AN ALGORITHM FOR CLIPPING POLYGONS OF LARGE GEOGRAPHICAL DATA

by

AREEJ SALEH ALGHAMDI

A THESIS

Presented to the Department of Computer and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Master of Science

March 2017

THESIS APPROVAL PAGE

Student: Areej Saleh Alghamdi

Title: An Algorithm for Clipping Polygons of Large Geographical Data

This thesis has been accepted and approved in partial fulfillment of the requirements for the Master of Science degree in the Department of Computer and Information Science by:

Hank Childs Chairperson

and

Scott L. Pratt Dean of the Graduate School

Original approval signatures are on file with the University of Oregon Graduate School.

Degree awarded March 2017

THESIS ABSTRACT

Areej Saleh Alghamdi

Master of Science

Department of Computer and Information Science

March 2017

Title: An Algorithm for Clipping Polygons of Large Geographical Data

We present an algorithm for overlaying polygonal data with regular grids and calculating the percentage overlap for each cell in the regular grid. Our algorithm is able to support self-intersecting polygons, meaning that some spatial regions may be covered by two or more polygons. Our algorithm is able to identify these cases and eliminate redundant polygons, preventing erroneous results. We also present an optimized version of our algorithm that uses spatial sorting through interval trees, and provide a performance comparison between the optimized and unoptimized versions. Finally, we apply our algorithm to geography data, specifically of bark beetle infestation.

CURRICULUM VITAE

NAME OF AUTHOR: Areej Saleh Alghamdi

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene, OR

DEGREES AWARDED:

Master of Science, Computer and Information Science, 2017, University of Oregon
Bachelor of Science, Computer and Information Science, 2014, University of Oregon

AREAS OF SPECIAL INTEREST:

Scientific Visualization
High Performance Computing

GRANTS, AWARDS, AND HONORS:

King Abdullah Scholarship for a Bachelor Degree, Ministry of Higher Education, Saudi Arabia, March 2009 – March 2014

King Abdullah Scholarship for a Master Degree, Ministry of Higher Education, Saudi Arabia, April 2014 – March 2017

ACKNOWLEDGMENTS

First and foremost, I would like to express my gratitude to Allah (God) for giving me the ability and strength to complete this work successfully.

I cannot begin to express my gratitude to my family for their genuine love and encouragement all through this journey. To my Father and Mother, for their unconditional and endless love and care throughout my life, who raised me to be the person I am today that always aspire to the best, and whom I consider my greatest role models in life. To my husband, my soul mate, for his constant love, care, motivation, and understanding. Also, to my siblings for always cheering me up and making me feel loved, appreciated and proud for whatever that I do small or big. Also, for my two little angels, the yet to understand anything, Maya, and the yet to be born my baby boy…I love you all.

My greatest appreciation and sincere thanks go to my advisor and mentor Dr. Hank Childs, for his willingness to offer his support, valuable feedback, and especially for his confidence in me. Dr. Childs, a great and valuable portion of my knowledge and love for the field of computer science came from you and I am so honored to have been one of your students.

This thesis is dedicated to my parents, my husband, and my siblings.

TABLE OF CONTENTS

Chapter Page

LIST OF FIGURES

LIST OF TABLES

Table Page

CHAPTER I

INTRODUCTION

The use of interactive and scientific visualization and data analysis techniques has become a necessity, and scientific researchers in various fields are becoming more inclined to use these techniques. It helps them understand their collected data, simulate experiments or propose methodologies to gain a better perspective into them. In this thesis, we explore visualization techniques to assist with problems from the geography domain, specifically those coming from professor Chris Bone from the Geography department.

The data that inspired this project is a large scientific data set collected over a period of 18 years. It has been collected aerially from tree-covered areas in the United States. It reflects the loss and "mortality over millions of hectares of forests" caused by the mountain pine beetle, which is "a native insect that occurs in pine forests over much of western North America, extending from northern Mexico to northwestern British Columbia (BC) and from the Pacific Ocean east to the Black Hills of South Dakota (Wood 1982)" [1].

This collected data is important because it captures the harmful effect of the mountain bark beetle insect on pine forests. Professor Bone is studying and analyzing this data to further understand the history and effect of the bark beetle on pine trees, one of the greatest national treasures. The research in this thesis was inspired by the idea of helping him to visualize his data.

The data is divided into five geographical regions of the United States map: Northern, Intermountain, Pacific Northwest, Pacific Southwest, Rocky Mountain, and Southwest. For every region, there is a shapefile that contains the data collected aerially of the areas of trees killed by the bark beetle. Each such region is stored as a polygon. Even

though each geographical region's data is independent of the others, within a single region it might overlap with that of another region. This overlap causes miscalculations and produces erroneous results.

This thesis describes an approach to calculate the percentage of trees attacked in a certain area while dealing with the problem of the overlap within the data. The goal is to create 2D histograms with a "percent filled" for each bin. The calculation of the percentage of a bin is done by calculating the area covered by the bin with the data that overlaps with it and then dividing it by the whole area of the bin. However, a naïve approach would lead to over-counting of the area covered, due to the overlap. We solve this problem by efficiently identifying overlapping data and clipping away the redundant regions to get the correct answer.

The contribution of this thesis is twofold. First, it describes an algorithm for efficiently and correctly producing 2D histograms that enable geographers to understand the impact of bark beetle on pine trees. This approach can be applied to other types of data as well. The second contribution is a study of the efficiency of the approach, varying over several configurations and choices for search structures.

In Chapter II, we survey previous work in polygon clipping and finding intersections between polygons. Chapter III describes the data that inspired this thesis, how it is configured and the specific problems with it, and provides samples of the data processed and visualized before and after the problems are addressed. In Chapter IV, we present our approach and describe the algorithms we developed for mitigating the problems within the data. Finally, in Chapter V we present our experiments and results.

CHAPTER II

RELATED WORK

In this chapter, we survey previous work in polygon clipping and finding intersections between polygons.

Weiler and Atherton [2] present an algorithm for removing hidden lines and surfaces. The algorithm can clip two general concave polygons with holes. It outputs a polygon of the same form as the input polygons. That means the outputted polygons can be entered back to the system to be processed further. The complexity of the final produced image is related to the processing time.

Huang and Liu [3] describe a new line clipping algorithm that clips general polygons. The algorithms transform the line segment from clipping to a horizontal line using shearing transformations. Also, using the same parameters as the ones used to transform the line, every edge of the polygon is transformed with shearing transformation. These transformations simplify the clipping resulting from faster processing because of the reduced number of calculations compared to previous clipping algorithms.

Sproull and Sutherland [4] developed a clipping divider. It is a technology that allows a display system to present a magnified portion of any large image. It frees the programmer from the hustle of dealing with the difficulties of bit-packing and resolution in displays.

Cyrus and Beck [5] describe a clipping algorithm and how it was implemented in 2D and 3D. The algorithm checks for intersection between a line and a convex planar polyhedron. Their implementation of the algorithm can run on parallel which results on reduction in execution time.

Sutherland and Hodgman [6] introduce reentrant polygon clipping. It uses 2D polygon clipping to render images in 3D by removal of hidden surfaces. This algorithm applies to convex, planar or non-planar polygons. The polygons in this algorithm are defined regarding their vertices only, not edges. This approach proved to simplify the process of clipping polygons. The extension to the algorithm can deal with concave and self-intersecting polygons by producing degenerate edges.

Barsky and Liang [7] present a new method of line clipping in 2D, 3D, and 4D against a rectangular clipping window. This algorithm maps the line segment to be clipped into a parametric representation. The main advantage claimed for this algorithm is that it rejects invisible segments, i.e., it rejects segments out of the boundary of the clipping window, and also that the algorithm can be generalized for clipping against any convex polygon. The authors claim that this algorithm is faster than the algorithm developed by Sutherland and Hodgman [6] but requires floating point computations. The main limitation is that the clip polygon must always be a rectangle with the sides parallel to the axes.

Nicholl et al. [8] developed a clipping algorithm of a rectangular window for clipping lines in 2D. The algorithm performs better than the Barsky-Liang algorithm and Cohen-Sutherland algorithm [7], [9:113], and the performance comparisons between their algorithm and the two old ones are machine independent. The algorithm finds the points of intersections between the line segment and the boundaries of the window defined by x-left and –right, and y-top and –bottom. If an intersection exists, the coordinates of endpoints of the line segment intersecting with the window are calculated.

