Point Cloud Surface Reconstruction

Andrew Morgan[1] (Sole Author), **Nathaniel Morgan**[2] (Project Mentor/Teacher)

[1]Los Alamos High School, Los Alamos, NM, USA

[2]Project Mentor and Teacher, Los Alamos, NM, USA

Abstract

The efficient and versatile reconstruction of the surface of point clouds remains a notable problem throughout computer science, physics, robotics, engineering, and other related fields. Many current methods struggle with noisy data, uneven density distribution, and discontinuities. This paper proposes a solution that uses a level-set-based method integrated with a linearized sparse octree, neighboring node caching, a min-heap binary tree, and surface tension simulation to parse large datasets to reconstruct point clouds as watertight meshes. A basic prototype in Python 3 validated the utility of this approach and provided a foundation on which to construct optimizations. Translations into C++ 17 and Rust implemented these additional concepts and demonstrated notable performance improvements over previous iterations.

1. Introduction

Point clouds are essential tools for countless fields and applications, including medicine, protein synthesis, robotics, computer graphics, video games, geology, lidar, 3D scans, and more. They provide a versatile and unique way to store many types of data and allow for novel algorithms. However, due to their unstructured nature, they have limited direct utility. Many applications require well-defined discrete surface topology, often as polygonal meshes, which point clouds cannot provide. Extracting the isosurface offers a solution and bridges this gap, extending its utility.

Many existing solutions utilize a wide range of methods and techniques. However, many of these have limitations, such as difficulty extracting the isosurface from noisy, incomplete, and

discontinuous data sets. Point clouds are often structureless and highly variable, making it challenging to form generalizations and find easy solutions. Additionally, specific applications require quick processing of point cloud data for real-time applications.

Point clouds can range from a few hundred points to sometimes over a billion, further complicating the matter. Creating a versatile algorithm to handle this extensive range of data and inconsistent and missing information proves challenging. Any solution has to balance performance and memory consumption, as these data sets can reach many gigabytes in size.

This paper proposes an optimized hybrid level-set-based method to help combat these problems. The method takes in an arbitrary point cloud and returns a watertight discrete mesh. Additionally, it is relatively versatile and can handle any number of points, from one to millions. Unless cited otherwise, all of the work and code created in this project was done by Andrew Morgan for the 2025 NM Supercomputing Challenge.

Section 2 discusses multiple approaches to this problem. Section 3 follows, breaking down the steps in the pipeline of the proposed solution. Then, Section 4 explores data structure optimizations, specifically a sparse, linearized, adaptive octree. Section 5 discusses the program's results and validates this approach's effectiveness. Section 6 summarizes the findings for the proposed method.

1.1 Background Information

A quick summary of some background information regarding a few topics mentioned may help in understanding some of the content of this paper.

1. A tree data structure is a structure that starts with a root node. Then, successively, each node starting at that root has a set number of children which may be filled, slowly branching outwards like how the branches of a tree start as one, and over time branch outwards with each branch having its own branches.

2. A perfect binary tree structure is a tree data structure where all branches get fully filled out and progress to the same depth. An imperfect tree would have branches that terminate early or don't have all their children. See [1] from *GeeksforGeeks* for further information.

3. Point clouds are an arbitrarily sized collection of unordered points; they often collectively represent a larger object or structure.

4. Hashes or hash codes refer to a unique index or value resulting from a given input. Some hashes are random, and others are structured. Hashes are fixed-size and often satisfy certain conditions that the former data couldn't. For example, converting a string into an unsigned integer using a hashing algorithm would allow the value to act as an index within an array.

5. Data structures refer to varying methods of storing memory as well as the associations one piece of memory has to another within the collection. Grids are single or multi-dimensional arrays representing a rectangular area in the form of evenly sized and spaced boxes. Two-dimensional grids are also called matrices.

7. Vectors are a continuous array with a dynamic size that never has holes in the middle (often called lists). Removing and adding items to the center or start does reduce performance, though; any values beyond it get shifted in memory to make room or fill in a void, resulting in a large amount of memory movement.

2. Related Work

Many solutions exist for surface reconstruction, with varying strengths and weaknesses. Some more traditional methods, like marching cubes, require a structured scalar field. However, the marching cubes algorithm is still practical as an intermediate step in a more extensive process; marching cubes standing alone is valuable in many other contexts involving more structured data. Other traditional algorithms, such as Delaunay triangulation, require structured data and can be very slow on large data sets. Requiring structured data presents a complication, as many

point clouds don't have an explicit structuring or order. However, this doesn't mean these methods don't have utility, as they're still widely used and play a key role in many areas.

Some newer approaches leverage artificial intelligence (AI) based methods, although they, too, have their strengths and weaknesses. High frequency, fine-tuned details, and more complex topology are intricate to capture with AIs. AIs often excel in a specific area, although they struggle in others. Additionally, AIs require extensive data sets, which, combined with the already significant size, complexity, and scale of point clouds, leads to high computational cost and time complexity. Furthermore, training an AI on complex surface topology proves challenging as there are countless variations and a lack of structure or unified patterns between or inside data sets. These factors limit the adaptability and generalization of AI implementations, making them fall short of the overarching goal of this project.

There are many other miscellaneous solutions, although this project focuses on level-set methods. Level-set-based methods rely on mathematically extracting the isosurface level through various means. Many implementations utilize signed distance fields (SDFs) to represent the point cloud. SDFs are often much more structured, even with an adaptive data structure (for example, an octree or kd-tree), allowing for more traditional methods, such as marching cubes or dual contouring, to be combined into a systematic pipeline. In other words, utilizing SDFs in conjunction with other techniques allows for hybrid methods, balancing accuracy, performance, memory consumption, and adaptability. This adaptability while maintaining reasonable performance makes a hybrid level-set-based method well-suited for the project's goal.

