Graphical Computing Solution for Industrial Plant Engineering

Marcelo Lima

Master's thesis in Software Engineering at

Department of Informatics,
University of Bergen

Department of Computing, Mathematics and Physics,
Bergen University College

June 2017

ABSTRACT

When preparing an engineering operation on an industrial plant, reliable and updated models of the plant must be available for correct decisions and planning. However, especially in the case of offshore oil and gas installations, it can hazardous and expensive to send an engineering party to assess and update the model of the plant. To reduce the cost and risk of modelling the plant, there are methods for quickly generating a 3D representation, such as LiDAR and stereoscopic reconstruction. However, these methods generate large files with no inherit cohesion. To address this, we propose to find a solution to efficiently transform point clouds from stereoscopic reconstruction into small mesh files that can be streamed or shared across teams. With that in mind, different techniques for treating point clouds and generating meshes were tested independently to measure their performance and effectiveness on an artifact-rich data set, such as the ones this work is aimed for. Afterwards, the techniques were combined into pipelines and compared with each other in terms of efficiency, file size output, and quality. With all results in place, the best solution from the ones tested was identified and validated with large real-world data sets.

Acknowledgements

I would like to thank my supervisors Harald Soleim and Atle Geitung for their support and help to this work. Their guidance and discussions throughout the whole process was immeasurable and this thesis would not have come to be without them. Also, I would like to thank the support from APIteq, especially the help given by Tor-Erik Rong and Per Erik Berger. They were invaluable in providing the data for this work as well helping to bring the objectives to a real-world scenario and problem solving.

Marcelo Lima

Contents

List of figures

List of tables

List of code snippets

Glossary

CAD	Computer Aided Design
CGAL	Computational Geometry Algorithms Library
CLI	Command Line Interface
Embarrassingly Parallel	A class of problems that can be easily broken into parallel tasks
Isosurface	A surface that represents points of constant value
Gaussian	A bell-curve mainly used in statistics to describe a normal distribution
GPGPU	General Purpose Graphical Processing Unit
GUI	Graphical User Interface
LAS	LASer File Format
LiDAR	Light Detection and Ranging
LOP	Locally Optimal Projection
MC	Marching Cubes
MLS	Moving Least Square
Octree	A tree with its internal nodes containing eight children that can be used for data modeling
PLY	Polygon File Format
WLOP	Weighted Locally Optimal Projection

1 Introduction

1.1 Motivation

In order to improve reliability, efficiency, accuracy, and effectiveness of engineering planning, decision making, and ultimately execution, the precise modeling and representation of data pertinent to the project is of utmost importance. This is especially true with industrial plants, such as the oil and gas refineries and production facilities, which have a high degree of complexity and present elevated consequences associated with deficiently planned and executed engineering projects.

The generation of these models, therefore, reside at the root upon which major engineering project are built, defining its final quality and success. In an ideal scenario and with the advent of computer-aided designs (CADs), the whole stream can be represented digitally, from the initial planning, construction, servicing, and upgrades, keeping a dependable record of the current state of the project to the level of detail required for any future work. Yet, in reality, this is often not the case, with changes to the plant not being recorded, be it minor alterations, ad-hoc solutions, missing documents, or deformations and damages from use and environment. On the extreme side of the spectrum, there are also many instances of older plants having their original plans represented as blue prints and no management of changes to the facility.

To assess the issue of unreliable models, engineers must rely on on-site surveys to gather an updated version of the necessary data, which can then be transmuted into a format compatible with the tools used in the project, such as CAD models. However, this solution has some mentionable drawbacks: it is possibly a time-consuming option, it can involve several personnel, it can have a high cost – particularly on offshore projects –, and it is prone to variance in the quality of the data and final product. The latter can be further expanded and attributed to the human component of the process, where different technical abilities, mistakes, and pressure to keep schedule might exacerbate the variance in quality

Currently, industries have the option of using automatically generated models of their industrial plants as a faster and lower-risk alternative to on-site surveying. Methods such as Light Detection and Ranging (LiDAR) and Stereo

Imaging can be used to produce point clouds, which represent three dimensional objects or scenes by means of infinitesimal points expressing the surface of the modeled subject.

Despite the lowered costs, turnaround, and operational risk, this surveying method has drawbacks of its own: the model generated is highly abstracted and, especially in topologically complex environments, susceptible to many artifacts, such as noise, missing data, and outlying points. This means that part of the cost saved during the automated surveying is actually transposed to the transmutation phase of the process, requiring skilled engineers to convert carefully the point cloud into a usable model.

1.2 Goals

The goal of this work is to compare techniques and strategies to generate a model of an industrial plant from point cloud data that is concise and small enough to be shared and streamed across teams while preserving the necessary information to assist engineering, design, planning, and operations.

Moreover, the scope deals with a particular type of point clouds: stereoscopically generated point clouds of large industrial plants. The data set used in this work comes from APIteq, a company in Bergen, Norway, specialized in visualization for the oil and gas industry. Their portfolio includes 360° Visual Assessment Management, a tool for visualizing oil and gas plants in a 360° format, and are now investing in the tridimensional visualization using the reconstruction of stereoscopic images taken from a plant. In that sense, the point clouds with which this work will be dealing are of varying density throughout the data, high artifact count, such as noise and “holes”, very large binary size, and topologically complex.

Finally, concerning the complexity of the point clouds that are in focus, lower computational times are therefore a major metric, and exploration into high performance computing languages, data models and algorithms, and exploration into parallelization and harnessing heterogeneous computational environments, are part of this work.

1.3 Research questions

This work aims to answer if it is possible to compress the data with little engineering information loss in an efficient manner. That is, can an industrial point cloud generated by stereoscopic reconstruction, with all their inherent properties, be distilled without sacrificing much fidelity in terms of information needed to assess an engineering decision, such as dimension and topology, while being completed in a quick and lightweight fashion?

2 Background

2.1 Point Clouds

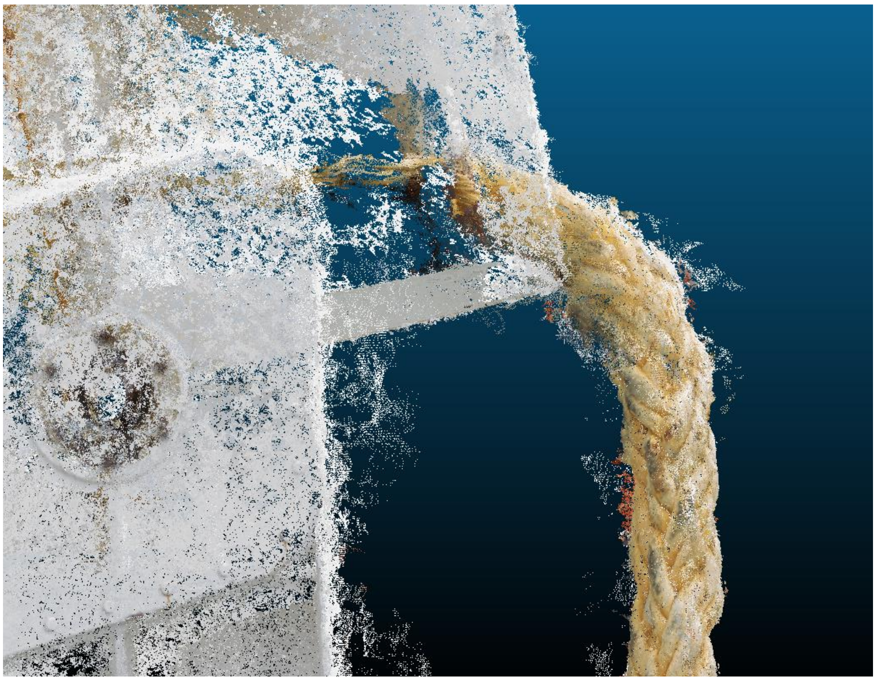
Point cloud is a representational method consisting of individual points in some coordinate system. The points are infinitesimal and carry no information of size or length; nevertheless, they can represent any property associated with that particular position in space, such as color, normal vector, global positioning system (GPS) coordinates. This format is often used to represent surfaces in three-dimensional space, such as terrains from aerial scanning or objects and environments from Light Detection and Ranging (LiDAR) scanning.

2.1.1 Point cloud treatment

Point clouds possess several properties and each can vary greatly. The density of the point cloud relates to how many points a given volume in space has; thus, in the absence of noise, a denser a point cloud will have a higher fidelity. Noise in a point cloud relates to the presence of outlying points, that is, points that do not belong to any surfaces of the modeled subject; e.g., a noisily modeled Euclidean plane will be misrepresented as a volumetric shape. Besides noise, another artifact that can be present in point clouds relates to the aforementioned density, in which the number of points representing a surface is lower than the Nyquist frequency that samples such surface, therefore rendering a lossy model. The size of the point clouds directly relates to the shear amount of points in a given collection; thus, larger datasets require heavier computing power. The last property relevant to this thesis is the topology of the modeled subject, which relates to the set of properties preserved when an object is deformed; for instance, an ellipsoid is topologically equivalent to a sphere, since it can be obtained by deforming the latter.

Figure 1 shows the detail of the data set being used in this thesis serving as an example of a noisy point cloud. In this example, outlying points and severe variation in density can be observed.

Figure 1: A point cloud with several artifacts



2.1.2 Reconstruction from point clouds

Due to its nature, there is no cohesion between the points as they are what the name suggests, merely a cloud of points. That means that they are very adequate to represent sample points – a discreet representation model, in which each point corresponds to a sampled value from a LiDAR scan, for instance – as opposed to continuous information, such as a parametric representation would. This is of utmost importance to this study, since the continuous representation of an industrial plant is key for precision and, depending on the representational model adopted, a decisive factor on the final file size.

In other words, if a parametric representation is used, that is, a representation that can describe the information through parametric functions, and therefore having infinite resolution, need not worry about the precision level. On the other hand, a discreet representation will only supply half of the resolution of its own sampling, meaning that, for high-precision applications, the number of samples can increase drastically and, by consequence, the file size.

The recovery of the original sampled object is a subject of intense study and techniques vary vastly. In its simpler form, a point cloud might represent a two-dimensional curve with enough sample points so that a linear interpolation might render sufficiently precise results. This is rarely the case; as it was discussed before, point clouds frequently possess artifacts and the modeled object is often more topologically complex.

Aiming towards industrial plant reconstruction, Pang et al [1] proposed a reconstruction solution that separates the process in three sub-problems: pipe modeling, plane classification, and object recognition. The work acknowledges that industrial plants share certain characteristics such as an abundance of pipes and large planes and leverage these to generate a CAD model of the original plant. On his dissertation, Rabanni [2] explored different techniques to avoid human intervention on the reconstruction of industrial installations from point clouds. The process involved point cloud segmentation, object recognition, and model fitting, and also relied on the intrinsic characteristics of industrial plants to focus the effort. With the expansion of the field of surface reconstruction in mind, Berger et al [3] released a state-of-the-art report on surface reconstruction in which different techniques are categorized per parameters, requirements, and robustness. Moreover, this work highlights similarities between the techniques and was an influential guide on the decisions taken during this thesis.

2.2 LAS FORMAT

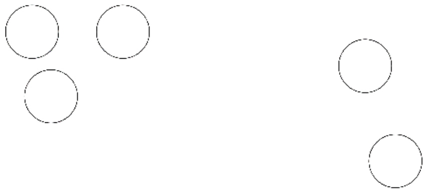
The LASer file format is a means to represent point cloud data in a standard fashion ratified by the American Society for Photogrammetry and Remote Sensing (ASPRS). The current latest version by the time of writing is 1.4, approved on November 14, 2011 [4]. Though primarily aimed at providing a public format for point clouds that would overcome the barriers of proprietary formats, the specification is actually capable of representing any 3-dimensional tuplet.

The format is able to carry additional information about a point besides its position, such as color, GPS time, and LiDAR specific information. However, the scope of this work focuses mainly on a point's location and color.

2.3 Weighted Local Optimal Projection

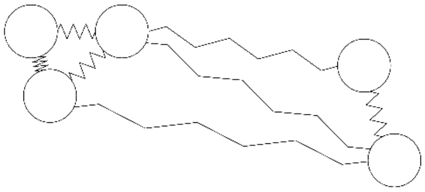
In order to address the artifacts of point clouds, Lipman et al proposed the *locally optimal projection* algorithm [5] to approximate the underlying surface, thus reducing outlying points and noise. It also strongly improves point density uniformity by defining repulsion and attraction parameters to each point, consequently keeping the projected point set at a regular spacing. These parameters can be thought of as springs connecting each point that becomes overstretched if they are too far apart and, similarly, become compressed if they are too close together. With each iteration, the points tend to converge to a uniform distribution. To illustrate the concept, a simple non-uniform point cloud of five points can be used with the points clustering around two regions.

Figure 2: Non-uniform point cloud



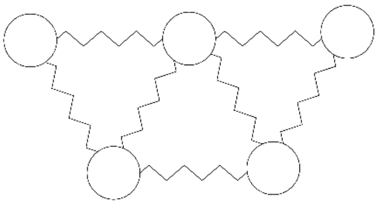
If the parameters of repulsion and attraction are considered, there will be tension and compression forces being applied to the points.

Figure 3: Repulsion and attraction components applied to point cloud



Upon convergence to a state closer to equilibrium, the points are more uniformly distributed.

Figure 4: Converged point cloud



Noticing a lack of clear-cut convergence on the algorithm, Huang et al proposed a modified version called *weighted locally optimal projection* [6], which introduces locally adaptive density weights that directly affect the repulsion and attraction of the points within a given region.

2.4 Moving Least Squares Smoothing

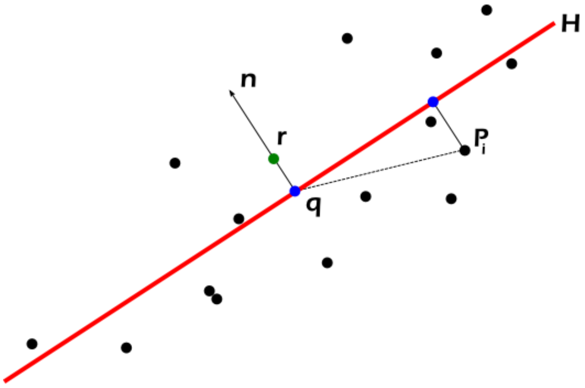
An alternative method for reducing artifacts such as outliers and noise is to use Moving Least Square projection [7] [8] [9] to generate a smooth point set projected into an implicit surface defined by the original data set. This method works by fitting a bi-quadratic polynomial, i.e. a quadratic polynomial with two variables, and projecting a given point onto it. It comprises of two steps.

First, to project a point r, a reference plane H is calculated using weighted least squares optimization, where the weights are based on the vicinity of r by using an always-positive smooth decreasing function. That means that a Gaussian function can be used as the weighing function. In that sense, the weights are based on the distance between q and the neighbor p_i. This step is trying to minimize the distances of p_i to H, as in:

$$\sum_{i=1}^{N} \langle n, p_i - r - tn \rangle^2 \, \theta(\|p_i - r - tn\|)$$

Where n is the normal, θ is the weighing function, and $q = r + tn$. See Figure 5.

Figure 5: MLS smoothing first step

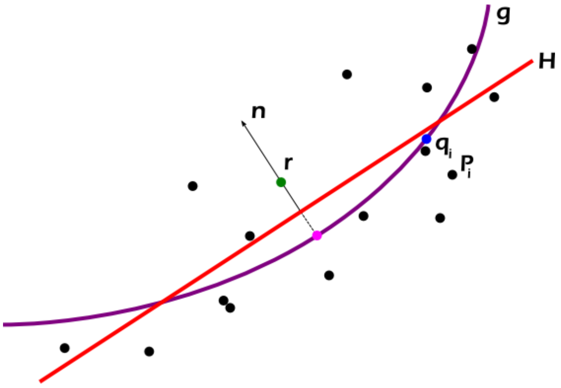


Second, with H defined, we can consider q_i to be the projection of p_i onto H, and f_i to be the orthogonal height of p_i to H as in $f_i = n \cdot (p_i - q)$. We can then minimize f_i in relation to a curve g, as in:

$$\sum_{i=1}^{N} (g(x_i, y_i) - f_i)^2 \, \theta(\|p_i - q\|)$$

Where (x_i, y_i) is q_i in terms of H coordinates. See Figure 6.

Figure 6: MLS smoothing second step



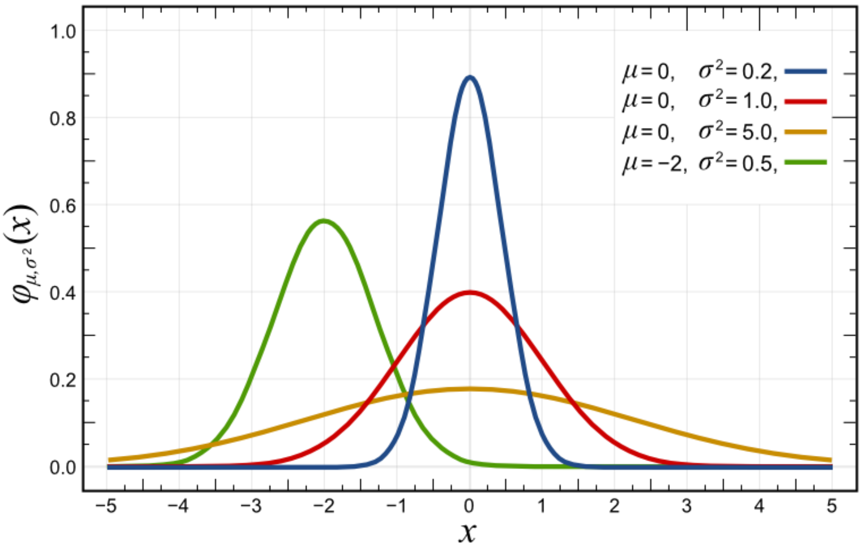
It is worth noting that the Gaussian is a bell curve centered on μ that tapers off according to σ with an amplitude given by a. The function is defined as:

$$f(x) = a\, e^{\frac{-(x-\mu)^2}{2\sigma^2}}$$

In this function, μ is usually referred as the mean and σ as the standard deviation. Figure 7* shows different shapes for the Gauss function according to the parameters presented here.

* This figure was not created by the author of this thesis. It was taken from Wikipedia [29] and is of public domain

Figure 7: Examples of Gauss functions with different parameters



2.5 Marching Cubes

With the growing presence of point clouds in modern industry and research, there is a great interest in converting this format into other domains, such as surface meshes, for several applications from medical imaging, geological reconstruction [10] to sculpture scanning [11]. Marching cubes [12] is an algorithm that extracts an isosurface, that is, a surface that represents a single value, and generates meshes by sampling a scalar field $f\colon \mathbb{R}^3 \to \mathbb{R}$. This technique is specially fitted for regular scalar fields structured as a tridimensional grid, though other representations have been used for marching cubes, such as octrees [13].

Given a grid, this method works by iterating over each cell and classifying it according to each of its corners in terms of its value, i.e. if it is above or below the isovalue α. That means that each cube can be classified as one of 256 possible values, or a byte: two values per vertex, and eight vertices per cell ($2^8 = 256$). If $\alpha = 0$, for instance, Figure 8 shows a cube where vertices 1, 3, 4, and 8 are positive, and the remaining vertices are negative. This would translate to 10110001, where low bits mean negative vertices and high bits mean positive vertices. With the classification in hand, marching cubes uses a lookup table to create a surface inside the cell, illustrated in Figure 9. The isosurface will therefore separate positive from negative vertex values, in our example.

Figure 8: Cube marching classification

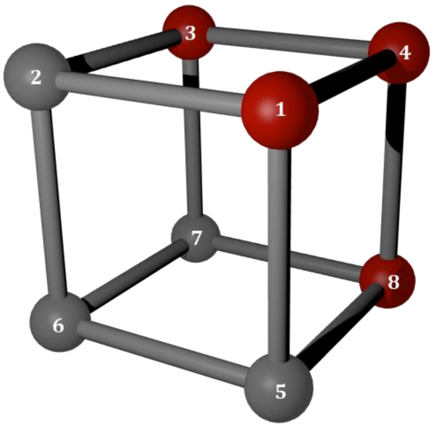
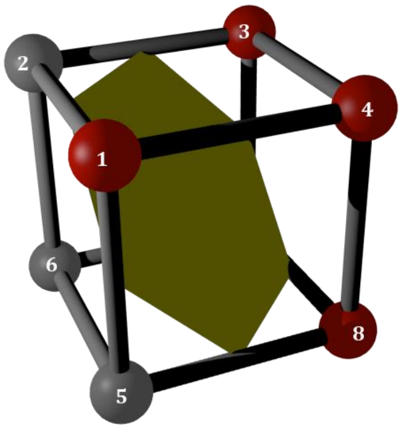


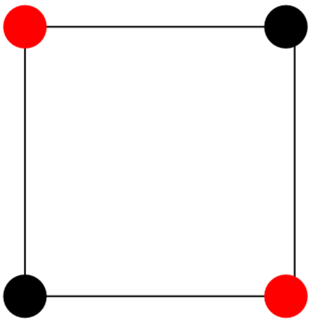
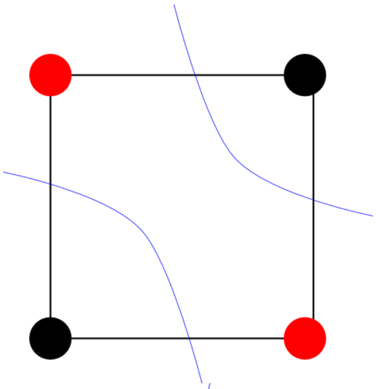
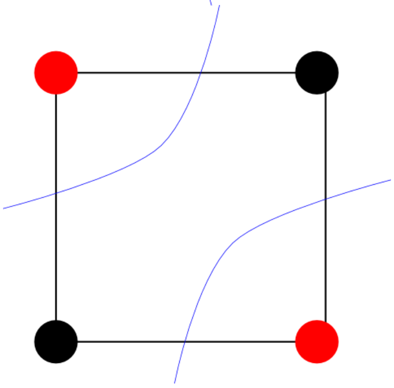
Figure 9: Face added per the lookup table



2.5.1 Lewiner's approach

This simplistic approach might generate ambiguities, leading to cracks and inconsistencies, and therefore various enhancements and techniques have been introduced. Table 1 below presents an example of such an ambiguity in a 2D domain. Both the solutions are correct but only one will maintain topological consistency. A particular improvement proposed by Lewiner [14] guarantees a topologically correct result and uses a more elaborate lookup table to solve ambiguities. His work proposed that, after performing the classical classification, the entries should be compared per face against a *case table* that would specify which test should be performed on the particular face of the cube. The results of the tests would then be mapped against one of 730 entries of the lookup table for face creation. To guarantee the correct topology, not only adjacent cubes are matched according to the tests performed, but also the internal ambiguities are resolved.

Table 1: Marching cubes ambiguity example

Situation	*Explanation*
	Original case. Considering $\alpha = 0$, vertices in red are positive and black vertices are negative.
	Connected case. The isoline can separate the square leaving the two positive vertices inside the same surface.
	Disconnected case. The isoline can separate the square leaving the two negative vertices in different surfaces.

2.6 Poisson Surface Reconstruction

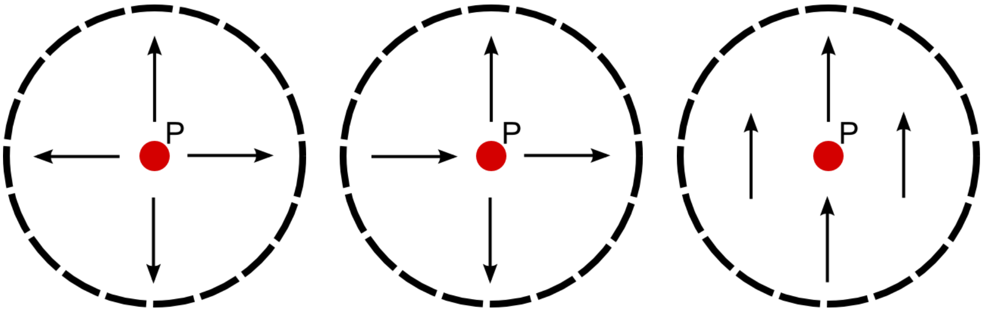
Similar to Marching Cubes, Poisson Surface Reconstruction [15] [16] is a method for generating meshes from point clouds; however, it models the problem as a spatial Poisson problem to find the indicator function and then extract the isosurface.

An indicator function is a function that, for every point inside the model, its value is 1, and for all other points, the value is 0. Therefore, its gradient is zero everywhere, except for the surface, where it is equivalent to the inverse of the normal of the surface. With that in mind, the authors of this method suggested that, by using the normals from a sampled model, they could calculate the indicator function by inverting the operation and trying to match an indicator function that best matches the normal vector field.

In other words, if we consider $\vec{V}$ to be the vector field that defines the surface normal and $\mathcal{X}$ to be the indicator function, we can model the problem as $min_{\mathcal{X}}\|\nabla\mathcal{X} - \vec{V}\|$. Moreover, if using the divergence operator, they can be equaled, as in $\nabla \cdot \nabla\mathcal{X} = \nabla \cdot \vec{V}$.

The divergence operator, denoted by $\nabla \cdot$, measures the flow out of a surface surrounding a given point, analogous to the curl function that measures the rotation of the vector field surrounding a point. For instance, Figure 10, shows three cases of a vector field surrounding point P. In the first case, the divergence of the field is positive, as there is a positive outflow from the point. The second case also has a positive divergence, however of a smaller magnitude. Finally, the third case has a value of zero, since all the inflow and outflow cancel each other.

Figure 10: Divergence operator examples



With that in mind, the problem can then be modelled as a Poisson's equation such that the Laplacian of $\mathcal{X}$, which is defined for the Euclidian space as the divergence of a gradient ($\nabla \cdot \nabla$, or simply Δ), can be equaled to the above statement and arriving at $\Delta\mathcal{X} = \nabla \cdot \nabla\mathcal{X} = \nabla \cdot \vec{V}$.

This method is very robust to noise and is able to keep high frequencies from the surface, that is, sharp corners or rapid variations on the surface normal, as demonstrated in [15], however it relies on having normals sampled with the point cloud, which is not the case for the stereoscopically generated data set used for this work. Nonetheless, they can be computed beforehand with the penalty of introducing one extra step.

2.7 OpenCL

OpenCL is an industry standard framework for programming heterogeneous systems, i.e., computing systems that involve multiple architectures within the same context. A ubiquitous example is the modern home computer and cellular phones, which usually have a general-purpose CPU and a dedicated GPU. OpenCL allows a program to be written in a single language and have it loaded into the compliant processors and executed with guarantee of the standard's definitions. Such definitions include minimum accuracy requirements, data type sizes, atomicity, among others; therefore, the developer can focus on the algorithm as opposed to architectural peculiarities, and trust that it will execute as expected on different architectures. Moreover, the framework defines how drivers should load the program, queue their execution, and handle memory.

A valuable advantage of OpenCL is being able to treat a GPU as a general-purpose processor, since, if it complies with the standard, it can be executed just like an OpenCL programmer would expect any other processor to execute. Considering this, the architectural properties of the GPU are still present, but they can be tapped into without necessarily aiming for graphical computation.

2.7.1 Graphical Processing Unit

The graphical processing unit, or GPU, is a specialized hardware conceived with the purpose of handling image generation and rendering. One common trait is the highly parallel architecture in which they are designed. More precisely, they leverage a data parallel model – consequence of handling pixels in an image – which consists of a single task being performed concurrently across separate parts of the data. This characteristic of the GPU fomented the exploitation of the hardware for other computations besides image processing, also known as general-purpose computation on graphical processing units (GPGPU). Though general-purpose, these applications usually are aimed at embarrassingly parallel problem solving, meaning solutions that require a large number of independent computations.

2.7.2 Validation

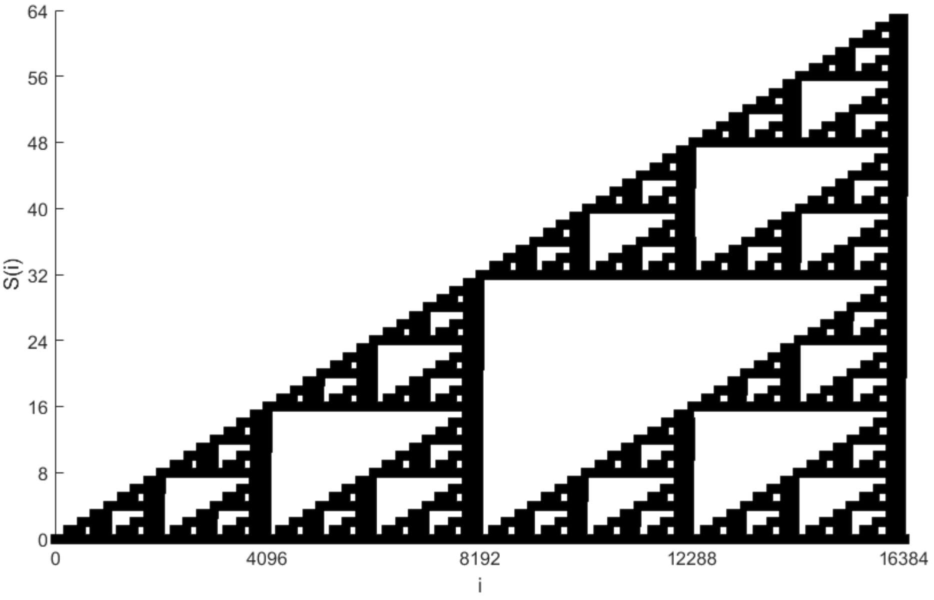
The motivation behind using this platform for performing the intense bulk of the computation relies on the aforementioned parallelism and ability to harness GPU processing characteristics, which are aligned with the properties of the problem-space being studied, i.e. an embarrassingly parallel workload involving many floating-point calculations.

