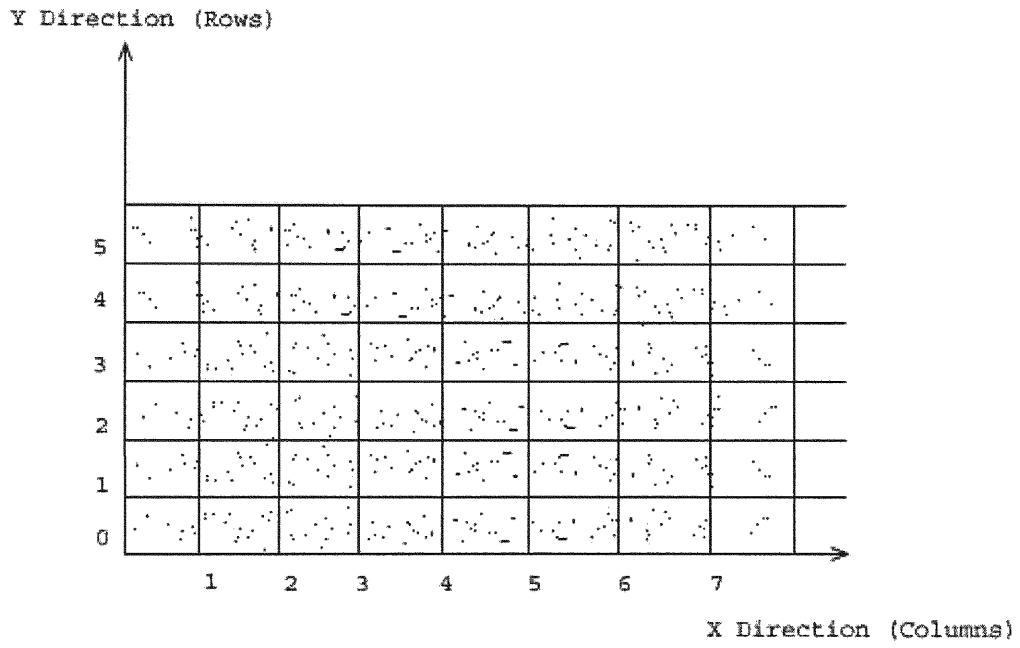

Figure. 4.4.2 Columns Partition of Sorted Tiling Method

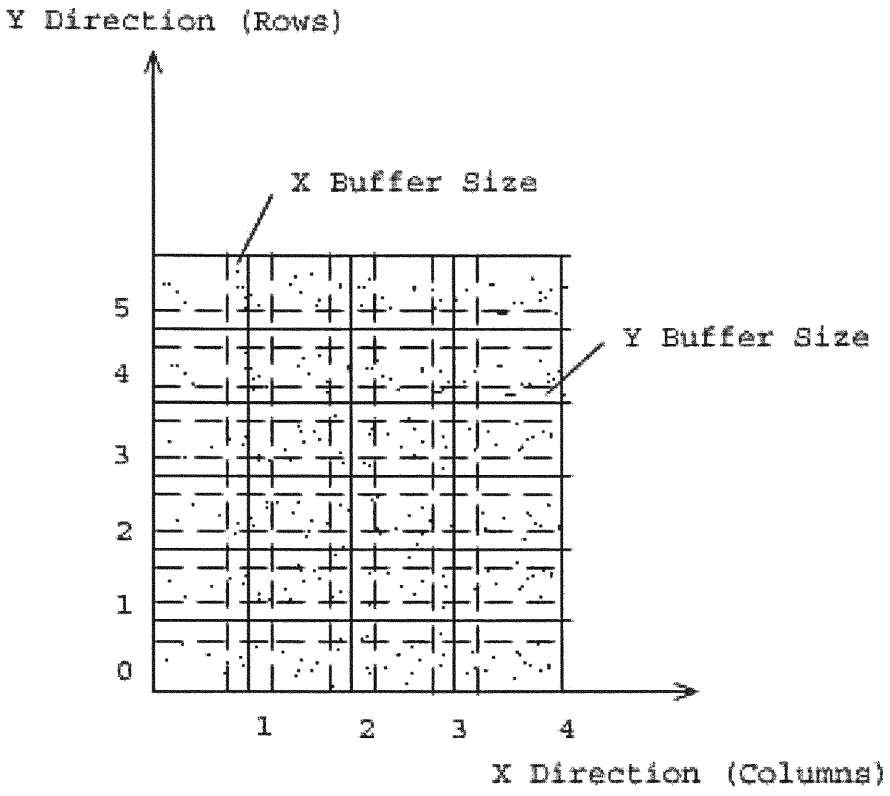

Figure. 4.4.3 Sorted Tiling with Buffer

4.4.1 Algorithm of single source sorted tiling with buffer

The pseudo-code for single source sorted tiling with buffer is as following:

```
Procedure altmTileOneFile
   01    Read the data file, get the size of data set;
   02    Read file into altmArray a;
```

```
03    a.getMinMaxXYZ(min_x, min_y, min_z, max_x,
      max_y, max_z); // Get the minimum and maximum
      values of x, y, z among all the points;
04    getNumOfRowCol(min_x, min_y, max_x, max_y,
      width_x, length_y, Cols, Rows); // Count the
      Columns and Rows of all the points area based
      on the given width on X direction and length
      on Y direction;
05    Call Procedure altmTileYData (OutputFileDir,
      a, min_x, min_y, width_x, length_y,
      buffer_size, rows, cols);
```

Line 01 is used to read the data from the file, after scanning the whole file, we can acquire the amount of the data points in the file, which is decided by how many lines are in the file, because each of line in the file represents a point.

After getting the number of the point set, through Line 02, we can create an altmArray with the number, and load all the data points in the file into the altmArray.

Next, we need to compute the minimum and maximum value of X, Y, which are used to decide the boundary of the data points in the file. Line 03 is functioning for it.

In Line 04, we compute the number of columns and rows of all the survey area through the given parameter -- width and length. Width is used on the X direction, and length is used on the Y direction. Accordingly, columns can be computed based on the width, because it reflects how many strips can be divided on the direction of X; on the

other hand, rows can be computed based on the length, because it reflects how many strips can be divided on the direction of Y. After that, we can split the whole area into tiles. Each point must fall into one tile. In Line 05, it calls the Procedure altmTileYData for tiling.

```
Procedure altmTileYData
  01    long int i=0, j=0, count=0, restart=0;
  02    double low_y = min_y;
  03    double high_y = min_y + length_y;
  04    long int start_index = 0;
  05    long int end_index = 0;
  06    bool bFirstTime = true;
  07    a.sortByY();  // Sort altmArray a by the Y
        value.
  08    For (i=0; i<rows; i++)
  09      bFirstTime = true;
  10      restart = 0;
  11      While (j < a.getSize() && (a[j].getY() >=
          low_y - buffer_size) && (a[j].getY() < high_y
          + buffer_size))
  12        If ((a[j].getY() >= high_y - buffer_size)
          && bFirstTime) Then
  13          bFirstTime = false;
  14          restart = j;
```

```
15          End if;
16          count++;
17          j++;
18       End while;
19       If (count > 0) Then
20          altmArray row(count); //Allocate memory for
         the row.
21          end_index = j-1;
22          long int m;
23          For (m=start_index; m<=end_index; m++)
24             row[m-start_index] = a[m];
25          End for;
26          Call Procedure
            altmTileRowData(OutputFileDir, row, min_x,
            low_y, width_x, length_y, buffer_size,
            cols);
27          count = 0;
28          If (!bFirstTime) Then
29             j = restart;
30             start_index = restart;
31          Else
32             start_index = end_index + 1;
33          End if;
34       End if;
35       low_y = high_y;
36       high_y = high_y + length_y;
```

```
37    End for;
```

This procedure is used to split the survey area into strips along the Y directions. That can partition the all area into rows according to the given length of each tile.