3. Signed Distance Field Representation

While point clouds may be highly variable, the signed distance to the nearest point at any given position has much more structure. A primitive way to choose which points to sample the signed distance is to create a 3D array with known bounds and positioning. This primitive solution is the exact approach taken for the prototype in Python 3. However, it has inherent flaws.

Because arrays are a fixed size and spacing, areas of low detail (i.e., very few or no points) require the same amount of memory allocation as an area with lots of detail. Additionally,

when computing the signed distance, low-detail regions will receive the same computation time and resources as those of high detail. Additionally, areas of low detail, which don't need a lot of expensive computation or significant memory allocation, receive a large portion of the available resources. The over-allocation of resources in low detail areas also takes away critical computation and memory necessary to evaluate complex topology regions accurately.

However, using an adaptive octree data structure can fix this issue. While octrees are far more complex than traditional grids, the implementation mentioned in this paper adaptively subdivides the structure in areas of high and complex detail while giving sparse areas more limited representation. This data structure, for one, saves a lot of memory. In a simple test case, it consumed nearly 29,000% less memory when storing just the signed distances (from 8MB down to 30KB for a basic grid of 64-bit floats, not including additional information on the actual structure). The benefit of the octree is further compounded because there are fewer nodes or points at which to sample the signed distance, and less computation is needed overall. This reduction in computation and memory allows for increased resources in more complex and intricate point cloud sections, resulting in greater detail and precision. More depth on this octree implementation and other data structures are in Section 4.

There is one issue with this current method. A known surface contour is necessary to create a signed distance field (SDF) instead of a regular distance field. Constructing an unsigned distance field from the point cloud instead of an SDF alleviates this problem. After this, an algorithm determines which sections are solid and which are hollow. A shell around the surface is then created by generating the exterior edges of the part(s). Because this shell is solid, the known surface contour allows for calculating a proper SDF. This pipeline process is broken down further in Section 3.1.

3.1 Signed Distance Pipeline

One inherent issue in generating a signed distance field, as discussed in Section 3, is that a known surface contour is necessary to get the signed part of an SDF. The solution is to break the

process into four steps: calculating an unsigned distance field, signs, solid edges, and finally, computing the complete SDF.

The initial step of creating an unsigned distance field is relatively trivial. The process involves looping over every node or grid cell in a given data structure and performing a nearest neighbor search on the point cloud (calculating the minimum distance to the nearest point). However, some complexity arises when optimizing and executing the search on an octree. Section 4.1 details the implementation of the nearest neighbor search on an octree.

A more straightforward optimized solution for a fixed grid is a chunking system, also referred to as hashing. The process relies on grouping all the points into unique vectors or arrays based on their local position. A good example is how the game Minecraft divides the world into 16x16 chunks. These chunks allow for a smaller, localized search to expand as needed to find the nearest point. Creating a smaller search radius improves performance by looking over fewer points in any given search. Implementing this solution in the prototype script in Python 3 gave decent performance gains, considering the reduced complexity compared to other algorithms.

Step two calculates the signs for the unsigned distance field using a novel algorithm. The algorithm calculates every grid cell by repeating the following set of 4 steps. (1) The initial step is to loop over all 1D slices facing a single axis and step through each cell one by one. (2) At each marched step through a given slice, check the unsigned distance; if the distance is less than the isosurface level, continue stepping along until the distance is greater than or equal to the isosurface level. (3) If the grid cell in the corresponding array for storing signs contains a filled point, save the current tracking sign as that sign and continue along; otherwise, flip the tracking sign and fill the entire region between the boundaries created by the isosurface and unsigned distance field with that sign. (4) Repeat these steps until every slice finishes its calculation. Like previous algorithms, octrees cause complications and require modifications to the underlying algorithm; Section 4.1 goes into these necessary modifications.

Step three involves calculating a shell around any object's exterior edges (in other words, voxelizing the distance field of the point cloud). Similar to the first step, the process is relatively simple. The primary step is to go through every grid cell or node and check a few conditions: if a

hollow point is directly adjacent to the cell or node (diagonals don't count) and the current position is solid, add a new surface point.

The final step builds upon the previous step to generate the final SDF. Similar to the first step, start by going through every point and calculating the unsigned distance. However, this time, use the surface shell rather than the point cloud to calculate the unsigned distance. After getting the distance, check the sign at the given node or grid cell position; if the sign indicates it's solid or the original unsigned distance is less than the isosurface level, flip the sign of the current distance. This final step concludes the calculation of a proper SDF, allowing a continuation in the larger pipeline.

3.2 Surface Tension Simulation

Due to the nature of the SDF generation, natural surface ungulations occur in the reconstructed part. However, a scalar field surface tension simulation solves this problem by smoothing higher frequency bumps on the surface; this method also preserves a lot of lower frequency bumps, although it won't work as well on every application. [2] breaks down the math behind the surface tension method. The surface tension simulation works by finding the curvature of the surface and raising the troughs while dropping the peaks. A summary of the math from [2] is as follows:

The first Equation (1) solves for the level set field while applying a front velocity of F. ϕ represents the level set field. i, j, k represent the position, and they can also represent the index within the grid. t represents time.

$$\frac{\phi_{i,j,k}^{n+1}-\phi_{i,j,k}^{n}}{\Delta t} = max(F,0)\nabla_{i,j,k}^{+} + min(F,0)\nabla_{i,j,k}^{-} \tag{1}$$