Maillot [10] describes a 2D clipping algorithm for clipping polygons that adopts the line clipping algorithm of Cohen-Sutherland. The author chose this algorithm rather

than the Weiler and Artherton [2] or Barsky and Liang [7] because it has an efficient process for acceptance and rejection cases and also because it covers more applications, as it can handle both integer and floating point computations depending on how it is implemented.

Day [11] presents a new algorithm for line clipping that can be used for both floating point arithmetic and integer arithmetic, i.e., in object and image spaces respectively. The clipping is done against rectangular windows. In comparison to other algorithms, this algorithm uses a fewer number of operations for the cases where the line segment being clipped resides completely within the clipping window. The algorithm also performed better with smaller window sizes.

Moller T. [12] introduces intersections tests for testing if two triangles intersect. The tests that can be used in developing algorithms for collision detection. The algorithm computes plane equations for the triangles and uses rejection of trivial points as well as computation of intersection intervals to test if the two subject triangles are intersecting or are co-planar.

Greiner and Hormann [13] present an algorithm for clipping 2D arbitrary closed polygons. The algorithm handles the general case of arbitrary polygons as did Weiler and Atherton and Vatti algorithms [2], [14]. The general case of arbitrary polygons is the case when the intersecting polygons are not convex and might self-intersect. This algorithm as well as the similar previous and compared ones [2], [14] all find the points of intersection between the two subject intersecting polygons. In this algorithm, they use doubly linked lists to represent all the polygons, i.e., the intersecting polygons as well as the resulted

polygons of the intersection between the two original ones are represented as doubly linked lists. Their results proved that their algorithm is faster than Vatti's algorithm [14].

Liu et al. [15] proposed a polygon clipping algorithm for concave and convex polygons with holes. The algorithm uses entry and exit point classification for intersection points, and they used a single linked list data structure. The algorithm is based on three main phases: intersection points are first found between the clip and the subject polygons and their entry/exit status and added to the linked list. Secondly, the polygon's edges are clipped using line clipping, and lastly, the list of the point of intersections is traversed and the resulting polygon is constructed. The algorithm used for clipping the line against any polygon is Liu's Algorithm, the first author of this paper.

Held [16] presents ERIT, a set of intersection tests for finding the intersection between pairs of a polygon and a polygon or a polygon and a line segment. Its 3D intersection works with lines segments, triangles, cylinders, cones and a concave or convex rotated circular arc around an axis. In 2D, an algorithm for finding points of intersections between a triangle and a line segment is provided as well as a small set of other intersection tests. They provide a triangle/triangle test in 3D, but their approach is to minimize it to a problem of a 2D triangle/line segment intersection test.

Źalik [17] introduces two new algorithms for finding intersection points between polygons. Both algorithms are sweep-line algorithms and can handle polygons with holes in them. One is a mid-point between time complexity and difficulty of implementation, and the second is faster but is harder to implement. The first is called a set-based intersection algorithm and the second is called a binary search tree intersection algorithm.

[18], [19], [2], [20] are convex hull algorithms for finding convex hull intersection points and sorting points sets. [2] is a variation of the Graham's algorithm [18] that has improved efficiency.

We have not fully adopted or based our algorithm or techniques used in any previous methods or algorithms because of how different our problem is. We are dealing with data with polygons with different types, but we use a filter to transform all polygons into a single type: triangles. The clipping of triangles is done on a square clipping window. Moreover, the intersection is found between two triangles at a time. The intersection area can be of any type and is calculated after sorting the points of intersections on clock-wise order.

CHAPTER III

DATA DESCRIPTION

The data is of the geographical region of the United States, divided into the five geographical regions: Northern, Intermountain, Pacific Northwest, Pacific Southwest, Rocky Mountain, and Southwest. Each region has its own data file. Areas of lost trees are represented by an enclosed region of a list of points (2D coordinates). From a computer scientist's point of view, the data is stored in Shapefiles for every region. "A shapefile stores nontopological geometry and attribute information for the spatial features in a data set. The geometry of a feature is stored as a shape comprising a set of vector coordinates" [21]. The area of lost trees in a specific region is represented by enclosed regions of polygons. Polygons are of different types: triangles, quadrilaterals, etc. When the polygons overlap, an overcounting occurs when performing certain calculations. What we mean by data overlapping is that multiple polygons intersect. In this thesis specifically, we are concerned about the overlap of polygons because the overlap happens within bins. The polygons cover a certain percentage of that bin, and the goal is to calculate that percentage. Polygon overlaps cause overcalculation when an area of intersection between any number of polygons is overcounted. Figure 1 shows two triangles overlapping. Within the highlighted bin, T1 and T2 intersect. The intersection area in red is considered twice when naïvely calculating the area that the two small triangles are covering in the blue bin. Figure 2 shows a 2D histogram of 4 million triangles of the Intermountain (IM) region of the bark beetle data in a 1000 by 1000 grid. This histogram is of data processed without dealing with the overlap within the data. The percentages covered in the bins ranges from 0 to

277.7%. This shows that overlap among the polygons within the bins cause overcalculations of the area covered and subsequently the percentage covered of the cell.

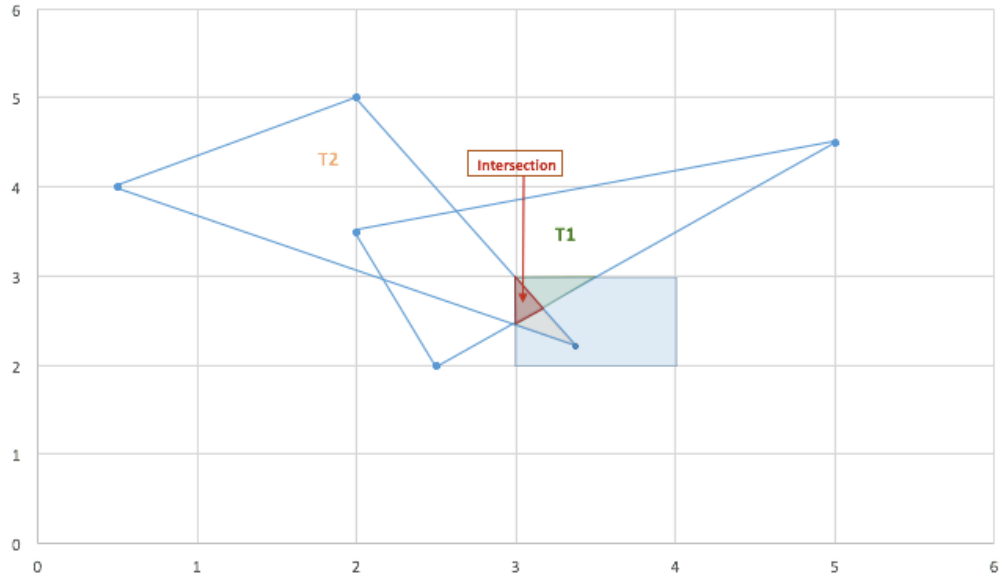

FIGURE 1. Data Overlap Illustration.

Figure 3 shows a 2D histogram of 4 million triangles of the bark beetle data in the IM region in a 1000 by 1000 grid. In this case, the overlap among the polygons has been dealt with by our algorithm. Figure 4 shows a different data set that is smaller in comparison than the size of the data set of the bark beetle, but it produces 2D histograms that make it easier to detect the difference in the results produced when the overlap between the polygons is dealt with and when it is not. Figure 4 is the 2D histogram of the Australia map where overlap among polygons is ignored, and Figure 5 is of the same data set but with overlaps among polygons removed.