First, we need to sort the data set by the Y value (in Line 07). Next, we will retrieve each row for further processing. Lines 08 to 37 are a For loop for each row. In Line 09, we set the Boolean variable *bFirstTime* as True. It is used to check whether or not we have retrieved any data located in the buffer of the next row. Long int variable j is functioning as the index of the data set array. Double variable *low_y* stores the low boundary of each row; double variable *high_y* stores the high boundary of each row. Since we start from the minimum Y value point, we set the initial *low_y* as *min_y*, and the initial *high_y* as *min_y* plus *length_y*. After one loop, we have to modify the *low_y* and *high_y*'s value, because we change another row to retrieve data.

We retrieve data for each row from Lines 11 to 18. In the condition of the while loop, we can see that we involve the buffer size. That will collect data in the buffer zone. If the buffer size is zero, that will be the case without the buffer. Long int variable *count* is used to count how many points are retrieved into each row. In Line 12, we check whether or not we have retrieved any data located in the buffer of the next row. If so, we have to record the index of the starting point located in the buffer zone in the variable *restart* that is used to modify the variable *start_index*. If there are some points retrieved into the row, we can give further process (from Lines 19 to 34). We copy the data located in the row into a new altmArray *row* with the size that is stored in the variable *count*. Long int variable *start_index* and *end_index* refer to the index of the starting point

and ending point in the whole data set altmArray *a*. The new created altmArray *row* stores all the data in each row. Then we call Procedure altmTileRowData to process each row data for tiling.

After finishing Procedure altmTileRowData, we have to reset the variable *count* to zero. As we know that the Boolean variable *bFirstTime* is used to check whether or not we have retrieved any data located in the buffer of next row. Thus, if it is False, it means we collect some data located in the buffer of the next row. So we have to adjust the index *j* and *start_index* to the index of starting point located in the buffer of next row, which is stored in the variable *restart*. If there is no data collected from the buffer of next row, we have to set the *start_index* as *end_index* plus 1. Finally, we have to adjust the *low_y* and *high_y* to the next row's boundary (Lines 35 ~ 36).

```
Procedure altmTileRowData
  01   long int i=0, j=0, k=0, count=0, restart=0;
  02   double low_x = start_x;  // Initial value of
       start_x is the minimum X value
  03   double high_x = start_x + width_x;
  04   long int start_index = 0;
  05   long int end_index = 0;
  06   bool bFirstTime = True;
  07   a.sortByX();  // Sort altmArray a by the X
       value.
  08   For (i=0; i<cols; i++)
```

```
09      bFirstTime = True;
10      restart = 0;
11      While (j < a.getSize() && (a[j].getX() >=
        low_x - buffer_size) && (a[j].getX() < high_x
        + buffer_size))
12        If ((a[j].getX() >= high_x - buffer_size)
          && bFirstTime) Then
13          bFirstTime = false;
14          restart = j;
15        End if;
16        count++;
17        j++;
18      End while;
19      If (count > 0) Then
20        altmArray tile(count); // Allocate memory
        for the row.
21        end_index = j-1;
22        long int m;
23        For (m=start_index; m<=end_index; m++)
24          tile[m-start_index] = a[m];
25        End for;
26        Output altmArray tile;
27        count = 0;
28        If (!bFirstTime) Then
29          j = restart;
30          start_index = restart;
```

```
31          Else
32              start_index = end_index + 1;
33          End if;
34      End if;
35      low_x = high_x;
36      high_x = high_x + width_x;
37  End for;
```

This procedure is used to split the row data retrieved from Procedure altmTileRowData into strips along the X directions. That can partition the row data into columns according to the given width of each tile.

First, we need to sort the data set by the X value (in Line 07). Next, we will retrieve each column in the row. Lines 08 to Line 37 are a For loop for each column. In Line 09, we set the Boolean variable *bFirstTime* as True. It is used to check whether or not we have retrieved any data located in the buffer of the next column. Long int variable *j* is functioning as the index of the row data set array. Double variable *low_x* stores the low boundary of each column; double variable *high_x* stores the high boundary of each column. Since we start from the minimum X value point, we set the initial *low_x* as *start_x* whose initial value is the minimum X value, and the initial *high_x* as *start_x* plus *width_x*. After one loop, we have to modify the *low_x* and *high_x*'s value, because we change another column to retrieve the data.

From Lines 11 to 18, we retrieve data for each column in the row. In the condition of the while loop, we can see that we involve the buffer size. That will collect the data in the buffer zone. If the buffer size is zero, that will be the case without the

buffer. Long int variable *count* is used to count how many points are retrieved into each column. In Line 12, we check whether or not we have retrieved any data located in the buffer of the next column. If so, we have to record the index of the starting point located in the buffer zone in the variable *restart* that is used to modify the variable *start_index*. If there are some points retrieved into the column, we can give further process (from Lines 19 to 34). We copy the data located in the column into a new altmArray *tile* with the size that is stored in the variable *count*. Long int variable *start_index* and *end_index* refer to the index of the starting and ending points in the row data set altmArray *a*. The new altmArray *tile* created stores all the data in each column. Then we output the altmArray *tile* and save it in the file.

After finishing the output of each tile data, we have to reset the variable *count* to zero. As we know that the Boolean variable *bFirstTime* is used to check whether or not we have retrieved any data located in the buffer of the next column. Thus, if it is False, it means we have collected some data located in the buffer of next column. So we have to adjust the index *j* and *start_index* to the index of the starting point located in the buffer of the next column, which is stored in the variable *restart*. If there are no data collected from the buffer of next column, we have to set the *start_index* as *end_index* plus one. Finally, we have to adjust the *low_x* and *high_x* to the next column's boundary (Lines 35 ~ 36).

4.4.2 Time Complexity

In the Procedure altmTileOneFile, we read the source file, load all the data into the altmArray, and compute the mininum and maximum X, Y value. These three steps

have the same time complexity, because they all have to scan the whole data set one time. Hence, the time complexity for these steps is O(N).

In Procedure altmTileYData, we need to sort the whole data set first. Thus, the time complexity for the sorting procedure is O(NlogN). Then we have to retrieve the row data from the original data set. The number of rows depends on the given length of the tile. The range for the number of row is from 1 to N. Thus, we build up a For loop for processing each row data. In each loop, we call Procedure altmTileRowData. In Procedure altmTileRowData, it is the same in that we have to sort the row data first. Let's suppose the number of the row is *k*. Thus, the time complexity for sorting each row data is $O(N_k \log N_k)$. Thus, the time complexity for Procedure altmTileYData should be $O(\sum_{i=1}^{k} N_k \log N_k)$.

Since, $\log N_k \leq \log N$, so

$$\sum_{i=1}^{k} N_k \log N_k \leq \sum_{i=1}^{k} N_k \log N$$

Also,

$$\sum_{i=1}^{k} N_k = N$$

So,

$$\sum_{i=1}^{k} N_k \log N_k \leq N \log N$$

Thus, the worst case time complexity of Procedure altmTileYData should be O(NlogN).

Hence, we know the time complexity of Procedure altmTileOneFile is O(NlogN).

4.4.3 Result test and analysis for sorted tiling

We still use the data source file in the previous section -- the survey in eastern Broward County. It is stored in a text file with the capacity of 8,568KB. There are 213,974 points in the file (Figure 4.1.2). We still choose the size of the tile with 400m in width and 250m in length. It splits the survey area into six tiles. Also we choose the buffer size with 20m in both X and Y directions.

Output File Name	Sorted Tiling	Non-Sorted Tiling
	Number of Points	Number of Points
572950_2890950_400_250_20.txt	50,709	50,709
572950_2891200_400_250_20.txt	38,583	38,583
573350_2890950_400_250_20.txt	58,123	58,123
573350_2891200_400_250_20.txt	39,721	39,721
573750_2890950_400_250_20.txt	36,456	36,456
573750_2891200_400_250_20.txt	27,648	27,648

Table. 4.4.1 Results of Two Tiling Methods

We know from the Table Two Tiling Method Result, we get the same amount of points in each tile file through two different tiling methods. We can utilize the software called TextPad (The version of this software is 4.1.01: 32-bit Edition Copyright © 1992-1999 Helios Software Solutions) to compare the two sets of files. First, we have to sort the two sets of files in the same way, it can guarantee all the data in both sets of files are in the same order. Then we use the function of TextPad to compare two files of each set. The result shows that the corresponding files in each set contain the same data. That shows these two different methods can get the same result.