Where:

$$\nabla_{i,j,k}^{+} = [max(D^{-x}\phi,0)^{2} + min(D^{+x}\phi,0)^{2} + ...$$

$$\ldots max(D^{-y}\phi, 0)^2 + min(D^{+y}\phi, 0)^2 + \ldots$$

$$\ldots max(D^{-z}\phi, 0)^2 + min(D^{+z}\phi, 0)^2]^{\frac{1}{2}}$$

$$\nabla^{-}_{i,j,k} = [min(D^{-x}\phi, 0)^2 + max(D^{+x}\phi, 0)^2 + \ldots$$

$$\ldots min(D^{-y}\phi, 0)^2 + max(D^{+y}\phi, 0)^2 + \ldots$$

$$\ldots min(D^{-z}\phi, 0)^2 + max(D^{+z}\phi, 0)^2]^{\frac{1}{2}}$$

D^{-x} refers to the backwards finite difference operation in the x direction. D^{x} refers to the forwards finite difference operation in the x direction. This applies to all three dimensions – x, y, and z. V is a constant representing a constant velocity inwards or outwards; it can either keep the object's size or shrink or expand the object depending on its value. F is defined as the front velocity and equals:

$$F = V - \kappa$$

V is a constant that moves the level set field in the normal direction. Kappa is the curvature of the front:

$$\kappa = \nabla \bullet \frac{\nabla\phi}{|\nabla\phi|}$$

The second term equates to the surface normal. Multiple iterations each solve these equations and adjust the scalar field, smoothing it over time. The more time steps (while shrinking the time duration), the better the results. Some parameters can lead to instability and undesired results if not correctly set.

3.3 Isosurface Extraction

Signed distance fields are the first step to reconstructing the surface of a point cloud. However, an intermediary step is necessary to provide a more discrete representation. Some traditional reconstruction algorithms, like marching cubes and dual contouring, become useful here. The previous year's submission, which implemented marching cubes, acted as an initial solution for the Python 3 prototype and basic C++ implementation.

However, complications arise when applying marching cubes to a more dynamic structure, like an octree. While many solutions exist, most create intersecting geometry, which can lead to inconsistencies in physics simulations and other applications, or have non-watertight gaps. A method that solves this, proposed in [3], not only creates watertight meshes but also does so efficiently and without modifying the underlying octree; in other words, the octree is unconstrained, allowing for optimizations tailored directly to point clouds. The method relies on constructing a set of edge trees and using them to properly align geometry to intersecting node boundaries when using a hybrid-dual contouring approach. Section 4.1 dives into the implementation a bit deeper.

3.4 Hybrid Reconstruction Pipeline

Combining these steps — unsigned distance field calculation (Section 3), calculating discontinuities and signs (Section 3.1), surface tension simulation (Section 3.2), and isosurface extraction (Section 3.3) — creates an efficient pipeline from a point cloud to a discrete polygonal mesh representing its approximate surface.

Each step serves a purpose in the greater pipeline. The first step is gathering a more useful and structured representation of the point cloud — this initial step contains multiple steps, which Section 3.1 breaks down. From there, the surface tension simulation can smooth any artifacts and unnatural surface undulations. Finally, extracting a level set of that final scalar field produces a discrete and water-tight polygonal mesh; this step produces an STL file, allowing seamless integration of the mesh into most commercial software along with many algorithms.

4. Octree Data Structures

While this pipeline can create great results, it also consumes excessive resources from unnecessary computation and a massive memory footprint. A solution to this overconsumption of resources, albeit far more complex than a fixed 3D array, is to use an octree data structure instead. By dynamically subdividing the octree's node structure — essentially just adding more children to create a deeper tree — in areas of high complexity and detail, the octree can represent sparse sections with limited memory and computation while also redirecting those resources to regions of higher topological complexity.

Like other tree-data structures, octrees have a root node but, from there, have exactly eight children, each of which are nodes capable of creating more children nodes to branch the tree further out. Each node represents a cube or rectangle, and each subdivision divides the parent bounding box into eight equally sized boxes. The leaf nodes, which represent the deepest nodes that have no children of their own, store a vector of indexes referencing which points in a static array, representing the point cloud, fit within their bounding box. One way to do this is to keep a constant address or pointer referencing the original array. Alternatively, when needed, provide a parameter for the point cloud in the methods of the octree structure in languages like Rust. By passing the point cloud in as a parameter, it keeps its ownership within its original scope, preventing ownership and borrowing errors.

To dynamically subdivide the octree, follow a set of 3 rules. (1) If the current depth has reached the maximum specified depth for the octree, push all point indexes within the bounding box into an array or vector for the current node, and then stop subdividing as it's now a leaf node. (2) If there is either one or no points within the current node's bounding box, turn the node into a leaf node; in other words, stop subdividing the particular node. (3) If there are multiple points within the node's bounding box, which means the previous two rules weren't satisfied, subdivide the current node into eight children nodes and continue the rule-set for each of those individual nodes.

Just these steps alone can provide significant performance gains. However, there are other possible improvements. These other optimizations rely on linearizing the data structure. Linearizing an octree involves representing all the data in a single, contiguous array. This paper's linearization implementation involves utilizing an array where each element contains another array of eight integers. Each of those eight integers represents an index to that exact same array to act as a pointer to the node's children. In the case of a leaf node, the eight indexes can either be replaced by null or by a value representing null; in the context of Rust, *Some(index)* represents standard indexes, while leaf nodes contain 8 *None's*. This structuring provides an efficient way to follow the tree's many branches and determine whether a node has children or is a leaf node. To store points in a leaf node, another array aligning with the original contains vectors to store references to points in the point cloud; an array with a predetermined size also works for representing the vector in memory. Utilizing that vector, adding a point is as easy as accessing the array at the index of the leaf node and pushing the points' indexes in the point cloud to the vector.