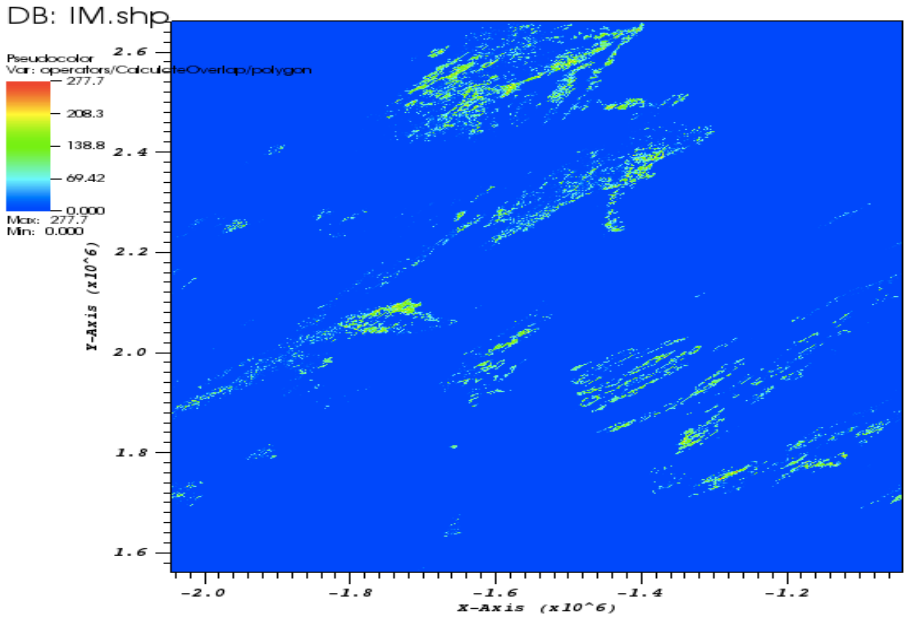

FIGURE 2. 4 Million Triangles of the Intermountain (IM) Data with Polygon Overlap.

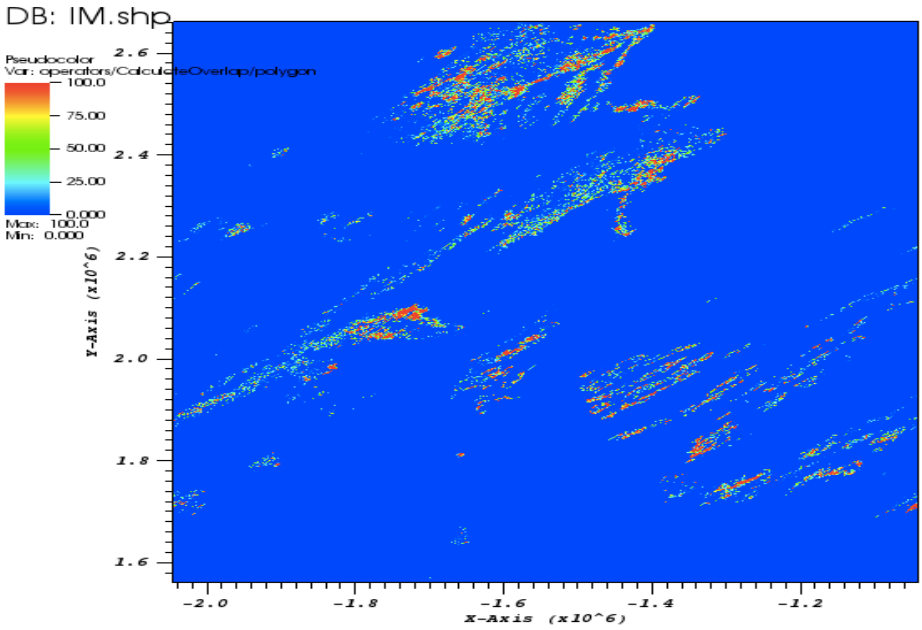

FIGURE 3. 4 Million Triangles of IM Data without Polygon Overlap.

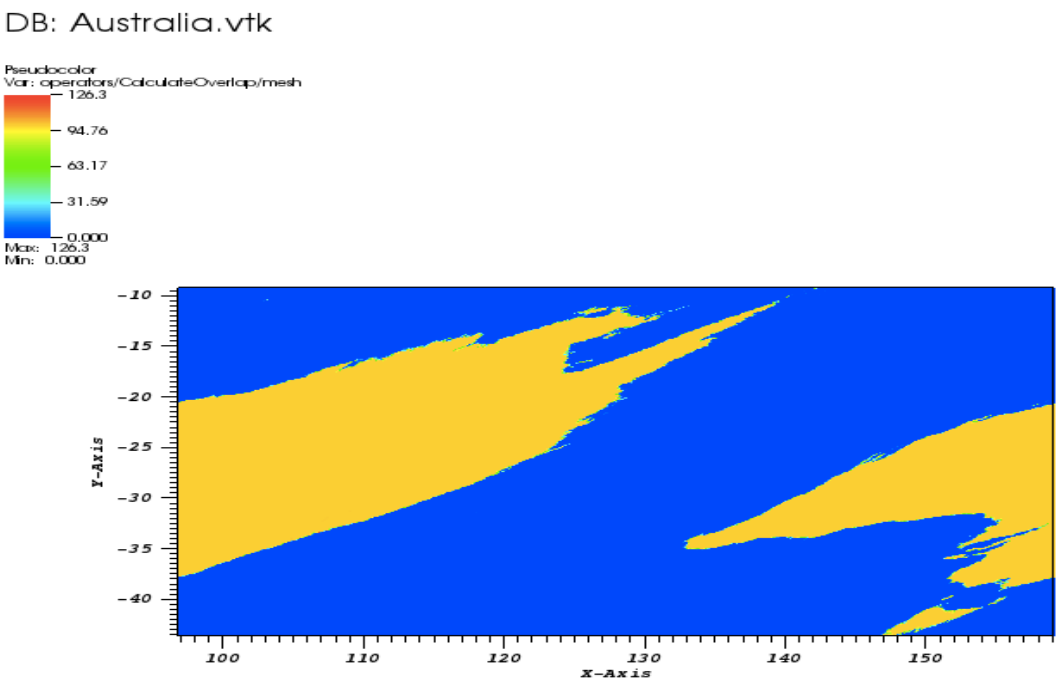

FIGURE 4. Australia with Overlap among Polygons.

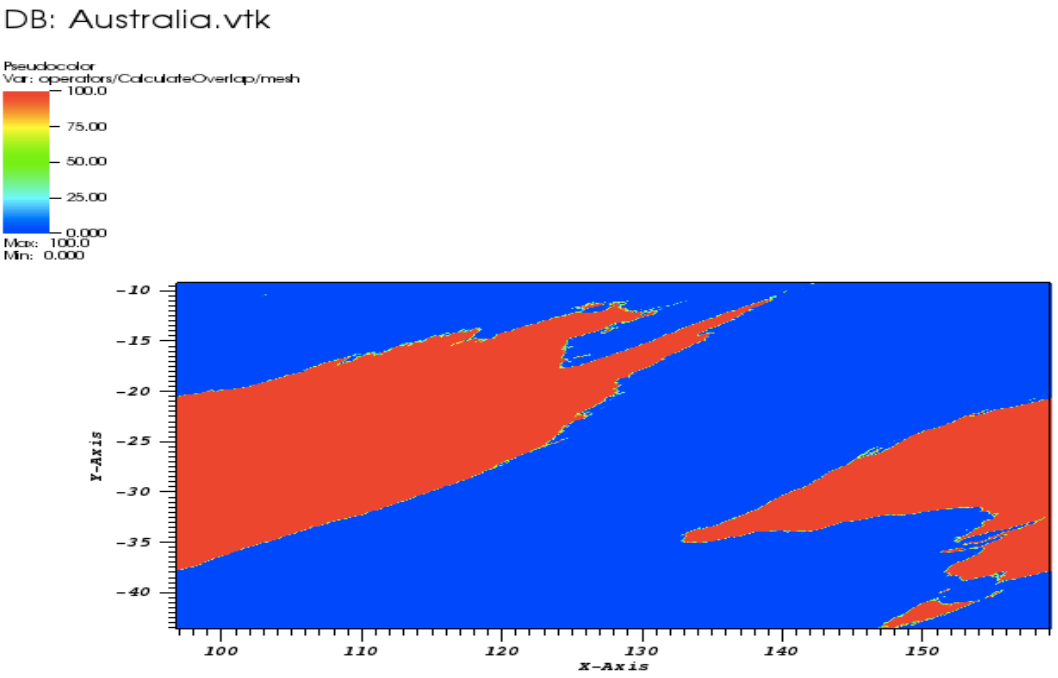

FIGURE 5. Australia without Overlap among Polygons.

CHAPTER IV

THE ALGORITHM

In this chapter, we present our algorithm's initialization step, its pseudocode, and the detailed descriptions of its routines.