The elapsed time on this original data file for the whole procedure including input and output on an Intel Pentium III 933MHZ CPU and 256M RAM is 12.328 seconds.

Comparing with the non-sorted tiling method on the perspective of time complexity, sorted tiling method has a greater time complexity, because it takes more time to sort the data set. However, the actual elapsed time of the sorted tiling method is shorter than the non-sorted tiling method. The cause of this result is that we use different data structures in these two methods. In the non-sorted tiling method, we use link list to store the data rather than array. It takes more time than array to handle. Therefore, it counteracts the advantage on the time complexity of the non-sorted tiling method.

4.5 Discussion

In this chapter, methods of making tile data on a single source data and multiple source data with or without buffers are presented to facilitate the breaking down of huge amounts of ALTM data and storing them into separate files. The test results based on the

LIDAR data collected in eastern Broward County show that the proposed methods are effective and promising in tiling the terrain data.

Since our current work is focused on initial processing of terrain data for further research, we have not proposed a more complicated model for tiling data. There are still some aspects of tiling data, such as adding some other parameters (Intensity) to analyze and process the original data according to different terrain characters.

After we have acquired the huge original ALTM data, we can use sparse or tile methods to shrink the data set for further processing. One kind of methodology will be addressed in this chapter. That is called the filtering method.

In order to gain more accurate terrain data, we have to remove those data points that do not describe the terrain character, such as building, transportation construction, and vegetation, because they do not reflect the real terrain character. According to the terrain character of Florida, the landscape is relatively flat. Hence, we can apply the following method to remove non-terrain information.

First, we have to partition the original data set into relatively small area and retrieve each area data for further processing. This can guarantee that there are no significant differences on the terrain character in each area. In each piece of area, we sort the data points by the height of the points (the z values of the points). Then we compare the height differences of the points, one after another, in the sorted list. When it comes across a sharp jump of the height difference according to the criterion given, the points in the list after that are removed. Finally, we can output the rest of the points into file. This is what we want to achieve.

The purpose of the filtering method is to remove those data points that do not reflect the terrain character, such as building, transportation construction, and vegetation. In another words, we only want to get those points on the ground. Thus, our job is to find a good way to remove the unuseful points as much as we can.

5.1 Tile filtering method

The tile filtering method is to filter the data set by splitting it into tiles. For filtering the original data file, first of all, we have to acquire the boundary of the survey area from the original file. Then, split the original data into tiles with given width and length. Each point belongs to one tile. The size of tile can be decided according to how large we want each tile. How to choose the size of tile depends on the terrain character of the survey area. It will be distinct due to different cases. The selection of size will cause a significant difference on the result. We will discuss how to choose the size of the tile in detail later in this chapter.

Specifically, we can filter the original data through the steps below:

1) Scan all the data in the original data file, acquire the amount of the data points in the file, which is decided by how many lines are in the file, because each of the lines in the file represents a point.
2) Get the boundary of the survey area by computing the minimum and maximum X, Y value among all the data. Create a rectangular boundary.
3) Inside the rectangular boundary, partition the whole area into tiles with given width and length. Each point belongs to one tile.
4) Scan each data point in the original data file, check whether or not the Z value of each point is within the given range. If it is in the range, continue to compute which tile it belongs to, and add it into corresponding tile point list.
5) Sort each list in ascending sequence according to the Z value of each point.
6) Scan each tile point list, compare the Z value of each point one after another, and check the difference of the Z value between two continuous points in each list. Specifically, use the Z value of the latter point minus that of the previous

point. If it is greater than or equal to the given threshold, remove the second point and the points behind it in the list.

7) Output each tile point list into file.

Our main idea for the filtering method is illustrated in Figure 5.1.1. The vertical coordinate represents the Z value of each point. The horizontal coordinate refers to each point in the sorted list by Z value. Since the Z value reflects the height of each point, when we go through the list, we check the height difference of each point with its previous point. When we come across a sharp jump that is greater than or equal to the given threshold, we can remove the points from it to the end of the list. Since we chose a relatively small area, there is no significant difference on the terrain character, if we find a big jump on the height of the point in the sorted list, we can infer that the point is not on the ground, it may be a building, transportation, construction, vegetation or something like that, which does not reflect the terrain character of that area. Hence, we can remove that point and the points beyond it.

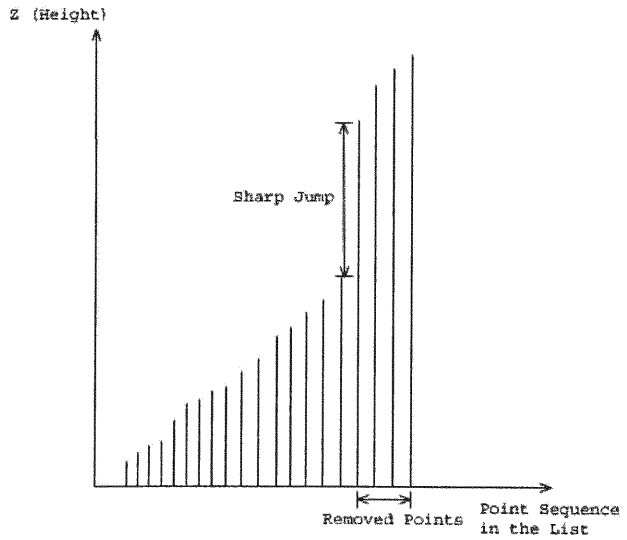

Figure. 5.1.1 Tile Filtering Method

5.1.1 Algorithm of tile filtering method

The Pseudo-code for the tile filtering method is as following:

```
Procedure Tile_Filtering
  01   Read the data file, get the size of data set;
  02   Read file into altmArray a;
  03   a.getMinMaxXYZ(min_x, min_y, min_z, max_x,
       max_y, max_z); // Get the minimum and maximum
       values of x, y, z among all the points;
  04   getNumOfRowCol(min_x, min_y, max_x, max_y,
       width_x, length_y, Cols, Rows); // Count the
       Columns and Rows of all the points area based
       on the given width on X direction and length
       on Y direction;
  05   lst = new std::list<altmPoint> [Cols*Rows]; //
       Create an ALTM point list array with the size
       of Columns*Rows.
  06   For (k=0; k<size; k++) // Go through each
       point of the Array
  07     If ((a[k].getZ() > floor) && (a[k].getZ() <
       ceiling)) Then
  08         lIndexX = (a[k].getX() - min_x)/width_x;
             // Count the X Index of each point based
             on the Columns and Rows Coordinate.
  09         lIndexY = (a[k].getY() - min_y)/length_y;
             // Count the Y Index of each point based
```

```
            on the Columns and Rows Coordinate.
10          lst[lIndexY*cols+lIndexX].push_back(a[k])
            ; // Add the point to the list array
            according to which tile it belongs.
11        End if;
12      End for;
13      For (i=0; i< Cols*Rows; i++) // Go through
        each point list
14        Sort lst[i] by Z value;
15        std::list<altmPoint>::iterator p =
          lst[i].begin();  // begin() returns an
          iterator that designates the first element.
16        bFilter = false; // bFilter is a Boolean
          variable for checking whether or not to
          filter the data.
17        dPreZ = p→getZ();
18        While (p != lst[i].end())
19          If (!bFilter) Then
20            dCurrZ = p→getZ();
21            dDiffZ = dCurrZ - dPreZ;
22            If (dDiffZ >= threshold) Then
23                bFilter = true;
24            End if;
25          End if;
26          If (bFilter) Then
27            p = lst[i].erase(p); // erase() Both
              returns an iterator that designates
              the first element remaining beyond any
```

```
                elements removed, or end() if no such
                element exists.
28          Else
29             dPreZ = dCurrZ;
30             p++;
31          End if;
32       End while;
33    End for;
34    Output each list in the list array into file;
```

Line 01 is used to read the data from the file, after having scanned the whole file, we can acquire the amount of the data points in the file, which is decided by how many lines are in the file, because each of line in the file represents a point.

After getting the number of the point set, through Line 02, we can create an altmArray with the number, and load all the data points in the file into the altmArray.