This linearized design has a few notable advantages. The first benefit is that the octree's data lines up contiguously in memory, allowing for more cache hits and quicker data fetches; cached information is also naturally aligned sequentially in increasing order, allowing for more efficient lookup algorithms, such as binary searches (also known as a bisect search).

```
pub fn BinarySearch <T: Eq> (points: &Vec <T>, searchValue: &T) ->
     Option <T> {
  let mut currentIndex: usize = 0;
  let mut dividedSize = points.len();

  let mut halfWidth: usize;
  // this could also be a loop; however, this prevents runaway code
  for _ in 0..MAX_BINARY_SEARCH_ITERATIONS {
    halfWidth = dividedSize / 2;
    dividedSize -= halfWidth;  // splitting the bounding size
    if let Some(middleValue) = points.get(currentIndex + halfWidth) {
      if middleValue == searchValue {
         return Some(currentIndex + halfWidth);
      } if middleValue < searchValue {
        // splitting the search
```

```
        currentIndex += halfWidth;
      }
    }
  } None
}  // Rust
```

Mortan codes (Z-order curves) [4] are another useful and performant hashing technique involving bit manipulation to create unique codes for any position that maintain spatial locality (two neighboring points will have similar hash codes). Hash maps are sometimes useful due to some variability in the codes' values. The algorithm works by taking three 32-bit numbers representing the three axes. The bits interweave, so every three bits contain a bit from the x, y, and z coordinates, creating the pattern: $x_1y_1z_1x_2y_2z_2$…. This results in a 96-bit unique hash, although Rust has 64-bit and 128-bit integers, so a larger number than the code is necessary. Because of the implementation within the octree, the code gets represented as an unsigned integer (this makes the final type a 128-bit unsigned integer or, in Rust, u128).

```
pub fn GetMortonCode (&self, xi: u32, yi: u32, zi: u32) -> u128 {
    let mut x = xi as u128;
    x = (x | (x << 16)) & 0x030000FF;  // magic nums -> stackoverflow
    x = (x | (x <<  8)) & 0x0300F00F;  // spacing the x bits out
    x = (x | (x <<  4)) & 0x030C30C3;  // creates room for y, and z
    x = (x | (x <<  2)) & 0x09249249;
    //... the same as above for y and z
    x | (y << 1) as u128 | (z << 2) as u128  // interweaving all bits
}  // Rust
```

Another performance gain occurs as accessing an array at an index is faster than chasing repeated pointers to other instances of nodes. This second point compounds with the previous ones and also alleviates the issue of slow neighboring node computation times; neighboring nodes are expensive to find and scale alongside the number of points and depth of the octree. This expensive calculation becomes problematic because the nearest neighbor search traverses the octree from node to node to identify nearby points while minimizing the search radius, and many neighboring nodes are needed to do this. Further compounding that issue, the nearest neighbor search must run for every node multiple times. However, because each node in the linearized structure is represented by a single integer index/identifier, another array that aligns

with every node's index can store a vector containing the indexes to every neighboring node for that given node. While this caching system requires an expensive computation for every node to find all its neighbors upfront, it prevents the necessity of doing these computations many times during each search. In fact, with this caching method, traversing the octree is possible in constant time, regardless of the number of points or the octree's depth (the search algorithm runs in log n time due to incorporating additional algorithms beyond traversal). Implementing a similar caching system with a nonlinear structure would be incredibly difficult; any solution would still involve an expensive descent from pointer to pointer every single time a cache lookup happens. While challenging to implement, these two major optimizations provide significant performance gains, making them worthwhile. The specific implementation of the nearest neighbor query on an octree is very complex compared to chunk or grid-based methods. Section 4.2 details the exact implementation used in this paper.

4.1 Octree Pipeline Integration

While the octree provides notable improvements, other algorithms within the greater pipeline are inherently unable to handle the variability. The sign generation algorithm falls short here because it traverses line by line, row by row; however, octrees don't have a perfectly aligned structure because of their adaptive structure. One solution is using a system where all nodes bounding the edges of the outline get pushed to a vector, and that vector acts as the starting point for further iterations; every iteration, the sign gets flipped, beginning at a point on the outside with a value of one representing hollow space.

The second issue falls within the surface tension algorithm. Octree-based scalar-field surface tension simulations are much more complex than their grid-based counterpart. The solutions go beyond the scope of this paper. Because of the complexity and scope, the surface tension simulation was discluded for the octree-optimized pipeline despite the notable improvements on lower resolution point clouds.

The final problem arises when reconstructing the surface of the scalar field. With a grid-based solution, marching cubes provided excellent results. However, Marching Cubes

doesn't translate as well to an octree. A solution proposed by [3] uses a hybrid method stemming from dual contouring. The approach proposed in that paper took in an unconstrained octree and returned a water-tight mesh. Again, this solution goes a bit beyond the scope of this paper. The paper cited below [3] provides an excellent breakdown, though.

4.2 Nearest Neighbor Query

One of the more complex aspects of integrating the octree is the nearest neighbor search algorithm (related to voronoi cells). [5] proposes an elegant solution and inspired some optimizations. For this paper's implementation, an expanding search radius provides the fastest results by pruning unnecessary data. However, finding the neighboring nodes to any given leaf node proves challenging. Additionally, searching for neighbors is too costly. A solution is to use a neighbor caching system; an array the length of the number of leaf nodes stores vectors containing the integer indexes of all neighboring nodes (discussed in Section 4).

From there, the solution is relatively trivial. Get all the neighbors starting at the node nearest to the sample position. Add all those neighbors to a priority queue (4.3) based on their distance to the query point. Then, for every iteration, pop the root node from the queue and continue. For every point encountered, keep track of the shortest distance. Finally, once the queue is empty or the shortest distance to the nearest node falls beyond the minimum distance found, return that minimum distance.

Locating the nearest leaf node to a given position is another challenge, though not nearly as complex. The first step is to force the query point into the bounding box of the octree using min and max. From there, start at the root node and iterate the number of times as the octree is deep. While iterating, keep track of the current node. The size of all nodes at a given depth is stored in a pre-computed array ($1.0 / 2.0^{depth}$). Take the query point and find its distance from the node's base. From there, divide that difference's x, y, and z components by half the node's size. Take the result and cast it to an integer of 0 or 1. Each node's eight children have different offsets from the node's base, which get stored in a constant order; using that known order, the current set of three integers allows for a reverse lookup of offsets to get the child's index within the current

node. Repeat that until encountering either a leaf node or the maximum depth (which would also be a leaf node). Some algorithms may require tracking the path to the node for traversal up the tree; usually, this only requires the last calculated leaf node, resulting in a negligible memory size.

