Icosatree Data Partitioning of Massive Geospatial Point Clouds with User-Selectable Entities and Surface Modeling

by

A Thesis Submitted
in
Partial Fulfillment of the
Requirements for the Degree of
Master of Science
in
Computer Science

Supervised by

Department of Computer Science

B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, New York

December 2016

The thesis "Icosatree Data Partitioning of Massive Geospatial Point Clouds with User-Selectable Entities and Surface Modeling" by has been examined and approved by the following Examination Committee:

Thesis Committee Chair

Dedication

I'd like to dedicate this to my wife; she's been supportive of me throughout all my procrastination and frustration.

Abstract

Icosatree Data Partitioning of Massive Geospatial Point Clouds with User-Selectable Entities and Surface Modeling

Supervising Professor:

Massive point cloud data sets are currently being created and studied in academia, the private sector, and the military. Many previous attempts at rendering point clouds have allowed the user to visualize the data in a three-dimensional way but did not allow them to interact with the data and would require all data to be in memory at runtime. Recently, a few systems have emerged that deal with real-time rendering of massive point clouds with on-the-fly level of detail modification that handles out-of-core processing but these systems have their own limitations. With the size and scale of massive point cloud data coming from LiDAR (Light Detection and Ranging) systems, being able to visualize the data as well as interact and transform the data is needed.

Previous work in out-of-core rendering [10, 3, 9] showed that using Octrees and k-d trees can increase the availability of data as well as allow a user to visualize the information in a much more useful manner. However, viewing the data isn't enough; applying work in context-aware selection [14] and surface creation [8] the visualization system would greatly benefit in usability and functionality.

This paper explores a new data structure called an Icosatree, or icosahedral tree, that

can be used to partition a point cloud dataset in the same fashion as an Octree is currently used. However, the Icosatree is made from triangular prism sub-cells which are tangential to the ellipsoidal surface used by Earth-based projected coordinate systems. In doing so, as new sub-cells are added to the rendering system, a much more uniform visualization emerges.

Along the same lines, this paper applies portions of the aforementioned context-aware selection and surface creation algorithms to the resulting visualization such that a user may triangulate, prune and/or export portions of the point cloud dataset using an intuitive three-dimensional interface and user-modifiable set of parameters. This allows the user to save items of interest for later analysis.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Massive point cloud datasets are becoming more prevalent as technology becomes cheaper and storage and rendering power grows by leaps and bounds. This thesis attempts to demonstrate a novel partitioning structure called an Icosatree as well as a three dimensional triangular coordinate used for positioning values inside the data structure. This icosahedron-based three dimensional data partitioning structure was developed in order to address a few failings seen with common data structures used for these types of datasets, such as the Octree. The algorithm itself is explained further in Chapter 3.

First, the Octree being axis-aligned is poorly suited to the projected surface the data is displayed in. This causes the cells in the tree to look asymmetrical and as each subsequent layer of the tree is displayed the asymmetrical nature of the visualization becomes more apparent. With the Icosatree, each cell is more closely aligned with the projected geospatial surface and each cell appears as a triangular area to the user. In the final visualization, the cells are added as they are accessed from the data source (local file system or HTTP requests) and as they are added to the rendering element it does indeed look much more uniform.

Second, with the Icosatree being much closer to the projected geospatial surface the hope is that less cells would be required in the output dataset and would more efficiently fill our rendering data structure. This has been seen to be the case, however, with limited resources on commodity graphics hardware a sweet spot between performance and data utilization was more difficult to determine automatically by the system and needs improvement. This will be covered in more depth in the future work section at the end of this

paper.

The Icosatree also uses a new triangular coordinate system in order to position points into the Icosatree cells, or Icosatet. The Icosatet is a triangular prism that is used as the basic sub-structure in the Icosatree. The initial Icosatree is split into twenty Icosatets and each subsequent level in the tree splits each Icosatet into eight smaller Icosatets. The triangular coordinates used allow the points in each level of the Icosatree to be as uniformly distributed as possible. This new coordinate system is explained in Chapter 4.

The resulting visualization was then used to test a hybrid user-definable triangulation algorithm loosely based on the work found in two separate papers [1,2]. The general idea behind this augmented algorithm is that the user selects a region of the screen and any points within that area are selected. This set of points will then be pruned using a combination of nearest neighbor density values, eigenvector comparisons between the entire selection and nearest neighbors, and altitude thresholds which are all definable by the user. Next, the set can be limited further by removing any points hidden by others based on a ray casting and occlusion test. Finally, the resulting set of points are fed into a Delaunay triangulation algorithm in order to create a triangle mesh.

Chapter 2

Background

The basis for this continued work stems from multi-dimensional data structures and how they are leveraged in a geospatial environment. These data structures, the Quadtree, Octree, and k-d tree, are used to partition a (usually) Cartesian coordinate system into a tree structure.

2.1 Quadtree

The Quadtree separates a two-dimensional coordinate system into a tree structure where each node in the tree is split along the center of both axes within its axis-aligned bounding volume creating four child nodes. This allows for much more efficient searching than a simple data structure like an array or list would. Below, Figure 2.1 shows a set of points in a two-dimensional Cartesian coordinate system stored in a Quadtree where each node that has only a single point remaining does not split again; thus saving computational time and storage space. The Quadtree structure is used in geospatial visualizations as it aligns to the longitude and latitude coordinate system, however, the disconnect along the 180-degree meridian and the difficult-to-handle north and south border meeting at a single point cause more issues.

2.2 Octree

The Octree structure is identical to the Quadtree structure except that it is applied to a three-dimensional coordinate system. Each node is split into potentially eight child nodes but it

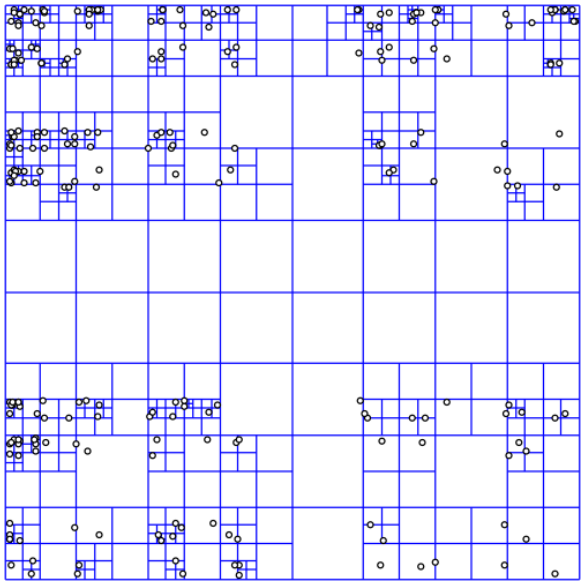

Figure 2.1: Quadtree Structure [2]

still uses a uniform split location by bisecting the span along each axis of its axis-aligned bounding volume. The Octree is used quite often in Computer Graphics as it fits well to an XYZ Cartesian coordinate system and allows culling, bounds intersection, and mouse picking for many objects quickly and efficiently. However, in a geospatial visualization, this structure does not align well to an elliptical surface such as the Earth. Below, Figure 2.2 is an image of an Octree in 3D as well as the flattened version of the tree.

2.3 k-d tree

The k-d tree is a specialized binary tree that allows for more control over how the tree is structured as well as lending itself to being more uniform and balanced than previously mentioned tree data structures. The k-d tree, at each level, chooses a single axis to split along. The axis chosen is defined by a set of criteria such as which axis has the longest span between its min and max values. Then, the split location is also defined by a set of criteria such as splitting each side into containing a uniform number of points. The actual

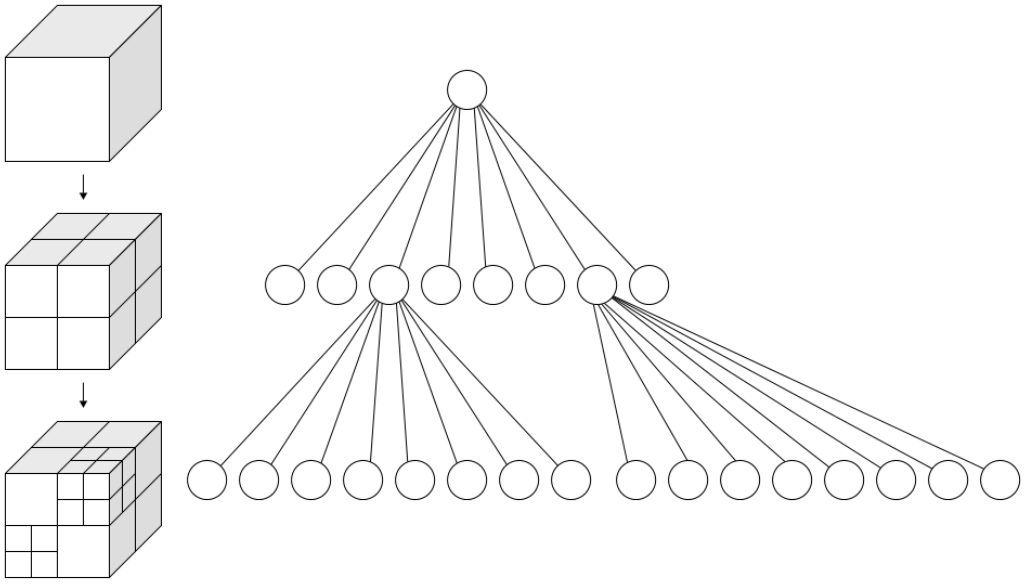

Figure 2.2: Octree Structure and Flattened Tree [12]

split location, unlike the Quadtree and Octree, is a single point that acts as the node in the tree at that depth. This continues until each node has a single point or until some threshold is reached based on tree depth or point count. The k-d tree is a binary tree but can be used with any dimensionality which allows it to be applied to a two-dimensional longitude-latitude coordinate system as with the Quadtree and an XYZ Cartesian coordinate system like the Octree. It has many of the same drawbacks when it comes to applying it to a geospatial coordinate system. However, one upside is that it can be built in such a way that each node has a relatively uniform number of points which can increase storage efficiency and render times by allowing the storage of nodes in chunks without wasting padded space but with the drawback that level-of-detail algorithms make the scene seem non-uniform as each node is rendered and the node dimensions are not consistent. Below, Figure 2.3 shows a two-dimensional k-d tree partitioning algorithm with red and blue lines as alternating split locations. Figure 2.4 shows the flattened tree structure for part of the previous figure and which axis was used at each level.