Next, we need to compute the minimum and maximum value of X, Y, which are used to decide the boundary of the data points in the file. Line 03 is functioning for it.

In Line 04, we compute the number of columns and rows of all the point areas through the given parameter -- width and length. Width is used on the X direction, and length is used on the Y direction. Accordingly, columns can be computed based on the width, because they reflect how many strips can be divided on the direction of X; on the other hand, rows can be computed based on the length, because it reflects how many strips can be divided on the direction of Y. After that, we can split the whole area into tiles. Each point must fall into one tile.

Based on the number of columns and rows, we can create an array which points to altmArray with the size of columns × rows for collecting data points in each tile. Actually, all the points in each tile can be stored in one element of the array, because each element of the array points to an altmArray which is used to store a bunch of data points. The index of the array represents each tile, because we can use the unique indexes (columns, rows) to represent a tile.

Lines 06 to 12 are functioning for collecting the tile data. We can scan each point, if the height of the point is in the given range which is between the variable *floor* and variable *ceiling*, then we can compute the column and row index of each point, in another words, we can determine to which tile the point belongs (Lines 08 and 09 perform this job). For computing the index of a tile in the *aTile* array, we can use the index of the column from Line 09 multiplied by the number of columns plus the index of the row from Line 08. Then add the point to the end of the point list in the list array. In this way, we can collect all the data points into the list array according to which tile it belongs.

Lines 13 to 33 are the main part of the Filtering algorithm. We will go through each point list to process the filtering job. For each point list, first we have to sort the point list by the Z value of each point in an ascending sequence (in Line 16). In Line 15, we create an iterator p that can refer to every point object for checking each point in the list. The Boolean variable "bFilter" is used to check whether or not we need to remove the data. We start the checking procedure from the first element in the list. Hence, we give the Z value of that element to the variable "dPreZ" in Line 17. It is used to store the Z value of the previous point. Then we check the continuous adjacent points, compute the difference of the Z value between the latter point and the previous one. When the difference is greater than or equal to the given

threshold, we can set the Boolean variable "bFilter" as True, that means we need to remove the data points beyond the latter point and itself from the list. If the difference is less than the threshold, store the Z value of the latter point in the variable "dPrez," in other words, we can treat it as the previous point, and continue to check and compare the one beyond it. Lines 18 to 32 are functioning for that. After we go through all the lists, we can output them into one file that is the result of the filtering (Line 34).

When we output the data into files, we have to use different file names to distinguish them. Thus, we use some parameters to create the file name, including the original file name, and the width and length of tile and the threshold value. That can guarantee each tile file has unique and clear file name. For example, the original file name is "573000_2891000," and the width and length of the tile are 5 and 10, the threshold value is 2, then the output tile file name should be "573000_2891000_5_10_2."

We use this method to do a test on the original source data from the survey in eastern Broward County. It is stored in a text file with the capacity of 8,568KB. There are 213,974 points in the file (Figure 5.1.4). We chose both the width and length of the tile as 5m, and the threshold as 0.5m. We got the result showed in Figure 5.1.5 Result of Filtering One Time. We can see from the figure, there are still some points with a relatively greater height. They are illustrated by the red color. The reason why we can't remove those points through one time filtering is because of the partition of the area. We can know that from how our method works. Let us suppose a piece of area we partitioned in the survey area is like the Figure 5.1.2. The black area represents the area with greater height and no significant difference on the height. Those points with lower height are located in the white area. According to

our filtering method, after we partition the area, we have to filter the data from each tile. Our method is to sort the tile by the height of point, find out the point that has a significant height difference with its previous point, and remove the points behind it. Thus, if there are lower points and much higher points in one tile, it will be easier to find out the point with the height difference beyond the given threshold. Also, the relative higher points in such tile will be removed. However, if the tile contains points with no big difference on the height, our method can hardly remove any point from it. The black area in the figure shows this case. If in that area, all the points with close height were those points on the ground, they would be what we want to keep. Otherwise, if they are the points with great height, we can hardly remove those points through our method, because those points have little difference on the height. We can know that from the result in Figure 5.1.3 Sample of Filtering One Time. We can see from the figure, the higher points in three tiles are removed, but the rest of the higher points are left in one tile. Thus, if we can partition those higher points into some tile with much lower points, we can remove those higher points. For example, suppose the black area represents the roof of the building. If part of the roof filled out the tile we partitioned, those points would not be removed. We can only remove such parts of the roof that are partitioned into tiles with some relative lower points like the tiles with white and black areas in the figure. Therefore, we can modify our method to partition the area with different parameters to let points with different combinations in each tile. That will be a great help for removing those higher points.

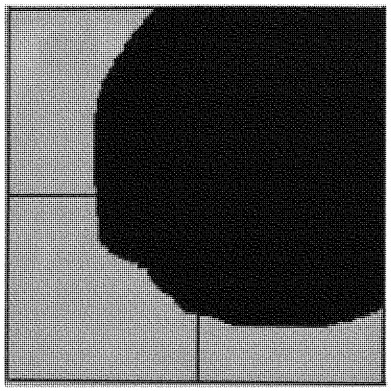

Figure. 5.1.2 Partition Sample

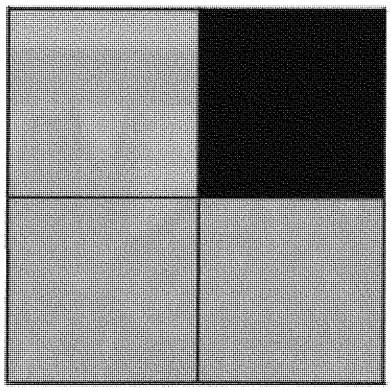

Figure. 5.1.3 Sample of Filtering One Time

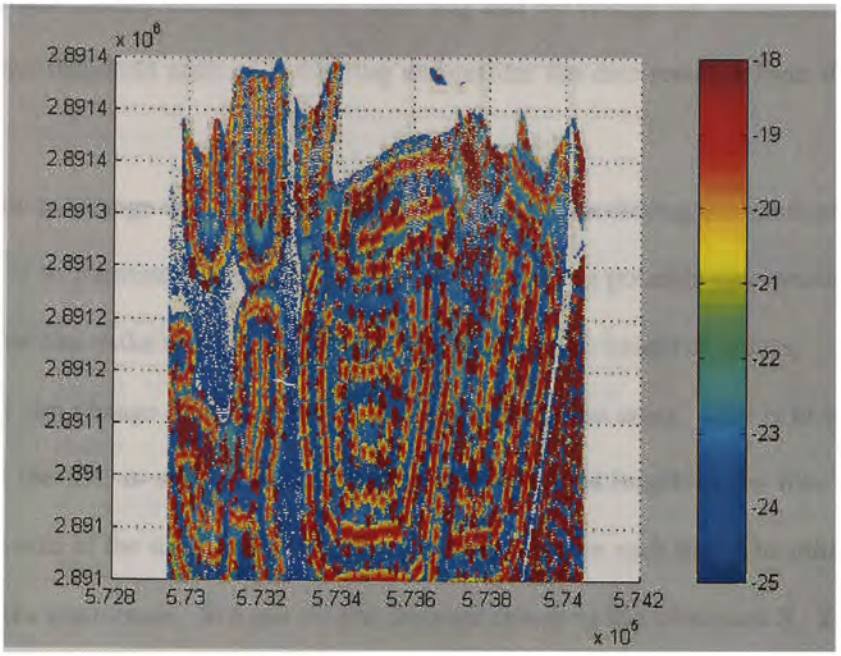

Figure 5.1.4 The Original data (213,974 points, 8,568KB ASCII file)