4.1. Algorithm Initialization

The input to the algorithm is a user-specified number of bins in the 2D histogram. For example, if the user specified 100x200, then a histogram with 100 bins in width and 200 bins in height would be created. There is also an option for the user to specify the spatial range to place the histogram. If no spatial range is specified, then the algorithm calculates the bounding box for all input data, and uses that bounding box as the spatial range. The program first reads the data from disk. The data is then converted from polygonal data to triangle data. This conversion makes subsequent overlap calculations easier to perform. The conversion itself is done by a module in the VTK library (vtkTriangleFilter). Next, an array is created for storing the cumulative percentage overlap between triangles and the 2D histogram. This array is referred to as "areaCoveredOfCells." The size of the array is the same as the number of bins in the 2D histogram. Each element of the array is initialized to zero. There are two algorithms. One is unoptimized without search structures, and another is optimized with search structures.

4.2. Algorithm 1 (Unoptimized)

The main code runs on all triangles. For every triangle, the algorithm finds the cells that are covered, performs clipping taking into consideration removing redundancy caused by overlap among triangles, and updates the bin with their overlap. The problem of overlap

within the data and its effect on the accuracy of its calculations are explained in Chapter III, section 3.

4.2.1. Pseudocode

In the pseudocode in Figure 6, we use the following short variable names: C is the total number of cells, T is the number of triangles in the data set, AC is the area of any single cell in the grid of size numX * numY, TAC is the number of triangles produced by clipping the original triangle t overlapping with some cell, NTC is the number of triangles in the current cell, and DataStats is the array that holds the dimension of the grid (numX, and numY), and the boundaries of a cell are denoted by minX, maxX, minY, and maxY for minimum X, maximum X, minimum Y, and maximum Y, respectively.

```
1   /* Initialization*/
2   Array percentageCoveredOfCells [ C ]                    // initialized to 0.0
3   Array areaCoveredOfCells [ C ]                          // initialized to 0.0
4   Vector<Vector<Double *> > cellsVectors (C)              // initially empty vectors
5   /* Find Cells Covered by Every Triangle */
6   Loop I =0 to I < T                                      //every triangle in the data
7       t ← get the triangle points
8       int bboxNormalized [4] ← BBox (t, DataStats)
9       Loop X = minXNormalized to X < maxXNormalized
10          Loop Y = minYNormalized to Y < maxYNormalized
11              /*Clipping The Triangle*/
12              cellBoundaries ← calculateCellBoundaries
13              cellId ← X + Y *numX
14              Vector<Double *> result ← Clip (t, cellBoundaries)      // clip the triangle  t
15              Loop K=0 to K < result.size()          // add those to the cell's vector of triangles
16                  double * tri = result[K]
17                  CellsVectors[ cellId ]. Push_back(tri)
18              EndLoop
19          EndLoop
20      EndLoop
21  EndLoop
22  /* Calculate Area Covered of Every Cell */
23  Loop CellId= 0 to CellId < C
24      Vector<double *> trianglesInCell = cellsVectors[CellId]
25      double AreaOfCurrentCellCovered = 0.0
26      Loop J = 0 to J < NTC
27          double * a = trianglesInCell [j]
28          AreaOfCurrentCellCovered += areaofTriangle (a)
29          /* Find and Remove Overlap Between Triangles */
30          Loop K = J+1 to K < NTC
31              vector<double * > pointsOfIntersectoins
32              double * b = trianglesInCell[K]
33              AreaOfCurrentCellCovered -= areaOfOverlapToSubtract (a, b,
                pointsOfIntersections)
34          EndLoop
35      EndLoop
36      /* Populate The Arrays with The Result */
37      areaCoveredOfCells → setTuple1(CellId, AreaOfCurrentCellCovered)
38      percentageCoveredOfCells → SetTuple1(CellId, (( AreaOfCurrentCellCovered / AC)*100))
39  EndLoop
```

FIGURE 6. The Unoptimized Algorithm Pseudocode

4.2.2 Full Description of Code

4.2.2.1 Finding Cells Covered by Every Triangle in the Data Set

In the algorithm for finding cells covered by every triangle section lines (6-20), the algorithm begins by retrieving the triangles from the Shapefile. For every triangle, we first find the bounding box of the triangle, and then calculate the normalized bounding box, that

is, the bounding box of the triangle mapped to the output mesh that starts from 0 to the number of x on the x-axis, and starts from 0 to the number of y on the y-axis.

We then loop from the minX of the normalized bounding box for the triangle to the maxX, and from the minY to the maxY of the normalized bounding box, to locate all the cells overlapping with this triangle. For each of these cells the triangle clipping routine is performed.

To illustrate this process, look at the example in Figure 3 alongside the pseudocode. We have a triangle with the vertices (1,3), (1.5, 2), and (2.5, 4). At line 6 of the pseudocode, we first get the normalized bounding box of the triangle, that is the bounding box of the original triangle mapped to the mesh. For this example, for simplicity the normalized bounding box is the same as the original bounding box of the triangle. So, minXNormalized = 1, maxXNormalized = 3, minYNormalized = 2, and maxYNormalized = 4. To find the cell numbers that overlap with the triangle, we loop from minXNormalized to maxXNormalized, and the inner loop from minYNormalized to maxYNormalized. The cell number is calculated by adding x and y*numX in the grid. For this example, numX = 3, numY = 4, minX = minY = 0, and maxX = maxY = 3. So, the first cell that possibly overlaps with the triangle is cell number = 1 + (2*3) = 7, the remaining cells are 8, 10, and 11.

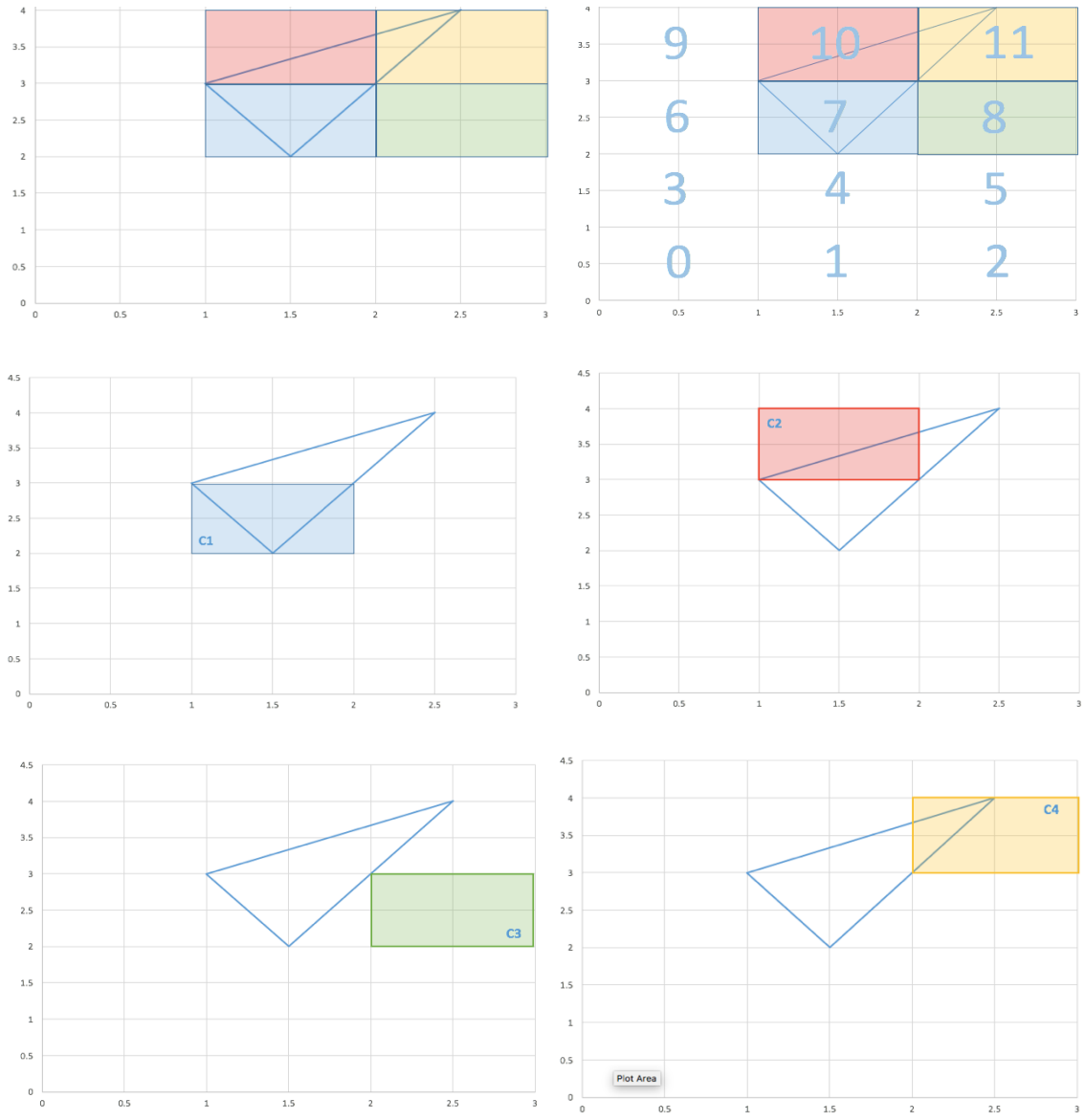

FIGURE 7. The Process for Finding Cells that Overlap with a Triangle.