Previously, Quadtree, Octree [10] and k-d tree partitioning systems have been applied

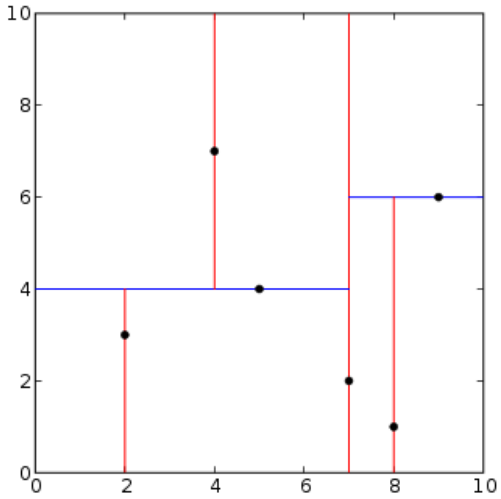

Figure 2.3: Two-Dimensional k-d tree Structure [4]

to a Cartesian system in order to add on-demand searching for view-dependent slices of data. However, once the data set is converted into a geospatial positioning system, either the Cartesian coordinates are now not surface aligned, or issues arise along the partitioning system boundaries where values are not continuous (such as with latitude 90 != latitude -90 but longitude 180 == longitude -180). Using a Cartesian projection from geospatial coordinates solves this issue but adds complexity when deciding what to render as it is no longer surface aligned. This paper looks into applying a surface-aligned data structure to a

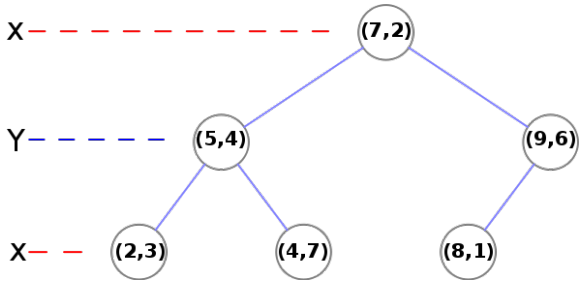

Figure 2.4: k-d tree Partitioning Example [7]

global data set such as a tetrahedral mesh. By using this structure to search for nodes within the view frustum the visualization system will be able to render progressively deeper nodes as the visualization's viewpoint moves closer or further away from the target and as the level of detail increases and will then pull these nodes from an on-demand point server or local file system cache.

Also, in recent work, real-time rendering of depth culling and surface representation [8] has been used to hide unseen data points as well as to fill surface information in on a massive unstructured point cloud. This paper proposes to use this information and apply it to a sparse context-aware selection algorithm [14] in order to adapt it to a more uniformly dense data set. This on-the-fly surface creation can be used to augment the context-aware selection algorithm in order to give it another avenue for object separation within a uniformly-dense data set.

Chapter 3

Design

Previous work in the field of massive point cloud creation and processing has moved toward data systems [5] such as OpenTopography.org as well as research in leveraging Octree and k-d tree data structures for taking massive point cloud data and partitioning it into manageable pieces for the visualization system. Separate work in selection and surface generation have increased the user interaction and utility of these types of datasets. However, tree structures in Cartesian or geodesic coordinate systems each have their own drawbacks and limiting the user to visualization alone gives them a powerful tool that does not allow the user to analyze the information further or export portions of the data for use in more specialized software packages.

The first hurdle is to look at the state of data partitioning and access. Currently, Octrees and k-d trees work in a Cartesian coordinate system. With point cloud data that is relatively limited in size and scope, using an arbitrary coordinate system can work well. Moving towards larger data sets and rendering them on a geospatial projected surface causes the data to no longer follow an axis-aligned format. This causes more issues with rendering such as data structures that clip the projected surface and do not fill the partitioned sections well. The first goal of this research is to modify the data structure used to store the massive point cloud data from an Octree/k-d tree structure where the XYZ Cartesian grid doesn't align well with a geospatial projected surface (WGS84) into an icosahedral grid using previous work in spherical self organizing grids [13] and traversal of triangle mesh elements [6] [11]. Allowing the data to be split into surface aligned data structures should look more pleasing to the end user as the level of detail is modified on-the-fly compared to an Octree/k-d

tree structure without the need of overdrawing or forcing the user to wait until all data is available before rendering.

The second hurdle is to add functionality to the system instead of just being a visualization system. Previous work in the area consisted of interacting with unstructured point clouds [14] by allowing the user the ability to select more-dense regions of data using a screen-space masking system. However, this implementation is limited by requiring the data be unstructured, or containing a very heterogeneous density throughout. Unfortunately, LiDAR data is rarely sparse; it is rarely pruned or processed at all before being accessed by researchers. In another area, on-the-fly surface creation has been applied to point cloud data in order to show physical structures [8] in the input data; this shows off actual objects in the virtual world without having extensive up-front processing of the data. This surface modeling can be leveraged against the context-aware selection algorithm in a uniformly dense point cloud to allow the user to select via screen-space masking controls objects in the LiDAR point cloud data. From this, the point data can be displayed separately for further study or exported for use in other applications.

In order to evaluate these assumptions, a number of utility applications have been developed as well as a simple visualization. The visualization has been designed to load a point cloud and allow the end user to modify the rendering settings, use a screen-selection lasso for object selection, and display and export the selected points and resulting triangulation. The utilities developed consist of command line applications that will access a binary file of point data along with a CSV file of point attributes. The application then partitions and exports the data as either an Octree or Icosatree for use with the visualization application. A command line application was also created to partition the binary input file into smaller portions for ease of use and testing. Lastly, the visualization system was developed using a number of previously developed graphics and computation libraries developed by the author for previous work as well as a number of third party, open source, libraries.

Chapter 4

Implementation

The system designed for this research contains four major pieces: A Partitioning application, A Visualization tool, The Rendering Component, and the Selection Algorithm. Below is a detailed explanation of their implementation and theory.

4.1 Partitioning Application

The partitioning application is a standalone command line application that reads from a number of binary point data files. It creates the different partitioning systems needed to evaluate the algorithm implementations and stores them in a custom binary format for ease of access at runtime. The files have been designed so that they can be accessed via the local file system or through a web server.

The data structure has been partitioned such that each level in the tree structure, for both the Octree and Icosatree, contains points which are left in each node. The nodes are split into a sub-cell grid structure so all nodes at a specific depth are uniformly dense in order for the level of detail algorithm to only have to deal with screen area and not point densities within the visualization tool when determining how deep to traverse. The data also contains information defining the number of points in the node, the number of child nodes, and the layout of the binary data (at the root, only).

The structure of the Octree and Icosatree file systems are fairly straightforward. Each is defined by a file tree originating at a given root directory. Then in each directory, including the root, a point data file exists along with an ASCII text file containing a list of the existing

child nodes at that location. The root directory also contains a CSV file defining the binary layout of the point data which contains information about attributes: name, offset, data type, size in bytes, min, max, mean, and variance.

The Octree and Icosatree structures are identical except for the fact that each Octree cell contains, at most, eight child cells. The Icosatree root node can contain up to twenty child cells and each cell after that can contain at most eight. The Octree structure is split along the XYZ axes as in Figure 2.2. However, the Icosatree is split into twenty triangular prism cells at the root as in Figure 4.1. Below that, each cell is split into eight additional triangular prisms as seen in Figure 4.2 and Figure 4.3.