4.3 Min-Heap Binary Trees

Min-heap binary trees, also referred to as priority queues, are binary trees where the root node always contains the smallest value. The counterpart would be a max heap binary tree; however, in the context of this paper, it isn't needed. Binary trees are similar to octrees. However, each node only has two children. *GeeksforGeeks* [6] provides a great article that breaks down binary heaps and inspired the implementation used in this paper.

The C++ standard library has a priority queue implementation that has min and max heap variants (the following is a min-heap binary tree, dictated by std::greater):

```
std::priority_queue<double, std::vector<double>,
     std::greater<double> > nodeQueue;
```

Some other languages' standard libraries may include an implementation with varying performance and versatility. For a manual implementation, the following dictates a min-heap binary tree; max-heap trees would be the same except search for the maximum instead of the minimum value. Insertion, popping, and swapping are the most important methods for this binary tree.

Swapping. The backbone of the other two algorithms relies on swapping values to satisfy the tree's rules: no value should be below a value greater than itself. When inspecting a node, look at the first branch; if the value is less than the current one, swap their values (ideally without altering the underlying data structure to reduce memory movement). Otherwise, check the right branch and do the same. If the condition fails for both branches, the value is in the correct position. Usually, this method gets called until the condition fails or the value reaches the bottom of the tree. This swapping method can also begin at the bottom of the tree and swap upwards to meet the condition until the parent is equal to or smaller than the current value.

Insertion. The first step requires constructing a new node with the given value. This new node becomes a child for one of the leaf nodes. After that, the swapping method iteratively places the value into the proper position.

```
pub fn Push (&mut self, value: (f64, usize)) {  ...  }  // Rust
```

Popping. The first step involves popping the root node and returning its value (and possibly co-value or index so it can reference additional information) – the root is always the smallest value. However, the binary tree requires a root node to function. The solution involves first removing one of the leaf nodes. From there, the leaf node replaces the root node. Then, the swapping method satisfies the conditions by swapping the root node downwards.

```
pub fn Pop (&mut self) -> Option <(f64, usize)> {  ...  }  // Rust
```

One way to optimize the queue is to attempt to balance the tree in the form of a perfect binary tree (1.1). Because the tree repeatedly gets restructured, there isn't always a perfect solution. The approximate solution explored in this paper relies on tracking the children of all leaf nodes. Two buffers are necessary to do this: one for future children below the maximum depth reached and one for the next deepest layer. The first buffer gets used when appending a new node; the final index of the first buffer represents the child, which receives the new value. After placing the value, the swapping method moves the value into position to satisfy the conditions. By having the two buffers, the tree will initially fill voids within the maximum depth of the tree before expanding the tree's depth. The second buffer dumps its contents into the first when it becomes empty. Most of the complexity comes from maintaining both these buffers as the tree mutates.

5. Validation

A few different scenarios validated the effectiveness of the approach and implementation. The first method used mathematical equations to generate a point cloud around a known shape. After that, the reconstruction pipeline took in the point cloud, and the results reasonably matched the inputted shape, as seen in Fig. 1. A

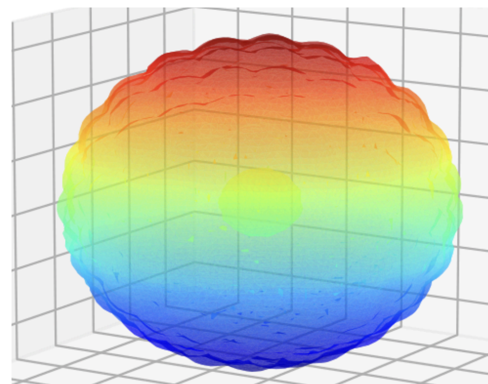


Figure 1. Hollow Fib. Sphere

known geometric model further tested the pipeline—an obj file from the Stanford Bunny (Fig. 2) allowed for a known comparison while also containing points that, on their own, have no spatial connection to each other. Similar to the first test, the results aligned reasonably well with the test model (Fig. 3). The specific Stanford Bunny model used had multiple holes in the surface geometry. However, the pipeline properly filled those holes while maintaining a reasonable level of accuracy in the overall model. This specific test didn't use surface tension smoothing.

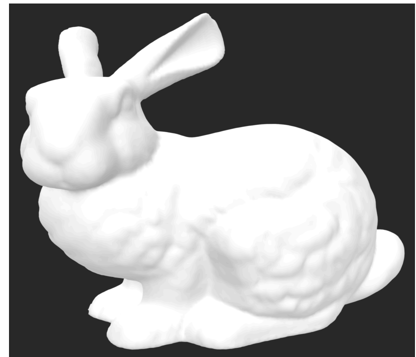


Figure 2. Stanford Bunny

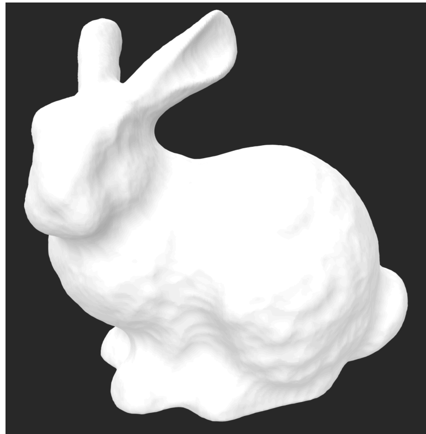


Figure 3. Reconstructed Bunny

The adaptive subdivision of the octree required rendering all corner points for every node to verify the structure and subdivision method. The corner points aligned with expectations, forming smaller nodes in regions of denser data while minimizing nodes in lighter areas (Fig. 4).