To validate the strategy, a simple computation was performed using three different computing models: serially on the CPU, in parallel on the CPU, and in parallel on the GPU. The computation was designed to be embarrassingly parallel; thus, it calculated a Sierpiński triangle [17] by defining the set S, with indices denoted by I as:

$$S = \left\{ \sum_{n=0}^{b} 2^n \left(\left\lfloor \frac{i}{2^n} \right\rfloor mod\ 2 \right) \left(\left\lfloor \frac{i}{2^{n+256}} \right\rfloor mod\ 2 \right) \right\}_{i \in I}$$

This equation is a modification of the original work by the Polish mathematician that operates in a binary architecture, repeating and compounding itself every 8 bits – represented by the division by 2^{256}, and saturating at b bits – represented by the sum of the individual b bits. To illustrate the expected result, a visualization of S with $|S| = 16384$ and $b = 6$ follows:

Figure 11: Binary Sierpiński triangle



Furthermore, a second step was added to generate a set P which would increase the cycle count per element computation by appending the following calculation to the value of S:

$$P = \{s_i^{\ seed}\}_{i \in I}$$

The set would then be generated x times, so that the average time required to populate S could be calculated.

With the above algorithm in hand, the test was performed on a 2.4GHz Intel Xeon CPU with 8GB of RAM and 8 logical processors, and with a 1GHz NVidia GeForce GTX 680 GPU with 6GB of RAM. The pertinent variables to run the test were defined as $|S| = 2(16^7)$, $a = 64$, and $x = 25$. The results are a good representation of the nature of parallel computing on GPUs, where the average time to generate the sets are as follows:

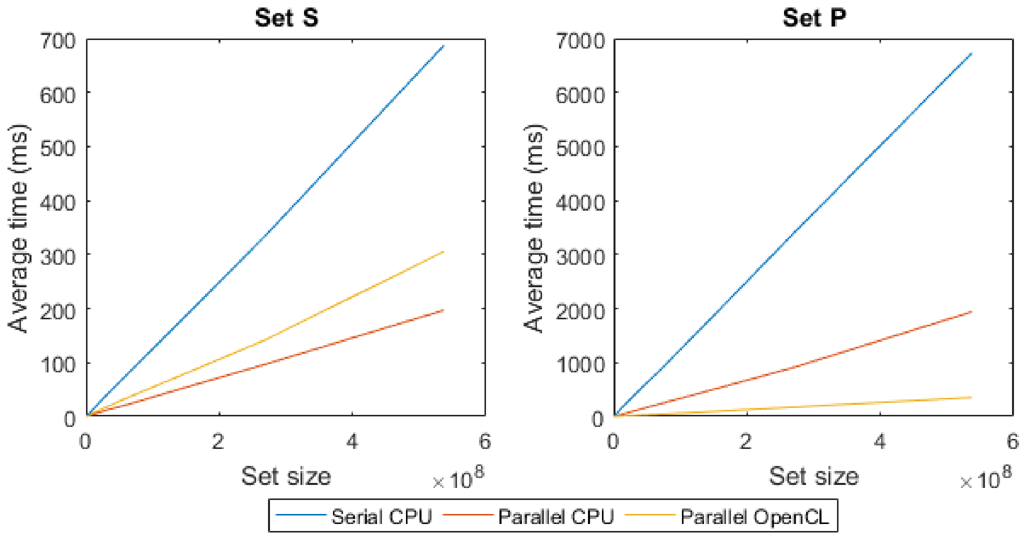
Table 2: Average time to generate set S

Method	*Average time (ms)*
Serial on CPU	687.11
Parallel on CPU	196.90
Parallel on GPU with OpenCL	305.53

Table 3: Average time to generate set P

Method	*Average time (ms)*
Serial on CPU	6,729.72
Parallel on CPU	1,938.32
Parallel on GPU with OpenCL	353.09

Figure 12: Average time to generate sets of different size



The results clearly show a consistent slower performance for the serial model in both sets; as well as the overhead involved in setting up and offloading processing to the GPU, reflected as a higher performance with CPU parallelized model when the cycle count per element computation does not compensate for the overhead. With that in mind, as long as the parallelized instruction block is complex enough, the overhead time will dissipate in relation to the overall time required to compute the whole instruction set. In the presented test case, the simple addition of an exponentiation was enough to overcome the underlying overhead.

3 Solution

Industrial plants share several geometrical similarities, such as the abundance of cylinders, boxes, and tori [18]. This can be expected to arise from the railings, piping, tanks, straight walls and ceilings, and valves, often present in industrial plants. As such, an automatic specialized algorithm could be anticipated [1]. However, the variance in the nature of the point clouds undermines a fully automated solution, and propositions that do not require human input often rely on a dense and artifact free point cloud [1] [2] [19]. Therefore, a compromise between automation and manual configuration would render the most general and useful solution.

Several open source software packages, such as MeshLab [20] and CloudCompare [21], provide a human supervised suite of tools, with pre-built algorithms for point cloud and mesh processing. However, MeshLab falls short in compatibility with LAS format, requiring a conversion prior to loading the data set. Furthermore, larger files, such as the ones this thesis focuses on, often result in crashes or require vast amounts of time in current common computers to perform most of the processing algorithms. CloudCompare, on the other hand, handles point clouds more efficiently, being built specifically for this matter; however, it too has an execution time which is larger than what this thesis is aiming to achieve, and does not present good robustness with regards to noisy point clouds. Both solutions, being built for general use, also fall short in ease-of-use when configuring the parameters for the processing algorithms. Consequently, a series of attempts with different parameters are needed to hone in the desired result, with the inconvenient fact that each try takes several minutes to compute.

As such, the proposed solution for the scope of this work is a lightweight software that can receive point clouds as LAS files and output a mesh representation of the data set, without presenting the shortfalls found on freely available open source packages. That is, the software should be able to handle large files without crashing or requiring too many resources – in terms of memory or time –, and perform well on common modern computers. It also means that it should be robust to noisy point clouds, outputting a reliable representation of the original dataset. Finally, the solution should be easy to configure and should output a small file size.

4 Journey

In order to achieve the desired results, this work was divided in three parts: the construction of a framework to handle point clouds and meshes, the comparison of different forms of point cloud treatment algorithms, and finally the generation of the mesh representation.

4.1 Framework

Considering the large size of LAS files, an efficient way to load them for handling and processing, consuming the minimal amount of memory and time as possible, is imperative. As such, C++ was the language of choice for its low-level capabilities and small footprint, stemming from the "you don't pay for what you don't use" philosophy behind the C++ community. Furthermore, OpenCL – a C-based API – can be tapped into with just a single wrapper header file, a characteristic that will be useful on the next stage of the process.

With that in mind, a `struct` for the data format was defined with no data padding. This allowed for an in place reading directly from the binary file into the software's local memory by using the built-in `reinterpret_cast` operation. More specifically, this method allows the memory from the LAS file to be directly mapped to the working memory of the application, with no need to interpret or parse its contents, as opposed to reading the file and generating each data packet individually depending on its contents. By using a direct read, the loading process was 38.7% faster than using a value conversion method as per Table 4, which describes the average loading time of a 4.6GB point cloud containing 181,372,708 points on an Intel Xeon 2.4GHz and 6GB of RAM.

Table 4: Loading time for LAS

Method	*Average time (s)*
Value conversion reading	99.40
In place reading	71.67

However, because of the direct mapping of memory through `reinterpret_cast`, the `struct` could not take advantage of inheritance to represent the different format described by the 1.4 LAS specification, which share a good portion of their fields. This is due to the fact that, when inheriting, the `struct` would require and extra field pointing to its base `struct`, and directly mapping would fail to populate the field properly. That meant that each different format was a separate `struct` and, consequently, all function calls that handled the point cloud had to be templated to match the format that was loaded.

This has advantages and drawbacks. The main advantage is that, by not using inheritance, there is no virtual table lookups to resolve the hierarchy, speeding up the execution and allowing for static compiler optimization. However, because of the templated functions, the compilation time is greater and the binary output is larger, since for each separate LAS format, a new version of every templated function had to be generated at compile-time.

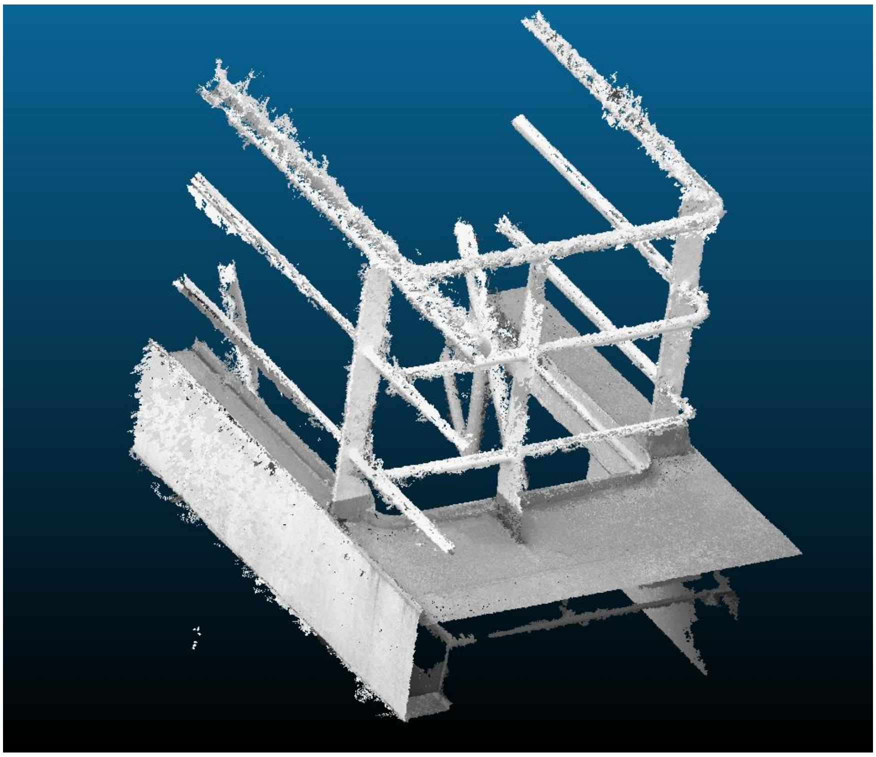
4.2 Point cloud treatment

Once the framework was in place, different methods on how to treat the point cloud in order to reduce its artifacts were compared.

4.2.1 WLOP

As a method to clean the point cloud, reduce its file size, and increase the uniformity of the data set, while still maintaining a reliable point cloud representation, the WLOP algorithm was first tried. The motivation behind the use of this technique can be traced back to State of the Art publication by Berger et al [3]. Given the nature of the point cloud, specifically the non-uniformity of the sampling, along with the noise, and outliers, Berger classified the LOP as a strong candidate for the treatment of the point cloud. With the improvements proposed by Huang et al [6] over the original work [5], an implementation from the CGAL library [22] was used to validate its applicability in the working set. For that matter, a subsection of the aforementioned point cloud was used to validate the method, shown in Figure 13, containing 8,728,342 points with a file size of 290MB.

Figure 13: Subset of point cloud representing a railing section



WLOP has three main parameters: radius, iterations, and reduction.

- Radius stablishes a radius of neighbors around a point. This parameter is very important as it results in irregular outputs if too small, or high impact on runtime if too large. The desired value usually means a radius that includes two neighboring radii. Moreover, this value can be automatically calculated by the algorithm by measuring the average spacing between points.
- Iterations refers to how many iterations will be run as the points converge. If the value is too high, there will be extra computation that does not contribute in a significant manner to the uniformization of the set, whereas a low number might not be enough to reach the optimum distribution.
- Reduction specifies the number of points that should be kept from the original set. It is expressed as a percentage of the original cardinality.

With that in mind, details on different tries and combinations can be noticed on the following figures.

Figure 14 is the original point cloud with no color information for a better comparison. Figure 15 presents the results of WLOP reducing to 2% of the original size, over 35 iterations, on a radius of five. The output file was 3.4MB, and the computation took 35 hours. Figure 16 presents the results of WLOP running an automatically detected radius (calculated to be 0.05), over 2 iterations, and 10% reduction. The output was 17MB, and the computation took one minute and ten seconds. Similarly, Figure 17 shows a very similar result with the same configuration, except for executing 35 iterations. The computation took eleven minutes and two seconds to finish. Finally, Figure 18 presents the most promising results that could be extracted. It is the result of reducing the point cloud to 10% of its original size over 10 iterations with a radius of one.

Figure 14: Original point cloud without color

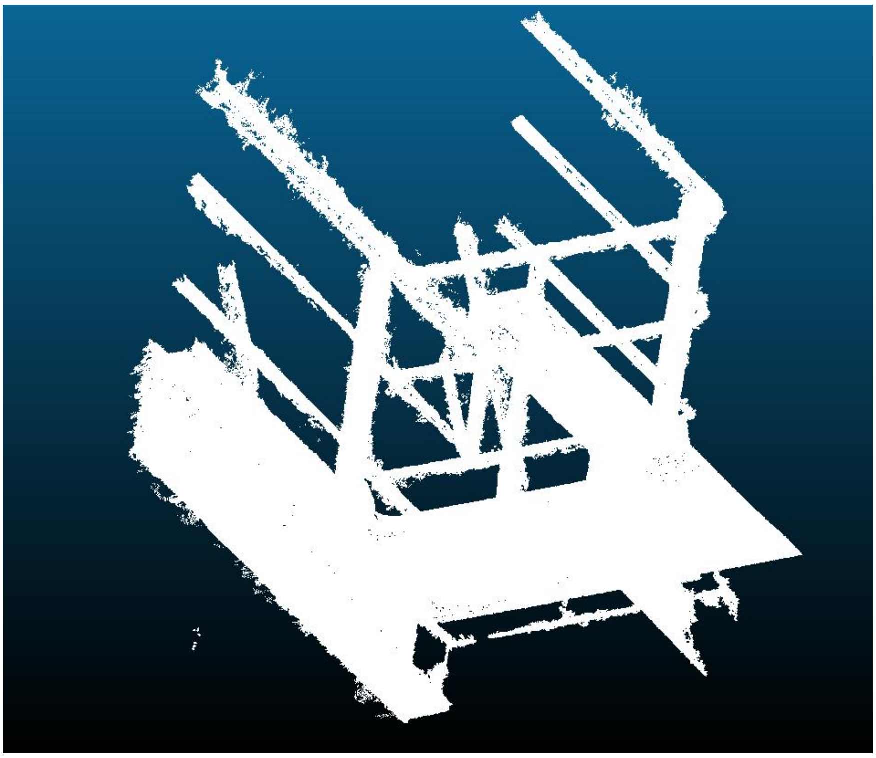


Figure 15: WLOP 2% over 35 iterations on radius of five

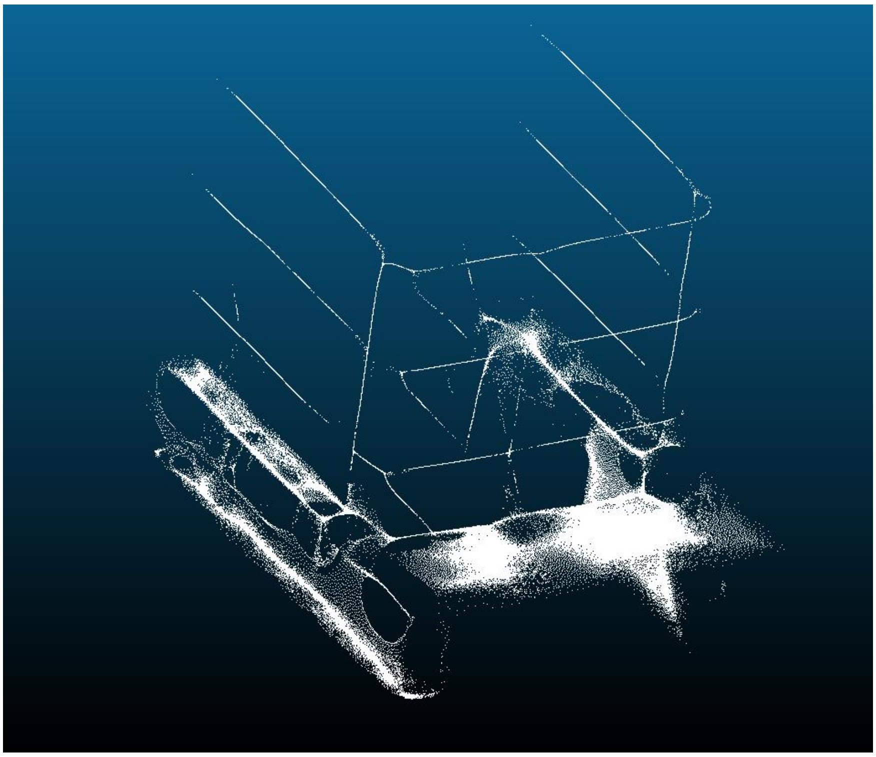


Figure 16: WLOP 10% over two iterations on a radius of 0.05

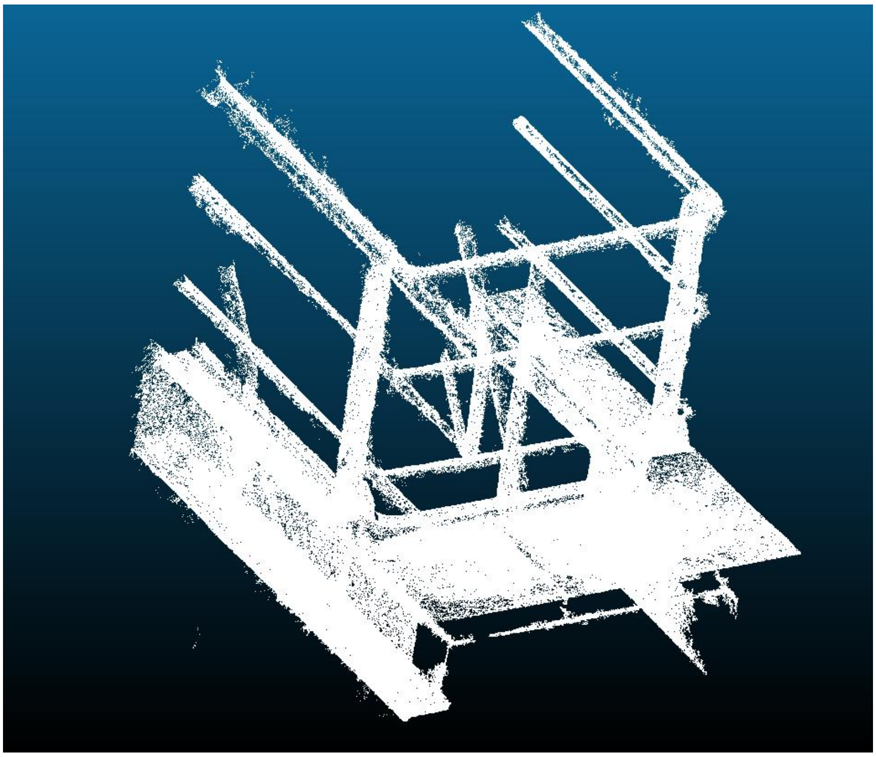


Figure 17: WLOP 10% over 35 iterations with a radius of 0.05

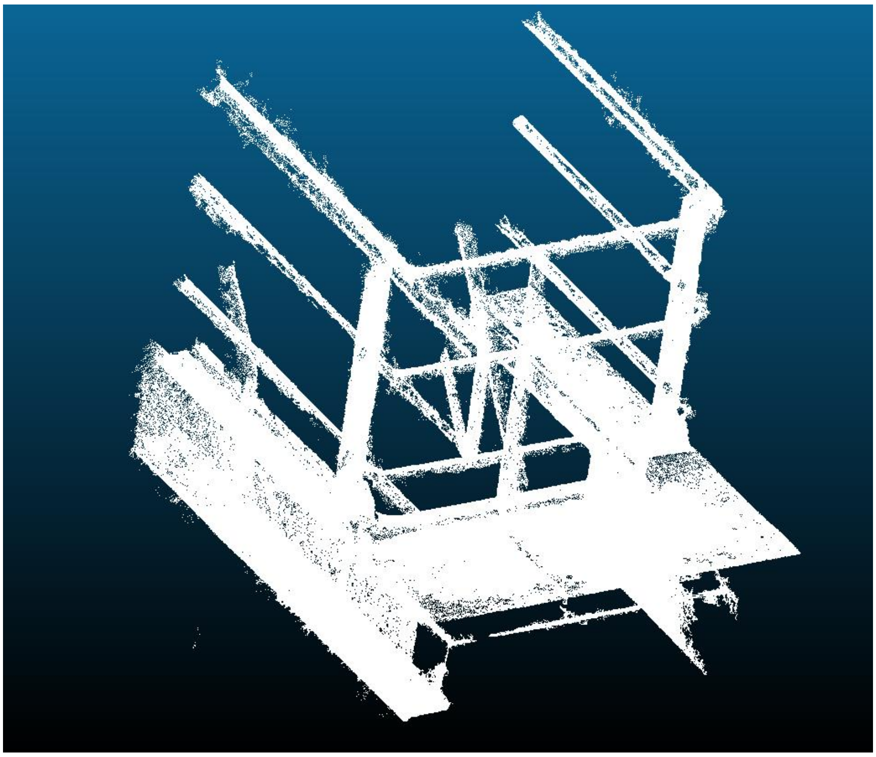
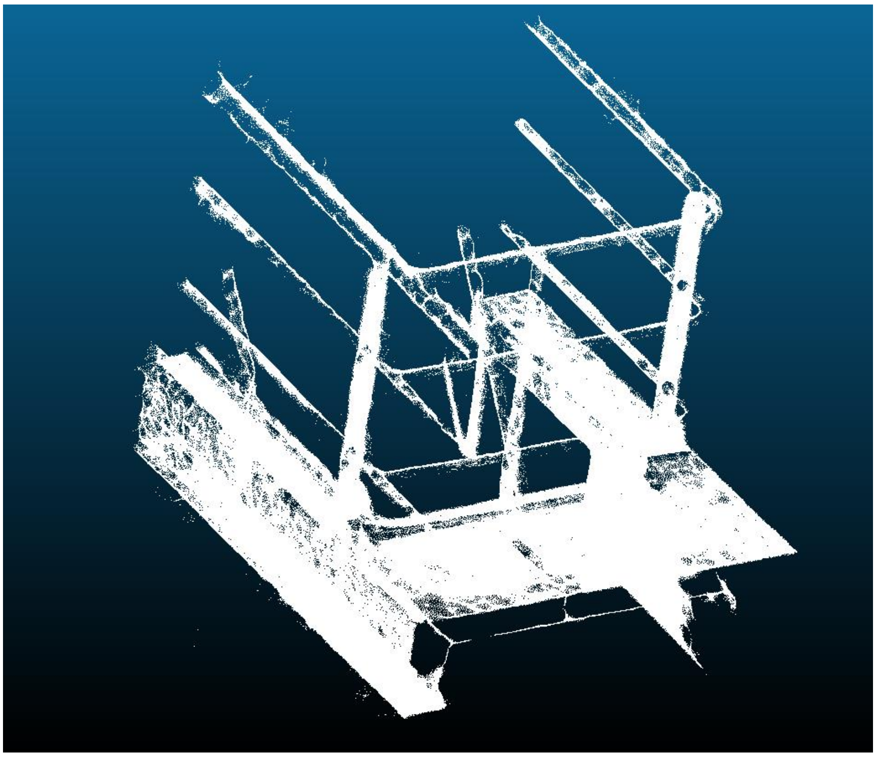


Figure 18: WLOP 10% over 10 iterations with a radius of one



It is worth noting how the radius is the largest contributor to how well the algorithm will perform, both in time as in output set. For instance, Figure 17 has over three times more iterations than Figure 16 with no significant gains. On the other hand, a big increase in the radius severely hampered the quality of the output, as observed in Figure 15 on page 27 with strings for railing and holes on the sidewalls. This is due to the radius being much larger than the diameter of the railing, compressing all points with the region into a single string.

4.2.2 MLS Smoothing

The second point cloud treatment to be tested was the moving least squares smoothing [7] [8] [9] [23], which can be used to do higher polynomial interpolation of the surrounding points. This method is meant to aid in the treatment of outliers, noise, and holes, as discussed in 2.4. The motivation behind using this method was its strength against outliers and noise, characteristics that are abundant in our data sets. Furthermore, it has been implemented into open source software, making it easier and faster to test, validate, and compare.

Its main parameters, among others, are the Gaussian standard deviation, which is discussed by Tekumalla et al [9] on section 4, and the search radius, which defines the local zone of influence. However, both parameters are related; as you increase the search radius, the standard deviation should also increase to account for the larger set of points. Therefore, they boil down to one main parameter: the search radius.

By using Cloud Compare's [21] implementation of the algorithm, the railing model was subjected to the MLS smoothing using different parameters. Using the same railing model of 8,728,342 points as before, the first setting used a search radius of 0.01 and 0.0001 for the Gaussian parameter, and completed in ~110 seconds with the output illustrated by Figure 19 below. As expected, increasing the deviation by one order of magnitude without increasing the search radius produced a very similar result with comparable compute time of ~105 seconds. In fact, the results are so similar that they are indistinguishable when presented as figures in this document; therefore, it was omitted.

However, in order to effectively extract the underlying surface function, a larger search radius is needed. Figure 20 shows the output of a search radius increased by one order of magnitude and 0.0001 for the Gauss parameter. This computation took significantly more time and lasted for 149 minutes. Similarly, using the same increased search radius, but also increasing the Gauss parameter for standard deviation by one order of magnitude produced the same results as before: similar outputs, similar computing times.

It is interesting to note that the weighting of the moving squares has little effect in our data set. As mentioned by Takumalla, this parameter has more of an impact on high frequency changes to the surface curvature. However, our data set suffers greatly from holes, which cannot be compensated for by applying a broader weighting function.

Using a larger search radius, on the other hand, gave use noticeable changes, as can be noticed by Figure 21 and Figure 22. The larger vicinity was even able to close the railing and restore a cylindrical surface shape.

Lastly, is worth mentioning that this method does not reduce the file size by itself. It could however be coupled with a down sampling algorithm to achieve this result.