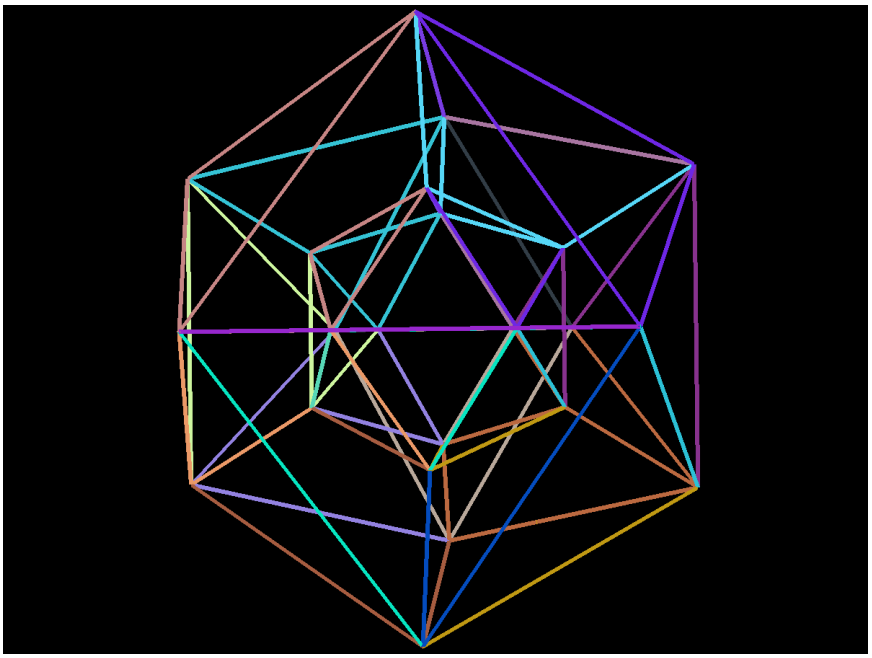

Figure 4.1: Icosatree Wireframe

Each cell is also split into sub-cells; this allows the tree creation to guarantee uniform density of point data at any specific level of the tree which greatly simplifies level of detail calculations in the renderer. As the tree structure is built, a point is inserted into the root node. The sub-cell is computed and if no point was previously added there, the current point is stored. If a point was previously stored at that sub-cell index, the point closest to

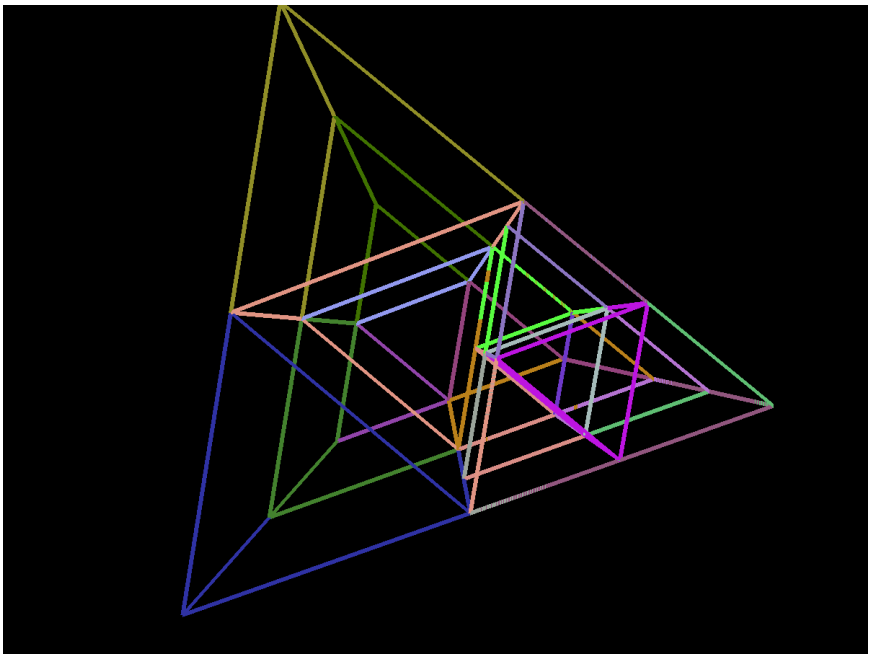

Figure 4.2: Triangle Prism Partitioning

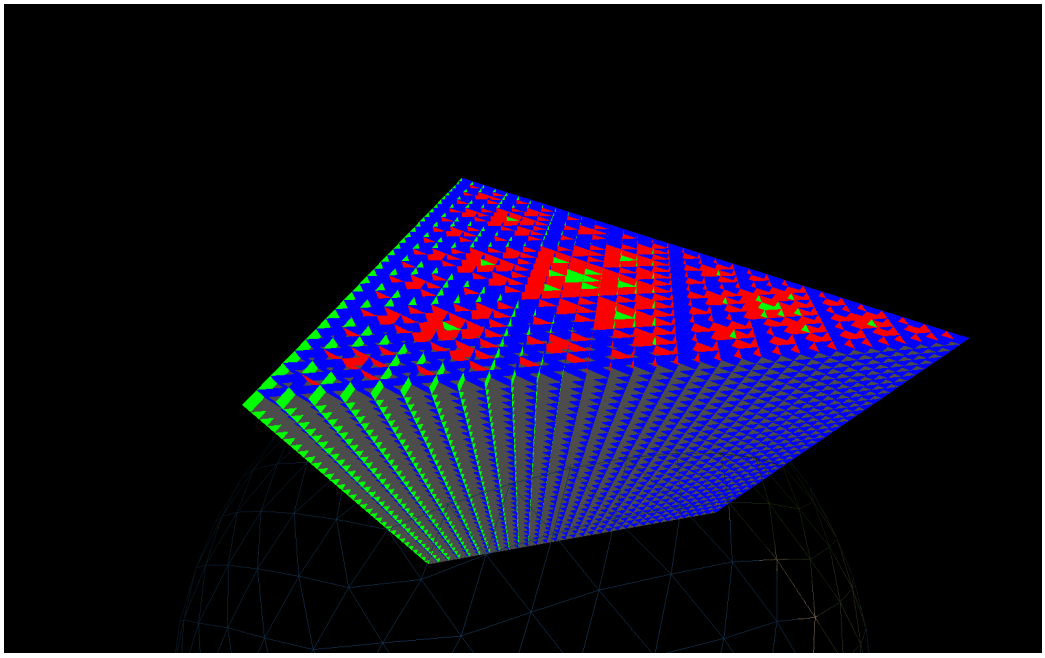

Figure 4.3: Icosatree Cells - Depth 5

its centroid is left there and the remaining point is sent to one of the cells' child cells. This continues until all points have been added to the tree. Once that is complete, the command line tool writes the tree structure as a set of directories and binary or ASCII files. A subset of this file structure can be seen in Figure 4.4. The DAT file contains the binary point data and the text file contains the child cell index values which contain point data of their own. These child cells also have a sub-directory within that nodes directory which continues to the leaves of the tree.

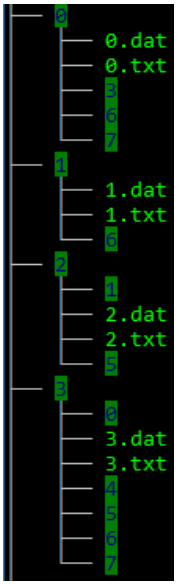

Figure 4.4: Tree File Structure

4.2 Visualization Tool

The visualization tool consists of a Java/OpenGL rendering system based off the Java OpenGL Library (JOGL) and a rendering toolkit the author has developed. Simple geospatial navigation and terrestrial terrain have been added as they aid the user when interacting and visualizing this form of data. The visualization renders a simple WGS84 projected globe, a spherical orbit navigator, the point cloud renderer which supports the different

data structures implemented by the partitioning application, and the selection algorithm used by the user which allows them to select objects, as the points that comprise them, from the scene using a screen-space lasso tool.

The visualization tool renders each item in the scene as its own scene element. The camera navigation is based on a geospatial anchor point and an azimuth/elevation/distance offset from the anchor. A local origin will offset the scene component's local coordinate system for floating point precision reasons and each scene element will update its own position based on this local origin.

There was also the need for a renderable Earth scene element as shown in Figure 4.5. This was developed by creating a GeodesicCoordinate to base the geometry from. The first level of detail consists of a number of rectangular sections each thirty degrees on a side. Then, as the camera moves closer to the surface, each portion splits into smaller sections which can be seen in Figure 4.6. In order to display accurate elevation data, support for Digital Elevation Model data was added and used as a lookup dataset for elevation values at each vertex; an example of the DEM data being used on a wireframe view can be found in Figure 4.7. Initially, a single high resolution texture was used for the imagery but even an 8k image did not add enough fidelity as the user zoomed close enough into the terrain so support was also added for the Slippy Map Tile URL protocol; specifically, OpenStreetMap and Stamen Terrain imagery as shown in Figure 4.8. The Digital Elevation Model data is downloadable manually and is indexed the first time the Visualization application is run whereas the imagery is accessed via HTTP requests as it is needed.

4.3 Rendering Component

The rendering component initially reads the root node and attributes layout information from the dataset (locally or over the web). It then uses the bounding volume of the node to determine if it should be rendered or not and if its children should be queried. It then uses a screen-space level of detail algorithm to determine how deep to traverse; this depth will be tunable based on user preference.