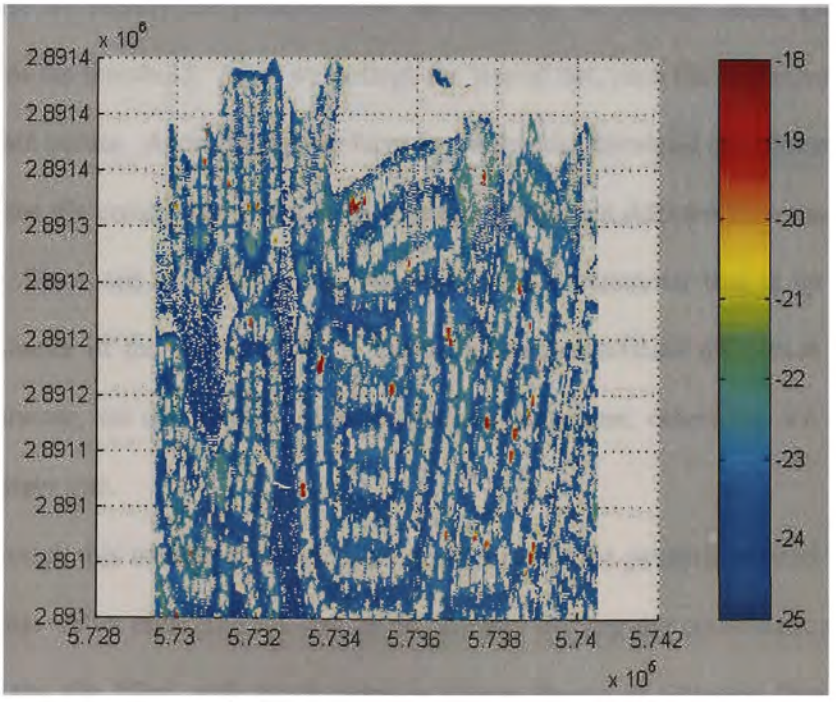

Figure. 5.1.5 Result of Filtering One Time (Width, Length, Threshold: 5, 5, 0.5 m)

We can modify the algorithm in such way that we change the parameter of the partition and threshold after each filtering and refilter the data resulted from the last filtering.

How to change the parameter is the key issue for modifying the method. The main idea is to partition the area into tiles with as many as possible combinations of points. That can make new groups of points to compare the height of points.

We can change the parameter of the partition in two ways. One is to change the size of the tile, in another words, change the width and length of the tile. If we change the size of the tile, we will make new points group in each tile. The other way is to shift the coordinate. We can do that through changing the minimum X, Y value of the data set. That will set up a new coordinate to partition the data set. That also can make new combination of points in each tile. Therefore, we can repartition the data set through shifting coordinate and enlarging the size of tile.

After we modify the parameter for the partition, we have to think about the parameter of the threshold. After we enlarge the size of tile, each tile will cover more area and data points. Accordingly, we have to enlarge the threshold correspondingly, because if the tile covers more area, it will make a significant difference on the height of terrain. Thus, we can enlarge the threshold with a parameter that is up to the terrain character of the survey area. If the there is no significant difference on the terrain character, we can choose a lower enlarged parameter; otherwise, we should choose a larger one.

Through this modified method, we can figure out the problem caused by one time filtering. After enlarging the size of the tile and shifting the coordinate, we can repartition the tile filled with black color in Figure Result of Filtering One Time.

Those points reflected by the black color will be partitioned into tiles with some lower points reflected by the white area. Then it will be removed.

The pseudo-code for modified method is as below:

```
Procedure Tile_Filtering
  01   Read the data file, get the size of data set;
  02   Read file into altmArray a;
  03   a.getMinMaxXYZ(min_x, min_y, min_z, max_x,
       max_y, max_z); // Get the minimum and maximum
       values of x, y, z among all the points;
  04   For (m=1; m<=3; m++)
  05      getNumOfRowCol(min_x, min_y, max_x, max_y,
          width_x, length_y, Cols, Rows); // Count the
          Columns and Rows of all the points area
          based on the given width on X direction and
          length on Y direction;
  06      lst = new std::list<altmPoint> [Cols*Rows];
          // Create an ALTM point list array with the
          size of Columns*Rows.
  07      For (k=0; k<size; k++) // Go through each
          point of the Array
  08          If ((a[k].getZ() > floor) && (a[k].getZ()
              < ceiling)) Then
  09             lIndexX = (a[k].getX() -
                 min_x)/width_x; // Count the X Index
                 of each point based on the Columns and
                 Rows Coordinate.
```

```
10          lIndexY = (a[k].getY() -
            min_y)/length_y; // Count the Y Index
            of each point based on the Columns and
            Rows Coordinate.
11          lst[lIndexY*cols+lIndexX].push_back(a[
            k]); // Add the point to the list
            array according to which tile it
            belongs.
12        End if;
13    End for;
14    For (i=0; i< Cols*Rows; i++) // Go through
      each point list
15        Sort lst[i] by Z value;
16        std::list<altmPoint>::iterator p =
          lst[i].begin();  // begin() returns an
          iterator that designates the first
          element.
17        bFilter = false; // bFilter is a Boolean
          variable for checking whether or not
          filter the data.
18        dPreZ = p→getZ();
19        While (p != lst[i].end())
20          If (!bFilter)  Then
21            dCurrZ = p→getZ();
22            dDiffZ = dCurrZ - dPreZ;
23            If (dDiffZ >= threshold) Then
24            bFilter = true;
25            End if;
```

```
26              End if;
27              If (bFilter) Then
28                 p = lst[i].erase(p); // erase() Both
                   returns an iterator that designates
                   the first element remaining beyond
                   any elements removed, or end() if no
                   such element exists.
29              Else
30                 dPreZ = dCurrZ;
31                 p++;
32              End if;
33          End while;
34      End for;
35      Load the lst into altmArray b;
36      delete [] lst; // Free the memory allocated
        for lst.
37      a=b;
38      If (m<3) Then  // Modify the parameter of
        partition and threshold
39          min_x = min_x - width_x/2;
40          min_y = min_y - length_y/2;
41          width_x = width_x + 1;
42          length_y = length_y + 50;
43          threshold = threshold * 1.2;
44      End if;
45  End for;
46  Output altmArray a into file;
```

In the modified algorithm, we add a For loop in Line 04, for refiltering the data set. After filtering the data set one time, we have to load the result to an altmArray for refiltering (Lines 35 to 37). Next, we modify the parameter of the partition and threshold. In Lines 39 and 40, we change the minimum X, Y value for shifting the coordinate. In Lines 41 and 42, we enlarge the size of tile on the width and length. Finally, we enlarge the threshold in Line 43. We process the filtering three times. The times of filtering can be set according to the requirement. After filtering the data set for a couple of times, we can output the altmArray into a file.

5.1.2 Time Complexity

In the Procedure Tile_Filtering, we first read the source file, load all the data into the altmArray, and compute the mininum and maximum X, Y value. These three steps have the same time complexity, because they all have to scan the whole data set one time. Hence, the time complexity for these steps is O(N).

After that, we scan the whole data set once to collect data into the tile file. The time complexity for this step is O(N).

Finally, we filter the link list of each tile since we have to sort each list element by Z value. Let's suppose the number of the link list is k. The time complexity for sorting each list data is $O(N_k \log N_k)$. Thus, the time complexity for sorting all the lists should be $O(\sum_{i=1}^{k} N_k \log N_k)$.

Since, $\log N_k \leq \log N$, so

$$\sum_{i=1}^{k} N_k \log N_k \leq \sum_{i=1}^{k} N_k \log N$$

Also,

$$\sum_{i=1}^{k} N_k = N$$

So,

$$\sum_{i=1}^{k} N_k \log N_k \leq N \log N$$

Thus, the worst case time complexity of sorting all the lists should be O(NlogN). Since searching for the start index of the filtered point needs to have linear time, the time complexity of this algorithm depends on the time complexity of sorting all the link list data. Therefore, the worst case time complexity of this algorithm is O(NlogN).

5.1.3 Result test and analysis for tile filtering method

We still chose the original source data from the survey in eastern Broward County. We chose both the initial width and length of the tile as 5m, and the initial threshold as 0.5m. The result is indicated in Figure 5.1.6 Result of Refiltering Method. We can see from the figure, the higher points have been removed.

The elapsed time on filtering this original data file on an Intel Pentium III 933MHZ CPU and 256M RAM is 10.765 seconds, including the time for reading and writing the file.