Figure 19: MLS with 0.01 search radius and 0.0001 for Gauss parameter

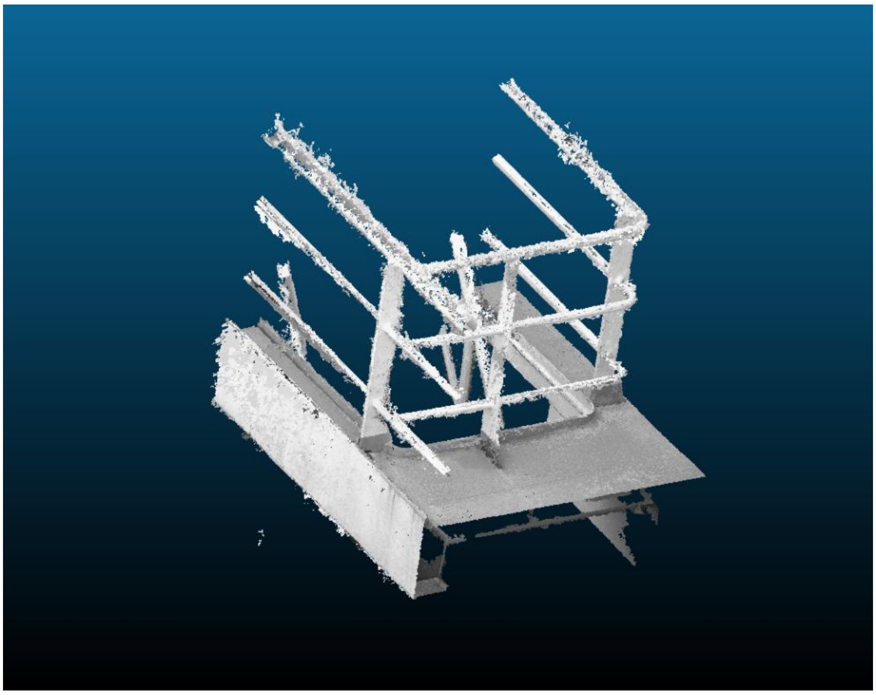


Figure 20: MLS with 0.1 search radius and 0.0001 for Gauss parameter

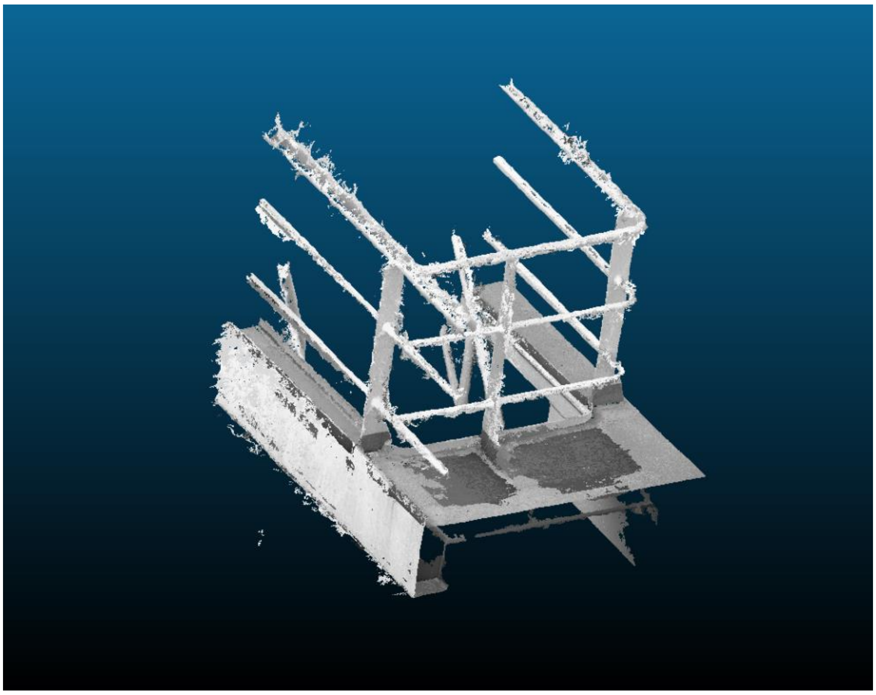


Figure 21: Railing detail from original point cloud

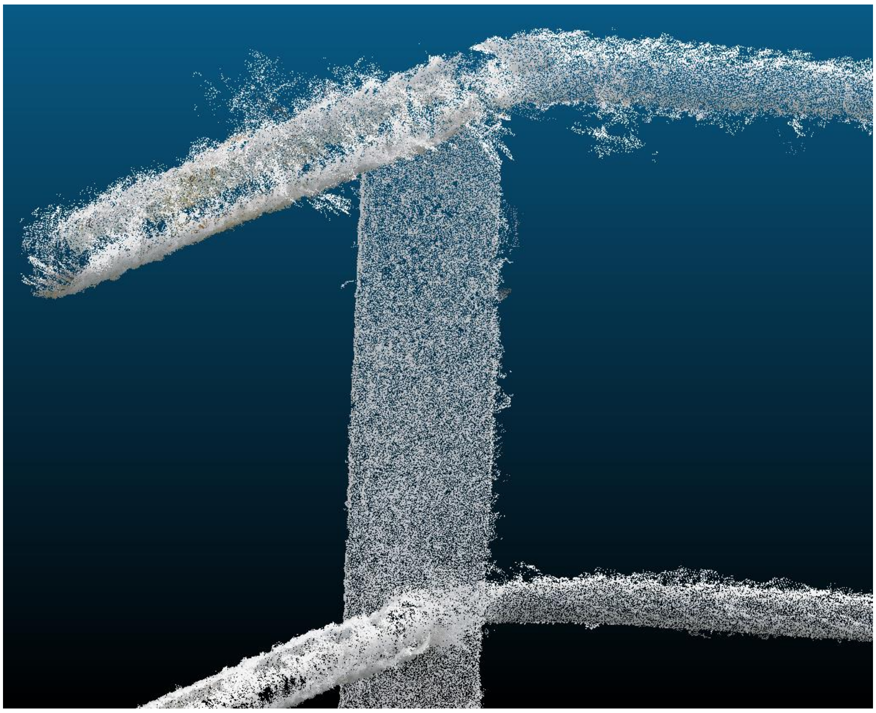
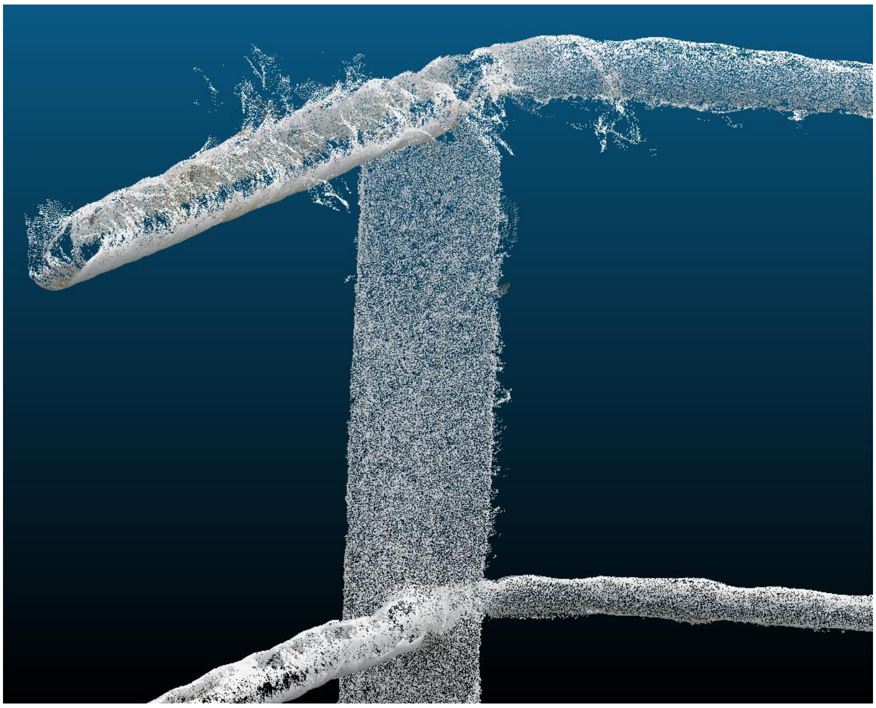


Figure 22: Railing detail after MLS smoothing



4.2.3 Naïve point cloud simplification

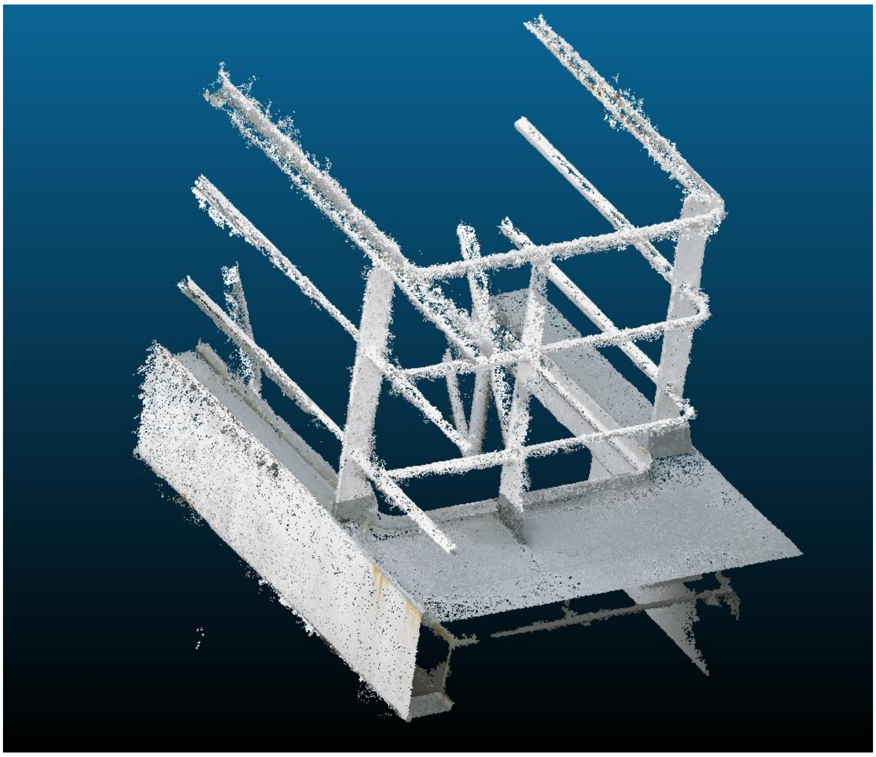
In order to simplify the point cloud without performing uniformization of the data, a naïve method was tested at this point. The main drive behind this approach is to reduce the file size so that it can be easily handled, while still working with a point cloud representation of the model. The process consists of shuffling the point cloud data set S and picking K elements, where $K \leq |S|$. This will randomly select a subset of the original data while preserving all information regarding the selected points. Note that this method does not perform any attempts to correct noise, outliers, or gaps in the point cloud. A C++ snippet can be seen in Code 1.Figure 23, where the file was reduced from 290MB to 79.4MB while maintaining all channels from the original file, such as color, number of returns, or any other present initially.

Code 1: Custom point cloud simplification

The algorithm picks a fraction of the points randomly and creates a new point cloud with the selected subset

```cpp
template<int N>
void simplify(const LASFile<N> & lasFile, const double factor) {

// Create a new file
LASFile<N> newFile(lasFile.filePath));

// Calculate the new size
uint64_t newSize = static_cast<uint64_t>(
  lasFile.pointDataCount() * factor / 100.0);

// Gather the indices and shuffle them
std::vector<uint64_t> indices;
indices.reserve(lasFile.pointDataCount());
for (uint64_t i = 0; i < lasFile.pointDataCount(); i++) {
  indices.push_back(i);
}
std::random_shuffle(indices.begin(), indices.end());

// Pick the first K indices
indices.resize(newSize);
newFile.pointData.reserve(newSize);

// Push the selected points
for (auto & index : indices) {
  newFile.pointData.push_back(lasFile.pointData[index]);
}

newFile.save();
}
```

Figure 23: Railing after naïve simplification



4.3 Mesh generation

With different techniques of point cloud treatment in hand; different mesh generation algorithms were tested with the output files. The goal is to identify the best pipeline now that all the pieces are in place. For that matter, different meshing algorithms were tested.

4.3.1 Marching cubes

The marching cubes algorithm [12] [13] [14] [24] was chosen for its simplicity and ubiquity. The motivation behind choosing an algorithm for its simplicity lies on the fact that this would be a stepping-stone into the scope of transforming the representation, therefore, a tried and tested simple approach would be quick to implement and verify.

As an actual implementation of the marching cubes algorithm, Lewiner et al [14] proposed an efficient implementation using an extended look-up table. Furthermore, the source code for CPU execution was made available by the authors, making it quicker to test and verify. However, the algorithm assumes the data is a scalar map, that is, a three-dimensional grid of cubes, with values associated to each cell. From this scalar map, an isosurface can be extracted, that is, a surface where all points have the same value. Therefore, the first step is to generate such a grid.

4.3.1.1 Grid generation

The strategy applied for the grid generation was to create a density scalar map of the point cloud. In other words, the point cloud would be divided into a regular grid of three-dimensional cells, and the number of points that were contained inside a given cell would give its density value. The reason for such generation approach was the simplicity of the concept and the ability to perform the generation in a highly parallel fashion. This strategy later also proved to be robust to outliers and noise.

By performing the creation of the grid in the graphics card, the process can leverage the high parallelization capabilities and take advantage of atomic operations to increment the density of a cube as points in the point cloud are visited. Code 2 shows how each individual point is visited and its location calculated. Because of the parallel nature of this computation, **`atomic_inc`** is called to increment the density of the respective cube in a lock-free fashion.

Code 2: Kernel for creating cuverille grid

The algorithm places a data point into a cell according to its spatial location

```
kernel
__attribute__((vec_type_hint(float3)))
void createGrid(const global uint3 * points,
global volatile uint * output) {
float3 point = convert_float3(points[get_global_id(0)]);

ushort3 grid = convert_ushort3((point - CONST_OFFSET) /
CONST_STEP);

if (grid.x == CONST_SIZE_X) grid.x--;
if (grid.y == CONST_SIZE_Y) grid.y--;
if (grid.z == CONST_SIZE_Z) grid.z--;

atomic_inc(&output[grid.z
          + grid.y * CONST_SIZE_Z
          + grid.x * CONST_SIZE_Y * CONST_SIZE_Z]);
}
```

For this code to be reliable, however, the global memory **`output`** should be placed in the same device that the code is running since the standard does not guarantee atomic operations across multiple devices as per Appendix A of the specification [25].

Furthermore, care should be taken when loading the memory as to avoid allocating more than the allowable amount given by **`CL_DEVICE_MAX_MEM_ALLOC_SIZE`**, defined in the specification as either a fourth of the total global memory or 134KB; whichever is larger. Depending on vendors, if this limit is not respected, the code will fail, which is the case for AMD cards. However, other vendors allow this value to be violated without failing in detriment of performance, such as for Nvidia cards. In the case of this work, the point cloud can be loaded in chunks that do not surpass the limit until the whole dataset is processed.

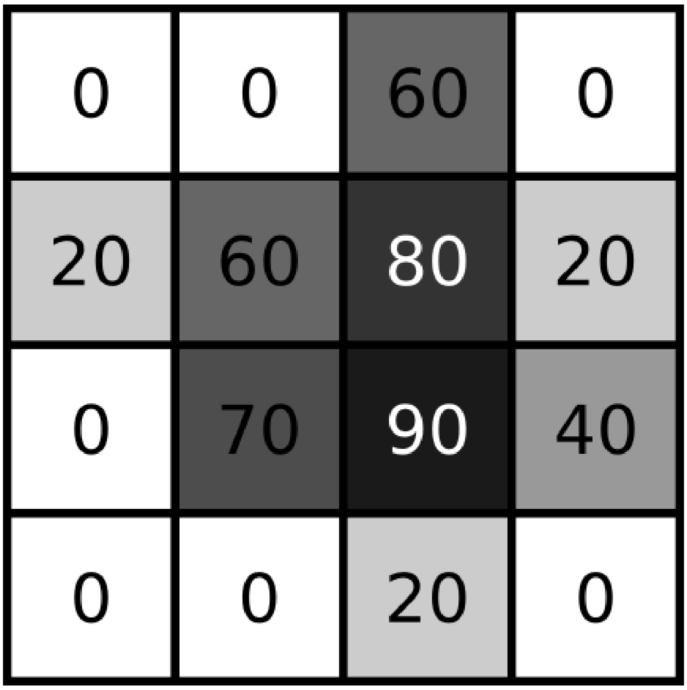
4.3.1.2 Vertices generation

Once the scalar field is generated, the algorithm then creates all the possible vertices that might be used while generating the triangles. This phase allows for further steps to reference pre-existing vertices, thus avoiding duplicated vertices, redundant vertex placement calculation, and minimizing output file size. The latter is due to the representation format for the mesh, where triangles are defined not by three vertices, but by three indices to a list of vertices. This technique is commonplace in modern graphics and available in different mesh representation formats, such as OBJ [26] and PLY [27]. In the case of OpenGL, such functionality is used when calling **`glDrawElements`**.

This step is achieved by taking the generated grid and connecting adjacent voxels with edges. Then, for each edge, if the connected voxels are on opposite sides of the threshold, a vertex will be created. Finally, the placement of the vertex along the edge is given by the linear interpolation of the two values.

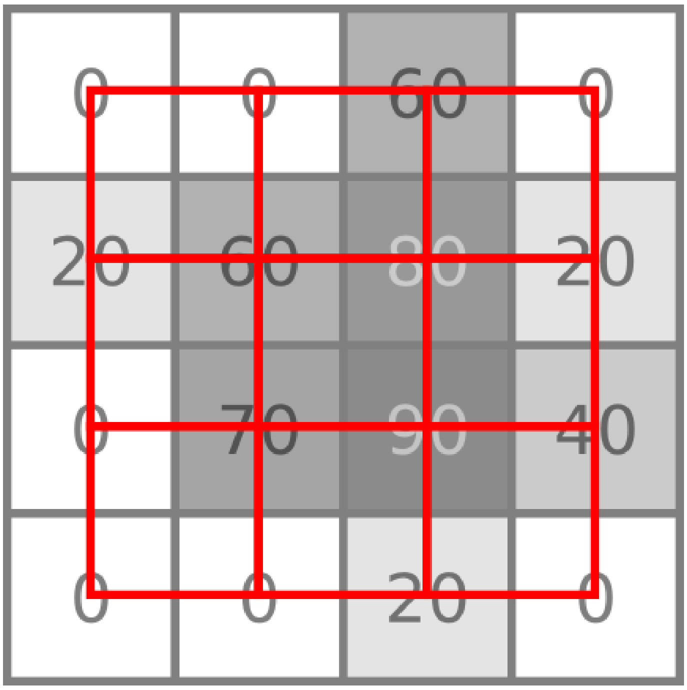
To illustrate the process, a bi-dimensional example will be used, to avoid cluttering and facilitate the explanation. Given a 4 by 4 grid, seen on Figure 24, where each pixel represents the density of points contained within that region, edges connecting the pixels will be created as on Figure 25. Note that the edges are not actually allocated in memory, only vertices are created in this phase. The use of edges is to better illustrate the process.

Figure 24: Bi-dimensional grid representing point densities



0	0	60	0
20	60	80	20
0	70	90	40
0	0	20	0

Figure 25: Grid with connecting edges

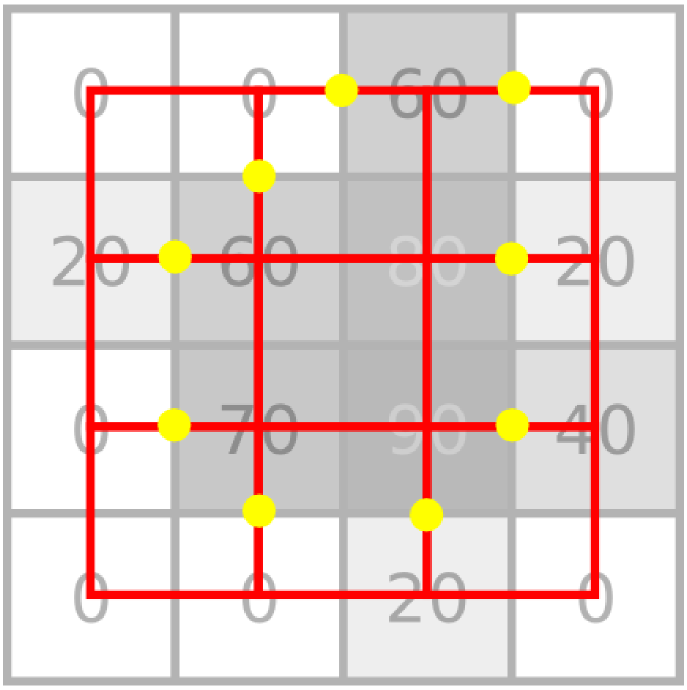


0	0	60	0
20	60	80	20
0	70	90	40
0	0	20	0

With the virtual edges in place, vertices will be created for each edge that connects pixels whose values are on either side of the threshold. Taking 50 as an example for a threshold, Figure 26 shows which vertices would be created in this example.

Finally, Figure 27 shows the vertices in their actual positions taking into account the values of the connected pixels and interpolating them to find where the threshold would lie. Figure 28 shows the final result.

Figure 26: Edges with created vertices



0	0	60	0
20	60	80	20
0	70	90	40
0	0	20	0

Figure 27: Vertices positions set by interpolation

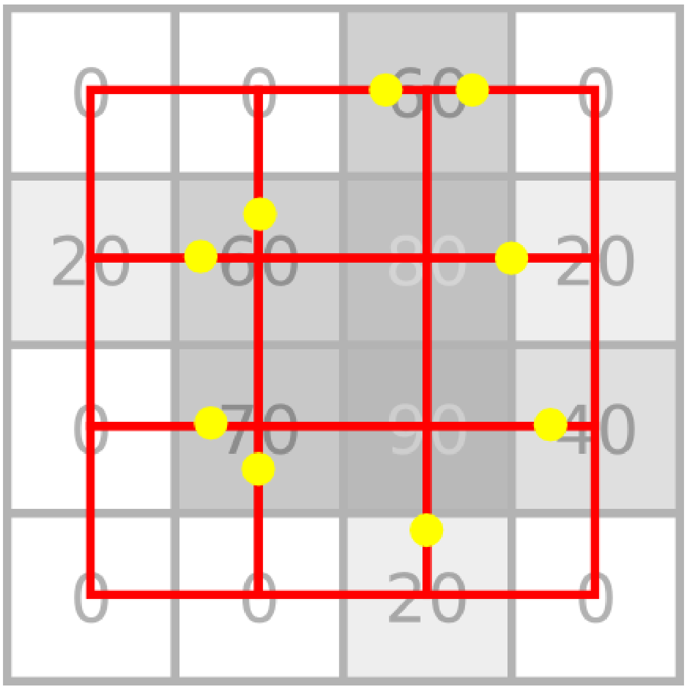
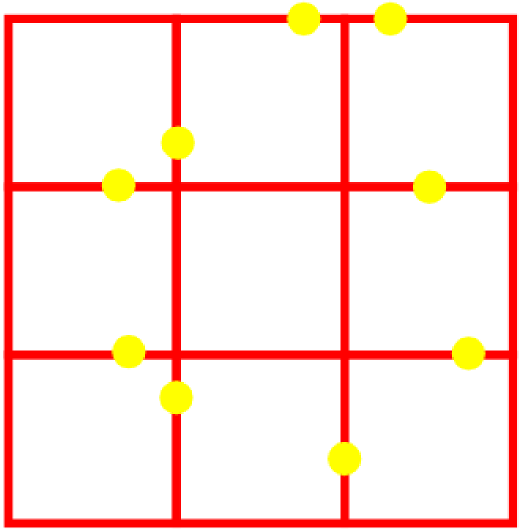


Figure 28: Resulting vertices



To illustrate the importance of this step in terms of memory consumption, let us take a bi-dimensional grid with V_{2D} pixels, where:

$$V_{2D} = Width \times Height$$

This grid will generate E_{2D} edges, where, if we consider:

$$A = Width - 1$$
$$B = Height - 1$$

We are using these values as base because we are connecting the grid with edges, as seen on Figure 25 above, and therefore reducing the dimensions by one. We then have:

$$E_{2D}(A, B) = 2(AB) + A + B$$

This can also be intuitively noticed since, for each cell on Figure 25, two edges are created, the bottom and left ones, giving $2(AB)$. This however leaves the top and right side of the grid open. Therefore, for the rightmost column, B edges have to be added to close the right side; and for the topmost row, A edges have to be added to close the top side of the grid. Hence $2(AB) + A + B$.

Similarly, in the tridimensional case, the potential number of vertices created is given by:

$$V_{3D} = Width\ \times Height \times Depth$$
$$A = Width - 1$$
$$B = Height - 1$$
$$C = Depth - 1$$
$$E_{3D}(A, B, C) = E_{2D}(A, B) + 3ABC + 2AC + 2BC + C$$
$$E_{3D}(A, B, C) = 3ABC + 2AB + 2AC + 2BC + A + B + C$$

Therefore, it is easy to notice that, in the worst case, the number of vertices created will be in the order of three times the number of voxels, using this generation phase. If vertices were to be created independently, this number could be in the order of 12 times the number of voxels.

This memory consumption also starts to degrade the performance of the GPU, since memory has to be moved in and out of the GPU only to perform trivial operations, such as comparison and interpolation.

4.3.1.3 Classification

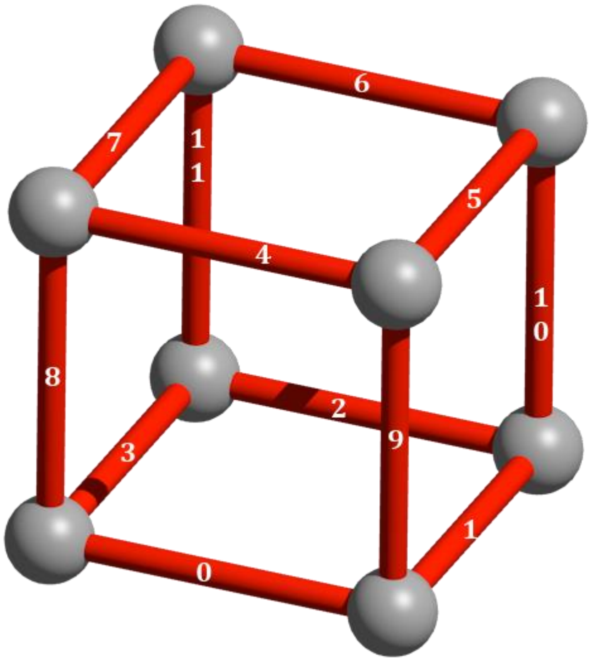
Once the vertices are generated, they are put aside and the algorithm then classifies each cell, where a cell is given by eight voxels representing the eight vertices of the cell. This process is straightforward and is exactly as described in section 2.5.

4.3.1.4 Triangle generation

The classification of the cell can then be used to get the rules for generating triangles inside the cell. However, there are ambiguity cases that need to be resolved in order to maintain the correct topology.

If no ambiguities were to be resolved, a lookup table containing 256 entries would dictate, according to the classification, which triangles to create by referencing which vertex indices. For instance, case 18 would cause the algorithm to lookup the 18th entry in the lookup table, which in turn would return the numbers $\{0,1,9,8,4,7\}$. This means that two triangles would be generated by connecting the vertices lying on edges $\{0,1,9\}$ and $\{8,4,7\}$. As reference, the edge numbers are labeled as in Figure 29.

Figure 29: Edge numbering for marching cubes

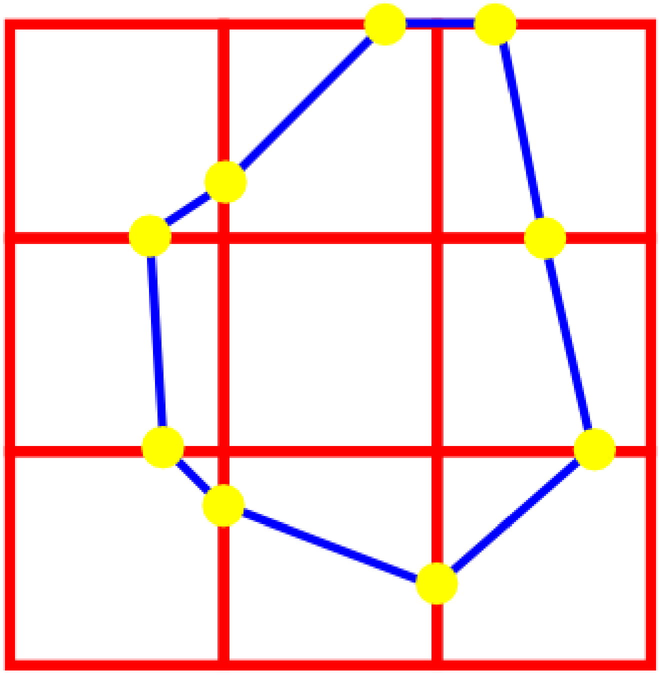


However, Lewiner [14] has proposed an implementation that uses ambiguity resolution, where certain cases would trigger tests to resolve which triangle should be created. It does not only resolve face ambiguities, but also solves internal ambiguities, and therefore is able to guarantee topological correctness.

To do so, the 256-entry lookup table maps which subcase and configuration to consider, and hence, which additional operations should be performed, including possible tests to further classify the case before generating the triangle. This lookup table is defined in a 2000-line header file.

For the example presented in section 4.3.1.2, this phase would generate the shape in blue seen on Figure 30.

Figure 30: Resulting shape



4.3.1.5 Output

Figure 31 shows the output of the marching cubes with a grid of 128×128×128 cells. From the original file size of 290MB, the marching cubes mesh had a file size of 4.2MB in 0.15 seconds.

Figure 32 show the same file marched in a 256×256×256 grid. The output file size was 20.3MB and was generated in 1.8 seconds.