Figure 4.5: Earth

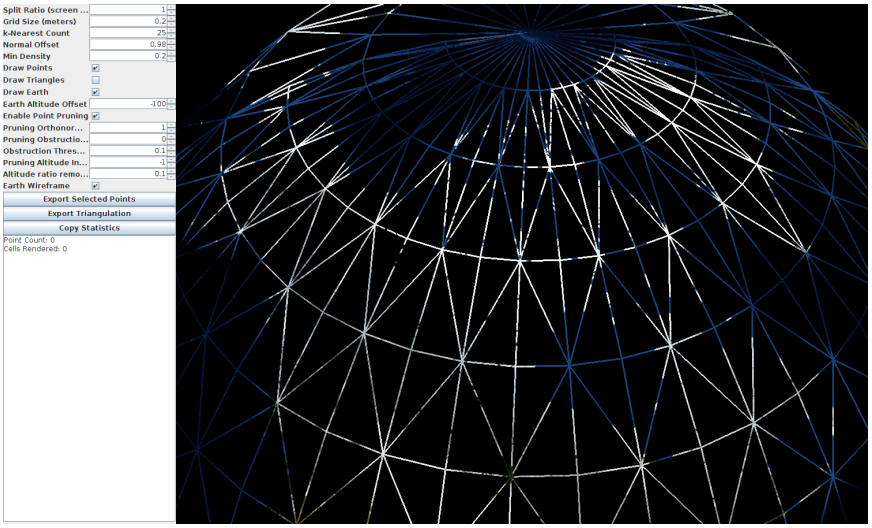

Figure 4.6: Earth Wireframe

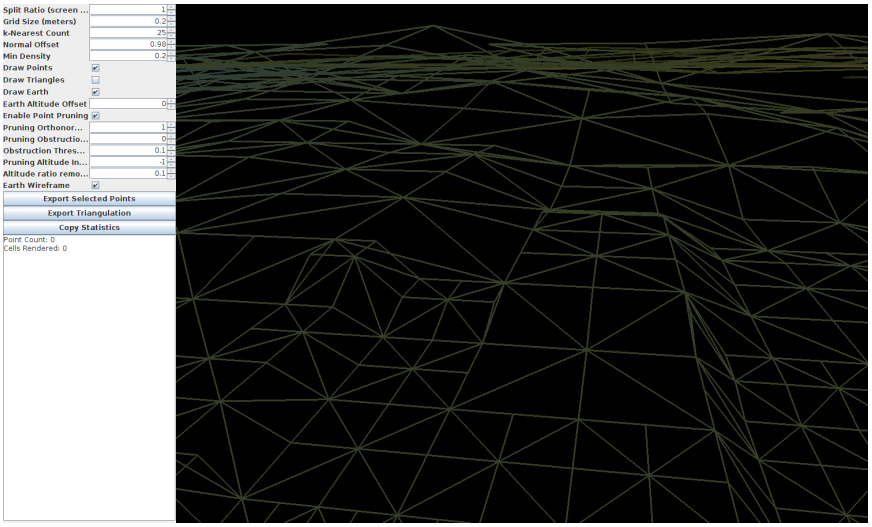

Figure 4.7: Earth Elevation

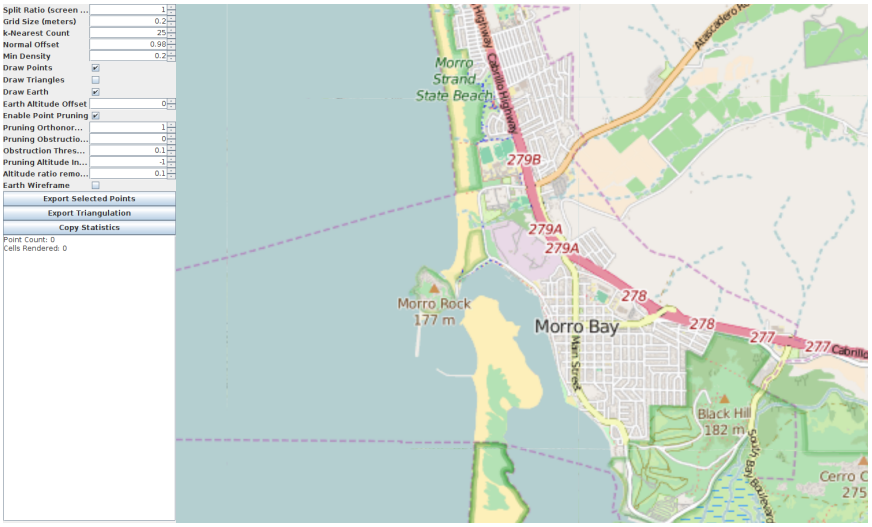

Figure 4.8: OpenStreetMap Tiles

The Octree and Icosatree are both supported by the rendering component as their data structures are identical aside from the number of children their root nodes may contain. The class TreeRenderable is given a path (either local or HTTP) to the root of the tree data and a ConnectionType. Each frame, it does a frustum culling pass to determine which tree cells should be included in the rendering step. Then, for each cell it checks its on-screen area. This area calculation is used to compare against two different thresholds; the first is if this node should render any points it may contain (5050 pixels) and the second is if its children should be checked (200200 pixels). The nodes are split into two sets; those that are complete and those that are pending. The complete nodes have their point data already retrieved while the pending nodes have been created but are still waiting for their point data to be accessed by the connection thread. Next, a small portion of the frame time is given over for uploading data to the graphics card; this is limited in order to keep the visualization at an interactive frame rate. The vertex buffer implementation is actually a number of buffer pools, each of which contain a number of vertex buffer objects and the pool is able to allocate and delete these objects as needed. The tree structure contains the maximum number of points necessary for any node in the tree; this is the starting segment size for the buffer pools. The first pool is defined by a number of segments (100 by default) which can each hold this maximum number of points per buffer object (byte size equal to the attribute-stride * max points * segments per buffer). Then, another pool is made by dividing the segment size by two and continuing this until the smallest buffer object is sliced into segments less than 100 points long. This allows a number of different sized cells to be inserted into a buffer that, at least, fills half of its total capacity but also allows the rendering system to intelligently add, remove, and update cells in the tree without having to pack or rearrange data on the graphics card every frame. Below, in Figure 4.9, is a screenshot of Morro Bay in California; this is a subset of the San Simeon, CA point cloud dataset and, in total, contains 13,725,592 points. At the current zoom distance and level of detail the rendering component is only required to render around three million points.

The TreeRenderable itself controls how the point data is displayed in the visualization

Figure 4.9: Visualization Application with Small Point Cloud

application. It creates a custom-built vertex buffer object pool which is then used to store and render the point data. The buffer pool contains a set of smaller pools internally that are defined by a set segment size (number of points available in a block) and a number of available segments. Each of these smaller pools is able to append itself with as many vertex buffer objects as it needs and uses a global index to find the specific segment location that is bound to each tree cell. When a segment is cleared, any empty vertex buffer objects on a segment pool will also be removed in order to free up memory. When the Tree Cells are uploaded to the GPU, the segment pool with the smallest segment size that will fit the tree cell's point count will be selected. Then, the first available index in that cell will be requested and the tree cell will store this pool index and segment index for rendering later. The number of internal segment pools is defined by the maximum number of points any given tree cell will contain in the point cloud collection. Figure 4.10 is a flow chart that goes through the steps in the rendering pass for the TreeRenderable.