4.2.2.2. Clipping Routine Algorithm

In "clipping the triangle" section, lines (14-15) in the clipping routine, we start with an empty vector. We then push the triangle to be clipped into this vector. The routine consists of four functions that take both a vector and a limit that we are clipping against. Those are maxXClip, minXClip, maxYClip, and minYClip. We clip the triangle first with

respect to the maxX of the cell. The maxXClip routine takes the vector with the original triangle, and returns the clipped triangle or triangles in a new vector.

The triangles produced from clipping with respect to maxX are clipped with respect to the minX of the cell, and the resulting vector of triangles from the minXClip is processed, and the triangles are clipped with respect to maxY. The same process is repeated with minYClip. After the result of clipping the original triangle against the boundaries of the cell, we add the triangles in the resulting vector to the cell's vector of triangles for use in the later stage of calculating the area covered of this cell by the triangles in it.

To illustrate the clipping routine, let's look at the triangle with the vertices (1,3), (1.5, 2), and (2.5, 4), in Figure 4. The clip routine creates a new empty vector, and pushes the triangle to be clipped to it. The vector of the single triangle is passed to the maxXClip function along with the maxX limit of the cell that the triangle is overlapping with. Inside the maxXClip function, the points of the triangle are sorted clockwise, and for every vertex of the triangle we check if it is greater than the maxX limit of the cell. For this triangle, the vertex (2.5, 4) is beyond the maxX limit of cell C1 which is equal to 2. Two points are interpolated on the x = 2 axis, and the two triangles T1 and T2 are produced as a result of the clip. The resulting vector of the function maxXClip is passed to the function minXClip function. The minXClip function takes the first triangle (T1) in the vector, and clips it against the minX limit of the cell C1 which is equal to 1. This clip produces the same triangle (T1). The minXClip function takes the second triangle from the vector (T2), and clips it against the minX limit of the cell. The clipping of the second triangle (T2) against x=1 produces the same triangle (T2). The produced vector from the minXClip function is of size two with the same triangles, T1 and T2, produced from the maxXClip routine. The

resulting vector is then passed to be clipped against the maxY limit of the cell C1 and minY limit of the cell C1. The resulting vector after the clipping is a vector with a single triangle with the same vertices as T2.

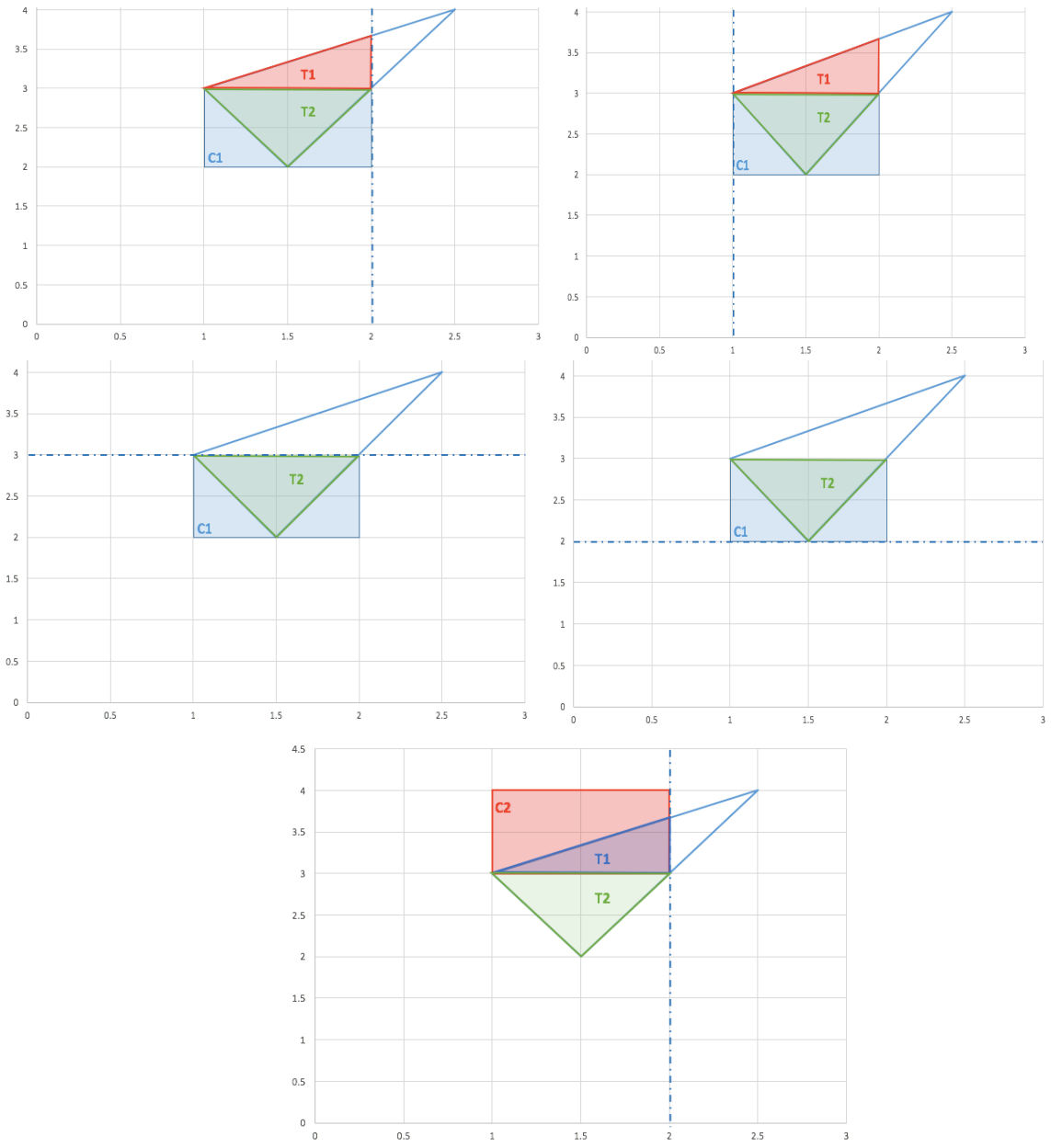

FIGURE 8. The Process of Clipping a Triangle against a Cell's Boundaries.

4.2.2.3. *Detailed Description of the Clipping Routine Functions*

To illustrate the process of the clipping routine functions, consider how the first two work. The remaining functions follow their main concept.

4.2.2.3.1. Processing Triangles for Clipping against the Maximum X Limit of a Cell

This is done with a function named maxXClip. It takes a vector of triangles represented by double arrays, and a double maxX value that represents the maximum X limit of the cell the triangles in the vector are being clipped against. For every triangle in the vector, we call the function maxXClipTri.

4.2.2.3.2 Clipping a Triangle against the Maximum X Limit of a Cell

In this function, named maxXClipTri , the triangle's vertices are checked first clock-wise by the X values. After that, every triangle vertex is checked against the maxX value. If the x value of the vertex is greater than the maxX value limit, the x value of the vertex is updated to the maxX limit value, and the y value of the vertex is changed to the new interpolated value along the original vertex and the other vertex of the triangle that is less than the maxX value. If the vertex is less than the maxX limit value, then the vertex's x and y values stay the same.