Figure 31: Marching cubes output for 128^3

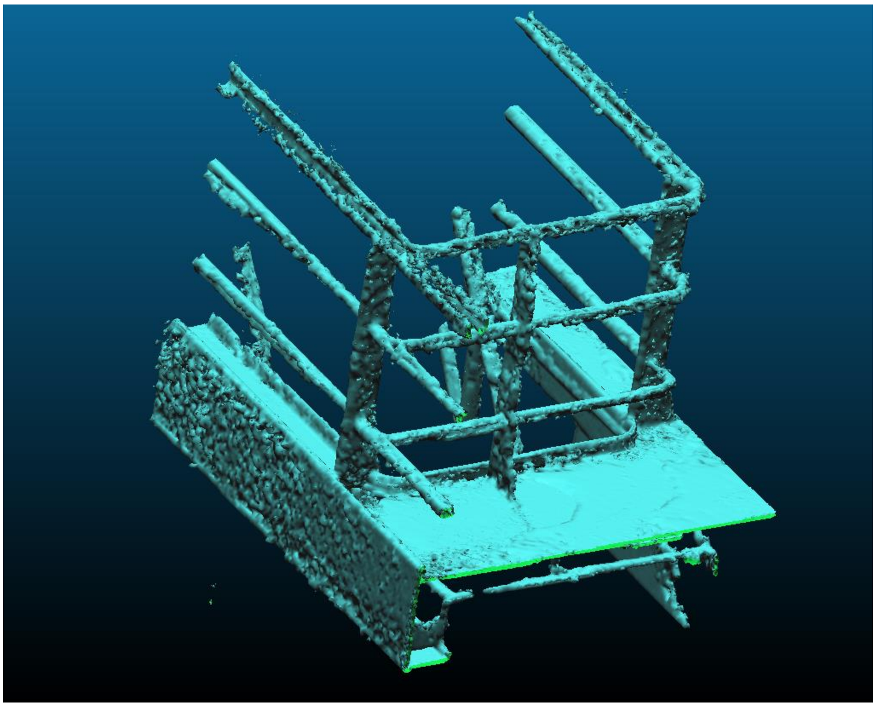
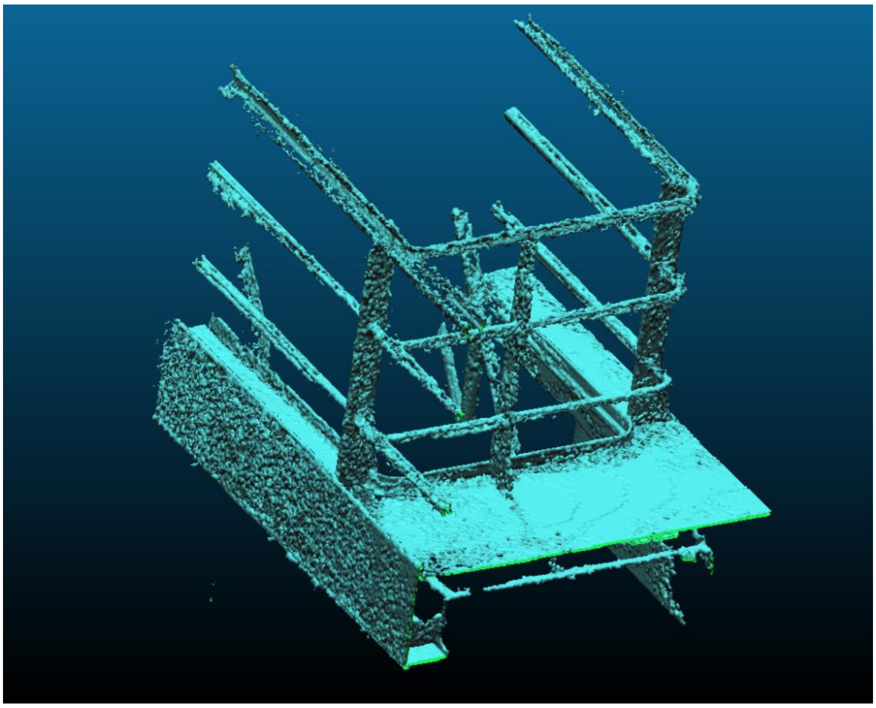


Figure 32: Marching cubes output for 256^3



4.3.2 Poisson Surface Reconstruction

The other surface reconstruction method tested was the Poisson surface reconstruction [15] [16]. This method renders results that are accurate and able to be resilient to imperfect data, outliers, and high frequencies in the surface. In addition, there is an available open source version implemented as part of Cloud Compare [21], making it easy to test and validate for this work.

However, as mentioned in section 2.6, this technique relies on normal information from the point cloud to extract the surface, and this information is not present in the data sets for the scope if this work. With that in mind, an additional step for normal calculation had to be taken beforehand. Fortunately, this step is also available as part of Cloud Compare.

The main parameter for the Poisson is the octree depth. This parameter changes how the discretization is performed, i.e. how deep the tree should be to represent the dataset. Since most of the model's volume will contain empty space, most of the precision is need around the surface and octree would be an efficient model to represent that. It is worth noting, though, that the size of the dataset grows exponentially with the depth of the tree, that is, the dataset will contain 8^n points, where n is the depth of the tree.

Figure 33 shows the output of the Poisson reconstructed surface with an octree depth of 8. The computation time was a total of 9.7 minutes, of which 8.75 minutes were spent calculating the normals, and 55 seconds were computing the Poisson surface. This particular output had a size of 9.5 MB.

Similarly, Figure 34 shows the same model, but with an octree depth of 10. The computation took 11.7 minutes broken down into 8.8 for normal computation and 2.9 minutes for surface extraction. The output file size was 185MB.

Figure 33: Poisson reconstruction with octree depth of 8

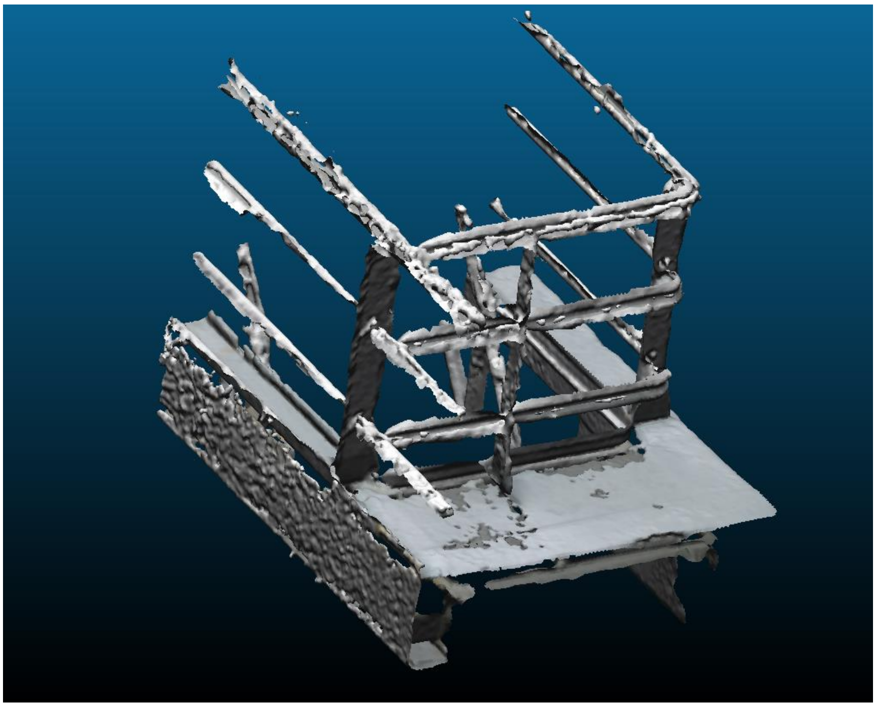
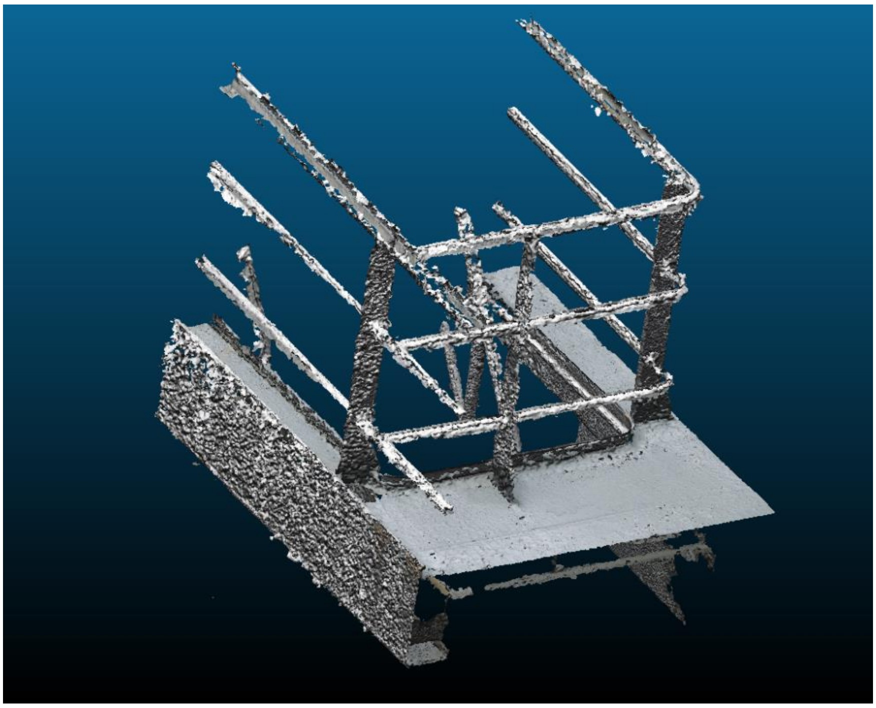


Figure 34: Poisson reconstruction with octree depth of 10



5 Results

In order to hone into the best pipeline to achieve the goals of this thesis based on the techniques assessed, the different algorithms that were implemented and the tests performed were analyzed to find such pipeline. As such, the results of the test will be independently analyzed and discussed. Later, the different methods will be combined with each other producing eight different pipelines that will then be analyzed and discussed.

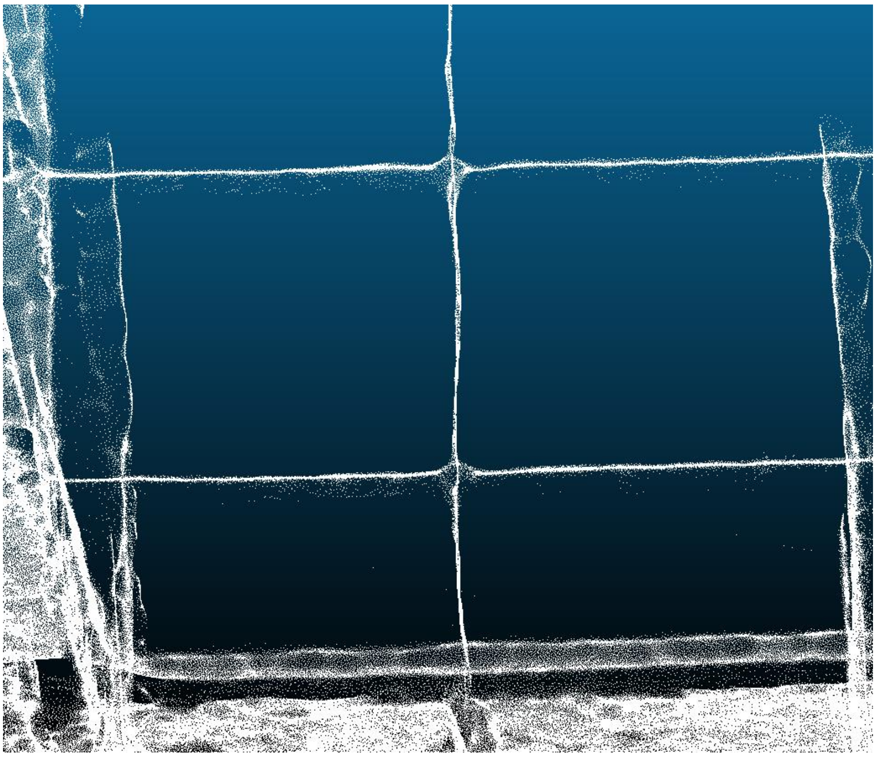
For each case, a comparison of the output was performed against the original point cloud, and a heat map was generated according to the distance between the two sets. In the case of meshes being compared, 3 million points were uniformly placed on the surface in order to perform the comparison. Furthermore, the scales are normalized to each case, so to not saturate the heat map. However, a detailed view of the histogram with the distribution is presented alongside the image to allow for an absolute comparison between the cases.

5.1 Point Cloud Treatment

5.1.1 WLOP

Overall, the results obtained from running WLOP were less than satisfactory, despite the promising compatibility proposed by Berger et al [3]. The most promising output – shown in Figure 18 on page 28 –, with a very well balanced radius size, failed to create uniform flat surfaces while still maintaining the shape of the railings, as seen in Figure 35. The main reasons for the dismissal of this strategy are the loss of color information and the tendency to collapse the railing into a string of points. The latter is a consequence of the radius of influence which, in order to be effective, should be large enough to encompass outliers in addition to the actual surface; however, as the radius grows, it becomes larger than the diameter of the railing, thus collapsing it. Moreover, though color can be restored based on the original file, significant improvement was not achieved with this method to justify the effort or its use.

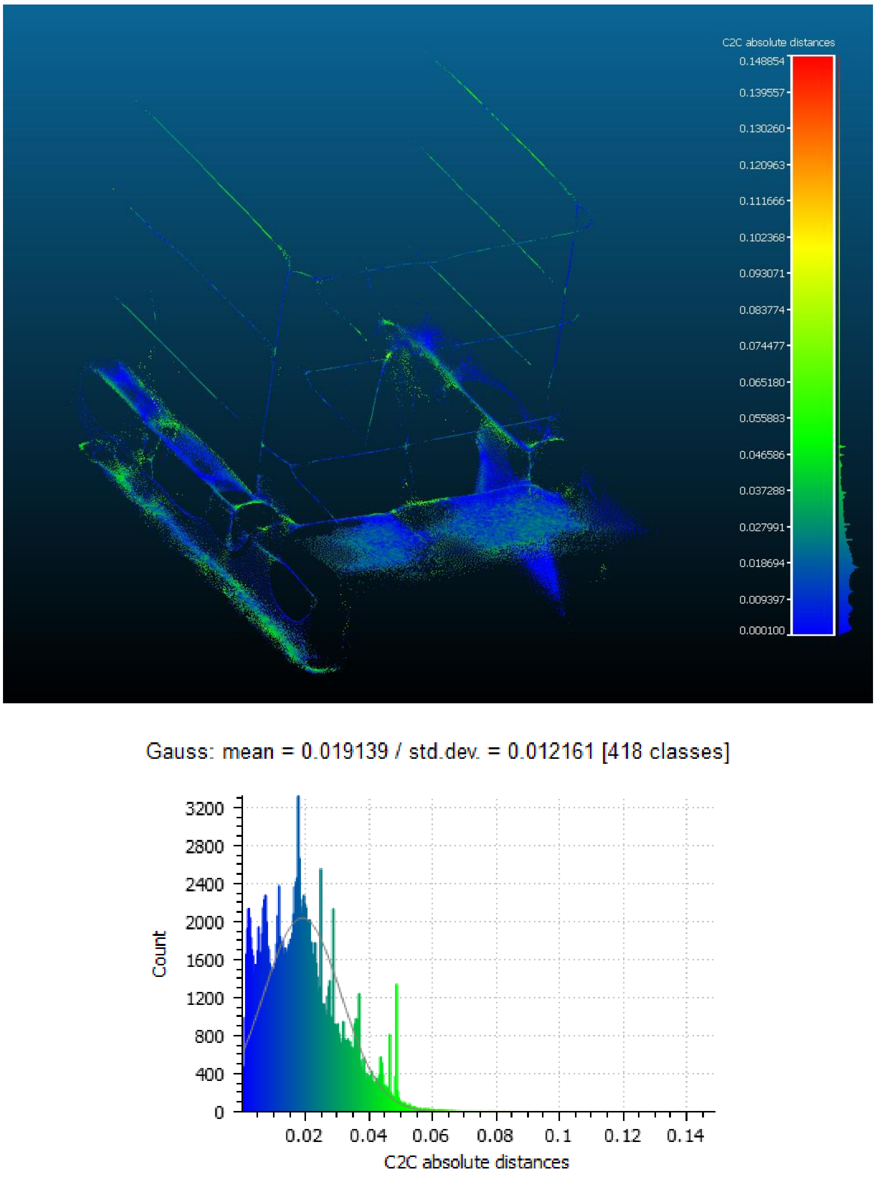
Figure 35: Detail of railing output from WLOP



5.1.1.1 Radius of 5

It is clear that when the radius is too large the points were compressed, rendering the railing unrecognizable. This is reflected as the largest absolute distances compared to the other cases with an average of 0.19, as seen in the heat map on Figure 36. Finally, as seen on Figure 39, the computation time turned out unfeasible for this project, taking multiple hours for a single iteration.

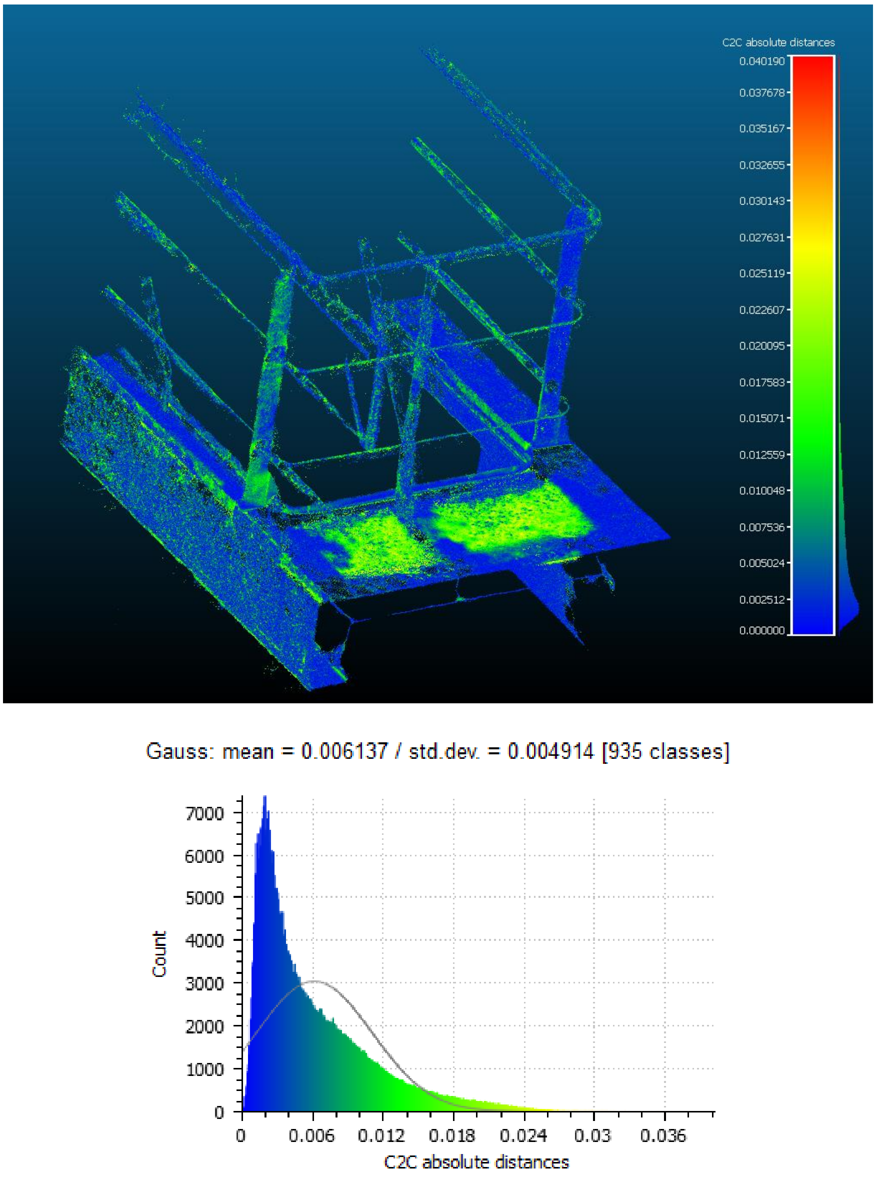
Figure 36: Heat map for WLOP with 2% reduction over 35 iterations on radius of five



5.1.1.2 Radius of 1

The best results using WLOP came from using a radius of one. However, the number of iterations was reduced to 10 to decrease computation time, after it was noticed that there was no detectable impact on the quality of the output. This configuration also showed high fidelity to the original model with an average distance of 0.006. It also proved able to successfully manage both high curvature, such as the railings, and flat surface, such as the floor.

Figure 37: Heat map for WLOP with 10% reduction over 10 iterations on radius of one

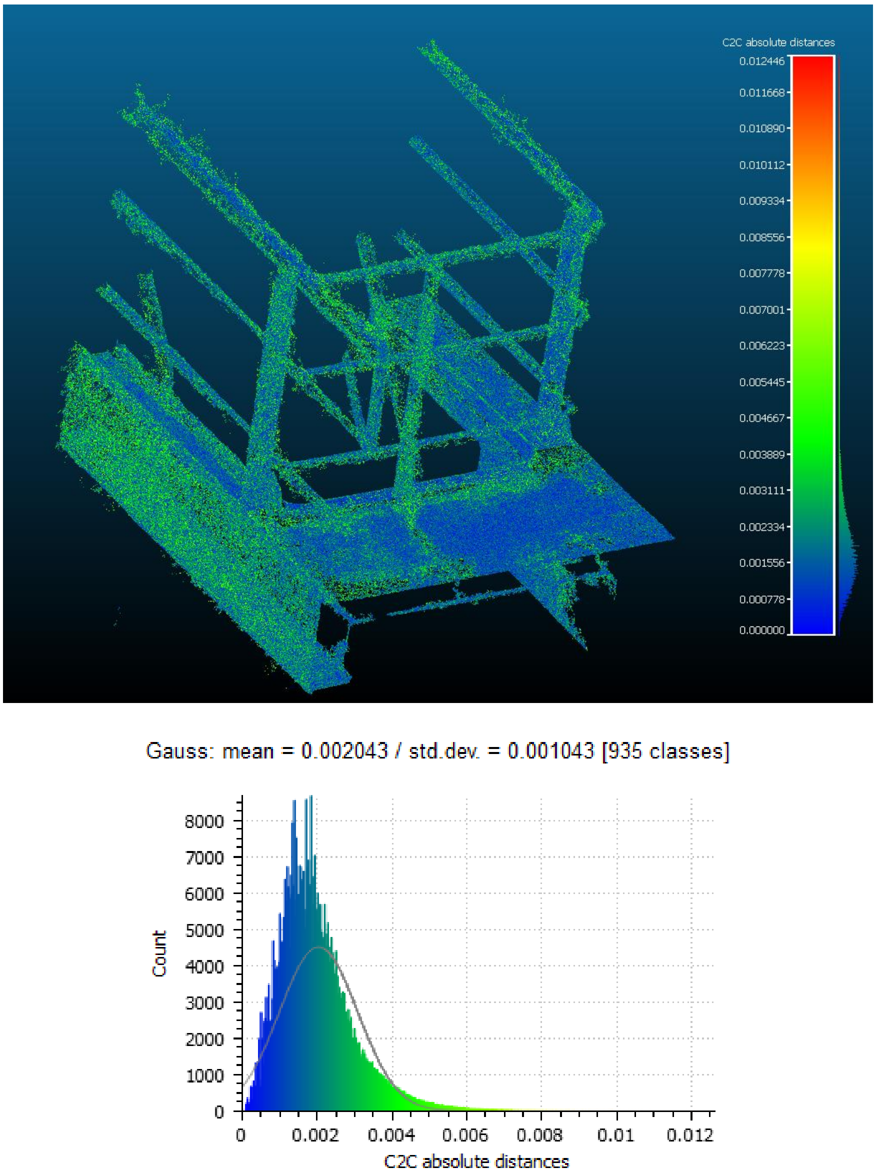


5.1.1.3 Radius of 0.05

In a more conservative approach and using a smaller radius of 0.05 – that would allow for keeping the integrity of areas with high curvatures, like the railing – and using 35 iterations, the output failed to properly resolve the lower curvature regions, such as the floor. This can be noted in the high distance region on the sidewall seen on the left side of Figure 38. Compared to section

5.1.1.1, it performed much better, reducing the computation time by several orders of magnitude and having an average distance about ten times smaller.

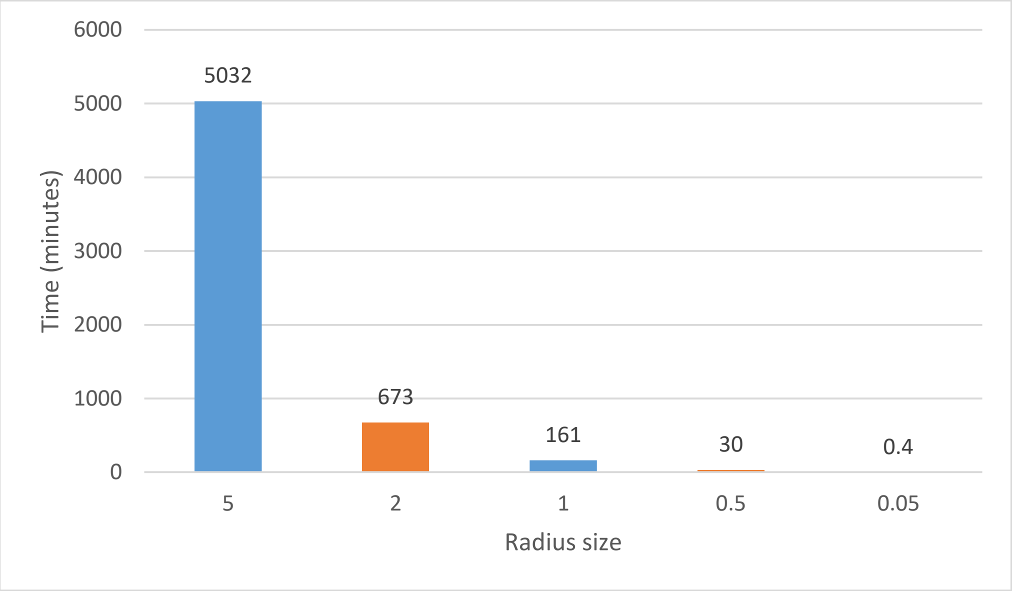
Figure 38: Heat map for WLOP with 10% reduction over 35 iterations on radius of 0.05



5.1.1.4 Computation time

The WLOP method is very computer intensive and grows exponentially with relation to the radius. Figure 39 shows the computation times for a single iteration and same reduction factor, but on different radii, including the ones presented in the above results (presented in blue). The results were taken from an Intel Core i7 2.2GHz, with 12GB of RAM and four cores.

Figure 39: Computation time for WLOP



5.1.2 MLS Smoothing

This approach rendered the most successful results, preserved colors, and was able to smooth the surface to a degree, as seen on Figure 22 page 32. In addition, it did not need any pre computation – such as calculating normals, since the algorithm does not rely on them as explained by Tekumalla et al [9] – prior to being executed, which might save computing time; however, it did require hours to complete in a subset of the point cloud. This method also was not able to handle the severe imperfections of our data set to its full extent, as it can be noticed on the protrusions from the side walls and the flaring on the railing, visible on Figure 40 and Figure 41, respectively. Finally, there was no reduction on the file size, which could potentially simplify the next step in the pipeline.

Notice that even though this algorithm also uses the search radius as its main parameter, it has no relation to the radius used by WLOP. Thus, the values used for the search radius are different from the ones used in section 5.1.1 for its WLOP radius.