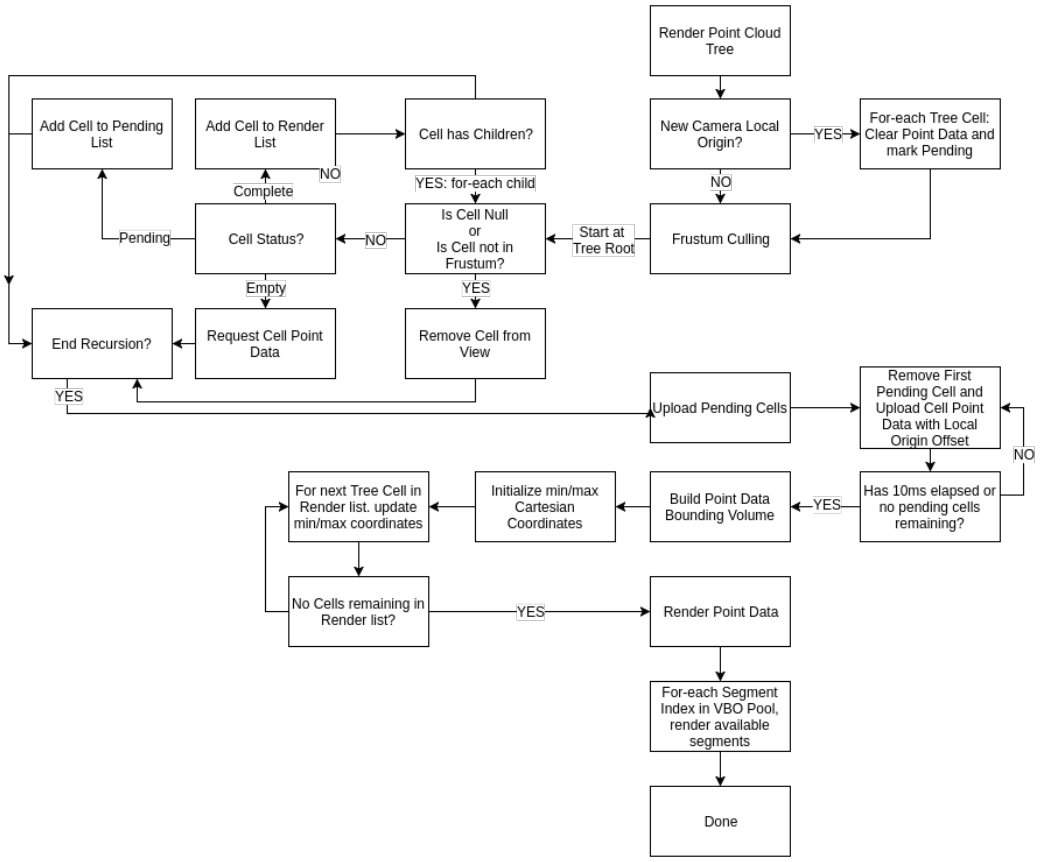

Figure 4.10: TreeRenderer Flow Chart

4.4 Selection Algorithm

The selection algorithm attempts to apply portions of two separate point cloud selection algorithms to the same set of data. The Screen-Space Operator Algorithm [8] is used to define surfaces inside the point cloud; this is useful for visualizing the walls of buildings and such. However, the algorithm itself doesn't handle selection of objects. The CAST algorithm [14] allows selection of more dense portions of a point cloud via two-dimensional mouse selection but it requires a dense cloud of points to handle selection. The selection algorithm developed for this thesis attempts to apply the screen-space lasso and point density techniques from the CAST algorithm and apply the surface creation and triangulation algorithm to the resulting selection in order to give a user more utility when analyzing a point cloud dataset.

The first step of the algorithm is for the user to define the search area. This is done in the visualization by holding the control key and holding the left mouse button while defining the lasso area. The shape is automatically connected to the initial selection point as the user outlines the lasso shape but the final polygon will be automatically closed the first time the lasso crosses itself in order to create a closed loop.

Then, a simple volume frustum is created from the camera into the scene along the screen-space lasso polygon. The tree is searched and any points that are contained within the lasso are returned. Next, for each point, the k-nearest neighboring points are queried and a covariance matrix is computed. If the normal of the best-fit plane of the entire selection is parallel (or within a user-defined threshold) the point is dropped from the selection. Finally, any points whose most distant neighbor is further than twice the user-defined grid size is also removed. This removes any points that are parallel to the ground as well as any points that are in small groups and aren't useful (less than the k-nearest limit).

In order to handle triangulation, the Screen-Space Operator Algorithm is used to determine which points in the selection are unobstructed. This is done by projecting a ray from each point towards the camera and, if any other point is within a specified angular distance and closer to the camera than the tested point, it is removed. These points are then

converted into a two-dimensional plane and a Delaunay triangulation algorithm is used to triangulate them. A map of the 2D projected point to the original 3D point is used to convert this triangulation back into a useful triangle mesh. The lasso (Figure 4.11), point selection (Figure 4.12), and triangle mesh (Figure 4.13) can be seen below. The follwing is pseudocode for the Selection Algorithm:

1. User selects area of screen
2. Create closed polygon from set of screen coordinates
3. Create projected 3D volume from 2D polygon
4. Get volume intersection on visible point data
5. If Pruning enabled, prune points based on user settings
   (a) Apply the following in user order
   - Orthonormal Pruning
     i. Compute global eigenvector for plane normal vector
     ii. For-each point, compute engenvector using k-nearest neighbors
     iii. if point eigenvector is within user-defined angle of global normal, discard
   - Obstruction Pruning
     i. For-each point, if any other point is closer to the camera and the angle between the two points is within a user-defined threshold, discard
   - Altitude Pruning
     i. Compute min/max altitude of all input points
     ii. For-each point, if altitude is within user-defined threshold of min value, discard
6. If user enabled, draw points

7. If user enabled, compute triangle mesh

   (a) Project points to screen space

   (b) Remove any occluded points

   (c) Compute mesh using Delaunay Triangulation

   (d) Project triangulation to world space

   (e) Render triangle mesh

Figure 4.11: Selection Lasso

Figure 4.12: Point Selection

Figure 4.13: Point Triangulation

Chapter 5

Analysis

Tree building was done on an HP Z820 Workstation with an Intel Xeon E5-2650, 256GiB RAM, and a 1TB and 2TB Samsung 850 Pro SSDs. The MapDB library was used for point storage by tree cell and index and in place of in-memory storage, a file system map was used and stored on the 1TB SSD. The input point data was stored on the 2TB SSD and the output tree structure was written there as well; this was to make sure the MapDB file was not interfering with reading or writing the point data.

The data set used was a subset of the Morro Bay area from the San Simeon, CA dataset [1]. The area used falls within (-120.88, 35.36) and (-120.84, 35.40). It contains a total of 163,251,931 points.

5.1 Octree and Icosatree Building Stastics

The Octree building statistics were run with a few different sub-cell sizes. A Table of sub-cell size, elapsed times, max and average points per cell, max tree depth, directory and file count, and total file size can be found in Tables 5.1 and 5.2. With the Icosatree, attempts were made initially using barycentric coordinates as the basis of the indexing system but because barycentric coordinates are a weighted average position within the triangle and not an absolute weight for each vertex the system didnt work as intended. Later, a naive axis-aligned index was used in the same fashion as the Octree sub-cell index but most of the index range went unused. Finally, a triangular coordinate system was developed that computes a position based on the projected position along two of the triangular edges.

The triangular coordinate system computes a sub-cell index by splitting one edge into n-sections. Then, a second edge is also split into n-sections, however, it is numbered in both directions; this creates two columns of sub-cells oriented left-to-right and right-to-left. Then, each sub-cell is numbered with a triple coordinate index. A point is inserted into a sub-cell along the top face of the triangular prism volume and then offset by depth into the volume by splitting the depth by m-sections. A diagram of the triangular coordinates can be seen in Figure 5.1.

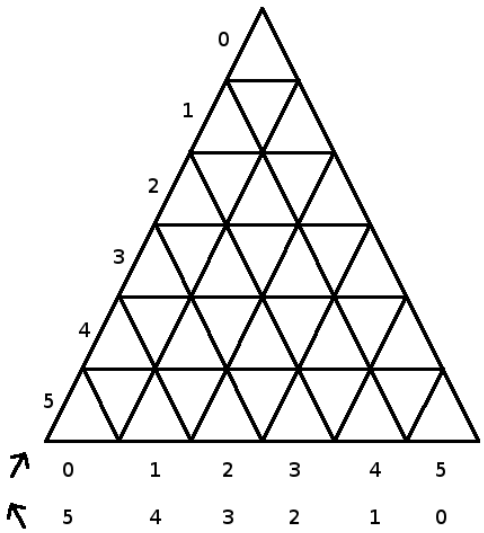

Figure 5.1: Triangular Coordinates

Sub-Cell Size	Tree Creation Time	Cells Created	Filesystem Export Time	Max Points Per Cell	Average Points Per Cell	Max Tree Depth	Directory Count	File Count	Total File Size
50x50x50	5h 8m	1,093,494	20m 48s	15548	150	32	1,093,494	2,186,989	11 GiB, 124 MiB
100x100x100	4h 42m	389,771	15m 55s	44223	419	31	389,771	779,543	11 GiB, 109 MiB

Table 5.1: Octree Building Statistics

Originally, a barycentric coordinate was used for inserting points into Icosatet sub-cells. However, once the system was tested it showed that because the barycentric coordinates cannot be positioned along two axes as with the Cartesian coordinate based Octree, all of the points were skewed towards one corner of the Icosatet. As a temporary replacement,

Sub-Cell Size	Tree Creation Time	Cells Created	Filesystem Export Time	Max Points Per Cell	Average Points Per Cell	Max Tree Depth	Directory Count	File Count	Total File Size
100x100x100	5h 25m	725,672	26m 41s	17123	225	32	725,672	1,451,345	11 GiB, 115 MiB
100x20	7h 18m	653,475	32m 38s	10233	250	31	653,475	1,306,951	11 GIB, 114 MiB
100x100	6h 27m	583,163	25m 43s	14569	280	31	583,163	1,166,327	11 GiB, 112 MiB
150x100	6h 14m	340,758	23m 54s	22618	480	30	340,758	681,517	11 GiB, 107 MiB
200x100	5h 49m	235,156	20m 4s	38081	695	29	235,156	470,313	11 GiB, 105 MiB

Table 5.2: Icosatree Building Statistics

the Octree split algorithm was used but caused a large number of sub-cells to be outside the bounds of the Icosatet and effectively wasted. Eventually, the triangular coordinate system was developed which alleviated the drawbacks of the other algorithms tested and fit well with the use case.

5.2 Point Selection and Triangulation

The Point Selection and Triangulation is implemented with a number of options. First, the CAST algorithm used a Marching Cubes algorithm to compute point densities and prune all points above a certain threshold; this was implemented early on but the issues with using a structured point cloud like San Simeon, CA became apparent. The marching cubes approach ended up removing most of the points selected or almost none at all. In the end, the screen-space lasso was the only portion of the paper implemented. This created a projected frustum into the scene which was used to cull points outside of the selection.