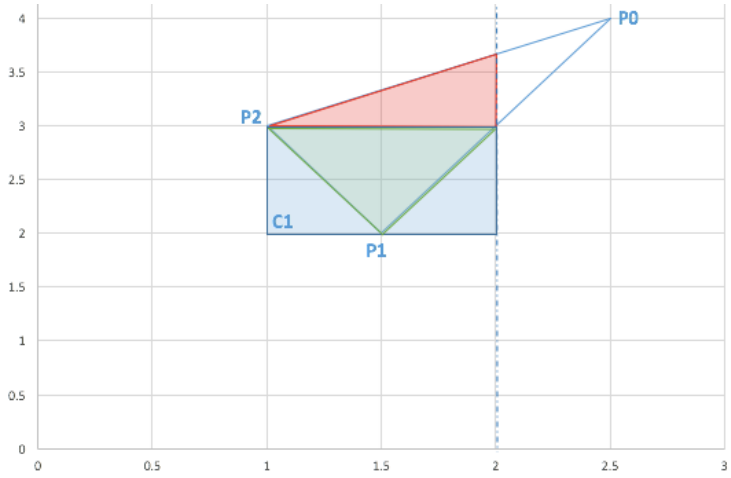

FIGURE 9. Example to Show the Process of Clipping a Triangle against the Maximum X Value of a Cell.

To illustrate the exact process of the clipping, look at Figure 9 as an example. The first vertex is the point P0, but since the x value of P0 is bigger than the maxX limit of the cell C1, P0's x value is updated to the maxX limit value. If P1 is less than the maxX limit of the cell C1, we interpolate the new y value for the point P0. The new x and y values of P0 produce a new point P0A, and if the other connected vertex to P0, i.e. P2 is less than the maxX limit of the cell, we get a second new point P0B with an x value equal to the maxX limit of the cell, and the y value interpolated for the maxX value along the edge P0-P2.

The clipping of the original triangle P0 P1 P2, produces two triangles, which are: P1→P2 →P0A, and P2 → P0A → P0B shown in Figure 10. The two triangles are pushed to the new vector, and returned to Clip function to be next processed by the next clipping routine, minXClip, and so forth.

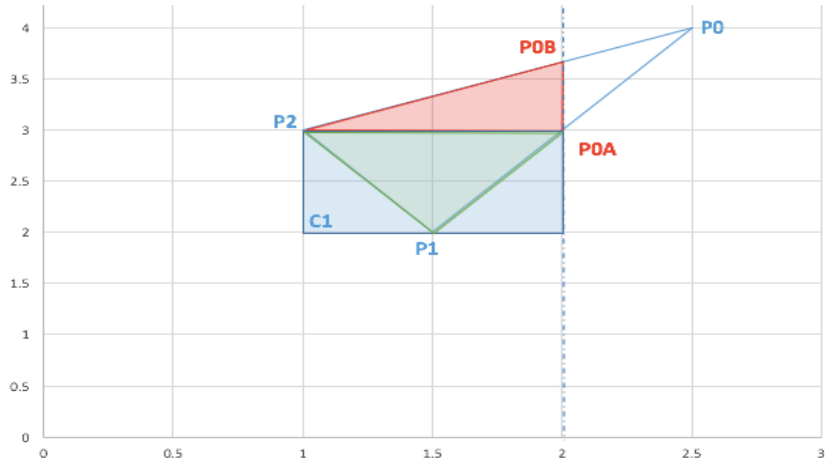

FIGURE 10. The Output after Clipping a Triangle against the Maximum X Value of a Cell.

4.2.2.4. Calculating Area Covered and Removing Overlap Between Polygons in the Data

In the "calculate area covered of every cell" section of the algorithm, we calculate the area covered from each cell by triangles residing within its boundaries. During this process, we consider the overlap between triangles within the cell, and remove it to prevent erroneous area calculation caused by it.

To calculate the area covered of the cell, let's look at the cell C1 in Figure 11 to illustrate the process. Refer to line 24 of the algorithm: we first retrieve the vector containing the triangles in the cell. The vector for C1 will be of size 4. In line 25, we initialize the variable that accumulates the area covered of the cell (**AreaOfCurrentCellCovered**) to zero. After that for every triangle in the cell, we add the area of the triangle to the cell and subtract the area of intersection between this triangle and every other one in the cell. For C1, we first add the area of T1 to **AreaOfCurrentCellCovered** and subtract the area of overlap between T1 and the remaining triangles in the cell, which are T2, T3, and T4. For the second iteration of the

loop, we add the area of T2 to **AreaOfCurrentCellCovered** and subtract the area of overlap between T2 and T3, and T2 and T4. For every triangle, we find overlap between the triangle in hand and only triangles following it in the vector, because possible intersections with triangles before the current one have already been considered with the previous triangles. In this specific example, since the overlap between T1 and T2 has been handled, when it is T2's turn to be processed, we don't bother subtracting the overlap with T1 because it has already been subtracted when T1 was processed. After T2, T3's area is added to the **AreaOfCurrentCellCovered** and the overlap between T3 and T4 is subtracted. Lastly, the area of the triangle T4 is added to the cell's area, and there are no remaining triangles in the cell to subtract the overlap with.

In the section "populate the arrays with the result" section of the algorithm, the processing of every triangle in the cell's vector is done, so we deposit the result of the current cell that is, the calculated area covered of the current cell in the areaCoveredOfCells array, and calculate the percentageCoveredOfCells array after dividing the AreaOfCurrentCellCovered by the area of a cell in the current grid and multiplying the fraction by 100.

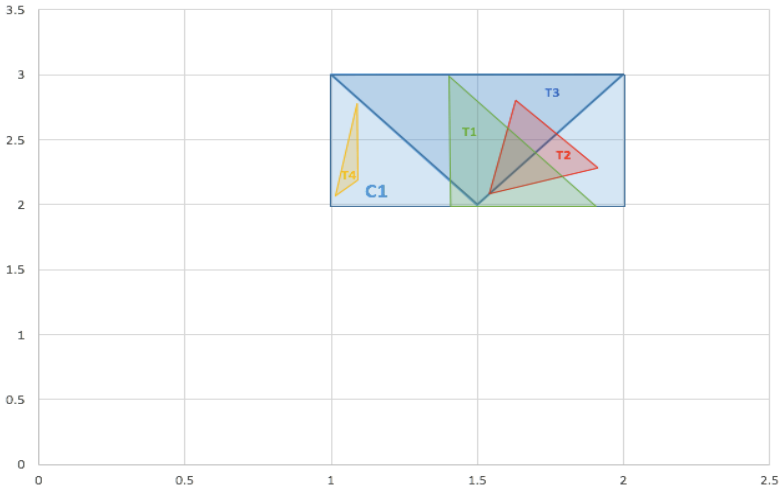

FIGURE 11. Example to Illustrate the Process of Calculating Area Covered of a Cell and Removing the Overlap.

4.3. Algorithm 2 (Optimized)

Optimizing our clipping and area calculation algorithm was a necessity in this project. The number of calculations and steps performed on the large data set resulted in having huge execution times and running out of memory. Optimizing the algorithm using interval trees cut the time of searching for overlaps, minimized the execution time, and allowed for processing of bigger portions of the data.

4.3.1. Pseudocode

The optimized version of our algorithm is shown in Figure 12 and described in detail in the following section.

```
/*Initialization */
Array percentageCoveredOfCells [ C ]                    // initialized to 0.0
Array areaCoveredOfCells [ C ]                          // initialized to 0.0
Vector<Vector<Double *> > cellsVectors (C)              // initially empty vectors
/*Find Cells Covered by Every Triangle */
Loop I =0 to I < T                                      // every triangle in the data
    t ← get the triangle points
     int bboxNormalized [4] ← BBox(t, DataStats)
    Loop X = minXNormalized to X < maxXNormalized
        Loop Y = minYNormalized to Y < maxYNormalized
            /* Clipping The Triangle */
            cellBoundaries ← calculateCellBoundaries
            cellId ← X + Y *numX
            Vector<Double *> result ← Clip (t, cellBoundaries)    // clip the triangle  t
            Loop K=0 to K < result.size()  // add those to the cell's vector of triangles
                double * tri = result[K]
                CellsVectors[ cellId ]. Push_back(tri)
            EndLoop
        EndLoop
    EndLoop
EndLoop
/* Calculate Area Covered of Every Cell */
Loop CellId= 0 to CellId < C
    Vector<double *> trianglesInCell = cellsVectors[CellId]
    double AreaOfCurrentCellCovered = 0.0
    /* Interval Tree Construction */
    avtIntervalTree itree (trianglesInCell.size(), 2)
    Loop J = 0 to J < NTC
        double * a = trianglesInCell [j]
        Double bbox [4]
        GetTriangleBoundigBox(a, bbox)
        itree.AddElement(j, bbox)
     EndLoop
     itree.Calculate()
    /* Finding Overlapping Triangles Using the Interval Tree*/
     Loop J = 0 to J < NTC
        /* Find Overlapping Triangles */
         double * a = trianglesInCell[J]
         AreaOfCurrentCellCovered  += areaOfTriangle(a)
         Double bbox [4]
        GetTriangleBoundingBox (a, bbox)
        Vector<int> elems
        Itree.GetElementsListFromRange (lo, hi, elems)
        /* Remove Intersection */
        Loop K =0 to K < elems.size
            if(elems[k] > J)
                vector<Double *> pointsOfIntersections
                double * b = trianglesInCell[ elems [k] ]
                AreaOfCellCovered -= areaOfOverlapToSubtract (a, b, &pointsOfIntersections)
        EndLoop
    EndLoop
    /* Populate The Arrays with The Result */
    areaCoveredOfCells → setTuple1(CellId, AreaOfCurrentCellCovered)
    percentageCoveredOfCells → SetTuple1(CellId, (( AreaOfCurrentCellCovered / AC)*100))
EndLoop
```

FIGURE 12. The Optimized Algorithm Pseudocode.