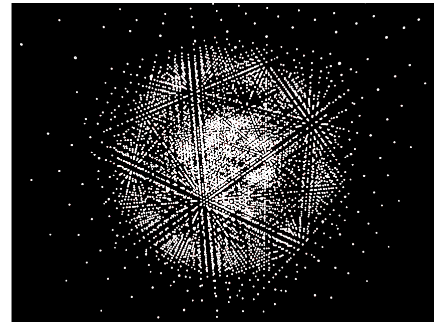
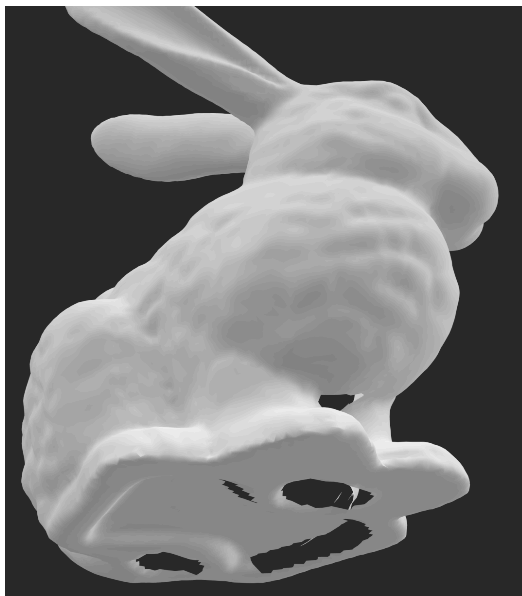
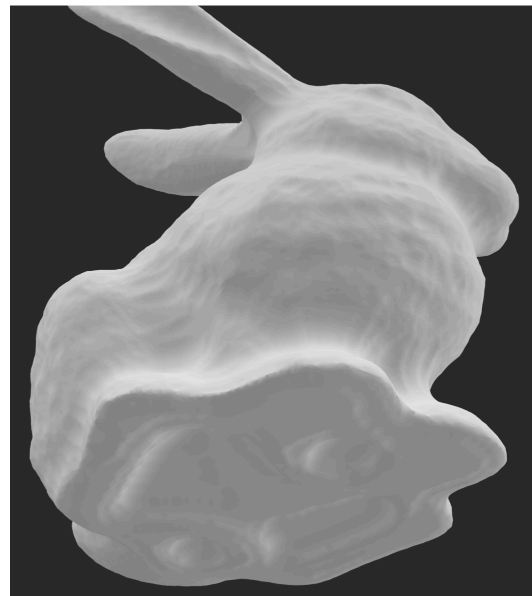


Figure 4. Hollow Sphere[1] Octree

The following are examples of the holes before and after reconstruction:





5.1 Results

All of the code and assets within this project are available on GitHub. The link is in Section 6.1. The point cloud data files (the program can load .pcd files, which are similar to .ply files that contain a slightly different header) used for reconstruction came from [7]. It provided a number of detailed point clouds, which were invaluable and allowed for excellent results.

Fig. 5 and Fig. 6 show the reconstruction of the Stanford Dragon. The reconstruction shows a decent improvement after running the surface tension simulation (Fig. 6). Additionally, the reconstructed model shows a decent handling of sharp corners (Fig. 7) and finer details (Fig. 8) while also keeping small gaps separated correctly (Fig. 9). Note that the mesh is water-tight.