Next, the screen-space point pruning was implement in a few different ways. The application of which is tunable as well as the run order of each portion implemented. The first pruning algorithm added was a naive altitude offset where the min/max altitude range of all points selected was computed and then the nth lowest percentage were removed. Second, an orthonormal threshold was added where, for each point, the k-nearest neighbors were queried. The orthonormal vector was found by computing the eigenvectors for the set of neighbors and if this vector was within a certain threshold of the global eigenvector normal, the point was discarded. Last, the screen-space operator pruning algorithm was used to remove any points that were obstructed by other points by using distance to the

camera and an angle threshold between the vector from the tested point and all other points in the selection. If the tested point was behind another point and the angle between its camera vector and the other points camera vector failed a threshold value, the point was discarded.

Once the point pruning is completed, all points are converted into x,y global eigenvector space and inserted into a map to preserve the local-to-world position pair. The local-space vertices were then sent through a Delaunay triangulation algorithm pass and all triangles exported are then converted back into world coordinates using the local-to-world coordinate mappings.

A few point selection and triangulation examples can be found in Figures 5.2, 5.3, 5.4, and 5.5.

Figure 5.2: Point Selection with Lasso

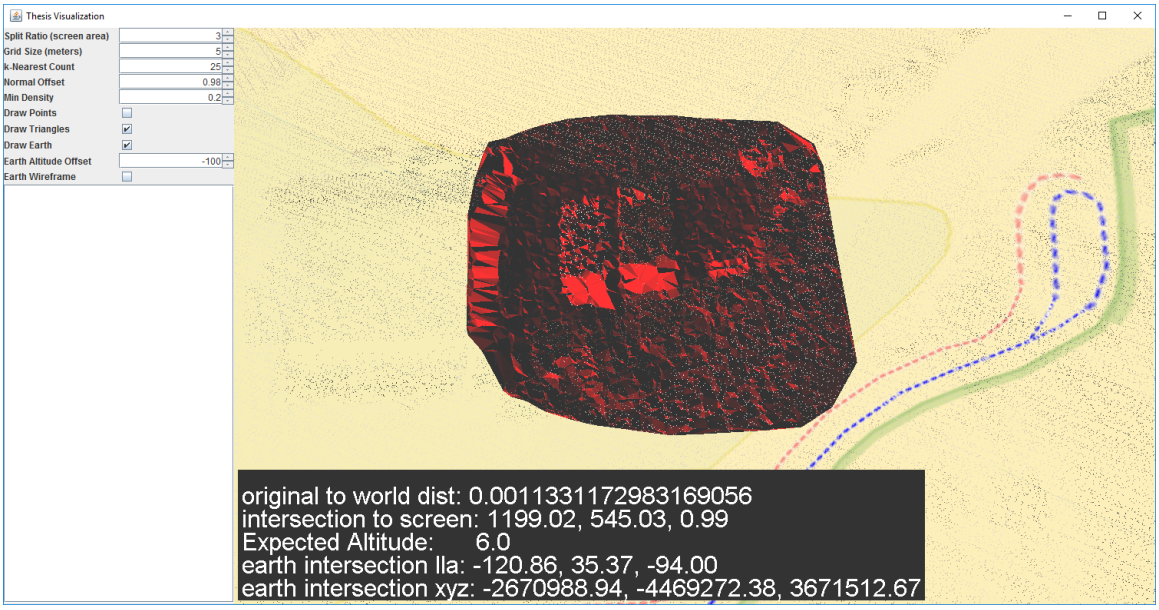

Figure 5.3: Selection Triangulation

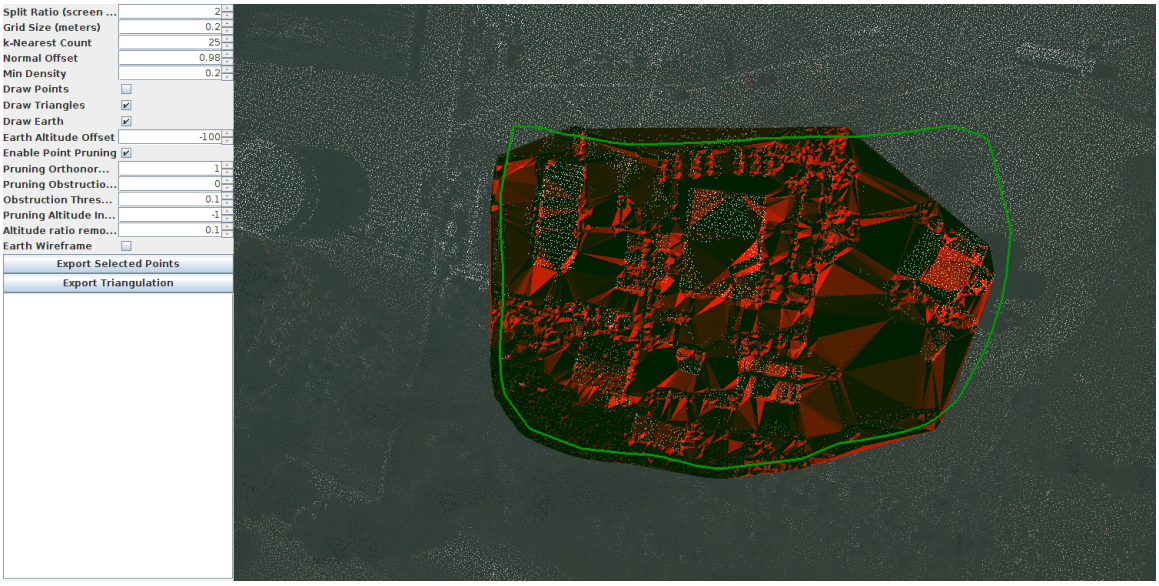

Figure 5.4: Selection Triangulation with Obstruction Pruning

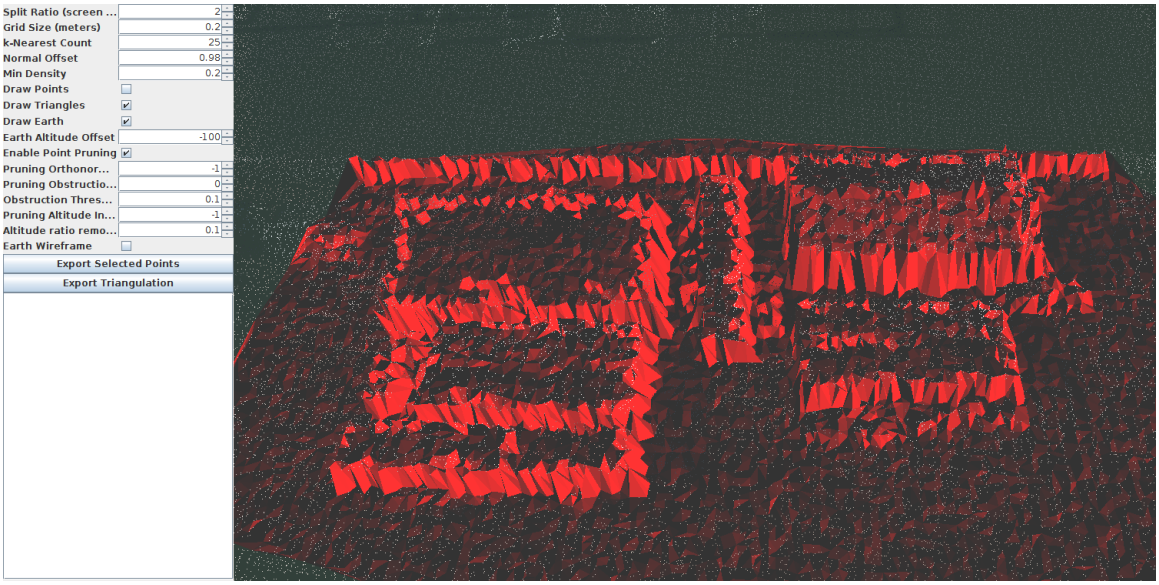

Figure 5.5: Selection Triangulation Closeup

5.3 Octree vs Icosatree Cell Size and Rendering Statistics

The Octree was initially chosen as the data structure for the point cloud data as it allowed a global dataset to be handled without the need to deal with geodetic coordinates and that the Cartesian coordinate space parallels directly with the graphics pipeline. The axis-aligned nature of the cells also allows for efficient culling of the dataset during rendering. However, the Octree structure does not follow the projected surface of the Earth well and there is a large portion of the Octree structure that is never used. The Icosatree structure was chosen as a potential alternative because it aligns with the Earths projected surface more closely, each cell in the tree splits into child cells whose volume can be filled more efficiently, and the handling of the tree file structure is identical to the Octree version so can be used as a direct replacement with very little additional modification and the tree can be traversed with little additional overhead. Also, because the tree cells are filled more efficiently, data usage in the vertex buffer should be less wasteful compared with the Octree implementation. Below, in 5.3 and 5.4, are tables of sub-cell sizes, including Cartesian dimensions, surface

area, and volume for cells at a number of select tree depths.