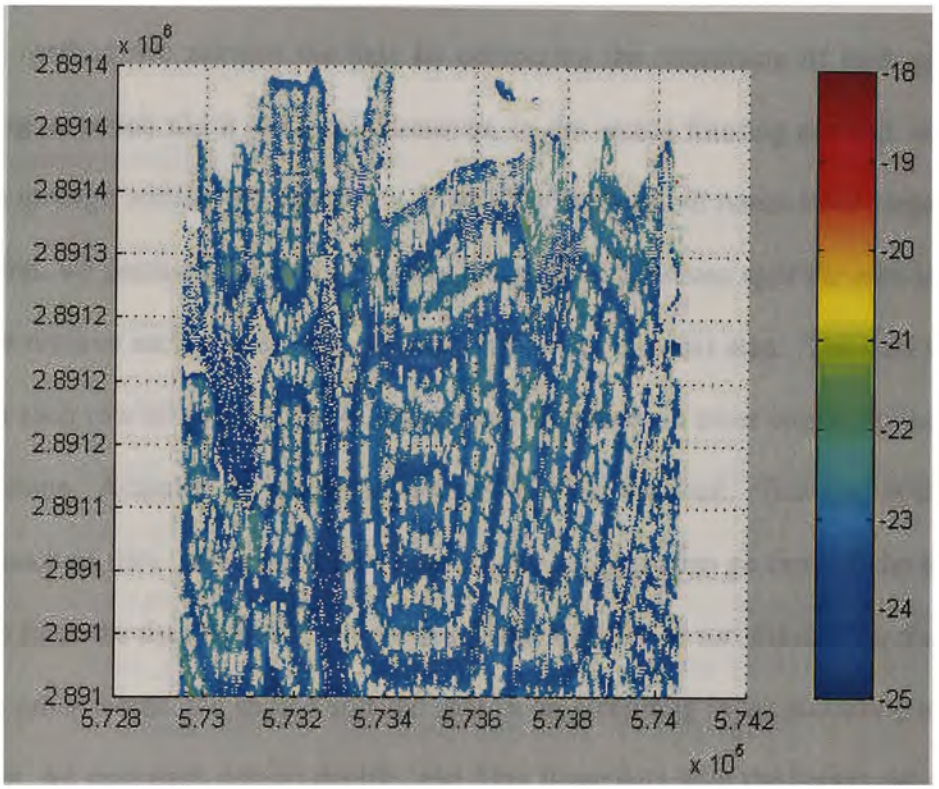

Figure. 5.1.6 Result of Refiltering

5.2 Sorted Filtering method

The main idea of the sorted filtering method is also to partition the survey into tiles, then filter the data from each tile. However, the algorithm is different from the tile filtering method. The difference is how we retrieve the data from the tile. In the tile filtering method, we retrieve the data by computing the coordinate of each point, and indicating to which tile it belongs. However, in the sorted filtering method, we collect tile data through sorting the data set in X and Y directions. It needs three steps to filter data. First, we partition the area along the Y direction. That can split the area into rows. After we retrieve each row of data, we can continue to the next step. The next step is to partition each row of data into strips along the X direction, in other words, split each row into columns. Actually, each column in a row is a tile we want. That makes the whole area a bunch of tiles. After we retrieve the column data, we can go through the third step that is to filter the data from tile. In the third step, we have to sort the data by the Z value first. In other words, we should sort the data in an ascending order according to height. After that, we scan each data in the tile, and filter those data with the height out of range. The range is decided by the given boundary value of height. Those data with the height that are beyond the high boundary or below the low boundary will be removed. At the same time, we compare the height of each data point with their previous ones. If the height difference is greater than or equal to the given threshold, we can remove the data behind it including itself. Since we need to partition the data along X or Y direction in this method, we have to sort the data along the corresponding direction, that is the reason why we called this method the Sorted Filtering method.

The process of collecting data in the tile is the same as in the Sorted Tiling method. After we filter the data set one time, we need to do refiltering work through shifting the coordinate of the data set, modifying the size of the tile, and changing the value of the threshold. The main idea of it is to partition the data set in different tiles. It can make different combinations of the data; thus it will be very useful to filter data.

5.2.1 Algorithm of tile filtering method

The Pseudo-code for sorted filtering method is as below:

```
Procedure FilterOneFile
  01   Read the data file, get the size of data set;
  02   Read file into altmArray arrPts;
  03   arrPts.getMinMaxXYZ(min_x, min_y, min_z, max_x,
       max_y, max_z); // Get the minimum and maximum
       values of x, y, z among all the points;
  04   getNumOfRowCol(min_x, min_y, max_x, max_y,
       width_x, length_y, Cols, Rows); // Count the
       Columns and Rows of all the points area based on
       the given width on X direction and length on Y
       direction;
  05   double shiftDist = 0;
  06   Call Procedure filterYX(arrPts, min_x, min_y,
       strip_width, strip_height, shiftDist,
       diffCriteria, heightFloor, heightCeiling,
```

```
      numRows, numCols, errMsg);
07    If Procedure filterYX Return -1 Then
08        Return -1;
09    End if;
10    For (i=1; i<3; i++)
11        shiftDist = i*10;
12        strip_width = strip_width+1;
13        strip_height = strip_height+50;
14        diffCriteria = diffCriteria*(i+1)*1.2;
15        Call Procedure removeNullData(arrPts);
16        Call Procedure filterYX(arrPts, min_x,
          min_y, strip_width,  strip_height,
          shiftDist, diffCriteria, heightFloor,
          heightCeiling, numRows, numCols, errMsg);
17        If Procedure filterYX Return -1 Then
18          Return -1;
19        End if;
20    End for;
21    arrPts.sortByZ(); // Sort the altmArray arrPts
      data by Z value.
22    Output altmArray arrPts into file;
23    Return 1;
```

Procedure FilterOneFile is functioning to filter one data file. Line 01 is used to read the data from the file, after having scanned the whole file, we can acquire the amount of the data points in the file, which is determined by how many lines are in the file, because each of line in the file represents a point.

After getting the number of the point set, through Line 02, we can create an altmArray with the number, and load all the data points in the file into the altmArray.

Next, we need to compute the minimum and maximum value of X, Y, which are used to decide the boundary of the data points in the file. Line 03 is functioning for it.

In Line 04, we compute the number of columns and rows of all the surveyed areas through the given parameter -- width and length. Width is used on the X direction, and length is used on the Y direction. Accordingly, columns can be computed based on the width, because they reflect how many strips can be divided on the direction of X; on the other hand, rows can be computed based on the length, because they reflect how many strips can be divided on the direction of Y. After that, we can split the whole area into tiles. Each point must fall into one tile. In Line 05, it calls the Procedure filterYXData for filtering.

In Line 06, we filter the whole data set one time by calling Procedure filterYXData. The filtering result will record in the original altmArray *arrPts*. If there are no data removed, Procedure filterOneFile will return –1.

Then we have to do some refiltering work. The main idea is to shift the coordinates of the data set, modify the size of the tile, and change the value of threshold. From Lines 10 to 20, we refilter the data set three times with different parameters of coordinates, size of tiles, and threshold.

```
Procedure filterYX
   01    start_y = start_y - strip_height;
   02    Call Procedure filterRowCol (a, start_x,
         start_y+shiftDist, strip_width, strip_height,
         diffCriteria, heightFloor, heightCeiling,
         numRows, numCols, errMsg)
   03    If Procedure filterRowCol Return -1 Then
   04        Return -1;
   05    End if;
   06    Call Procedure removeNullData(a);
   07    If Procedure removeNullData Return True Then
   08        double end_x, end_y, min_z, max_z;
   09        a.getMinMaxXYZ(start_x, start_y, min_z,
             end_x, end_y, max_z);
   10        getNumOfRowCol(start_x, start_y, end_x,
             end_y, strip_height, strip_width, numRows,
             numCols);
   11    End if;
   12    start_x = start_x-strip_height;
   13    Call Procedure filterRowCol(a,
         start_x+shiftDist, start_y, strip_height,
         strip_width, diffCriteria, heightFloor,
         heightCeiling, numRows, numCols, errMsg);
   14    If Procedure filterRowCol Return -1 Then
```

```
15        Return -1;
16    End if;
17    Return 1;
```