Figure 40: Protrusions from sidewalls from MLS

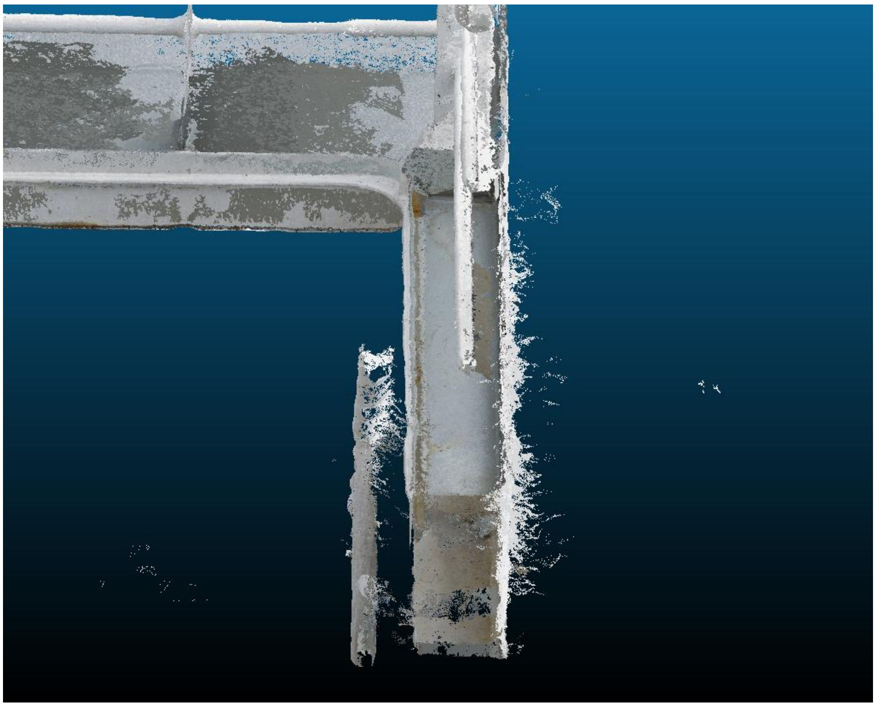
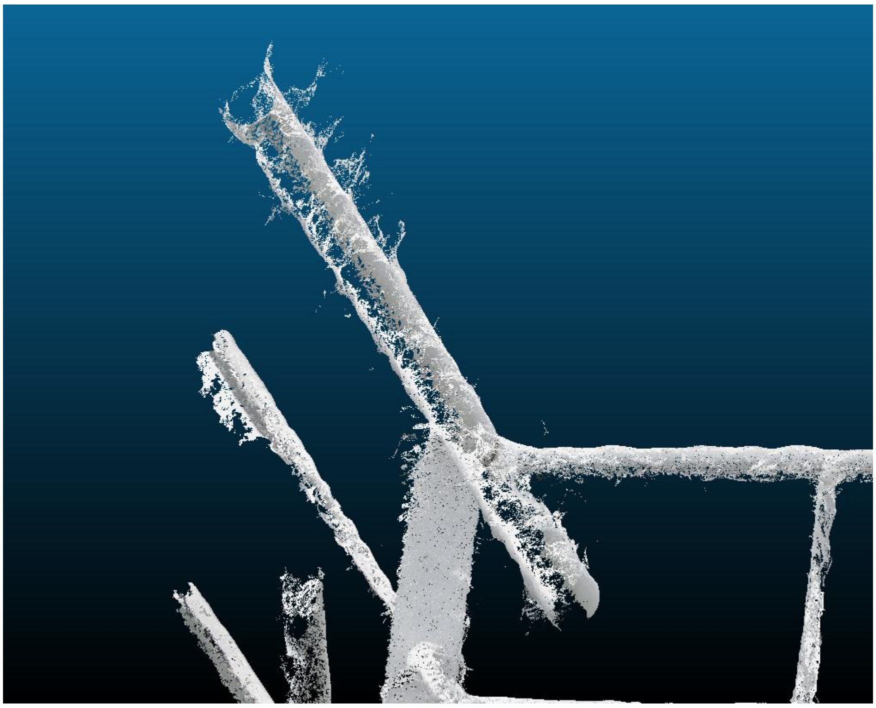


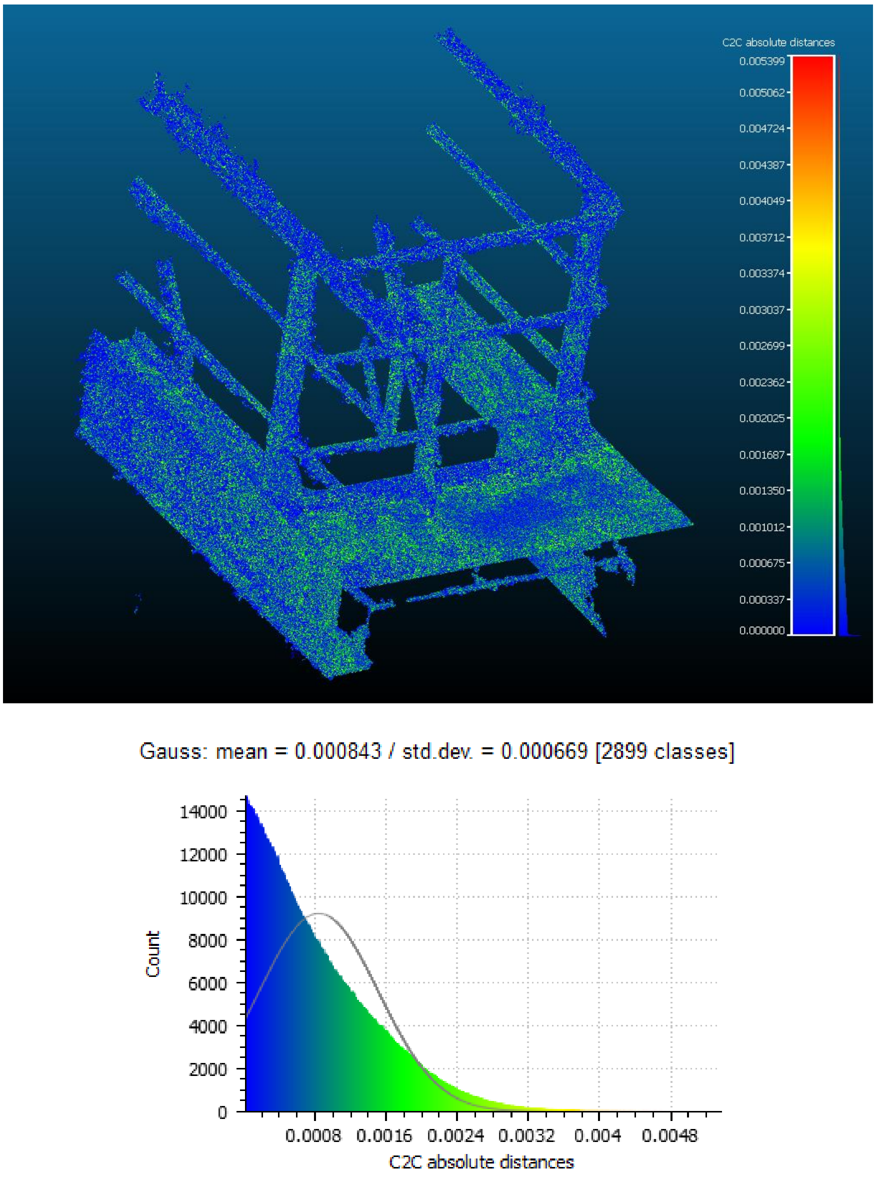
Figure 41: Flaring on railing from MLS



5.1.2.1 Radius of 0.01

When the search radius is not large enough, the algorithm did not present any smoothing results, as seen on Figure 42. This could be due to the radius not including enough points to reliably represent the intrinsic surface; therefore, the interpolation was based on non-representative points from a localized region. As such, there is very little difference between the projected points and the original point cloud and this is reflected in the small distances shown in this result.

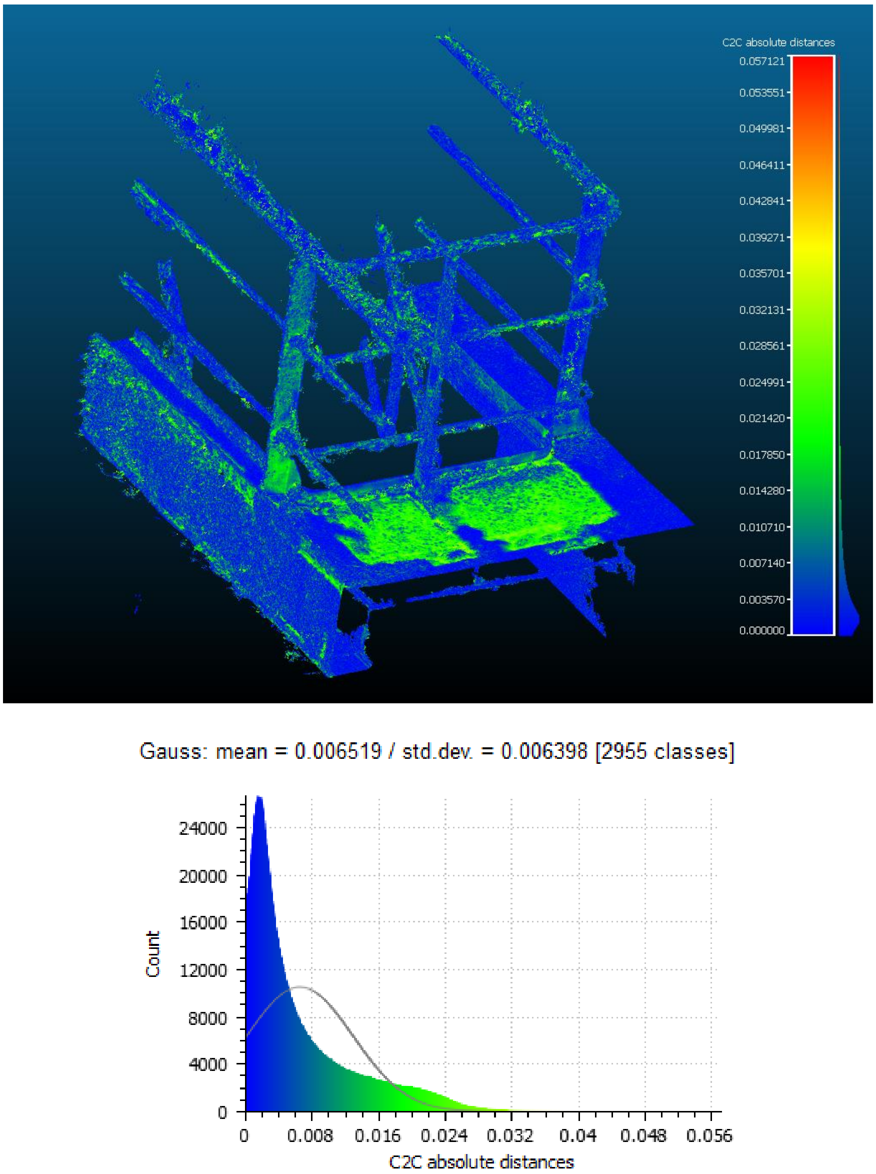
Figure 42: Heat map for MLS with a search radius of 0.01



5.1.2.2 Radius of 0.1

By using a larger search radius, the smoothing procedure was able to extract the surface function with greater success, due to the radius being able to encompass enough of the surface and fit a better matched curved, as mention in section 2.4 and illustrated on Figure 22 page 32. In addition, this heat map is comparable to the best WLOP result seen on Figure 37 page 48, in terms of absolute distances and heat clusters; however, this smoothing method took considerably less time to compute than WLOP with the addition of preserving colors.

Figure 43: Heat map for MLS with a search radius of 0.1



5.1.2.3 Radius of 0.5

With a larger radius, the computation time grew exponentially, as noted on Figure 46, but granting poorer results with severe deformations. The larger radius was not able to cope with higher frequencies in the surface function, meaning that it was unable to handle sharp edges. This is clearly visible in Figure 45, where the intersection between the horizontal railings meet the vertical supporting poles, creating a region of deformation.

Figure 44: Heat map for MLS with a search radius of 0.5

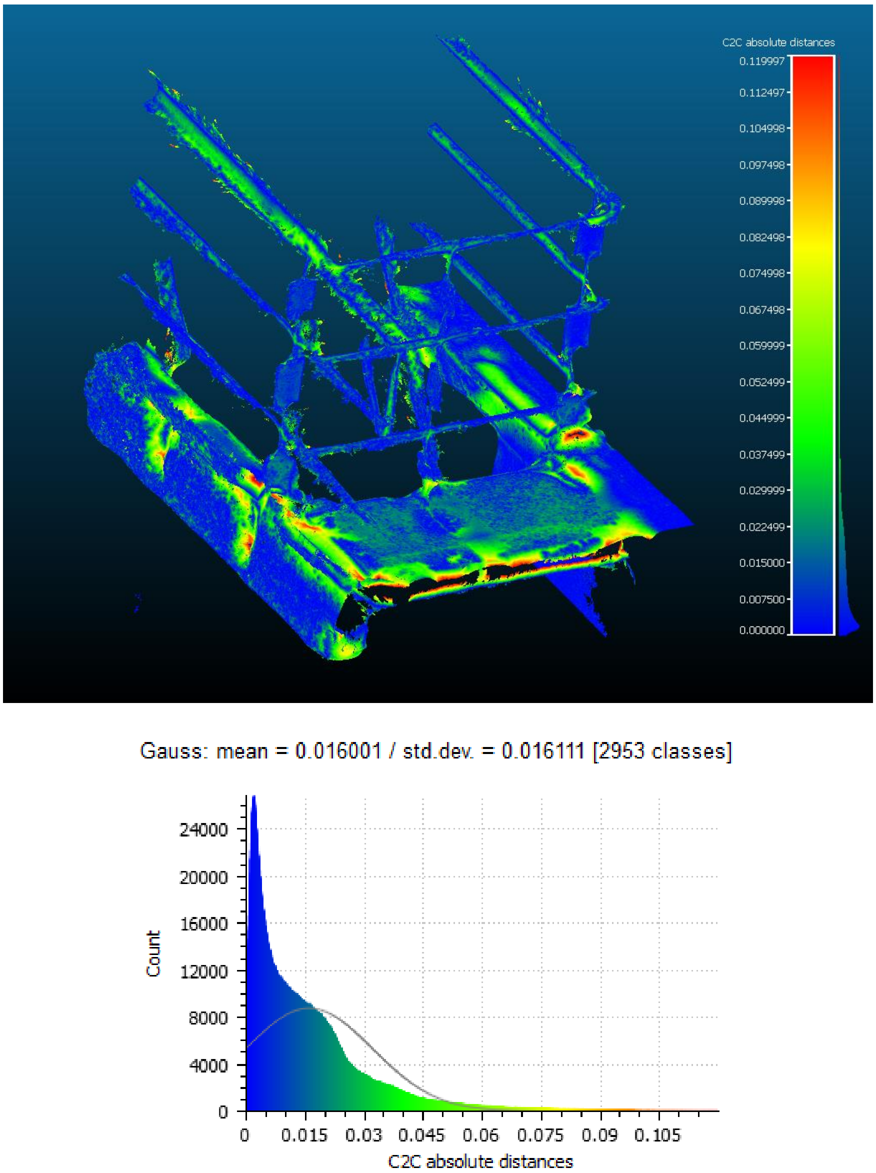
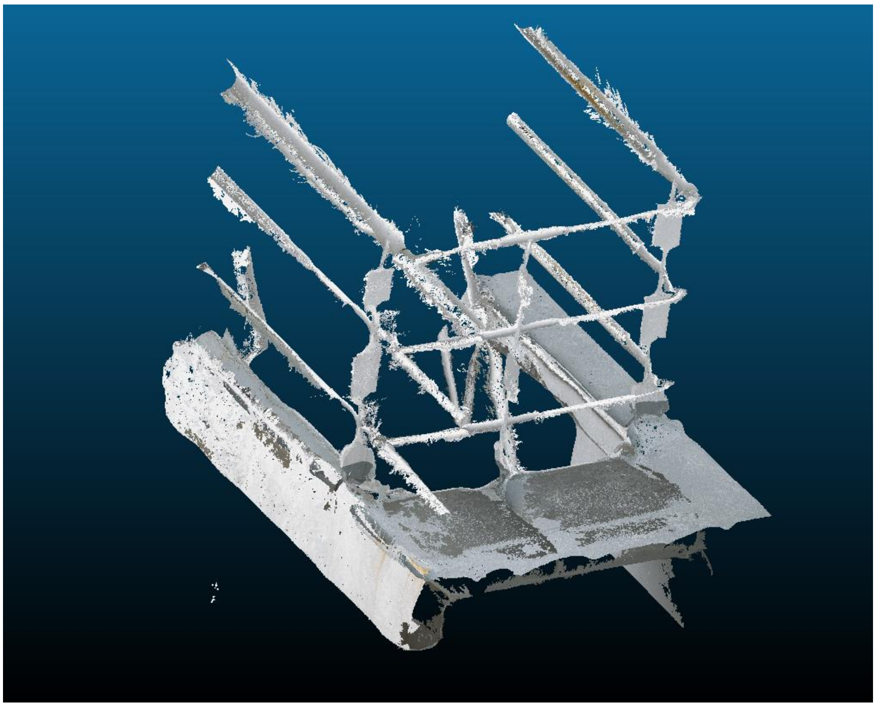


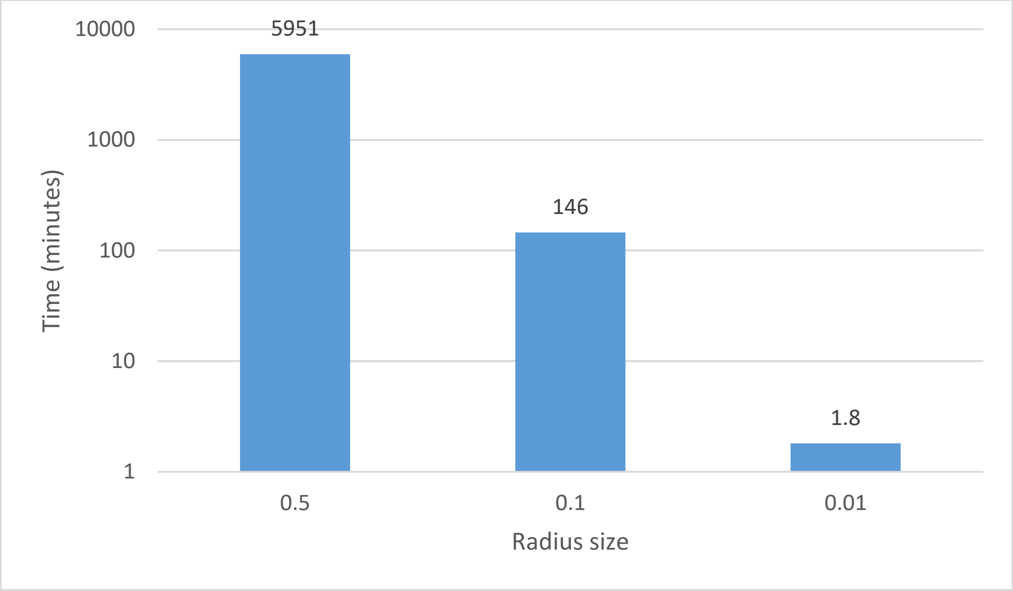
Figure 45: MLS smoothing with a radius of 0.5



5.1.2.4 Computation time

The MLS smoothing is very sensitive to the search radius, growing exponentially with the radius size. This means that, for the desirable results from section 5.1.2.2, there is a 2h30 computation time associated with it. Depending on the robustness of the meshing algorithm down the pipeline, this might be unnecessary waste of resources. Figure 46 shows the computation times for the outputs presented on this section. The results were taken from an Intel Core i7 2.2GHz, with 12GB of RAM and four cores.

Figure 46: Computation time for MLS



5.1.3 Naïve approach

The results were satisfactory, preserving color and still representing the model with sufficient points. However, this simplistic approach would only make sense if further improvements were to be performed in a point cloud representation format, that is, if a transmutation into a different representation, such as a density scalar map, were to be applied, the computational time might not be justifiable. Therefore, while new techniques were explored, this method could be applied if such computation would benefit the overall simplification pipeline, as for MLS smoothing.

Furthermore, since the naïve approach does not modify the positions of the points in any way, there is no reason to study its fidelity to the original model, as it was done to the previous methods.

Lastly, the computation time is linear with the size of the original file, meaning that, different reduction factors will run in always about the same time if the input file is the same; therefore, there is no need to study its computation time since, for the test file being used, the run time was always ~1.5 seconds.

5.2 Mesh Generation

5.2.1 Marching Cubes

The marching cubes implementation turned out to be exceptionally fast and able to generate small files. It proved to be reliable to outliers as a consequence of the gridification process; as explained in section 4.3.1.1, if the density of outliers is not high enough, it will not be reflected in the grid, essentially phasing out the outliers when fitting the isosurface. However, it produced unnecessary triangles for the same reason, i.e. since the grid size is fixed, there will always be multiple triangles generated for a surface that crosses multiple cells.

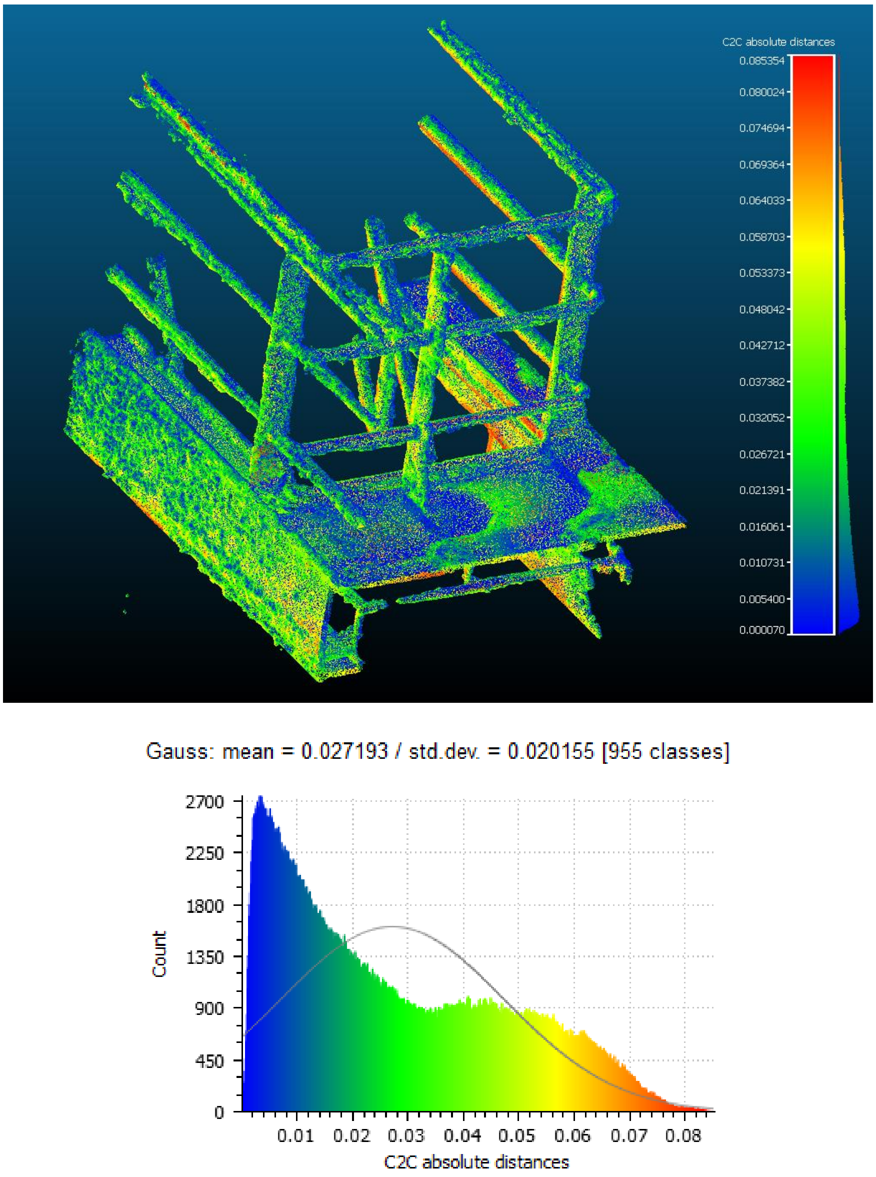
The algorithm was very efficient in terms of memory and time, running in the realm of under five seconds, and consuming about the same amount of memory as the input file.

5.2.1.1 Grid 128^3

Using a coarse grid, the algorithm ran in a fraction of a second, as seen on Figure 52 and generating an output file of 4.2MB – 72 times smaller than the original point cloud. It is interesting to note the pattern of green and blue on the floor in Figure 47; this is the consequence of a grid-based algorithm, where the floor is not perfectly horizontal and therefore is slowly creeping from one grid plane to the next. This is the main cause for loss in fidelity for this implementation of marching cubes.

This configuration obtained the best results by being the fastest, generating the smallest output, and being effective against artifacts. By using larger cells, the average density was higher, meaning that sporadic outliers had less of an impact and were phased away. In addition, the larger cells were able to encompass regions of noise and with that interpolate the value for the surface fitting within a single cell. If the noisy region were to cross several cells, the threshold from section 4.3.1.2 could be crossed multiple times in adjacent cells, generating multiple faces.

Figure 47: Heat map for MC with a grid of 128^3



5.2.1.2 Grid 256^3

By using a 256×256×256 grid, higher frequencies were captured, and with that, more of the artifacts came with the mesh. The processing time was still under two seconds, shown in Figure 52, and the output file size 20.3MB. Though both values are quite good and acceptable, the quality of the mesh degraded. A comparison of the noise can be seen on Figure 49 and Figure 50, where the sidewall on the 256^3 version contains severe corrugations, and the railing support corrodes away at the top.

Figure 48: Heat map for MC with a grid of 256^3

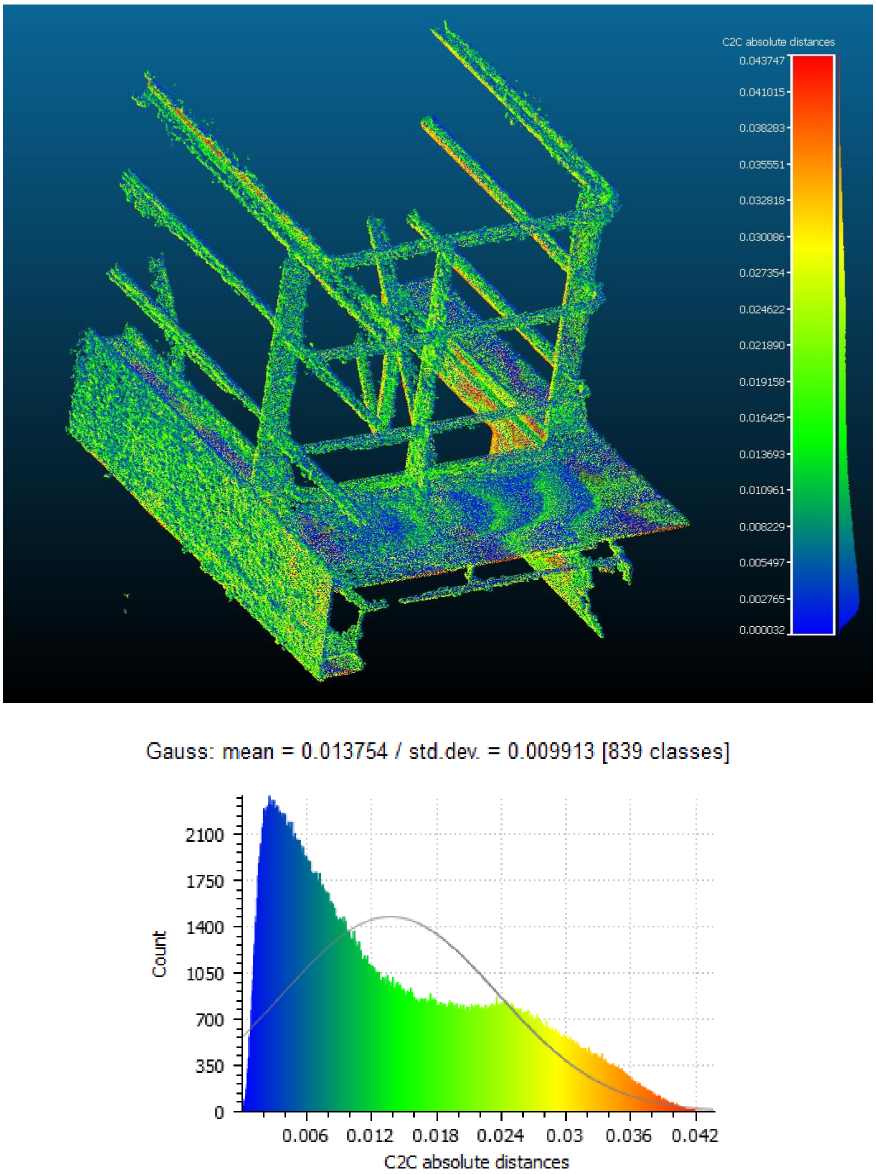


Figure 49: Comparison of sidewalls: MC 128^3

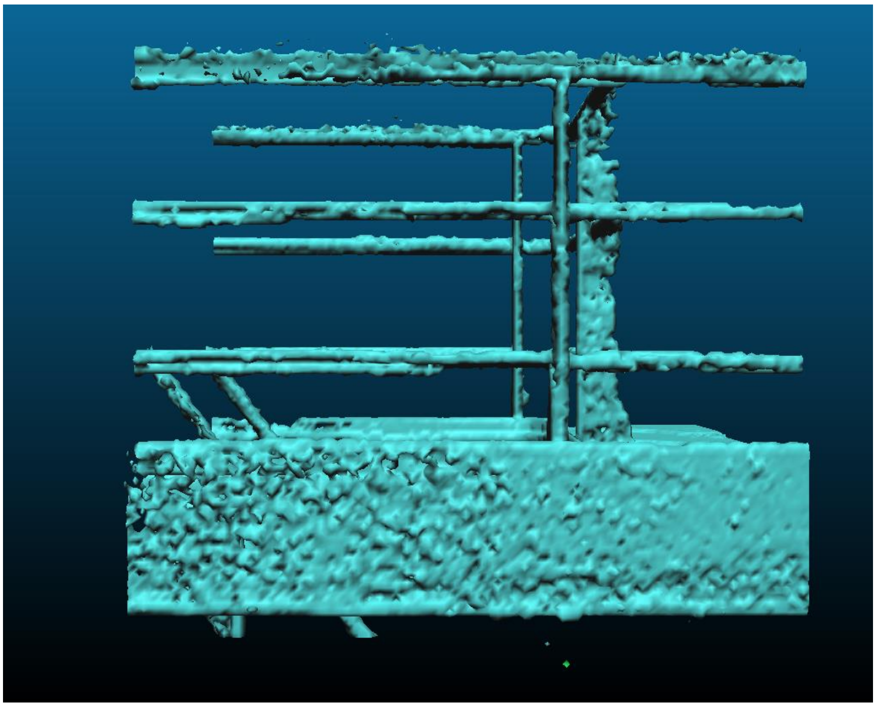
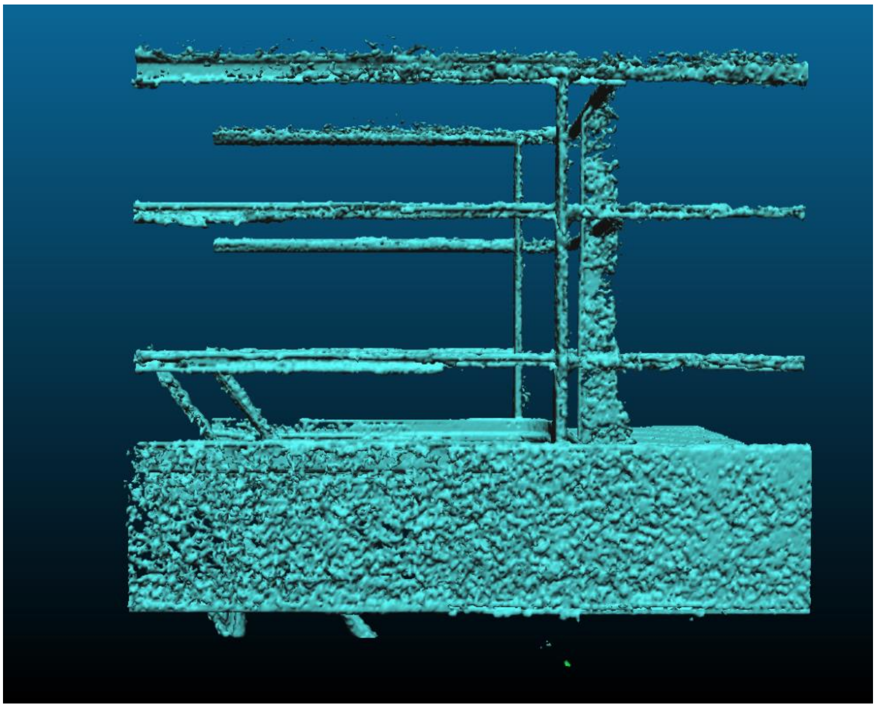