Tree Depth	Type	X-Span (m)	Y-Span (m)	Z-Span (m)	Side Area (m^2)	Volume (m^3)
18	AABB	64	64	64	4096	262144
19	AABB	32	32	32	1024	32768
20	AABB	16	16	16	256	4096
21	AABB	8	8	8	64	512
22	AABB	4	4	4	16	64

Table 5.3: Octree Cell Statistics

Tree Depth	Type	X-Span (m)	Y-Span (m)	Z-Span (m)	Depth (m)	Top Area (m^2)	Bottom Area (m^2)	Volume (m^3)
18	Triangle Prism	83.7769	95.9998	64.0	29.8935	2709.8502	2709.8296	81006.642
19	Triangle Prism	41.8885	48.0	32.0	14.9468	677.4626	677.46	10125.8496
20	Triangle Prism	20.9443	24.0	16.0	7.4734	169.3656	169.3653	1265.7324
21	Triangle Prism	10.4721	12.0	8.0	3.7367	42.3414	42.3414	158.2166
22	Triangle Prism	5.2361	6.0	4.0	1.8683	10.5854	10.5853	19.7771

Table 5.4: Icosatree Cell Statistics

As can be seen from the tables above, The dimensions and volumes of the two tree variations are comparable in absolute Cartesian bounds but their internal volumes are much different. With the Icosatree cells being smaller but aligned with the projected surface of the dataset they are filled more efficiently as seen in Tables 5.1 and 5.2. This allows the rendering system to display fewer cells in order to render the same amount of data while only adding a small amount of processing time when computing frustum culling per cell which, overall, is still comparable by per cell time (0.16 ms/cell vs. 0.19 ms/cell) as seen in Table 5.5. However, it also adds a bit of point overdraw where the cells that fill the frustum contain more points outside the viewable area but are still drawn as their parent cell intersects the frustum and contains some points within this area. Since the cell fits better with the dataset projection, points will tend to fill the larger, less deep, cells in the tree. The oriented axis cells perform better than the axis-aligned cells as seen by comparing the Octree performance and the Icosatree with axis-aligned cells compared to the Icosatree with oriented cells. The test performed below was on the 100x100x100 sub-cell datasets with the level-of-detail set to maximum; this requires a higher-than-normal per-frame time but was used to show the tree rendering between each type as close to comparable as possible.

The camera was set at a specific location before starting the application so that each run was consistent. An example of the rendering result can be seen in Figure 5.6.

Cell Type	Point Count	Cell Count	Culling	Rendering	Building Bounds
Icosatree (100x100)	3355393	1276	252 ms	1 ms 172 μs	265 μs
Icosatree (150x100)	6065591	1533	236 ms	1 ms 402 μs	220 μs
Icosatree (200x100)	7325988	1522	187 ms	1 ms 765 μs	325 μs
Octree (AABB)	3692901	630	103 ms	755 μs	91 μs

Table 5.5: Rendering Statistics

Figure 5.6: Rendering Statistics Sample Image (depth = 10.0)

If the depth ratio is reduced to a more reasonable use-case, the comparisons between using the Octree and Icosatree become less pronounced and more manageable as seen in Tables 5.6 and 5.7. The number of points rendered is slightly higher in the Icosatree implementation than with the Octree, and per-frame times are increased as well. This is due in part to the smaller altitude component of the icosahedron cell handling less point data

than the much more massive Octree cells. As seen in Tables 5.3 and 5.4, the depth of the icosahedron cells limits the amount of data the cells can handle vertically along the projected surface when compared to the Octree cells. In future implementations, this might be alleviated by not splitting every cell along its depth; such as all cells above a certain depth threshold or every third split, for example.

Cell Type	Point Count	Cell Count	Culling	Rendering	Building Bounds
Icosatree (100x100)	1097116	200	67 ms	636 μs	53 μs
Icosatree (150x100)	2280826	201	47 ms	408 μs	60 μs
Icosatree (200x100)	3825534	201	96 ms	332 μs	52 μs
Octree (AABB)	880582	99	39 ms	379 μs	35 μs

Table 5.6: Rendering Statistics

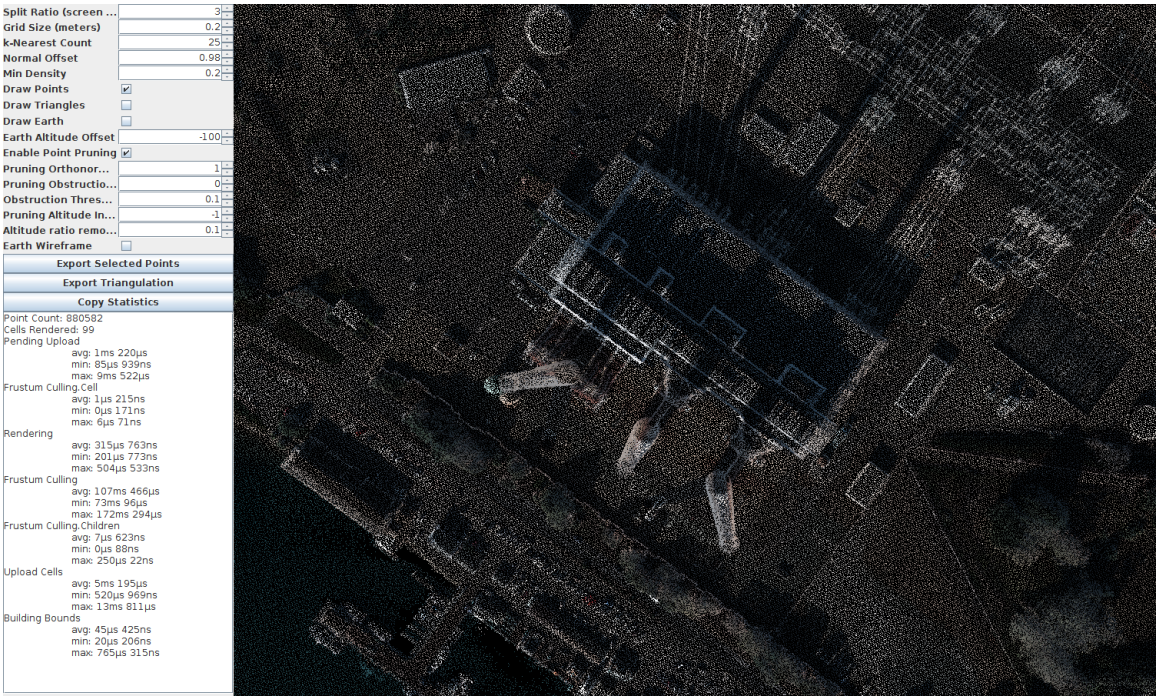

Figure 5.7: Rendering Statistics Sample Image (depth = 3.0)

Cell Type	Point Count	Cell Count	Culling	Rendering	Building Bounds
Icosatree (100x100)	2391155	494	138 ms	910 μs	310 μs
Icosatree (150x100)	4368414	494	92 ms	630 μs	140 μs
Icosatree (200x100)	6239669	493	110 ms	502 μs	87 μs
Octree (AABB)	1915896	223	126 ms	312 μs	46 μs

Table 5.7: Rendering Statistics

Figure 5.8: Rendering Statistics Sample Image (depth = 6.0)

Chapter 6

Conclusions

6.1 Current Status

The algorithms and data structures are in place and are able to build and visualize point cloud datasets from raw point data. The data requires values for X, Y, Z, Red, Green, Blue, Altitude, and Intensity. There is support for custom sets of attributes for the input point data but it has not been implemented at this time.

The visualization supports a camera navigation control based on an anchor location on the Earth's surface as well as a spherical coordinate offset from the defined anchor. This allows the user to set the point along the surface that the camera is bound to but gives them the freedom to rotate around it and move closer to or further from it.

The Earth rendering component supports level of detail so the geometry will progressively become more or less detailed as the distance from it changes. Digital Elevation Model data is used to define the altitude about the ellipsoid for each vertex and Slippy map tiles are supported for imagery (such as OpenStreetMap or Stamen).

The point selection and triangulation algorithms have been implemented such that the user can define a number of toggles and thresholds from the visualization user interface. Then, the user can select a portion of the screen where a lasso will draw on screen and select any points within its bounds. Next, any pruning functions will be applied and any triangulation steps will be executed.

The visualization also supports exporting the selected points or resulting triangulation in the PLY model format.

The performance of the Icosatree was overall promising. The surface-aligned nature of the tree cells was much more visually pleasing as data was updated in the visualization and the tree cells were filled more efficiently for better use of hardware resources. However, sub-optimal tree partitioning values limited the performance of the system overall as the initialization parameters for the Tree data structure are much more important to the functionality of the visualization system with the Icosatree than they are with the Octree. Also, the triangular prism bounding volume requires more calculations than the axis-aligned bounding volume and the sub-optimal tree parameters requires more tree cells to be rendered for the Icosatree so the added complexity was not offset by limiting the number of required tree cells.

6.2 Future Work

The visualization and data creation software currently supports a very specific set of input data in order to build the tree data structures and the visualization assumes specific point attributes will be available (as stated in the previous section). However, support to allow the user to define what attributes are applied to the visualization and support for other coordinate systems, such as longitude, latitude, and altitude instead of XYZ would be helpful. Also, reading a set of LAS/LAZ files directly would be convenient.

The initial testing of the Icosatree data structure turned out promising. However, when building the initial Icosatree dataset and deciding what cell split values to use there were a few glaring issues.