4.3.2. Full Description of Code

The optimized algorithm's "initialization and finding cells covered by every triangle" section (lines 1-21) are the exact same as the one in the unoptimized version.

4.3.2.1. Interval Tree Construction and Usage

In the section for calculating area covered by every cell (lines 22 – 55), we loop over every cell in the bin retrieving the triangles that lie inside each of them. For every cell, an interval tree is constructed of size equal to the number of triangles in the cell. For every triangle in the cell, we calculate the bounding box of the triangle and add the extents as intervals, such as [maxX, maxY], and [minX, minY] to the constructed interval tree of the cell. When all the intervals of the triangles are added to the interval tree, the interval tree is built. Now the interval tree is ready to find which of the triangles of the cell overlap with the triangle under investigation, which is the following step.

In the section for finding overlapping triangles (lines 37 – 43), for every triangle in the cell, we first add the area of the current triangle to the **AreaOfCurrentCellCovered,** and remove the redundancy, i.e., area of intersection with any other triangle in the cell. The removal of this redundancy is done in the find overlapping triangles section as follows. We first calculate the bounding box of the current triangle. After that, we use the interval tree search structure to find all the triangles that overlap with the current triangle's intervals. A pool of triangle identifiers is returned, and for each of these triangles, we calculate **areaOfOverlapToSubtract** between them and the current triangle, and subtract it from the AreaOfCellCovered. This is done in the "remove intersection" section of the pseudocode.

In the “populate the arrays with the result” section of the code, the processing of every triangle in the cell’s vector is done. We deposit the result of the current cell that is the calculated area covered of the current cell in the **areaCoveredOfCells** array. Then the percentage of the current cell is deposited in the **percentageCoveredOfCells** array after dividing the **AreaOfCurrentCellCovered** by the area of a cell in the current grid, and multiplying the fraction by 100.

4.3.2.2. Detailed Description of Finding the Area of Overlap between Two Triangles Functions

The areaOfOverlapToSubtract function takes two triangles as its parameters, and a pointer to a vector which holds the points of intersections between the two triangles. The first step is to find the points of intersection between the two triangles using the two functions: **findIntersectionsBetweenEdges** and **isInside**, and the second step is to calculate the area of intersection between the two triangles using the points of intersection found by the previous step.

4.3.2.2.1. Finding the Intersection Points between the Edges of Two Triangles

The FindIntersectionsBetweenEdges function takes an edge from the first triangle, and an edge from the second triangle. This function is called on every edge from the two triangles. That is, nine times. Specifically, we find points of intersection between the first edge of the first triangle with the three edges from the second triangle, and the points of intersection between the second edge from the first triangle with the three edges from the

second triangle, and lastly, the points of intersection of the third edge from the first triangle with the three edges from the second triangle.

4.3.2.2.2. Testing if any of the Two Triangle Vertices are Inside of the Other Triangle

The isInside function takes a triangle vertex, and the triangle represented by a double array. The function checks if any of the points from the first triangle is inside of the second triangle. This function handles the case where the edge from the first triangle is not intersecting with one of the edges from the second triangle but one of that edge's end points is inside of the triangle which means intersection point with the triangle exists, and it is the end point of that edge that lies within the inner area of the triangle.

The function is called on every vertex of the two triangles. So, it is called six times. It checks if any of the first triangle's vertices is within the second triangle. The **isInside** function works as follows: it checks if the areas of triangles formed by the vertex we are testing with the triangle's vertices sum up to the original triangles' area; if not, we can conclude that the tested vertex is not inside of the triangle. If it is within the triangle, we check if the point of intersection has been already detected by the **findIntersectionsBetweenEdges**, that is, the vertex is already in the **pointsOfIntersection** vector; if not, we push the vertex to the **pointsOfIntersections** vector; if it is in the vector, we ignore this point.

4.3.2.2.3. Calculating the Area of Overlap between the Two Triangles Using Found Points of Intersections

The CalculateAreaOfIntersection function takes the **pointsOfIntersection** vector as a parameter, after all possible points of intersections have been found. The number of vertices of the area of intersections must be at least three; if the points of intersection between the two triangles are less than three, then there is no overlap between the areas of the two triangles. If the points of intersections are more than two, the points are ordered in a counter-clockwise direction to ensure the correct calculation of the area of the resulted polygon.

CHAPTER V

TESTING AND RESULTS

5.1. Experiments Overview

Our study was designed to test the correctness and performance of both the optimized and unoptimized versions of our algorithm. The algorithm is running as a filter through VisIt which is an open source visualization tool for 2D and 3D meshes. Through its GUI, we can easily choose different data sets and vary the grid sizes for testing and experimenting. The tests varied three factors: optimization setting, grid sizes, and data sets.

5.2. Factors

5.2.1. Optimization Settings

The experiments we did were run on two different optimization settings, either optimized or unoptimized.

5.2.2. Grid Sizes

The grid sizes are the number of elements in width by the number of elements in height. In this study, our tests used grids with equal dimensions. That is, the number of X elements equaled the number of Y elements. The grid sizes reported in the results table below starts from grid size 50 by 50 to 2000 by 2000.

5.2.3. Data Sets

The performance results presented here are for the intermountain data's first million triangles processed using the optimized and unoptimized versions of the filter. The

intermountain data X values range is -2045100 to -1040320, and the Y values' range is 1560084 to 266110, with almost 10 million and 750 thousand triangles.

	50X50	100X100	300X300	500 X 500	800 X 800	1000X1000	2000X2000
Optimized	22.58	22.42	23.08	24.53	29.53	31.8	50.8
Unoptimized	776.66	271.26	73.99	48.68	43.38	43.19	51.77

FIGURE 13. Performance Trend Comparison between the Optimized and Unoptimized Algorithm.

5.3. Results and Performance Analysis

The performance of the optimized version of the filter is better when the number of triangles in one cell is big. That is, the interval tree built for the triangles ranges is used enough times to reap the benefits of this efficient searching structure. The numbers in Table 1 are execution times in seconds for the different grid sizes. Figure 13 shows the performance trend comparison between the optimized and unoptimized algorithms.

TABLE 1. Execution Times Comparison between the Optimized and Unoptimized Code.

Grid Size	*Optimized*	*Unoptimized*	*% Optimized faster*
50 X 50	22.579536	776.66	97.09 %
100 X 100	22.418807	271.26	91.73 %
200 X 200	22.141781	106.50	79.2 %
300 X 300	23.080292	73.99	68.8 %
400 X 400	24.052942	57.14	57.9 %
500 X 500	24.526272	48.68	48.9 %
800 X 800	29.526272	43.28	31.78 %
1000 X 1000	31.799353	43.19	26.37 %
2000 X 2000	50.79175	51.77	1.88 %

TABLE 2. Execution Times for Building the Interval Trees.

Grid Size	*Total Execution Time*	*Interval trees construction time*	*Interval trees usage time = Total execution time – Interval trees construction time*
50 X 50	22.579536	9.791923	12.787613
100 X 100	22.418807	10.053468	12.365339
200 X 200	22.141781	10.333979	11.807802
300 X 300	23.080292	10.409227	12.671065
400 X 400	24.052942	10.804534	13.248408
500 X 500	24.526272	11.072370	13.80017
800 X 800	29.526272	12.436397	17.089875
1000 X 1000	31.799353	13.268166	18.531187
2000 X 2000	50.79175	19.904474	30.819699

By observing the numbers in Table 1, we can see that the total execution time of the optimized code increases as the number of bins in the grid increases, while the

execution time of the unoptimized code decreases until we reach the 2000 by 2000 grid size. The optimized code runs faster than the unoptimized code until we reach the 2000 by 2000 grid size, even though we kept the same number of triangles across the different codes and different grid sizes. The average density of triangles, that is, the number of triangles per bin, gets smaller as the number of bins increases. Therefore, we can see that the unoptimized code runs faster as the number of bins increases. But the same is not true about the optimized code, as the performance of the optimized code drops as the number of bins increases. We provide the analysis for this trend below.