Figure 50: Comparison of sidewalls: MC 256^3

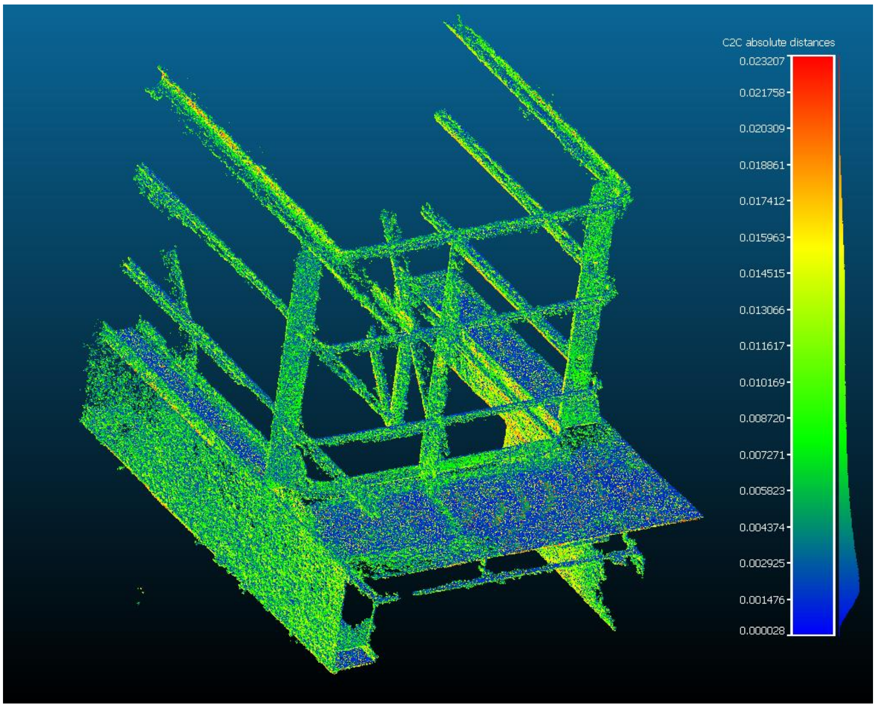


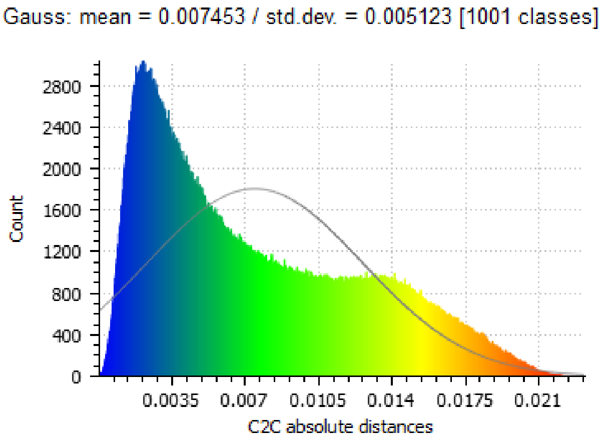
5.2.1.3 Grid 512^3

A finer grain grid was then tested, with 512×512×512 cells. The computation time was still under ten seconds, but the file size already reached 79MB. Once there are too many cells in the grid, the gain from the meshing process becomes hazed. Consider an extreme case: it is possible to have an arrangement such that, for each point in the point cloud, a whole triangle would have to be generated to represent it, thus three new points would emerge, as the three vertices from a triangle. This would actually increase the output file size and defeat the purpose of this thesis.

Furthermore, this amount of cells enabled much of the noise to come through, as the cells were not large enough to smooth out the variance of the noise, as can be seen on Figure 51. As such, the distance from the original model is quite small, with an average of 0.007; but that only reflects the persistence of undesired qualities.

Figure 51: Heat map for MC with a grid of 512^3

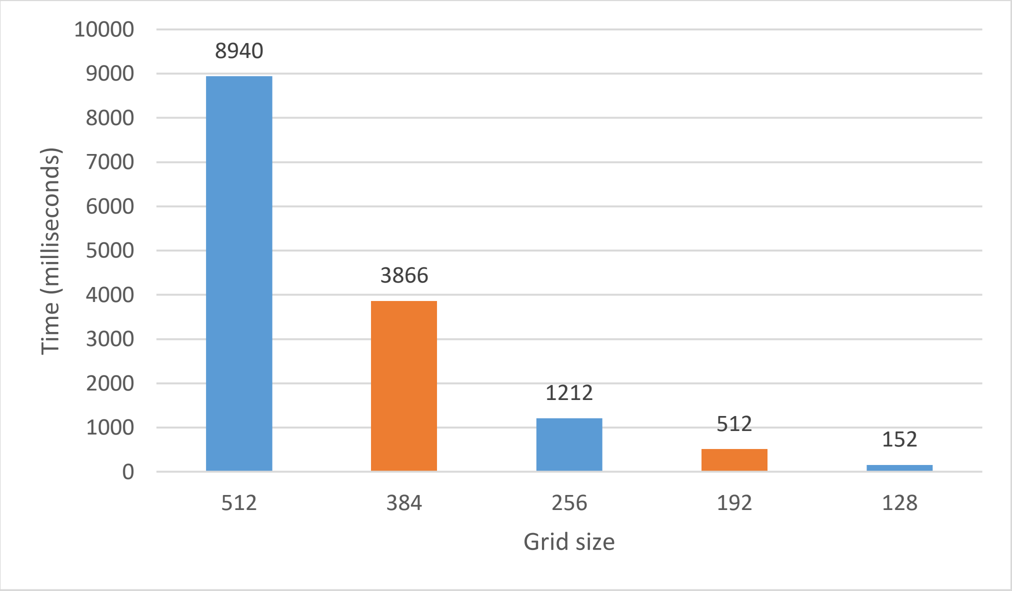




5.2.1.4 Computation time

Marching cubes proved to be exceptionally fast, growing linearly with the number of cells in the grid. Even using over 134 million cells, the algorithm executed in under ten seconds. Figure 52 shows the computation times in milliseconds for the outputs using different grid sizes, including the ones presented on this section (shown in blue). The results were taken from an Intel Core i7 2.2GHz, with 12GB of RAM and four cores.

Figure 52: Computation time for marching cubes



5.2.2 Poisson Surface Reconstruction

The Poisson reconstruction showed very good results, able to fit reliable surfaces on the artifact-rich data sets. Smaller octree depth levels showed great distortions, but higher values were able to model the surfaces appropriately, though with a tendency to reverse normals of the generated faces.

However, this method is much slower than the marching cubes. Not only it requires normals for each point in the point cloud, it also takes vastly more time in the surface generation phase. The data set this work is aimed at does not contain normals, therefore the time required to compute them should be considered as part of the nature of this algorithm.

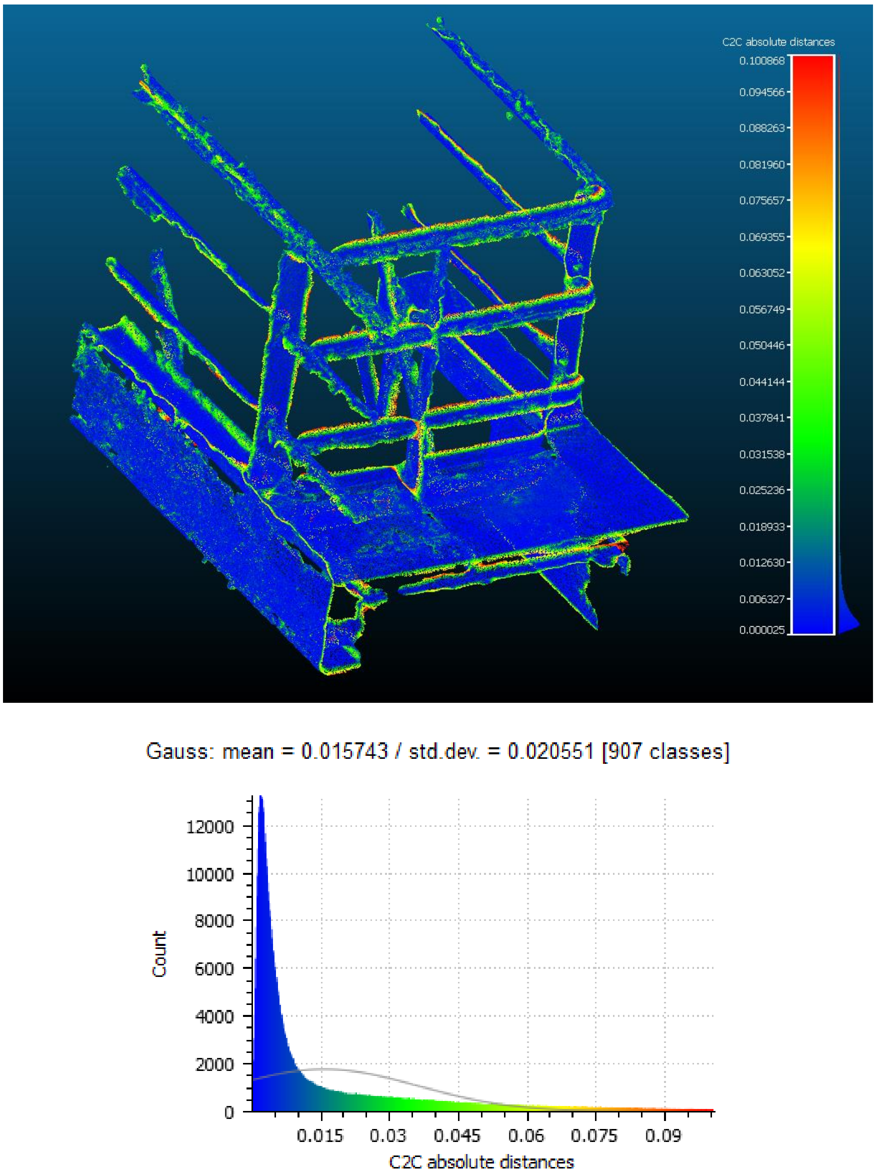
5.2.2.1 Level 8

By using an octree of depth 8 for the Poisson reconstruction, the results were poor and distorted. Some heavy artifacts emerged as flaring from the rails, which can be easily noticed by their high distance values on the heat map on Figure 53. In fact, their distance skewed the whole heat map normalization, making most of the figure be blueish, however the histogram shows that the average distance is in reality 0.016.

The process itself was quite fast, taking only 55 seconds to generate the surface. However, 525 seconds were required beforehand in order to compute the necessary normals for this algorithm. The output file was small, measuring 9.5MB.

Overall, the distortions are too great to make this method viable for the purpose of this project.

Figure 53: Heat map for Poisson with octree depth 8



5.2.2.2 *Level 9*

By adding an extra level to the octree, most of the distortions disappeared, but not completely. The surface also had a much higher fidelity, with an average distance of 0.007, but the flaring is still present. In fact, the flaring can be removed by selecting a different density threshold, but in detriment of have holes in the model, as seen on Figure 55; there is not enough levels to discriminate the density. This effect is even more accentuated in shallower octrees.

Figure 54: Heat map for Poisson with octree depth 9

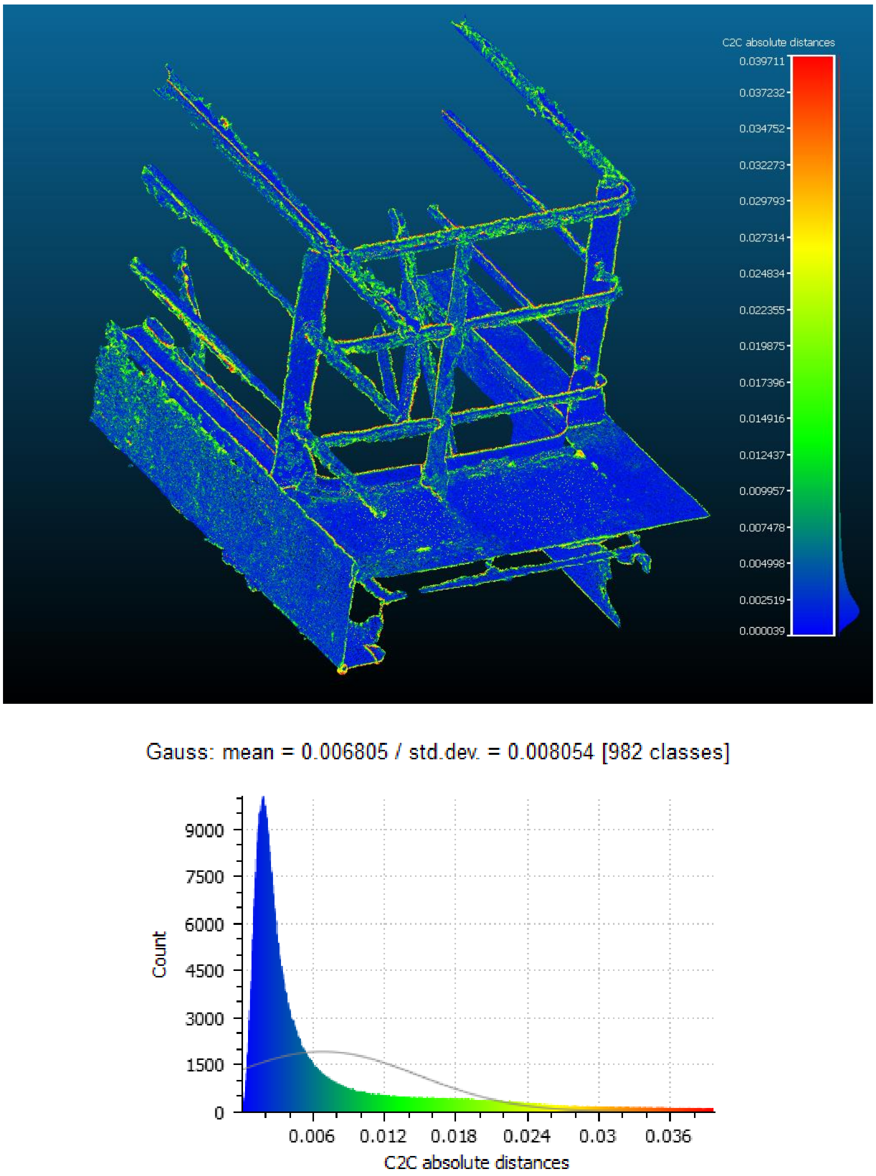
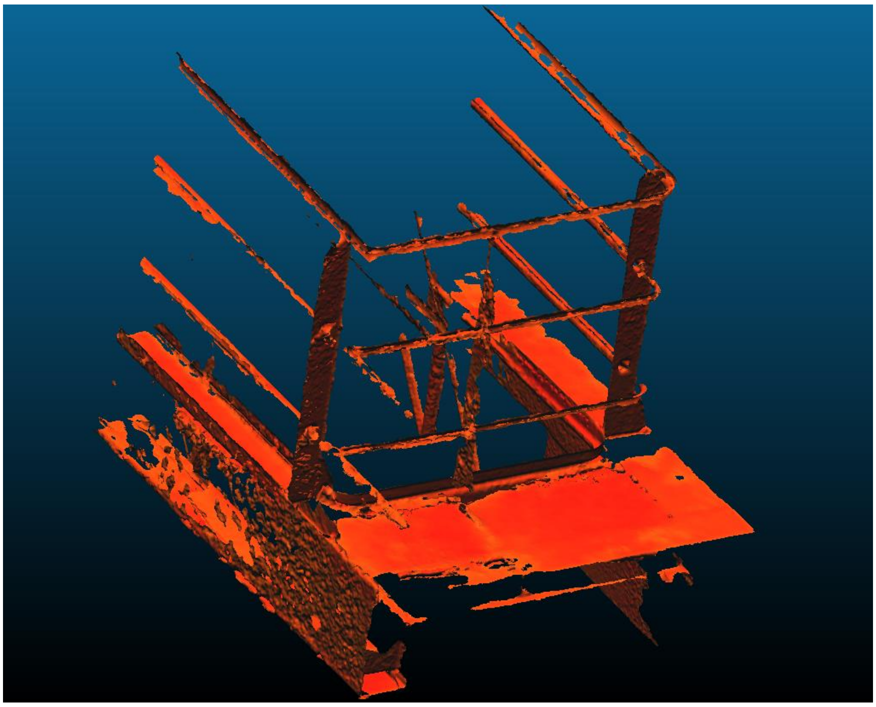


Figure 55: Poisson 9 with higher threshold

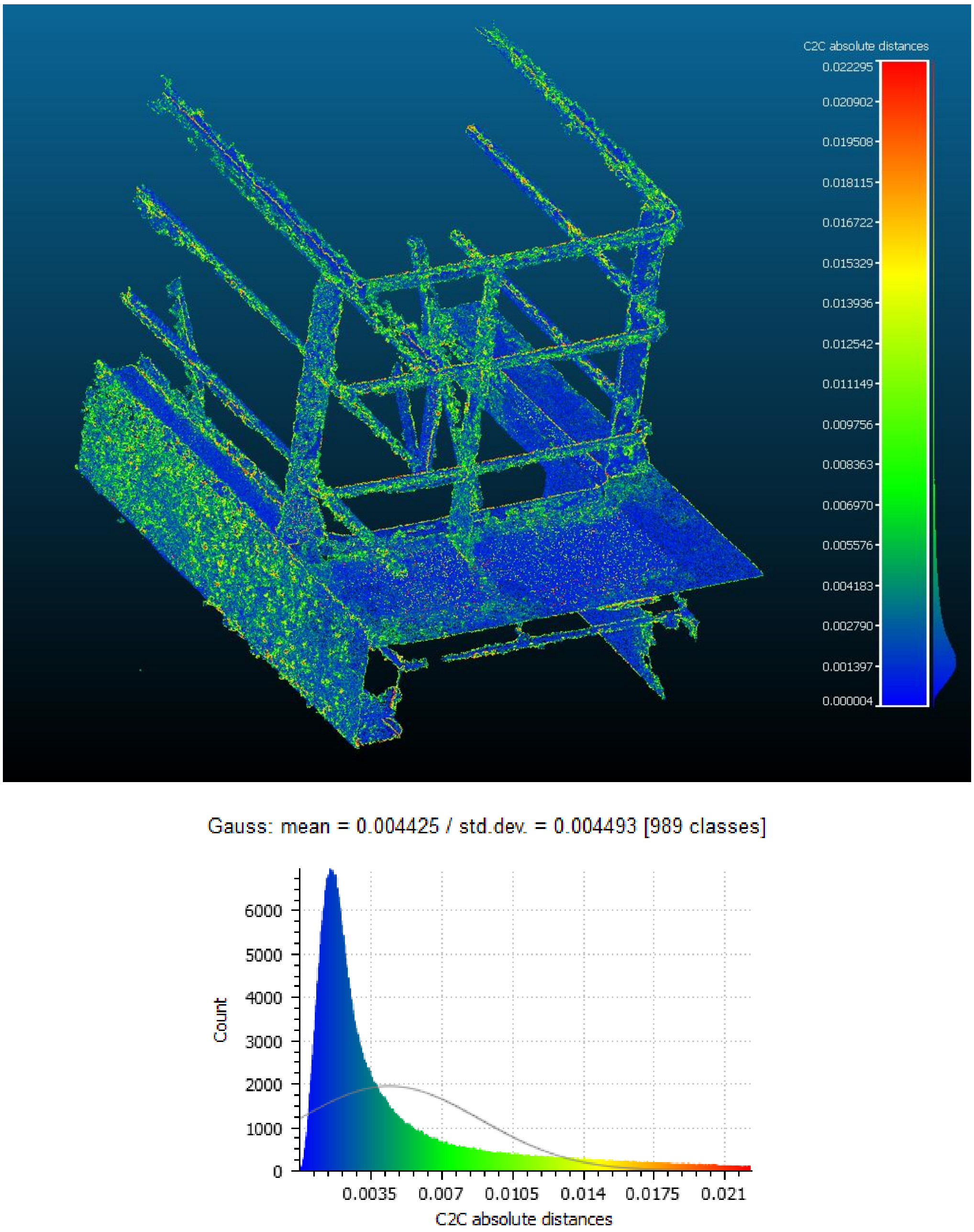


5.2.2.3 Level 10

Using a deeper octree of ten levels, the algorithm managed to fit a better surface onto the model. The flaring is no longer present and the surface is very close to the original model, reflecting the high fidelity that was achieved.

Similar to other octree levels, the normals had to be precomputed, increasing the computing time: 528 seconds for normals calculation and 174 seconds for the mesh generation. However, the file size is significantly larger, nearing the original point cloud at 185MB. Overall, the results were satisfactory, the best Poisson output, in fact; but the output file was larger than desired and the computation time long.

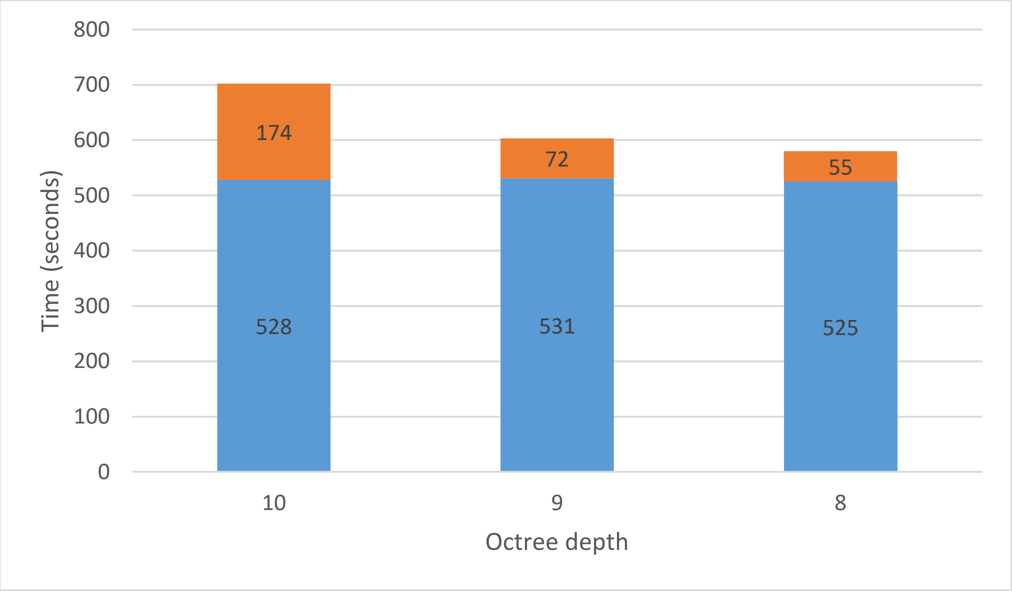
Figure 56: Heat map for Poisson with octree depth 10



5.2.2.4 *Computation time*

The computation time, interestingly enough was fairly constant across the different parameters. The reason for this is the fact that about 90% of the time is spent calculating normals, shown in blue, which is not affected by the octree depth. However, considering just the meshing part, shown in orange, the algorithm was orders of magnitude slower than marching cubes; still, the data set that this thesis is aimed at does not contain normals, therefore its calculation time has to be considered. The results were taken from an Intel Core i7 2.2GHz, with 12GB of RAM and four cores.

Figure 57: Computation time for Poisson reconstruction



5.3 Pipelines

With the different techniques tested and analyzed, their best outputs were matched with each other in order to create point-cloud-to-mesh pipelines. A total of eight pipelines were evaluated:

1. WLOP + MC
2. MLS + MC
3. Naïve + MC
4. Crude + MC
5. WLOP + Poisson
6. MLS + Poisson
7. Naïve + Poisson
8. Crude + Poisson

The parameter chosen for the methods are as follows:

- WLOP: Radius of 1

- MLS: Radius of 0.1
- Naïve: 25% reduction
- Marching Cubes: 128^3 cells
- Poisson: 10 levels depth

"Crude" here stands for the unprocessed point cloud, with no treatment applied to it before being fed into the meshing algorithm. Since the crude versions were analyzed in the previous sections, their results will be presented here only briefly so that all pipelines have an equal footing for comparison.

5.3.1 Marching Cubes

Marching cubes proved to be very fast and efficient. With the computation time varying around 0.15 seconds, the overall time shown in Figure 66 is heavily weighted by the point cloud treatment phase. There is also the significantly smaller file sizes generated that average 3.9MB as seen on Figure 67 on page 87, which is a direct consequence of the grid size.