Procedure filterYX is functioning to filter the data set. In Line 01, we modify the *start_y* for shifting the coordinate. Then we call Procedure filterRowCol for filtering one time. During the process of filtering, we will set the Z value of points that should be removed as Null. After we filter the data set one time, we can call Procedure removeNullData to remove those points with Null Z value. If we indeed remove data after the first time filtering, we have to compute the minimum and maximum X, Y value again. According to those values, we can compute the number of rows and columns. In Line 12, we set the *start_x* for shifting the coordinate, and filtering the new data set again.

```
Procedure filterRowCol
   01    long int i=0, j=0, count=0;
   02    double low_y = start_y;
   03    double high_y = start_y + strip_height;
   04    long int start_index = 0;
   05    long int end_index = 0;
   06    unsigned long int size = a.getSize();
   07    a.sortByY();  // Sort altmArray a by the Y
         value.
```

```
08    For (i=0; i<=numRows; i++)
09        While ((j < size) && (a[j].getY() >= low_y)
          && (a[j].getY()<high_y))
10            count++;
11            j++;
12        End while;
13        If (count > 0) Then
14            altmArray rowPts(count); // Create
              altmArray for storing row data points
15            end_index = j-1;
16            getRowPoint(a, start_index, end_index,
              rowPts); // Store row data into
              altmArray rowPts.
17            Call Procedure filterRow (rowPts,
              start_x, strip_width, diffCriteria,
              heightFloor, heightCeiling, numCols, i,
              errMsg);
18            If Procedure filterRow Return -1 Then
19                Return -1;
20            End if;
21            Call Procedure writePointBackToArray (a,
              start_index, end_index, rowPts);
22            count = 0;
23            start_index = end_index + 1;
24        End if;
```

```
25        low_y = high_y;
26        high_y = high_y + strip_height;
27    End for;
28    Return 1;
```

This procedure is used to split the data set into strips along the Y directions, retrieve the data in each strip, call Procedure filterRow to filter each row of data, and write the result back to the original row data set.

First, we need to sort the data set by the Y value (in Line 07). Next, we will retrieve each row of data for further processing. Lines 08 to Line 27 are a For loop for each row. Long int variable j is functioning as the index for the data set array. Double variable *low_y* stores the low boundary of each row; double variable *high_y* stores the high boundary of each row. Since we start from the minimum Y value point, we set the initial *low_y* as *min_y*, and the initial *high_y* as *min_y* plus *strip_height*. After one loop, we have to modify the *low_y* and *high_y*'s value, because we change another row to retrieve data.

```
Procedure filterRow
   01    unsigned long int i=0, j = 0;
   02    unsigned long int k = 0, count=0;
   03    double low_x = min_x;
```

```
04   double high_x = min_x + strip_width;
05   unsigned long int start_index = 0;
06   unsigned long int end_index = 0;
07   unsigned long int size = a.getSize();
08   a.sortByX(); // Sort altmArray a by the X value.
09   For (i=0; i<=numCols; i++)
10       While ((j < size) && (a[j].getX()>=low_x) &&
         (a[j].getX() < high_x))
11          count++;
12          j++;
13       End while;
14       If (count>0)
15          altmArray colPts(count);
16          end_index = j-1;
17          getRowPoint(a,start_index,
         end_index,colPts);
18          Call Procedure filterStrip (colPts,
            diffCriteria, heightFloor, heightCeiling,
            errMsg);
19          If Procedure filterStrip Return -1 Then
20             Return -1;
21          End if;
22          Call Procedure writePointBackToArray
            (a,start_index, end_index, colPts);
23          count = 0;
```

```
24            start_index = end_index + 1;
25        End if;
26        low_x = high_x;
27        high_x = high_x + strip_width;
28    End for;
29    Return 1;
```

This procedure is used to split the row of data retrieved from Procedure filterRowCol into columns along the X directions, call Procedure filterStrip to filter each column data, and write the result back to the original column data set. That can partition the row data into columns according to the given width of each tile.

First, we need to sort the data set by the X value (in Line 08). Next, we will retrieve each column data for further processing. Lines 09 to 28 are a For loop for each column. Long int variable j is functioning as the index of the data set array. Double variable *low_x* stores the low boundary of each row; double variable *high_x* stores the high boundary of each row. Since we start from the minimum Y value point, we set the initial *low_x* as *min_x*, and the initial *high_x* as *min_x* plus *strip_height*. After one loop, we have to modify the *low_x* and *high_x*'s value, because we change another column to retrieve the data.

```
Procedure filterStrip
   01    unsigned long int size = a.getSize();
   02    If (size>1) Then
```

```
03        For (i=0; i<size; i++)
04            If ((a[i].getZ()>heightCeiling) ||
              (a[i].getZ()<heightFloor))
05                a[i].setZ(C_ALTM_NULL_Z);  // Filter
                  the point with the height beyond
                  heightCeiling or below heightFloor.
06            End if;
07        End for;
08        a.sortByZ();
09        unsigned long int start_index =
              getCriticalStartIndex (a, diffCriteria,
              errMsg);
10        If (start_index == -1) Then//if start_index
              is incorrect
11            Return -1;
12        End if;
13        If (start_index < size) Then
14            For (i=start_index; i<size; i++)
15                a[i].setZ(C_ALTM_NULL_Z);
16            End for;
17        End if;
18    Else
19        a[0].setZ(C_ALTM_NULL_Z);
20    End if;
21    Return 1;
```

Procedure filterStrip is the main part of the Filtering algorithm. After we retrieve the data from the column of each row, we can use this procedure to filter the data. Actually, each column in any row is the tile we partitioned based on the given width and length. First, we have to get the size of the tile data set (Line 01). If the data set contains more than one point, we go through each point; otherwise, remove the data by setting the Z value as Null. Next, we have to check the Z value (height) of each point, if it is out of the range, we need to remove the point by setting the Z value of the point as Null. Then, we sort the data set by Z value. In Line 09, we have to check from which point we need to remove the point in the data set. Since the data set has been sorted according to the Z value, we can check the height difference between each point and its previous point. If the height difference is beyond the given threshold, we can remove it from the checked point to the end of the data set. Variable *diffCriteria* is used to store the value of threshold. After we get the index of the starting point that should be removed, we can set the Z value of those points from the starting point to the end of the data set as Null.

```
Procedure writePointBackToArray
   01    unsigned long int i = 0;
   02    For (i=start_index; i<=end_index; i++)
   03        a[i] = rowPts[i-start_index];
   04    End for;
```

Procedure writePointBackToArray is used to write the processed data back to the original array.

5.2.2 Time Complexity

In sorted filtering algorithm, we also first read the source file, load all the data into the altmArray, and compute the mininum and maximum X, Y value. These three steps have the same time complexity, because they all have to scan the whole data set one time. Hence, the time complexity for these steps is O(N).

After that, we have to retrieve some strip data for filtering process. Before we retrieve any strip data along X or Y direction, we have to sort the data along the corresponding direction, that decides the time complexity of this algorithm. In the procedure filterRowCol, we have to sort the whole data set by Y value, thus, the time complexity should be O(NlogN). However, in other procedures, we do not need to sort the whole data set, but only part of it, so the worse case time complexity for those sorted procedures is O(NlogN). Furthermore, other procedures without the sorting job only need linear time, so the time complexity for the sorted filtering method is O(NlogN).

5.2.3 Result test and analysis for sorted filtering

We still chose the original source data from the survey in eastern Broward County. We chose both the initial width and length of the tile as 5m, and the initial threshold as 0.5m. The result is indicated in Figure 5.2.1 Result of Sorted Filtering Method. We can see from the figure, the higher points have been removed. The elapsed time for processing all the jobs is 11.25 seconds.