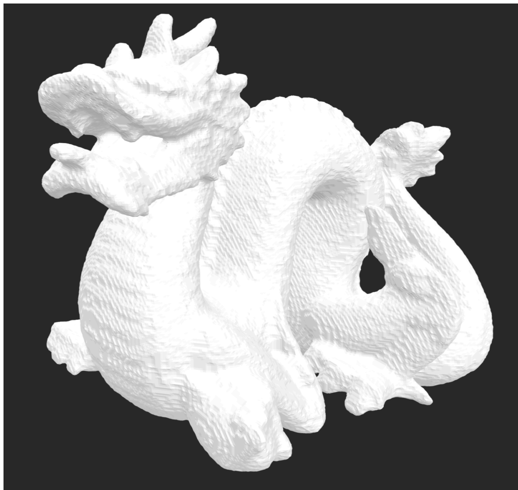


Figure 5. Unsmoothed Dragon

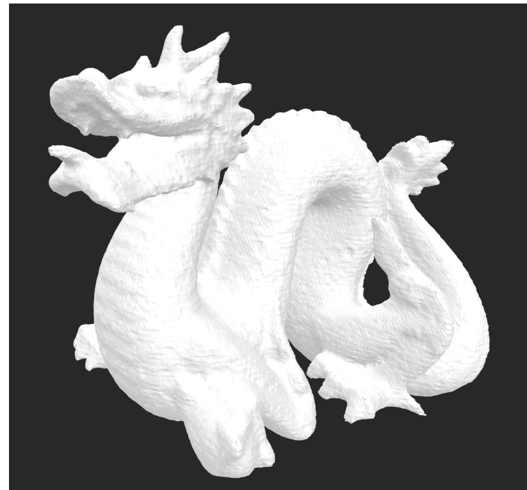


Figure 6. Semi-Smoothed Dragon

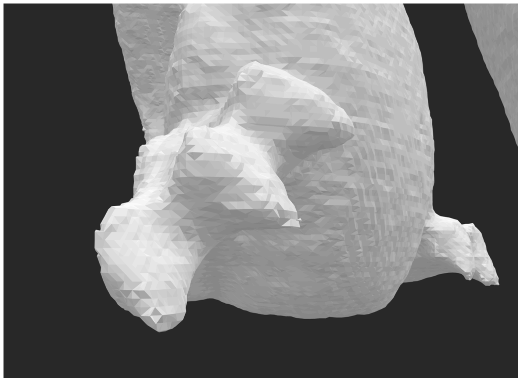


Figure 7. Sharp Claws

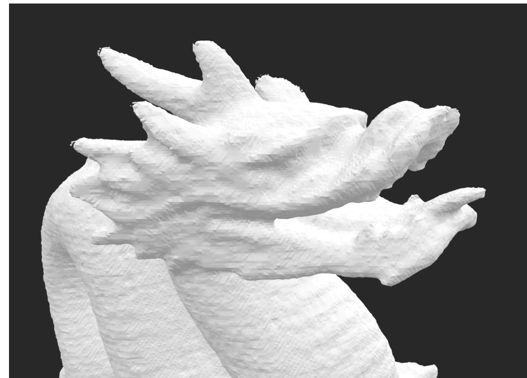


Figure 8. Finer Face Features

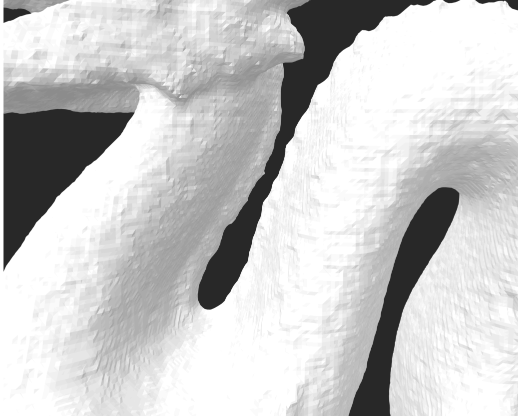


Figure 9. Narrow Gaps

The second result comes from the Egyptian mask [7]. The model contains a thin mask without a bottom. Additionally, the point cloud is somewhat noisy, complicating the reconstruction. Fig. 10 shows the reconstructed mesh, which handled the noisy data and gaps in the model well, only containing a few minor artifacts. Fig. 11 shows the bottom of the mask where the opening is along with the thinness of the mask (the mask should be fairly thin).

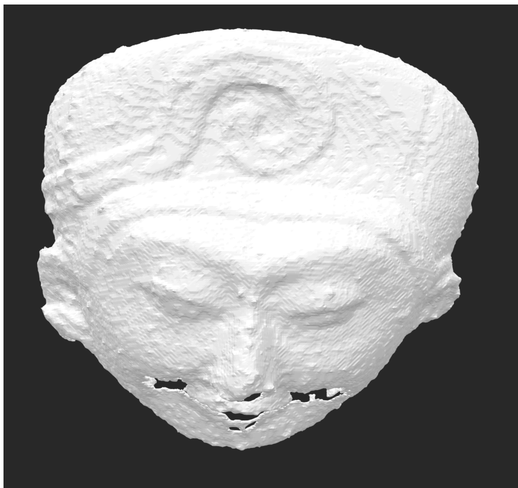


Figure 10. Egyptian Mask Front

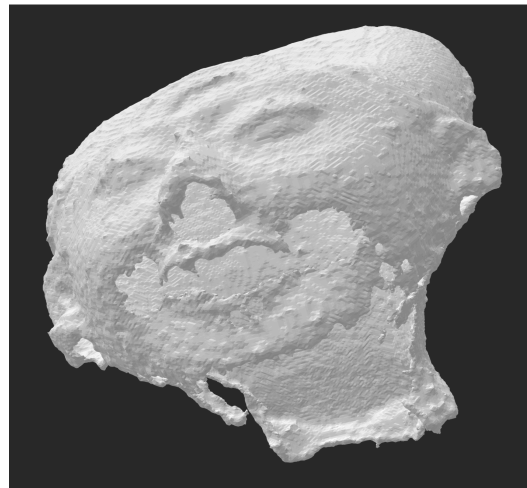
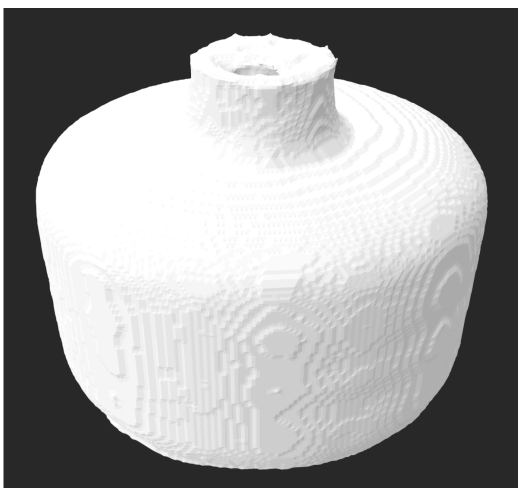
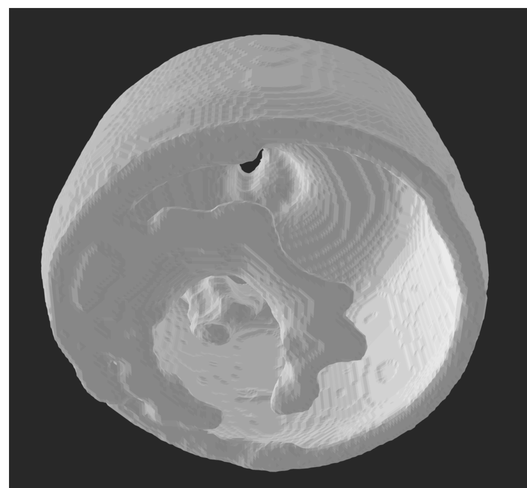


Figure 11. Bottom of the Mask

The final point cloud is a 3D scanned vase [7]. Similar to the mask, the vase has thin walls and a hollow interior (the Stanford Bunny and Dragon both had solid interiors). Additionally, the vase has a couple of sharper corners, presenting a challenge that the algorithm handled well.





5.2 Versitility and Limitations

The implementation covered has many strengths but also many weaknesses. Starting with the strengths, the algorithm can handle anywhere from a single point to millions – some other algorithms can't handle such significant disparities in data sets. Additionally, the algorithm can perform well on noisy or incomplete data sets (as seen with the Stanford Bunny in Section 5). In addition, while computational parallelization hasn't been implemented yet, many steps within the pipeline are well suited to run in parallel, which would provide significant performance improvements over the current results.