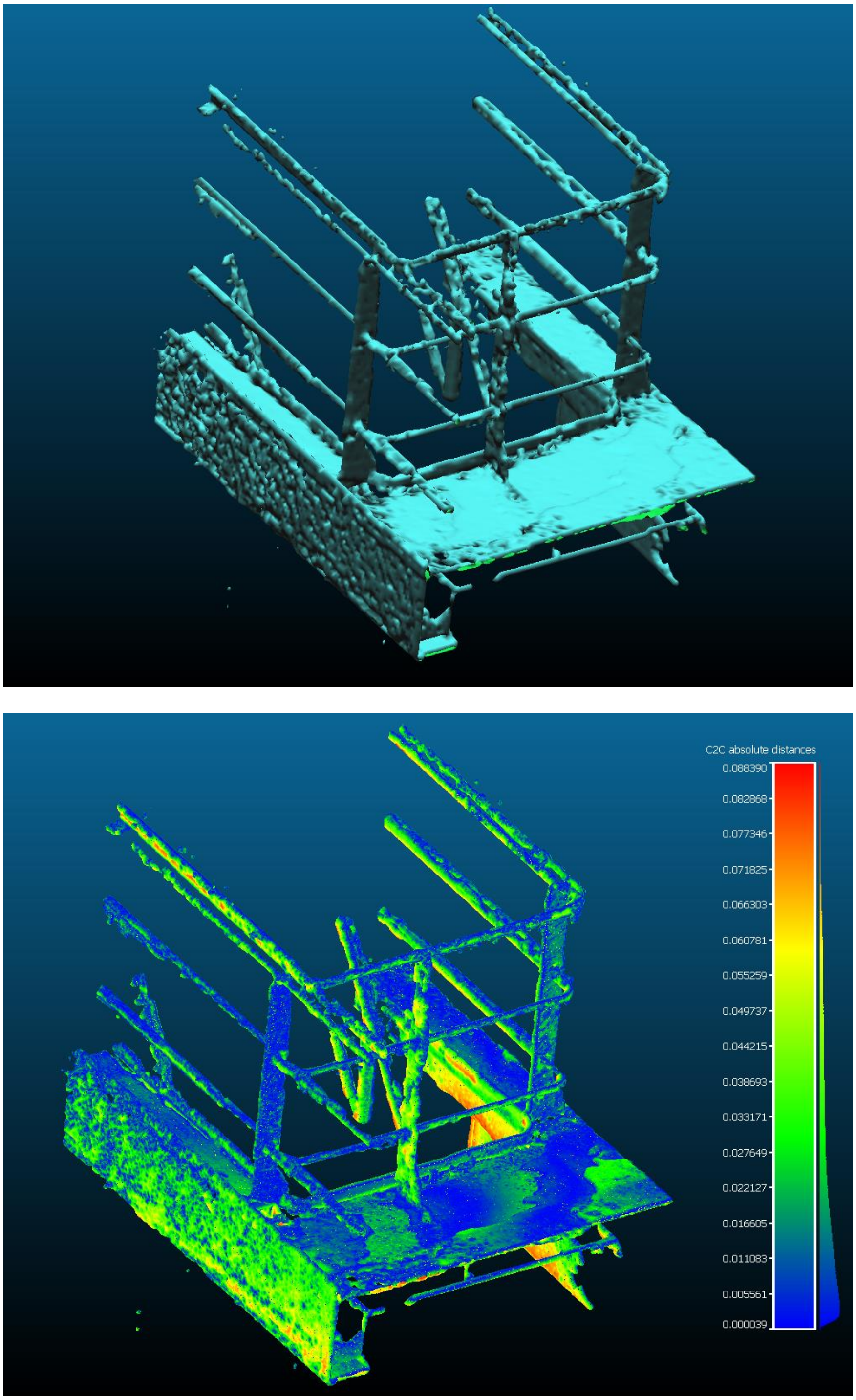
The method however does not output colors as it is implemented by the time of this assessment and this can be considered as loss of engineering information. With that in mind, when color support is added to the algorithm, the file sizes will grow to reflect the storing of color information, that is, 24 bits per vertex. That means that, in average, the files will grow 1.9MB on average once colors are implemented, for the point cloud being used in this test.

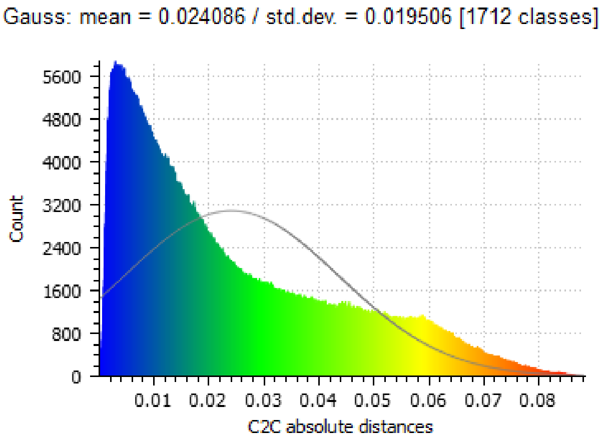
5.3.1.1 WLOP

When combining with WLOP, the output was subpar with several holes present in the model. In addition, the computation time was the highest of all the pipelines involving marching cubes, as seen on Figure 66 on page 86. However, the results presented the lowest average distance from the original model with a standard deviation comparable to the other marching cubes pipelines.

Due to the holes and the vast time required to finish this pipeline, this combination was not deemed acceptable for our parameters set in our objectives, i.e. the necessity for a fast and efficient algorithm and a fidelity to the original model, discussed in section 1.2.

Figure 58: WLOP and Marching Cubes



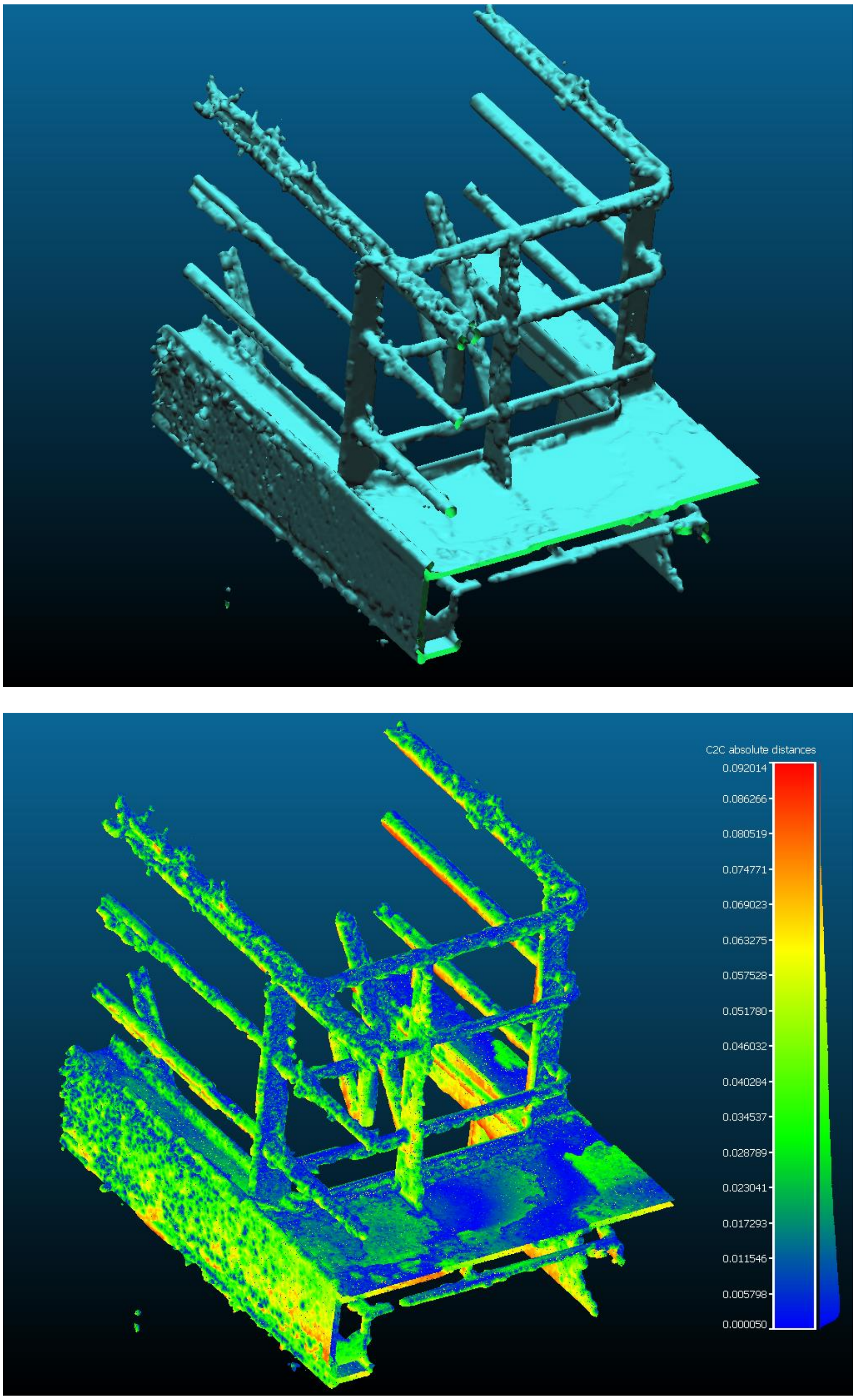


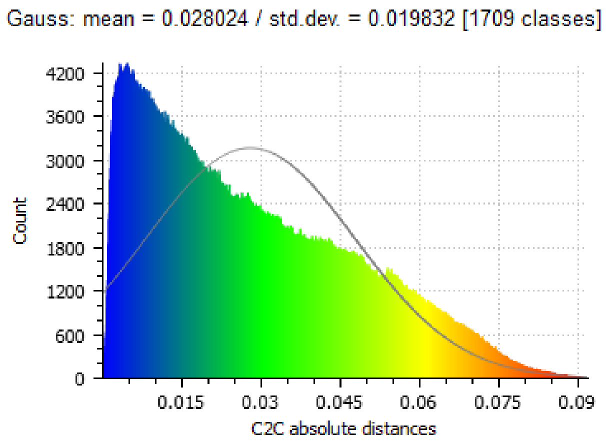
5.3.1.2 MLS

The MLS showed considerably better results. The holes present in the WLOP pipelines are gone and the model exhibits a smooth quality to its surface, as can be noticed by comparing Figure 58 above and Figure 59 below. Moreover, the railings retained their cylindrical shape, which was not the case for the WLOP. The main downside for this combination is the computation time, though lower than WLOP, it is still in the order of hours, as seen on Figure 66 on page 86.

Since the file sizes for the marching cubes pipelines are all very similar, the main conditions for acceptance is the efficiency of the algorithm and the quality of the output, as stated in the objectives. As such, the MLS + MC pipeline does present good results, the best for marching cubes pipelines, in fact. However, the computing time is a hindrance, meaning that the other pipelines must present significantly worse output quality to justify the computing time of almost three hours, seen on Figure 66 on page 86.

Figure 59: MLS and Marching Cubes



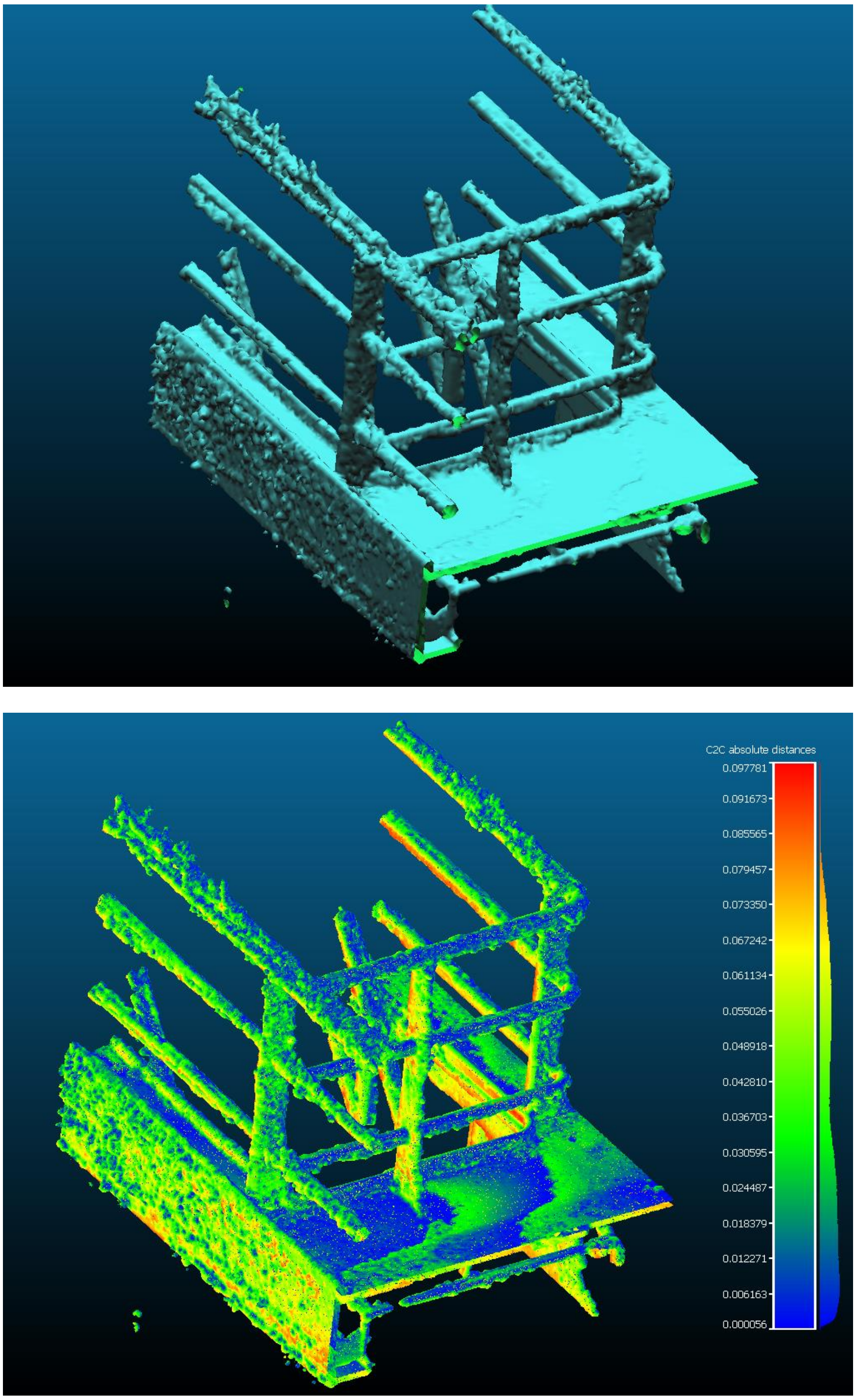


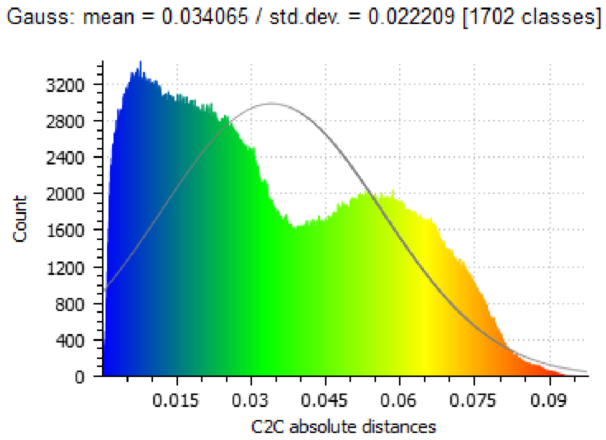
5.3.1.3 Naïve

The naïve method proved to be the worse marching cubes pipeline. There is not much in its favor expect for the quick processing time, though the time required is still greater than running the crude point cloud.

It is interesting to note that, even though the points were removed across all regions randomly, the noise turned out to be accentuated in the pipeline when comparing to the crude input. This is especially visible on the sidewall, where the protuberances are heightened in the naïve method when compared to the crude method. In fact, this combination presented the largest distances of all the marching cubes pipelines, averaging at 0.034, as seen on Figure 60.

Figure 60: Naïve and Marching Cubes



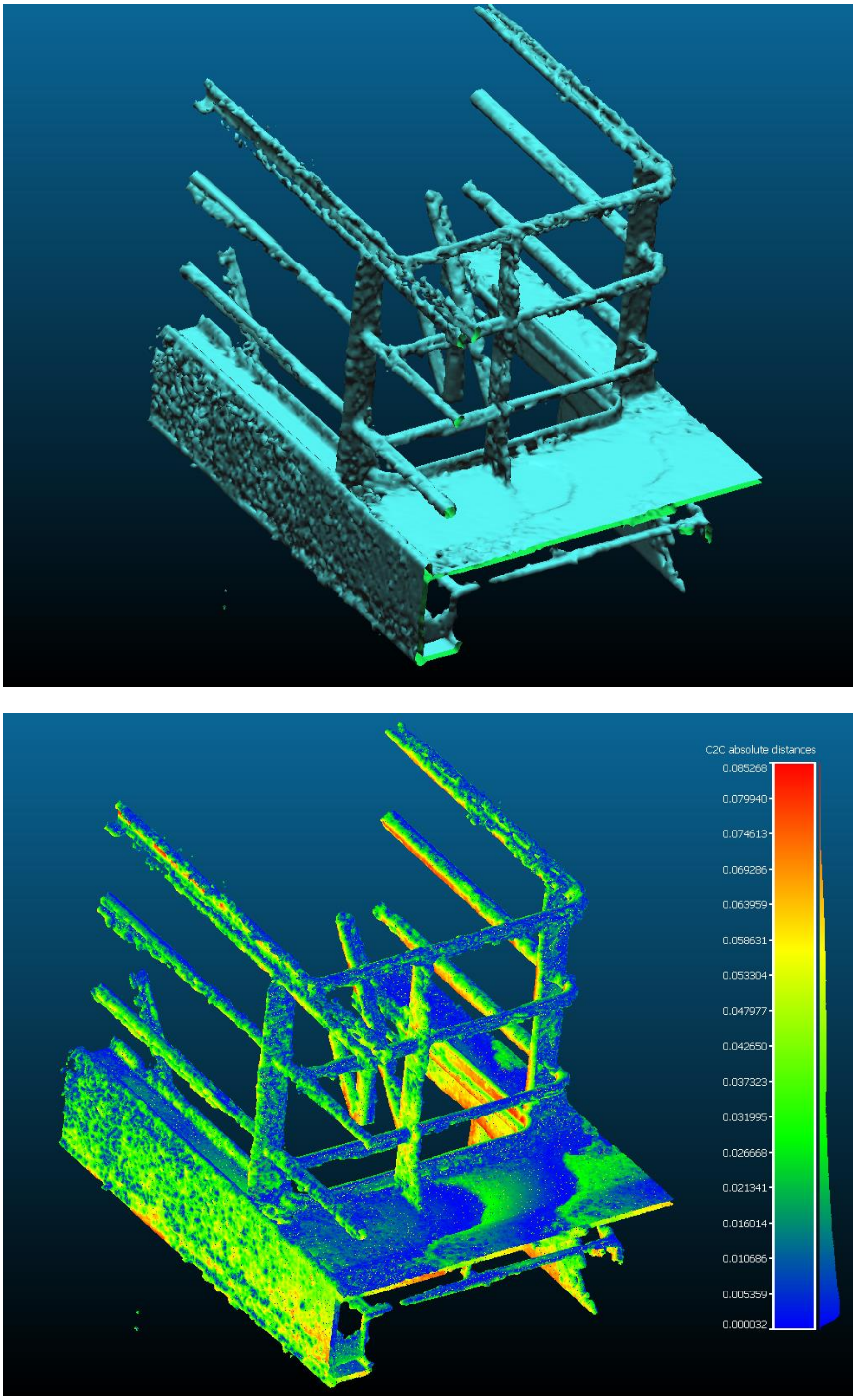


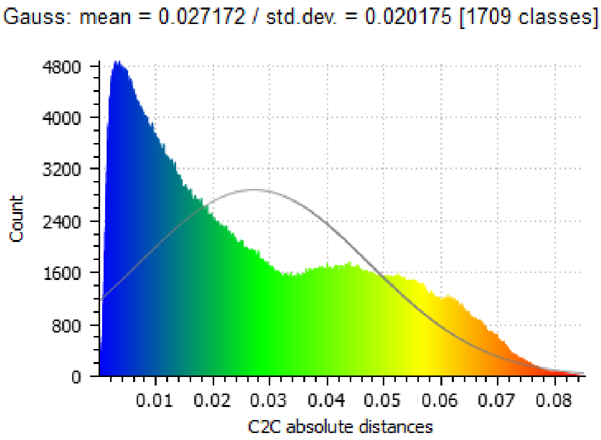
5.3.1.4 Crude

The crude method as by far the fastest method, compared across all pipelines, including the Poisson ones, requiring 151 milliseconds to complete, noticed on Figure 66 on page 86. The distances are comparable to all the other marching cubes pipelines and it generated the second smallest average distance to the original model.

Though the MLS point cloud treatment coupled with marching cubes generated a smoother surface, the results are very similar to each other due to the gridification process. All shapes are retained from the original model, and engineers can perform measurements just as well. Consequently, the three hours required to handle this small data set does not justify the use of MLS. This is, therefore, the best pipeline using marching cubes.

Figure 61: Crude and Marching Cubes





5.3.2 Poisson

Poisson generated very smooth surfaces with short computing time. However, it requires that the point clouds contain normals associated with the points. This is not the case for the stereoscopically generated data sets that this work is aimed at. Therefore, there is an overhead of computing normals, which take a significant longer time to conclude.

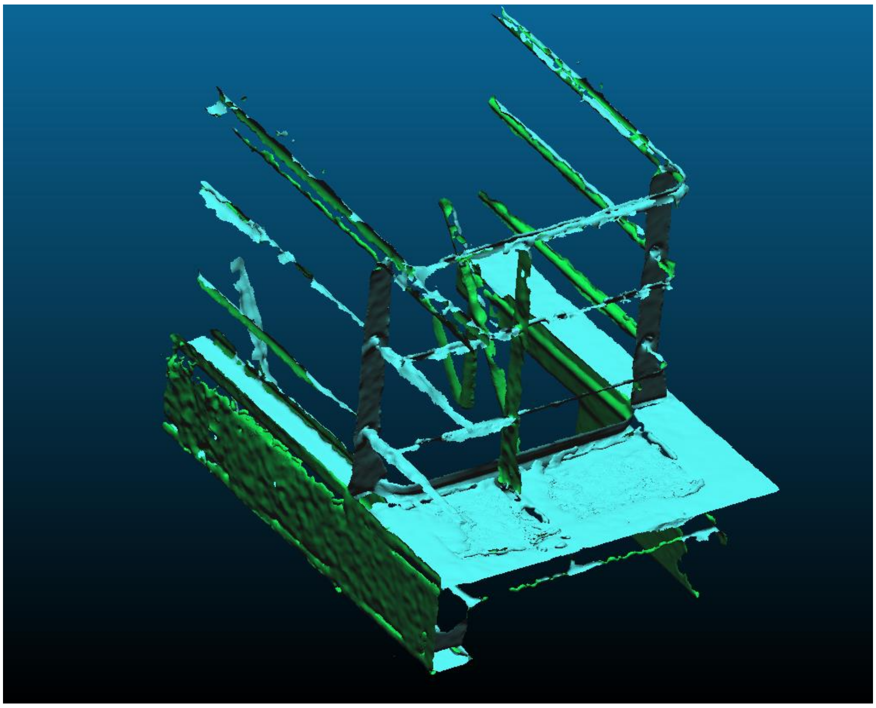
In this section, most of the pipelines presented were able to remove the outlying cluster on the lower left side of the model, reflecting the resilience to outliers that this method promises [15]. It was also able to generate single surfaces on the floor, which is not the case for the marching cubes. However, this method had a tendency of generating inverted normals on the final mesh. This happens because a surface can only have one direction to its normal, and a single plane, such as the manner which Poisson represents the floors and sidewalls also must choose a single direction for its normals. This way a ceiling might have its normal facing up, or a floor might have its normal facing down. This is especially visible on Figure 62, where there is no color information and the normal directions can be identified by the shading of green.

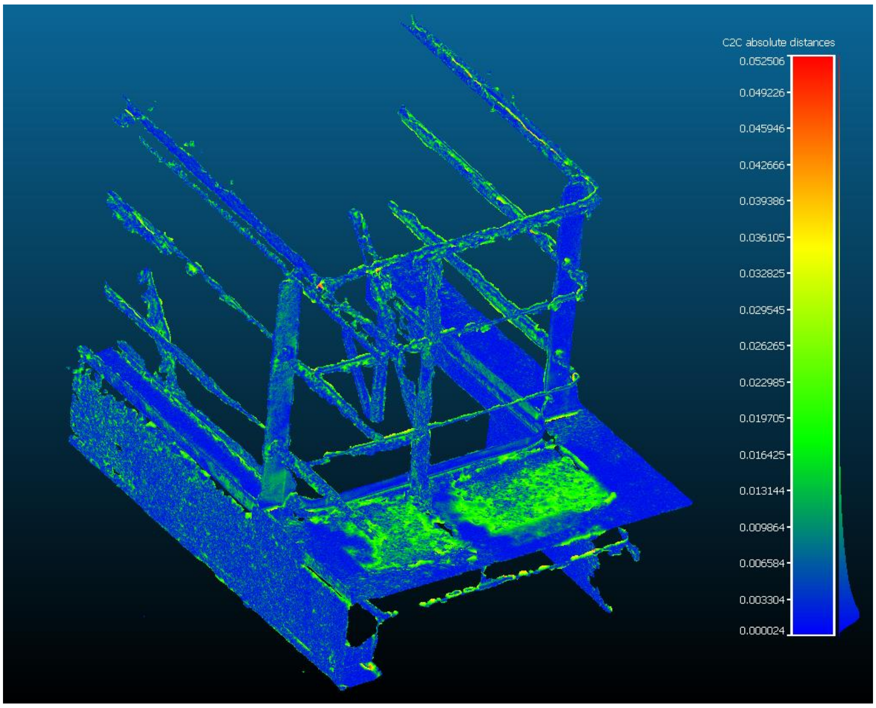
Furthermore, this method generated much larger file sizes, several orders of magnitude larger than the Marching Cubes pipelines, as seen on Figure 67 on page 87.

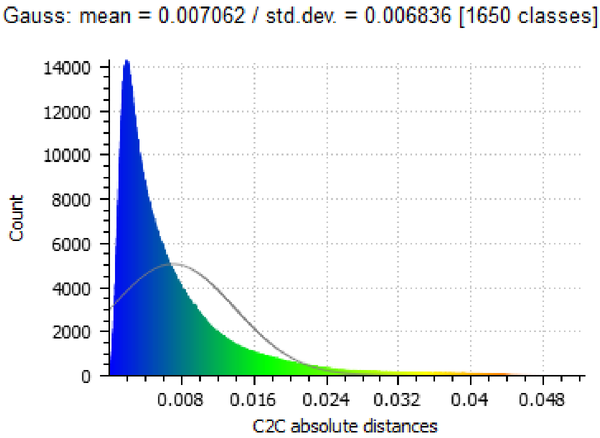
5.3.2.1 WLOP

Similarly to the marching cubes pipeline, when using WLOP the results were mediocre: there are severe deformations and holes, with major segments of the railing missing and flattened. This poor result, nonetheless, took the longest time to calculate and generate, as seen on Figure 66. In addition, due to the loss of color information from the WLOP treatment, it fails to output colors in the result. However, it generated to smallest file size of any Poisson pipeline. Finally, it presented the second highest average distance from the original model, measuring at 0.007.

Figure 62: WLOP and Poisson





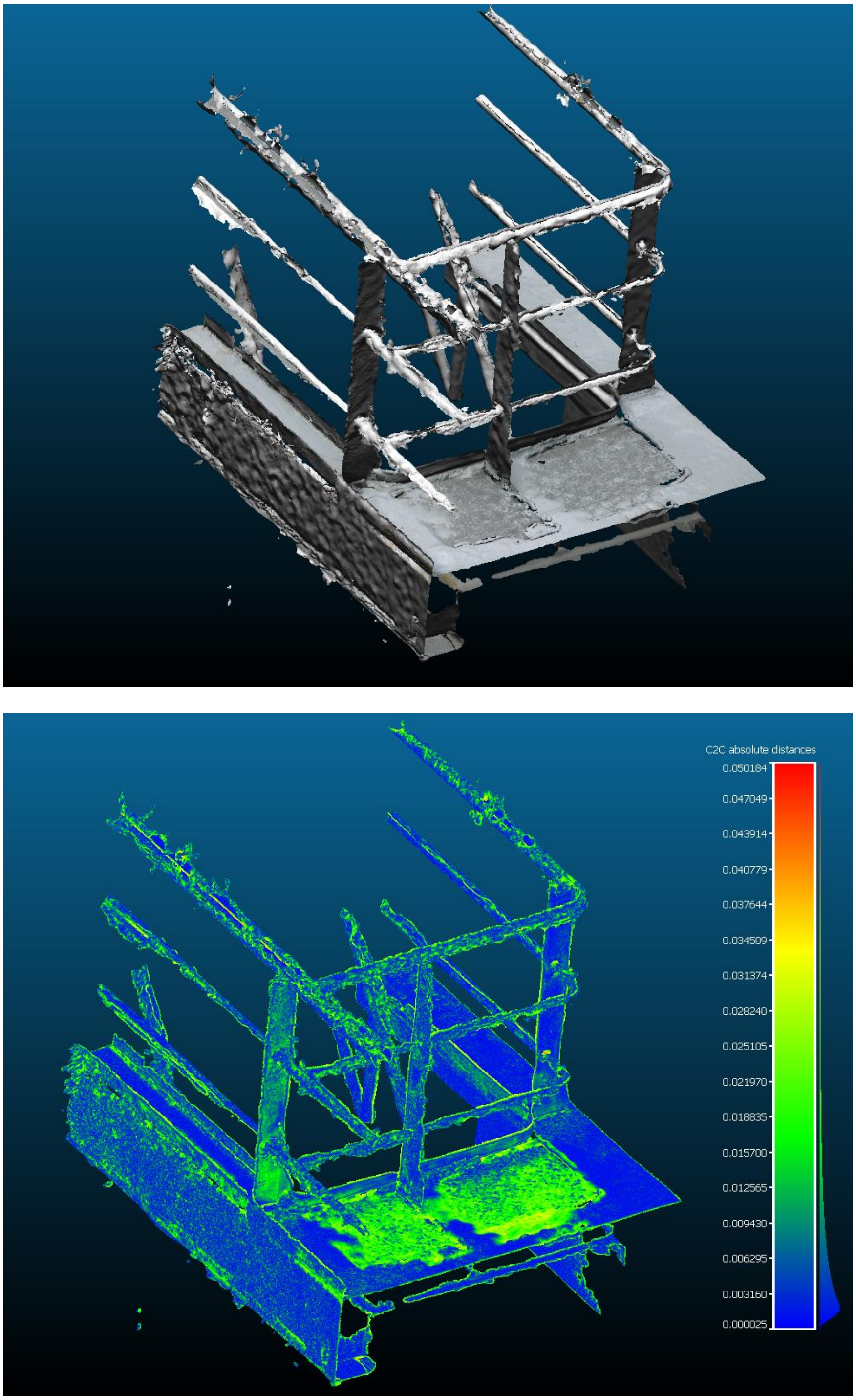


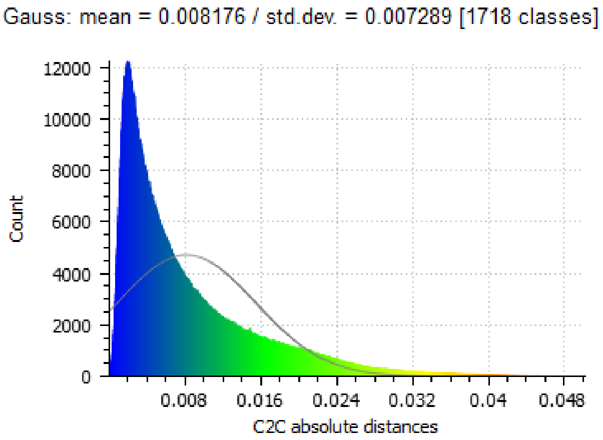
5.3.2.2 MLS

MLS, in contrast to WLOP, performed worse when combined with Poisson. There are severe depressions on the floor, clearly visible on the heat map on Figure 63, which were not present when combined with MC. Furthermore, this is the only Poisson pipeline that heavily retained the outliers on the lower left side of the figure. The method also presented the lowest fidelity to the original model with the distance averaging 0.008.