First, more time needs to be taken in selecting proper sub-cell split values for the Icosatree. Selecting the same values as used with the Octree creation does not yield comparable results. The Octree cell split values determines an exact X-Y-Z cell index such that multiplying the split values together give the total number of cells. Another issue here is that the Octree cells are 90% unused in the majority of instances so creating a more dense sub-cell structure is less likely to cause issues with memory management in the long run; this is not the case with the Icosatree. The Icosatree cell split values are used to determine

two separate split coordinates; the first is based on the triangular cell face and the second is based on the depth of the volume. Therefore, the split value for the Icosatree's depth is only loosely based on the same data as the Octree; the Octree is axis-aligned so there is no depth in the same way there is with the Icosatree which is pseudo-surface-aligned (the center points of the faces are aligned with the surface directly below them but the further from the center the less aligned they become). The depth split for each still separates the sub-cells in a similar way; the distance along the axis of separation creates the same number of new sub-cells. For the top face split value, however, is used as an index along its edges. The surface of this face is also much less area than the Octree cells. An Octet and Icosatet of comparable size with split values of six for each dimension will both turn into 216 sub-cells but the Octet will take up more than twice the volume so the Icosatet will store many more point instances in a much smaller volume, not to mention the higher sub-cell utilization from being more surface-aligned than the Octet.

Second, the dataset creation tool needs to be streamlined, multi-threaded, and updated to run on a cached file storage system. Storing the working copy of the tree building data in memory is infeasible for even a moderately-sized dataset. When testing, the initial dataset was a few gigabytes which fit in memory without much issue. However, once a larger dataset was selected, an order of magnitude larger than the original dataset, the in-memory limitations of the original dataset creation tool were very apparent. The tool was then updated to use a memory-mapped database for storing the point data (MapDB) but even that has its limitations. It was not able to create a mapped database for all the Tree cells being created (could not keep all files mapped at once) so a single mapped database was used for all the points. This causes some synchronization slowdowns the larger the dataset becomes. Adding support for in-place file caching and multi-threading could greatly increase performance of the dataset creation tool. Unfortunately, the projected completion time for the larger dataset with the current iteration of the dataset creation tool precluded the testing of the visualization with a larger point cloud.

Next, the point selection and triangulation portion of the paper could be optimized

further, adding a more real-time option to the visualization. As the various pieces of the algorithms were developed they were added ad-hoc to the visualization system. This allowed the pieces to be tested in a modular fashion, however, it is very likely that much of the calculations done could be consolidated or simplified. Also, moving many of these calculations off to a general purpose processing engine such as OpenCL or CUDA, much of the time taken to calculate output could be sped up considerably.

Last, the visualization tool was not the focus of the work for this paper but was necessary in order to display and test the results. However, it is a bit rough around the edges and needs some work. Fixing the graphics pipeline issues in the current graphics engine would be preferable but replacing it with an Open Source alternative might be a viable option depending on the time required to port the current code base to the new platform.

The visualization itself was built on top of a rendering library written by the author but not intended to support direct manipulation of the camera. For this paper it was updated to support mouse interaction and a navigation class was written for intuitive and easy to use controls for moving around an elliptical surface; in this case, a WGS84 projected Earth. However, the interaction with these two systems is not perfect and needs some fine tuning, debugging, and various improvements. Such improvements include support for camera movement based on velocity and acceleration, frustum to bounding volume intersection validation, and world-screen coordinate conversion validation.

Bibliography

[1] Lidar and raster datasets covering 1,500 sq. km. of the ca central coast, 2016.

[2] David Eppstein. File:point quadtree.svg - wikimedia commons, 2005.

[3] Prashant Goswami, Yanci Zhang, Renato Pajarola, and Enrico Gobbetti. High quality interactive rendering of massive point models using multi-way kd-trees. *2010 18th Pacific Conference on Computer Graphics and Applications*, 2010.

[4] KiwiSunset. file:kdtree 2d.svg - wikimedia commons, 2016.

[5] Sriram Krishnan, Christopher Crosby, Viswanath Nandigam, Minh Phan, Charles Cowart, Chaitanya Baru, and Ramon Arrowsmith. Opentopography. *Proceedings of the 2nd International Conference on Computing for Geospatial Research & Applications - COM.Geo '11*, 2011.

[6] Michael Lee and Hanan Samet. Traversing the triangle elements of an icosahedral spherical representation in constant-time. *Proc. 8th Intl. Symp. on Spatial Data Handling*, pages 22–33, 1998.

[7] MYguel. file:tree 0001.svg - wikimedia commons, 2016.

[8] Ruggero Pintus, Enrico Gobbetti, and Marco Agus. Real-time Rendering of Massive Unstructured Raw Point Clouds using Screen-space Operators. In Franco Niccolucci, Matteo Dellepiane, Sebastian Pena Serna, Holly Rushmeier, and Luc Van Gool, editors, *VAST: International Symposium on Virtual Reality, Archaeology and Intelligent Cultural Heritage*. The Eurographics Association, 2011.

[9] Jürgen Richter, RicoDöllner. Out-of-core real-time visualization of massive 3d point clouds. *Proceedings of the 7th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa - AFRIGRAPH '10*, 2010.

[10] K Wenzel, M Rothermel, D Fritsch, and N Haala. An out-of-core octree for massive point cloud processing. In *PROCEEDINGS, IQMULUS 1ST WORKSHOP ON PROCESSING LARGE GEOSPATIAL DATA*, page 53, 2014.

[11] Denis White. Global grids from recursive diamond subdivisions of the surface of an octahedron or icosahedron. *Monitoring Ecological Condition in the Western United States*, pages 93–103, 2000.

[12] WhiteTimberwolf. file:octree2.svg - wikimedia commons, 2016.

[13] Masahiro Wu, YingxinTakatsuka. Spherical self-organizing map using efficient indexed geodesic data structure. *Neural Networks*, 19(6-7):900–910, 2006.

[14] Lingyun Yu, Konstantinos Efstathiou, Petra Isenberg, and Tobias Isenberg. CAST: Effective and Efficient User Interaction for Context-Aware Selection in 3D Particle Clouds. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):886–895, 01 2016.

Appendix A

Glossary

- AABB : Axis-Aligned Bounding Box is a bounding volume used by defining the minimum and maximum values an object occupies within a given coordinate system
- API: Application Programming Interface; a set of programming definitions and tools for building software
- ASCII : American Standard Code for Information Interchange is an character encoding standard
- Barycentric Coordinates : A coordinate system based on the ratio of the area of three internal triangles made by inserting a fourth point within a larger triangle
- Best-Fit Plane : A plane that defines the average position of a set of points through a three-dimensional coordinate system
- Binary Tree : A data partitioning structure that starts with a single root node and then connects to, at most, two child nodes
- Cartesian : a three-dimensional coordinate system with three perpendicular and uniform axes
- Coordinate System : A numerical system for positioning objects within a defined space
- Covariance : Defines how much two variables change in relation to one another

- Covariance Matrix : Defines the covariance between a set of items
- CSV : Comma Separated Values is an ASCII file format for defining tabular data
- CUDA : a Parallel Computing language developed by nVidia that is run on GPU hardware
- Culling : The removal of objects from view; used in computer graphics when renderable objects are not rendered because they are either outside the viewing frustum or are unseen from the current viewpoint
- Delaunay Triangulation : An algorithm for creating a triangle mesh from a set of two-dimensional points
- Eigenvector : A vector that does not change direction when a linear transformation is applied to it
- Frame Time : The amount of time it takes between the start time of a rendered frame to the start of the next rendered frame
- Geodesic : The shortest path between two points on a curved surface
- Geospatial : Data and information relating to geography
- GPU : Graphical Processing Unit; computer hardware used for displaying images to a screen or monitor
- GPGPU : General Purpose GPU; a GPU that can be used for other, non-graphical, computations
- HTTP : Hyper-Text Transfer Protocol; used as the communication protocol for the World Wide Web
- Icosahedron : A three-dimensional shape with twenty faces
- Icosatet : A cell in an Icosatree

- Icosatree : A data partitioning structure based on the Octree but initially defined by a regular Icosahedron and a depth
- Java : A programming language
- JOGL : Java OpenGL is a wrapper library for Java that allows OpenGL to be used
- K-Nearest Neighbors : The closest k objects to a specific item where k is a specific number of objects
- k-d tree : k-dimensional tree is a more specialized form of an Octree
- LiDAR : Light Imaging, Detection, and Ranging is a surveying method that computes distance and intensity of objects by using a laser
- MapDB : A key-value dataset library
- Octet : A cell in an Octree
- Octree : A three-dimensional version of a binary tree
- OpenCL : Open Computing Language; a framework for general purpose computing
- OpenGL : Open Graphics Language; an API for rendering 2D and 3D graphics
- Orthonormal : Two vectors that are orthogonal and normalized
- Picking : Selection of objects using the mouse
- Quadtree : A two-dimensional version of a binary tree
- SSD : Solid State Disk; storage medium used in computers
- Surface Normal : The normal vector for a given surface
- Triangle Mesh : An inter-connected set of triangles

- Triangular Prism : A three-dimensional prism made from two triangular bases connected by three rectangles
- Vertex Buffer Object : A storage mechanism in OpenGL for sending data to a GPU
- WGS84 : World Geodesic System 1984; A standard coordinate system and associated reference surface for converting between a Cartesian coordinate system and Geodesic Coordinate System