However, there are some limitations facing the current iteration of this project. The first is that the many steps in the pipeline each require allocated memory, and when combined, they result in a larger memory footprint than ideal; according to the activity monitor, mac's version of task manager, the program was using upwards of 1.25GB of ram while running on a dataset of a quarter million points when not using an octree. Additionally, thin objects can get slightly thickened due to the SDF generation. Furthermore, because of the SDF generation, surface tension simulation, and Marching Cubes in the non-octree pipeline, sharp corners and higher-frequency data can sometimes get smoothed over if the resolution becomes too small relative to the data. Also, the pipeline as a whole isn't running quite as efficiently as desired, and as such can't be used in real-time applications – although, with additional improvements, there is potential for significant performance gains over the current iteration. Reconstruction speeds varied from 30 seconds to 45 minutes (from around 40,000 points to upwards of 500,000 on a high-resolution grid), depending on the version, optimizations, and parameters like grid size or the number of points in the point cloud. Note that all benchmarks were taken on an older Mac M1, so a more performant computer would likely provide better benchmarks.

6. Summary

A hybrid level-set method provides a dynamic and versatile means to reconstruct the surface of arbitrary point clouds. Combinations of other algorithms expand that versatility and also offer greater performance while reducing the memory footprint. The first step of approximately voxelizing the point cloud provides a decent base to work from – some applications may only want the voxelized data and nothing beyond. From there, the generation of a proper SDF allows for a smoother and more accurate representation of the object. Additionally, that SDF is compatible with the surface tension simulation, Marching Cubes, and other algorithms.

A surface tension simulation is one such algorithm, providing a smooth output that removes artifacts and surface ungulations formed from the nature of SDFs. Additionally, a seamless integration of the simulation into the greater pipeline is relatively trivial.

Sparse, linearized octrees prove promising, providing excellent performance and memory with the only tradeoff being the complexity. Caching neighboring cells utilizing the benefits of the linearization further improves the nearest neighbor search, which has to run many times. Min-heap binary trees can further optimize the search query, efficiently generating signed and unsigned distance fields.

6.1 Final Remarks

I would like to thank Nathaniel Morgan for helping me come up with the initial idea for this project. I would also like to thank him for helping me interpret some of the math involved in the surface tension simulation.

In addition, I would like to thank a group at LANL for allowing me to present my project and for providing feedback on everything. Their library, MATAR, was also very helpful by providing a memory-safe multi-dimensional array structure for C++ (in place of alternatives like std::shared_ptr, std::unique_ptr, or unsafe raw pointers).

All of the code and assets used within this project are on GitHub: https://github.com/AndrewDMorgan/Point-Cloud-Surface-Reconstruction. The code is 54%

C++, 29% Rust, and 17% Python 3. In total, all three program versions came out to a total of a little over 4,800 lines. Through development, prototyping, iteration, and revision, over 8,000 lines were written, although much of that got refined and compressed over time.

References

[1] "Types of Binary Tree," *GeeksforGeeks*. [Online]. Available: https://www.geeksforgeeks.org/types-of-binary-tree/. [Accessed: Mar. 30, 2025]

[2] S. Osher; R. and Fedkiw, *Level Set Methods and Dynamic Implicit Surfaces*, Springer-Verlag, 2003. ISBM 978-0-387-95482-0

[3] M. Kazhdan, A. Klein, K. Dalal, and H. Hoppe, "Unconstrained isosurface extraction on arbitrary octrees," in *Proc. Eurographics Symp. Geometry Processing*, 2007

[4] "Z-order curve," *Wikipedia*. [Online]. Available: https://en.wikipedia.org/wiki/Z-order_curve. [Accessed: Mar. 30, 2025]

[5] B. H. Drost, S. Lilc, "Almost constant-time 3D nearest-neighbor loopup using implicit octrees," 2018 *Machine Vision and Applications*. [Online]. Available: https://doi.org/10.1007/s00138-017-0889-4. [Accessed: Mar. 30, 2025]

[6] "Priority Queue using Binary Heap," *GeeksforGeeks*. [Online]. Available: https://www.geeksforgeeks.org/priority-queue-using-binary-heap/. [Accessed: Mar. 28, 2025]

[7] "RG-PCD: Reconstructed Geomatry Point Cloud Dataset," [Online]. Available: https://www.epfl.ch/labs/mmspg/downloads/reconstructed-point-clouds-results/. [Accessed: Mar. 30, 2025]

Future Reading

E. Alexiou, M. Bernardo, L. S. Cruz, L. G. Dmitrovic, R. Duarte, E. Dumic, T. Ebrahimi, D. Matkovic, M. Pereira, A. Pinheiro and A. Skodras, "Point Cloud Subjective Evaluation Methodology based on 2D Rendering," 2018 *Tenth International Conference on Quality of Multimedia Experiance (QoMEX)*, Cagliari, 2018, pp. 1-6. Doi: 10.1109/QoMEX.2018.8463406

B. H. Drost, "Almost constant-time 3D nearest-neighbor loopup using implicit octrees," *Springer Nature Link*. [Online]. Available: https://link.springer.com/article/10.1007/s00138-017-0889-4. [Accessed: Feb. 14, 2025]

S. Lague, "Coding Adventure: Marching Cubes," *Youtube*. [Online]. Available: https://www.youtube.com/watch?v=M3iI2l0ltbE. [Accessed: Feb. 12, 2025]

"The PCD (Point Cloud Data) file format — Point Cloud Library 1.14.1-dev documentation," *Point Cloud Library*. [Online]. Available: https://pointclouds.org/documentation/tutorials/pcd_file_format.html. [Accessed: Dec. 20, 2024]