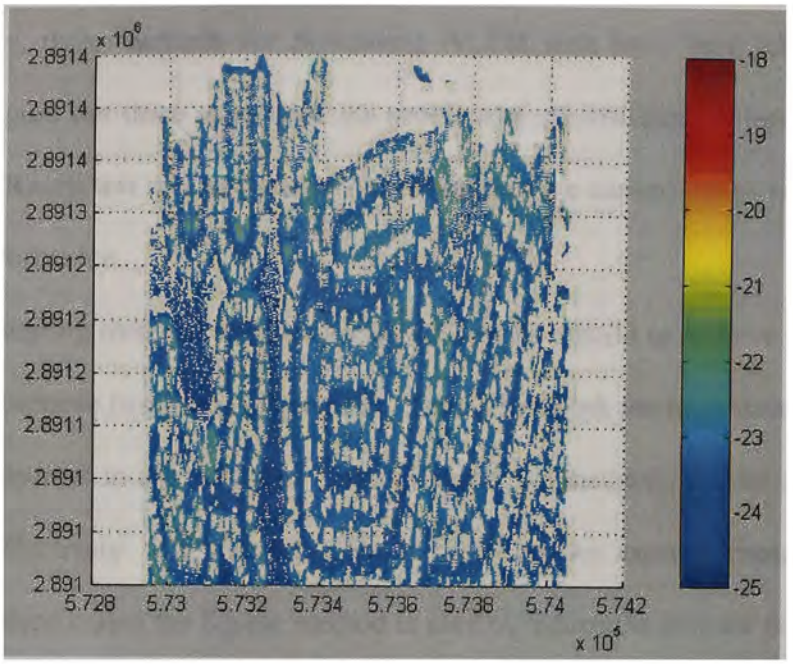

Figure. 5.2.1 Result of Sorted Filtering Method

In this study, three methods for processing ALTM data have been addressed. They are used to figure out three aspects of the problem of ALTM data, however, they are related tightly. Result test and analysis of our techniques are carried out to verify the correctness and performance.

For the resampling method, we concentrate on how to rebuild or retrieve the data set. Therefore, we propose two kinds of methods. One is to shrink the huge data set; the other is to retrieve data set in any polygon. Through these two methods, we can simplify our survey object effectively. The simplification is based on two aspects: capacity and shape. The first method called the Sparse method is used to reduce the amount of ALTM points, but it keeps the terrain character at the mean time. This is sort of a simplification on the capacity. It will be of great help on further processing and storage. Furthermore, the retrieving method is working as the simplification on the shape, because we can exploit it to cut any data set in a polygon shape according to our requirement. Based on our present work, we can suggest some new research content in these two aspects in the future. For the sparse method, we can design more sophisticated and accurate methods to shrink the data set and match the terrain character. We can introduce more parameters such as intensity of the ALTM point or some new criteria to get the representative point of each partitioned area. On the other hand, for retrieving data from a certain area, we can design new algorithms to retrieve data from some more complicated geometrical shapes, in order to satisfy more requirements of visualization.

The second method we discussed in the study is the Tiling method. That is used to partition the huge data set into small pieces. It is very helpful for storage and further processing. We implemented the tiling method through two different ways. One is non-sorted tiling, the other is sorted. The non-sorted method has a simpler algorithm and less time complexity, and is easier to implement. However, the Sorted method has better performance on processing huge amounts of data. Based on our present research, the speed and efficiency of processing and storage should be increased by improving the algorithm and data structure in future work.

The third part of this study is focusing on the filtering method. That is the key part of processing ALTM data. Two different methods were addressed in the study. As a matter of fact, the main idea of these two methods has no significant difference. They both filter the data in the area with given size through checking the starting point index of the sharp jump on the height difference. After that, we can change the size and refilter. From the test result, we know this kind of method can effectively handle the survey area without too much undulation. Since the filtering criteria for this kind of method is linear, it cannot work well on the terrain with massive undulation. Therefore, we can do a lot more work on designing a non-linear filtering criteria function for different terrain character in future work. It will make a big breakthrough for the filtering method.

List of References

[Bruce2000] Bruce Eckel, "Thinking in C++"Prentice Hall, 2nd edition, Published March 2000.

[Bruno1998] Bruno R. Preiss, John Wiley, "Data Structures and Algorithms: With Object-Oriented Design Patterns in C++," Published August 1998.

[Carter1999] W E Carter, R L Shrestha, S P Leatherman, "Airborne Laser Swath Mapping: Applications to Shoreline Mapping", 1999.

[Clarke1995] Clarke, Keith C., "Analytical and Computer Cartography 2nd Edition", pp. 207-208, 1995.

[Carter1997] Carter, W. E, R. L. Shrestha, P.Y. Thompson, and R. G. Dean; "Project LASER: Final Report to FDEP," Department of Civil Engineering, University of Florida, Gainesville, FL 32611, pp 27, April 4, 1997.

[Estep1994] Estep, L.L., Lillycrop, W.J., and Parson, L.E., "Estimation of Maximum Depth of Penetration of a Bathymetric Lidar System Using a Secchi Depth Database," Marine Technology Society Journal, Vol. 28, No. 2, pp. 31-36, 1994.

[Everett2000] Everett McKay, Mike Woodring, "Debugging Windows Programs: Strategies, Tools, and Techniques for Visual C++ Programmers," Addison-Wesley, Published August 2000.

[Hawg1998] Hawg, P.A., Walsh, E.J., Krabill, W.B., Swift, R.N., Manizade, S.S., Scott, J.F., Earle, M.D., "Airborne Remote Sensing Applications to Coastal Wave Reasearch," Journal of Geophysical Research, Vol. 103, No. C9, pp. 18,791-18,800, 1998.

[Herbert1998] Herbert Schildt, McGraw Hill, "C++: The Complete Reference, 3rd Edition" 3rd edition, Published June 1998

[Irish1999] Irish, J.L., W.J.Lillycrop, "Scanning Laser Mapping of the Coastal Zone: The SHOALS System," ISPRS Journal of Photogrammetry & Remote Sensing , Vol. 54, Nos. 2-3, pp. 123-129, 1999.

[Ivor1998] Ivor Horton, "Beginning Visual C++ 6," Wrox Press, Published September 1998

[Krabill1995] Krabill, W.B., R.H. Thomas, C.F. Martin, R.N. Swift, and E.B. Frederick; "Accuracy of Airborne Laser Altimetry Over the Greenland Ice Sheet," International Journal Remote Sensing, Vol. 16, No. 7, pp 1211-1222, 1995a.

[Mike1999] Mike Blaszczak, "Professional MFC With Visual C++ 6," Wrox Press, Published August 1999.

[Milbert1991] Milbert, D.S.; Computing GPS-Derived Orthometric Heights with Geoid90 Height Model, Presented at ACSM Fall Meeting, GIS/LIS, Atlanta, GA, 1991.

[Nicolai1999] Nicolai M. Josuttis, "The C++ Standard Library: A Tutorial and Reference," Addison-Wesley, Published August 1999.

[Parson1996] Parson, L.E., Lillycrop, W.J., and Irish, J.L., "Surveying Florida Bay Using Airborne Lidar Technology," Proceedings, 2nd International Airborne Remote Sensing Conference and Exhibition, Environmental Research Institute of Michigan, June 24-27, San Francisco, CA, pp, 1996.

[Robert1998] Robert Sedgewick, "Algorithms in C++, Third Edition," Addison-Wesley, 3rd edition, Published December 1998.

[Smith1999] Smith, R.A., West, G.R., "Airborne Lidar: A Surveying Tool for the New Millennium," Proceedings, Oceans '99 MTS/IEEE, 13-16 September, Seattle, WA, 1999.

[Thomas1990] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, "Introduction to Algorithm," MIT Press, Hardcover, Published July 1990.

[Whitman2000] Whitman D., K. Zhang, S.P. Leatherman, W. Robertson, and Baig S., 2000. An Airborne Laser Topographic Mapping Study of Eastern Broward County, Florida with Applications to Hurricane Storm Surge Hazard, American Geophysical Union Spring Meeting, Washington DC.

[West1999] West, G.R., and W.J. Lillycrop, "Feature Detection and Classification with Airborne Lidar - Practical Experience," Proc. Shallow Survey 99, October 18-20, Sydney, Australia, 1999.

[Zhang2000] Zhang K., D. Whitman, S.P. Leatherman, W. Huang, W. Robertson, R. Shrestha, and W. Carter, 2000. Airborne Laser Mapping of the Erosion Caused by Hurricane Floyd near Vero Beach, Florida, American Geophysical Union Spring Meeting, Washington DC.