The optimized code uses the interval tree searching structure. The cost of constructing new interval trees is O (n log n), and the query cost is O (log n + k) time, where k is the number of segments reported to be overlapping [22:220]. In this problem, every bin in the grid constructs its own tree with the triangles residing within its boundaries. Let N be the total number of triangles in the data. So, for every triangle, we have its bounding box inserted in the interval tree as an interval, and B the number of bins in the grid. But for each step of processing, i.e., building the interval tree, and using it, there is a fixed overhead cost. Constructing an interval tree takes O (n log n + C1) where C1 is a fixed overhead cost. As the number of bins increases in the grid, the fixed cost is multiplied, which increases the overall execution time. For example, you have 9 million triangles, and 900 bins. Then the average number of triangles per bin (n) = 9000000/ 900 = 10000 triangles. The cost of constructing interval tress = B * O (n log n + C1). In this example, it is 900 * O(10000 log 10000 + C1) = 3.6 million + 900 * C1. As the number of bins increase, the cost of building the trees, and the overhead cost drastically increase. When the number of bins is increased to B = 9000, then the new average number of triangles per bin (n) =

9000000 / 9000 = 1000. The cost of building the interval trees = 9000 * O(1000 log 1000 + C1) = 27 million + 9000 * C1.

The same analysis applies for the increasing cost of using the built interval trees. Thus, even though the general cost of constructing an interval tree, i.e., O(n log n) might seem bigger than the recurring cost of using it, i.e., O (log n), in our code, as the fixed cost associated with using the interval trees is way bigger than the fixed cost associated with constructing the interval trees.

From our previous example, B = 9000, and n = 1000. The query and usage cost O (log n + C2) where C2 is the fixed cost associated with using the built interval trees.

$$9000 * O (\log 1000 + C2) = 27000 + 9000 * C2$$

Referring back to Table 2, we observe that the time for using the interval trees is always bigger than that for constructing the trees. Therefore, we can form the following inequality:

The cost of constructing the interval trees < The cost of using the interval trees

$$27 \text{ million} + 9000 * C1 < 27000 + 9000 * C2$$

$$26973000 + 9000 * C1 < 9000 * C2$$

$$2997 + C1 < C2$$

We can therefore conclude that the optimized version of the algorithm is efficient and is helping the performance, but there is an overhead cost that might be from the initialization steps needed or the memory allocation.

CHAPTER VI

CONCLUSION

This thesis was inspired by the need of Prof. Chris Bone from the Geography department for a way to visualize a big data set. The data was collected over a span of many years of the loss of trees in the United States caused by the bark beetle insect. We developed an algorithm to process and visualize the data accurately and efficiently.

There were two main goals of this thesis. The first was to develop an algorithm that reads and processes the data and can be applied to different types of data that need to be processed and visualized in a similar way. The second goal was to implement the algorithm, test it, and also optimize it. The algorithm successfully calculates the percentage covered of a cell by the triangles residing within its boundaries and removes redundant intersection areas, i.e., the overlap among triangles, that causes erroneous results. The algorithm is running as a filter through VisIt, and reads Shapefiles or VTK files to process the data according to our developed algorithms, and produce a 2D histogram of the data. We performed extensive testing and experiments for accuracy and performance, discovered our implementation's limitations, and arrived at insights with regard to the performance of our code's optimization.

Our future work includes developing the algorithm further to make it run faster and able to handle larger data sets. Furthermore, we would like to produce a parallelized version of our algorithm, and experiment with the different techniques. Also, we would like to make use of the power of supercomputers and run our algorithms on them with the bigger data sets, with no limit on the number of polygons that can be processed before running out of memory.

REFERENCES CITED

[1] L. Carroll and L. Safranyik, “The bionomics of the mountain pine beetle in lodgepole pine forests: establishing a context.,” in Mountain pine beetle symposium: Challenges and solutions , 2003, pp. 21–32.

[2] K. Weiler and P. Atherton, *Hidden surface removal using polygon area sorting*, ACM SIGGRAPH Computer Graphics, vol. 11, no. 2, pp. 214–222, Jan. 1977.

[3] Y. Q. Huang and Y. K. Liu, *An Algorithm for Line Clipping against a Polygon Based on Shearing Transformation*, Computer Graphics Forum, vol. 21, no. 4, pp. 683–688, DEC. 2002.

[4] R. F. Sproull and I. E. Sutherland, “A Clipping Divider,” Proceedings of the December 9-11, 1968, fall joint computer conference, part I on - AFIPS '68 (Fall, part I), Dec. 1968.

[5] M. Cyrus and J. Beck, *Generalized Two- and Three-Dimensional Clipping*, Computers & Graphics, vol. 3, no. 1, pp. 23–28, 1978.

[6] I. E. Sutherland and G. W. Hodgman, *Reentrant Polygon Clipping*, Communications of the ACM, vol. 17, no. 1, pp. 32–42, Jan. 1974.

[7] B. A. Barsky and Y. D. Liang, *A New Concept and Method for Line Clipping*, ACM Transactions on Graphics, vol. 3, no. 1, pp. 1–22, Jan. 1984.

[8] T. M. Nicholl, D. T. Lee, and R. A. Nicholl, *An Efficient New Algorithm for 2-D Line Clipping: Its Development and Analysis*, ACM SIGGRAPH Computer Graphics, vol. 21, no. 4, pp. 253–262, Aug. 1987.

[9] Van D. Foley, S. K. Feiner, and J. F. Huges, *Computer Grpahics Principles and Practice*, 2nd ed. Reading, Mass: Addison- Wesley, 1990.

[10] P. G. Maillot, A New, *Fast Method for 2D Polygon Clipping: Analysis and Software Implementation*, ACM Transactions on Graphics, vol. 11, no. 3, pp. 276–290, 1992.

[11] J. D. Day, *A New Two-Dimensional Line Clipping Algorithm for Small Windows*, Computer Graphics Forum, vol. 11, no. 4, pp. 241–245, Aug. 1992.

[12] T. Möller, *A Fast Triangle-Triangle Intersection Test*, Journal of Graphics Tools, vol. 2, no. 2, pp. 25–30, 1997.

[13] G. Greiner and K. Hormann, *Efficient Clipping of Arbitrary Polygons*, ACM Transactions on Graphics, vol. 17, no. 2, pp. 71–83, 1998.

[14] B. R. Vatti, *A Generic Solution to Polygon Clipping*, Communications of the ACM, vol. 35, no. 7, pp. 56–63, 1992.

[15] Y. K. Liu, X. Q. Wang, S. Z. Bao, M. Gomboši, and B. Žalik, *An Algorithm for Polygon Clipping, and for Determining Polygon Intersections and Unions*, Computers & Geosciences, vol. 33, no. 5, pp. 589–598, 2007.

[16] M. Held, *ERIT—A Collection of Efficient and Reliable Intersection Tests*, Journal of Graphics Tools, vol. 2, no. 4, pp. 25–44, 1997.

[17] B. Žalik, *Two Efficient Algorithms for Determining Intersection Points between Simple Polygons*, Computers & Geosciences, vol. 26, no. 2, pp. 137–151, Mar. 2000.

[18] R. L. Graham, *An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set*, Information Processing Letters, vol. 1, no. 4, pp. 132–133, 1972.

[19] T. M. Chan, *Optimal Output-Sensitive Convex Hull Algorithms in Two and Three Dimensions*, Discrete & Computational Geometry, vol. 16, no. 4, pp. 361–368, 1996.

[20] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa, *The Quickhull Algorithm for Convex Hulls*, ACM Transactions on Mathematical Software, vol. 22, no. 4, pp. 469–483, Dec. 1996.

[21] *ESRI Shapefile: a Technical Description*. 1998.

[22] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars, *Computational Geometry - Algorithms and Applications*, 3rd ed. Springer, *2008*.

[23] *Aerial Survey Geographic Information System Handbook: Sketchmaps to Digital Geographic Information*. USDA Forest Service, Nov. 2005.