All these factors, coupled with long computing time of almost three hours, rendered this pipeline not a good option for the final pipeline.

Figure 63: MLS and Poisson



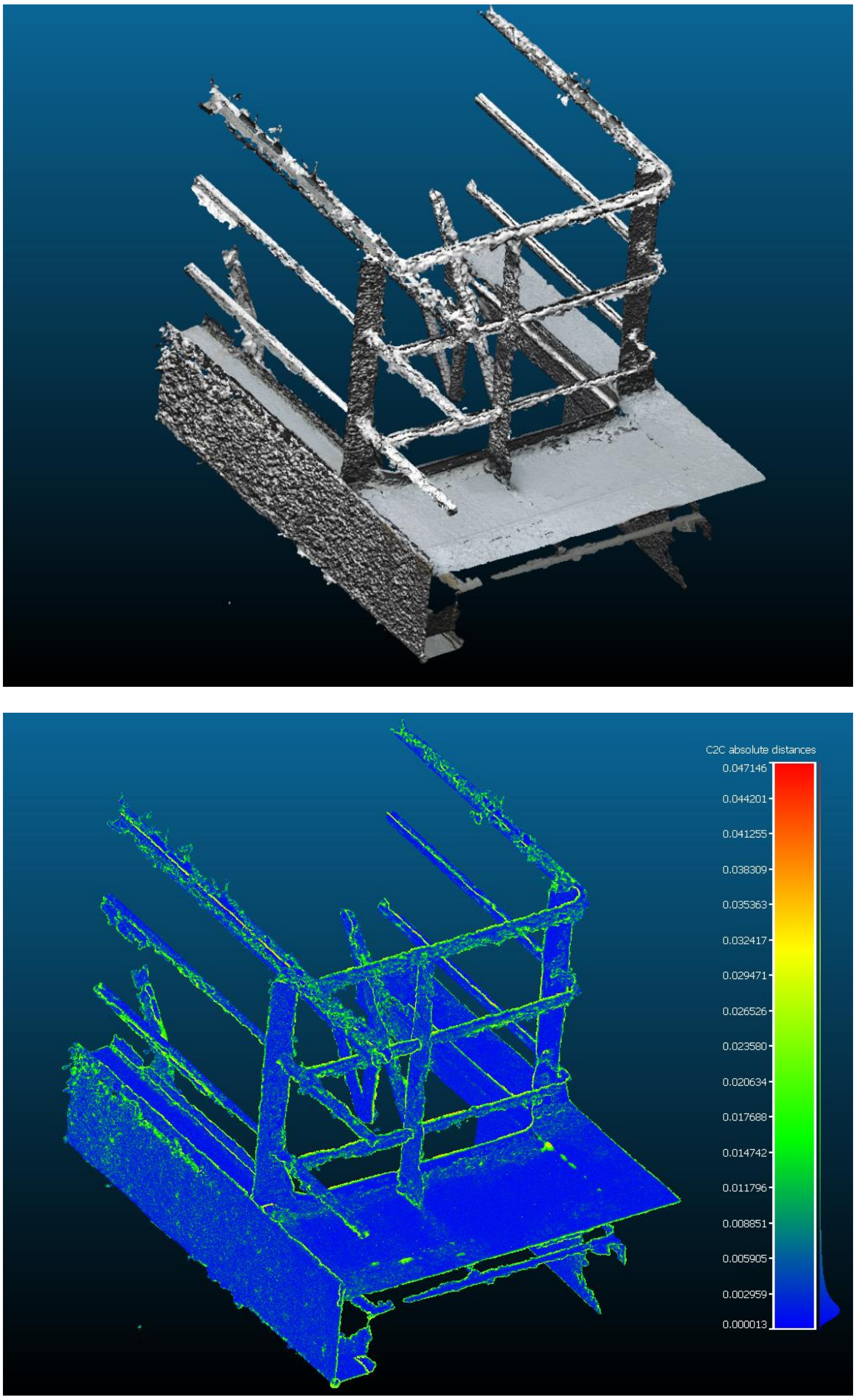


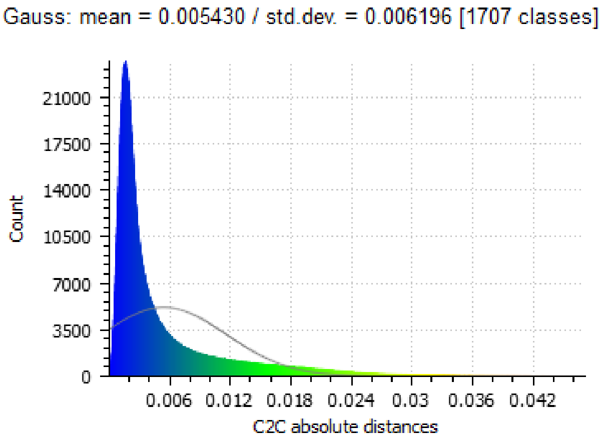
5.3.2.3 Naïve

The naïve method performed much better than the WLOP and MLS, when coupled with Poisson. The distances were considerably smaller compared to the original model, averaging 0.005, as shown on Figure 64, and the computation time was orders of magnitude faster, as Figure 66 demonstrates. Though the smoothness is not as high as the MLS version, with the dimpled sidewall being the telltale, the overall shape of the model is noticeably better. There are no depressions on the floor and the color is retained.

However, similarly to the naïve method with MC, the results were actually worse than the crude pipeline. The file size was the largest of all Poisson pipelines and the output presented more artifacts. It is interesting to note that the naïve method did simplify the pipeline, by reducing the complexity of calculating the normals, and reducing the pipeline computing time significantly, as seen on Figure 66 on page 86.

Figure 64: Naïve and Poisson



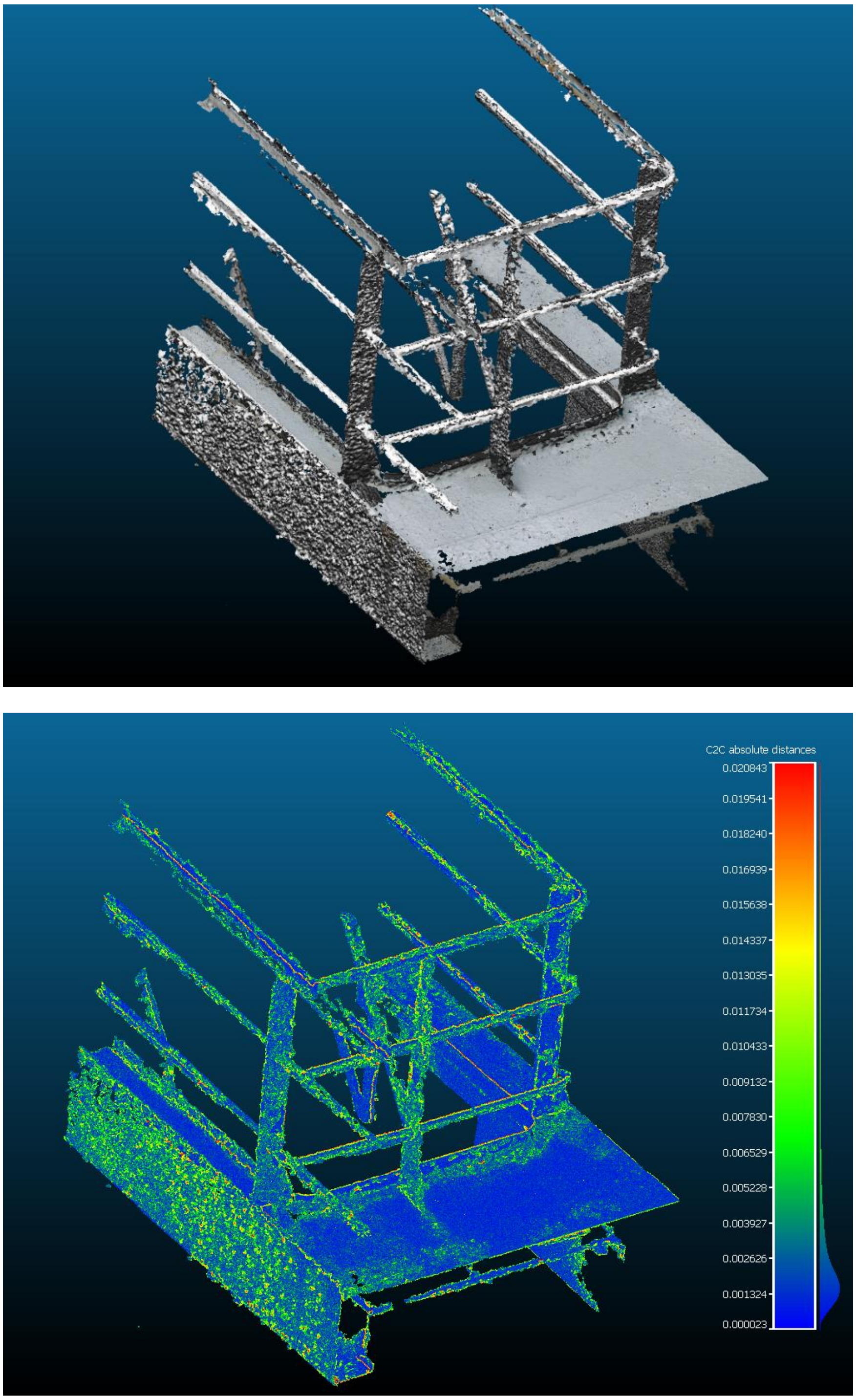


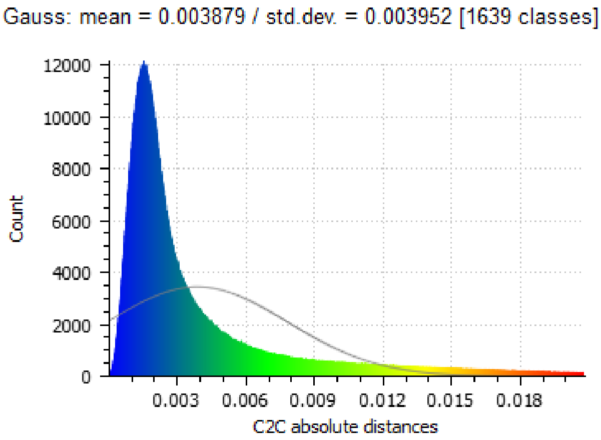
5.3.2.4 Crude

In the case of the Poisson pipelines, the crude output again presented the best results. Despite of exhibiting corrugated and dimpled sidewalls and railing posts, visible on Figure 65, this pipeline generated the smallest distances, averaging at 0.004. It also preserved colors and was able to remove the outlying cluster to the left of the sidewall.

However, it was not the fastest method, though. The naïve simplification helped to reduce the normals computation time, thus reflecting in a faster pipeline. Moreover, the output file size was nearly twice as that of the WLOP pipeline, taking up 132 MB and being the second largest output file.

Figure 65: Crude and Poisson

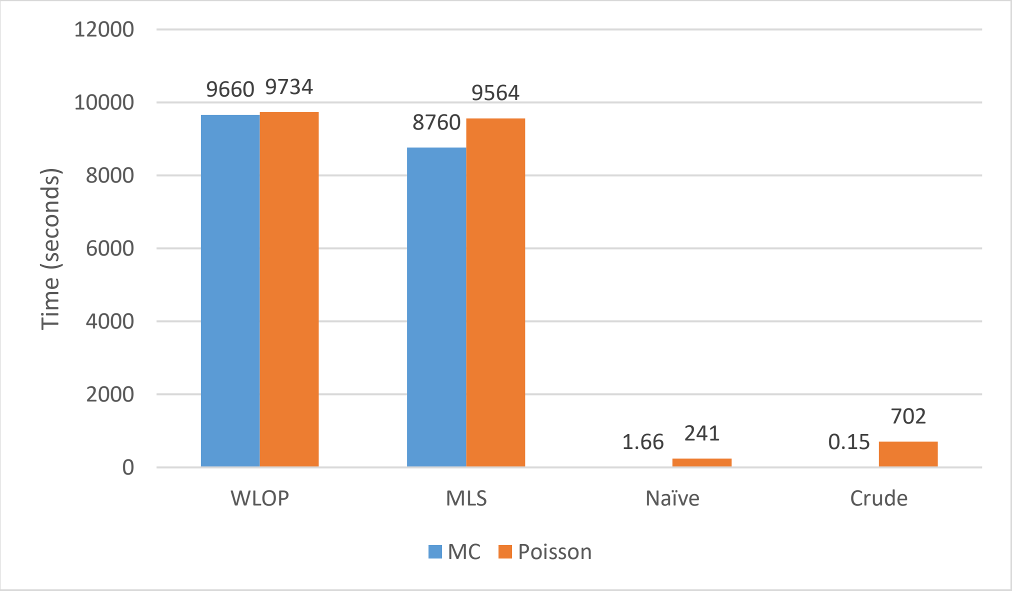




5.4 Pipelines Comparison

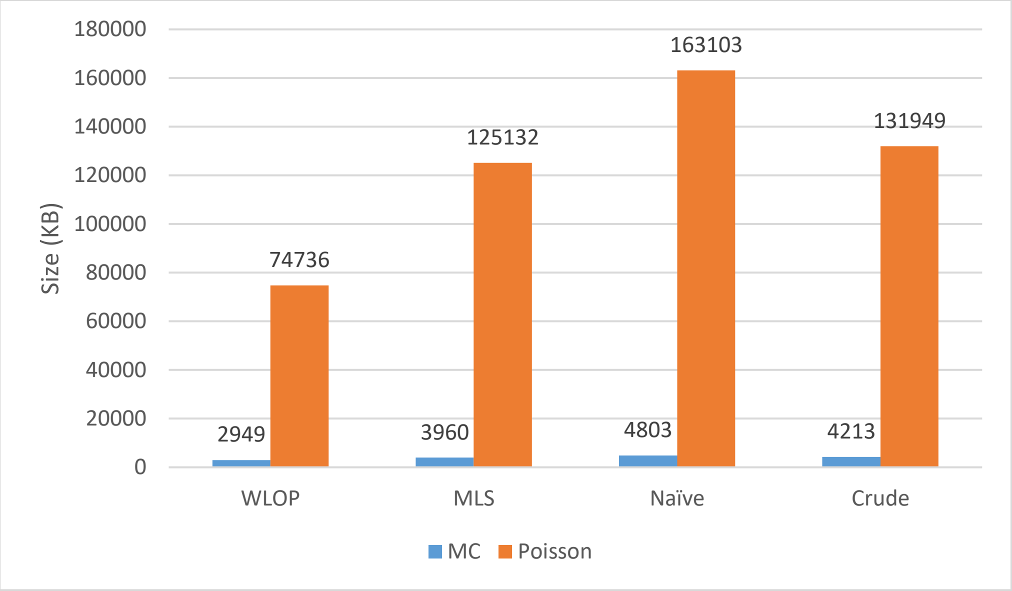
When comparing times, it is clear the impact that the preprocessing of the point cloud has. So much so that it shadows the pipelines that do not require intensive computation on the point cloud, which can be noticed on Figure 66. It is interesting to note that the naïve method did have a positive impact when coupled with Poisson, reducing the computing time three fold. The results were taken from an Intel Core i7 2.2GHz, with 12GB of RAM and four cores.

Figure 66: Computation time for the pipelines



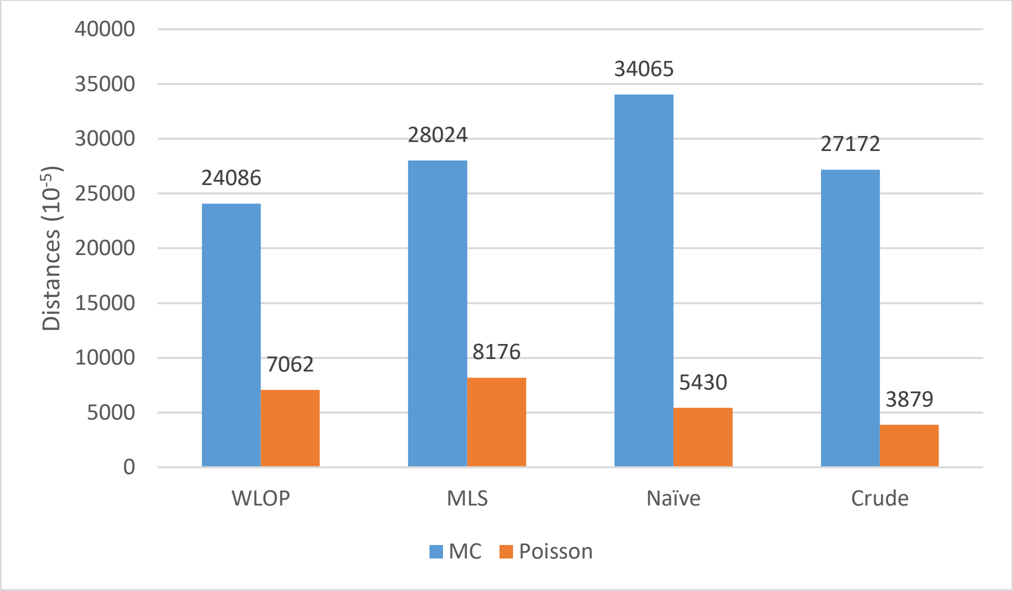
As noticed on Figure 67, it is clear that the output file sizes are in favor of the marching cubes pipelines. In cases such as Naïve + Poisson, the output file is over half the size of the original file, thus degrading Poisson pipelines' stance towards the objective of having small file sizes without engineering information loss.

Figure 67: File size for the pipelines



When analyzing the distances generated, Poisson pipelines are considerably better and fitting the original model much closer than the marching cubes. This is due to the precision of the 128^3 grid used, which generates smaller files, but fail to match the inherit surface properly.

Figure 68: Distance for the pipelines



6 Conclusion

In this work, different techniques for point cloud treatment and surface generation were tested separately on a point cloud to assess their individual strengths and weaknesses against a data set rich in artifacts such as noise, holes, and outliers. The point cloud was provided by APIteq, a company that specializes in oil and gas offshore installation visualization, and was generated based on stereoscopic images of an offshore installation, more specifically a support vessel's main deck. After all techniques were tested independently, they were combined into pipelines and again the results were assessed.

In this section, the results obtained from combining the different techniques will be assessed in terms of the objectives described in section 1.2. That is, mainly, we are trying to find a quick and efficient algorithm that can reduce the file size of the point cloud without losing information that might help in taking engineering decisions, such as dimensions, colors, and topology. In this section, the best solution, from the ones tested, to answer the question posed in section 1.3 will be identified from the set analyzed in the previous sections.

6.1 File Size

Assessing the file size reduction first, the results gathered from the comparisons were very enlightening and clear distinctions can be easily observed. There was a large discrepancy in file size output from using Poisson or MC, as seen on Figure 67 on page 87, with MC outputs averaging 3.2% the average size of Poisson. With file size output being a major driver for this project, MC performed considerably better, regardless of the pipeline used.

6.2 Efficiency

Analyzing the efficiency, another clear distinction on the results can be seen on the computing time. Figure 66 on page 86 shows two divisions: whether a point cloud treatment is used in the pipeline or not, and whether MC or Poisson is used as the meshing algorithm. That is, when WLOP or MLS is used prior to the mesh generation, there is a 40 fold increase in the average computing time, regardless of meshing algorithm. The average time for pipelines running WLOP or MLS was 9429.5 seconds, while the Naïve and Crude pipelines averaged 236.2 seconds. Furthermore, Poisson was consistently slower than MC, mainly due to the normals calculation, and that is visible on Figure 66 on page 86. However, to better grasp the time difference and comparing both methods without considering the point cloud treatment, it is easy to note that the average computing time for Poisson is much larger than that of MC, as seen when comparing Figure 52 and Figure 57, on page 63 and 69, respectively. The time scale for Poisson was in the order of seconds, as opposed to the milliseconds scale for MC.

Therefore, crude point cloud combined with MC performed more satisfactory than any other pipeline. This conclusion further enhances the MC position as the meshing algorithm of choice, as well as it detriments pipelines containing WLOP or MLS as constituents for a final solution.

6.3 Fidelity

To assess how much information was lost, Figure 68 on page 88 shows that Poisson kept truer to the original data set with consistently lower distances. Furthermore, by analyzing section 5.3.2, it is also noticeable that the standard deviation of the distances is much tighter than the ones exhibited by MC on section 5.3.1.

It is interesting to note that Poisson models the floors and sidewalls as a single plane, which is clean and representative, however loses thickness information of the steel plates. Because of this single plane nature, the normals also get inverted in some cases. This can be noticed as a slight difference in color on the WLOP + Poisson pipeline, which does not retain color, and therefore outputs the normals encoded in the surface color, as seen on Figure 62 on page 79.

6.4 Pipeline Identification

With all assessments considered, the best pipeline from the ones analyzed can be identified. Both the file size and efficiency assessments point to the MC meshing algorithm as the best fit to the objectives of this thesis. However, the efficiency assessment further restricts the pipeline by demonstrating how much longer a pipeline containing WLOP or MLS takes to execute. That means that either the Naïve or Crude approaches would be a good combination, but the Naïve approach did not perform any better in terms of computing time, therefore rendering the Crude + MC the best solution from the ones analyzed.

Furthermore, despite of the lower distances with Poisson, MC still presents an honest representation of the original model, without normals inversion, preserving the thickness of planes, and not exhibiting the flaring characteristics noticed on Poisson reconstructions. However, MC, as it stands, does not preserve color information. This is can be easily considered and added to future versions of the algorithm.

6.5 Solution

Finally, to answer the question posed in section 1.3, we can affirm that it is indeed possible to reduce a stereoscopically reconstructed point cloud of an industrial plant in an efficient manner, with minimal engineering information loss. We were able to reduce the original point cloud from 290MB to 4.2MB in 0.15 seconds, retaining all the measurements and topology of the model with minimal variation.

However, color information was lost as it stands. Nevertheless, as proposed in section 8, this can simply be remediated, thus improving significantly the outcome.

Furthermore, section 7 will expand this conclusion and validate the pipeline against real-world point clouds.

7 Validation

To validate the solution proposed in this work, the selected pipeline was applied to a larger point cloud and, finally, to the complete data set representing the full working deck of a support vessel. This is made to reflect a real-world scenario, as these point clouds are full products delivered by APIteq, the company mentioned in section 1.2 that provided the data sets for this work.

First, a 1.7GB point cloud representing a cable spool was passed through the pipeline. Figure 69 shows the original point cloud and the railing used to test and compare can be seen on the right-hand since of the image. Figure 70 shows the resulting mesh from the pipeline; a 192×192×192 grid generated the 18.6MB surface in 0.55 seconds. The grid was slightly larger to reflect the larger volume that the spool comprised. Figure 71 shows the calculated distances from the original point cloud with an average distance of 0.052.

Figure 69: Cable spool point cloud

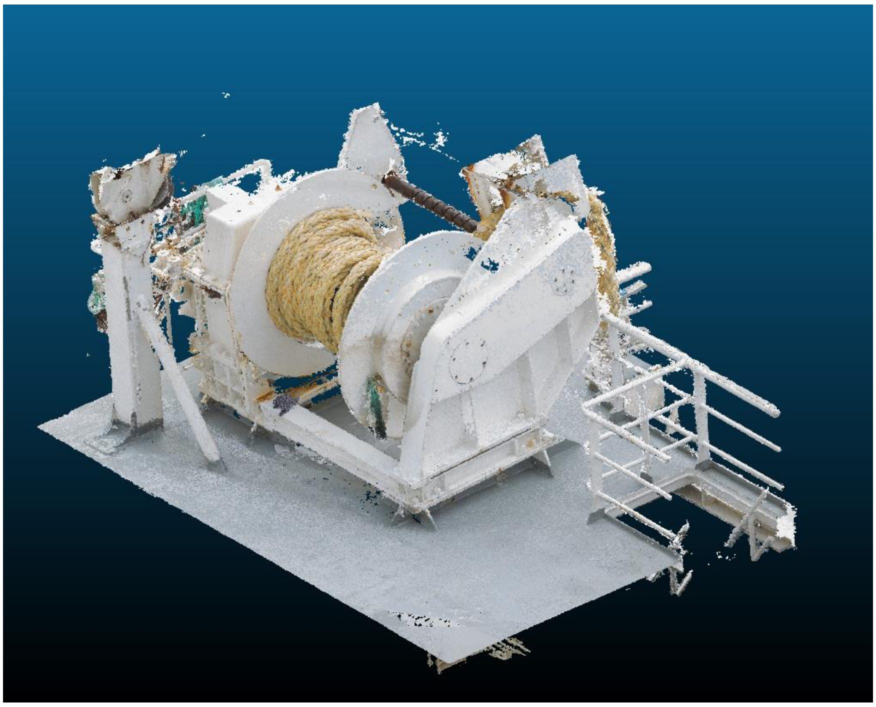


Figure 70: Cable spool mesh

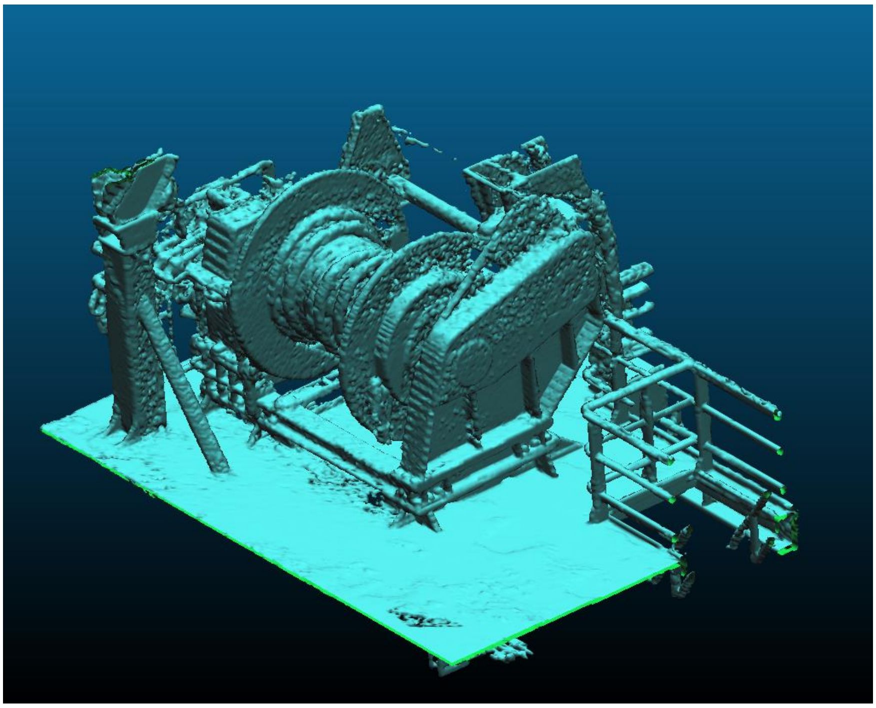
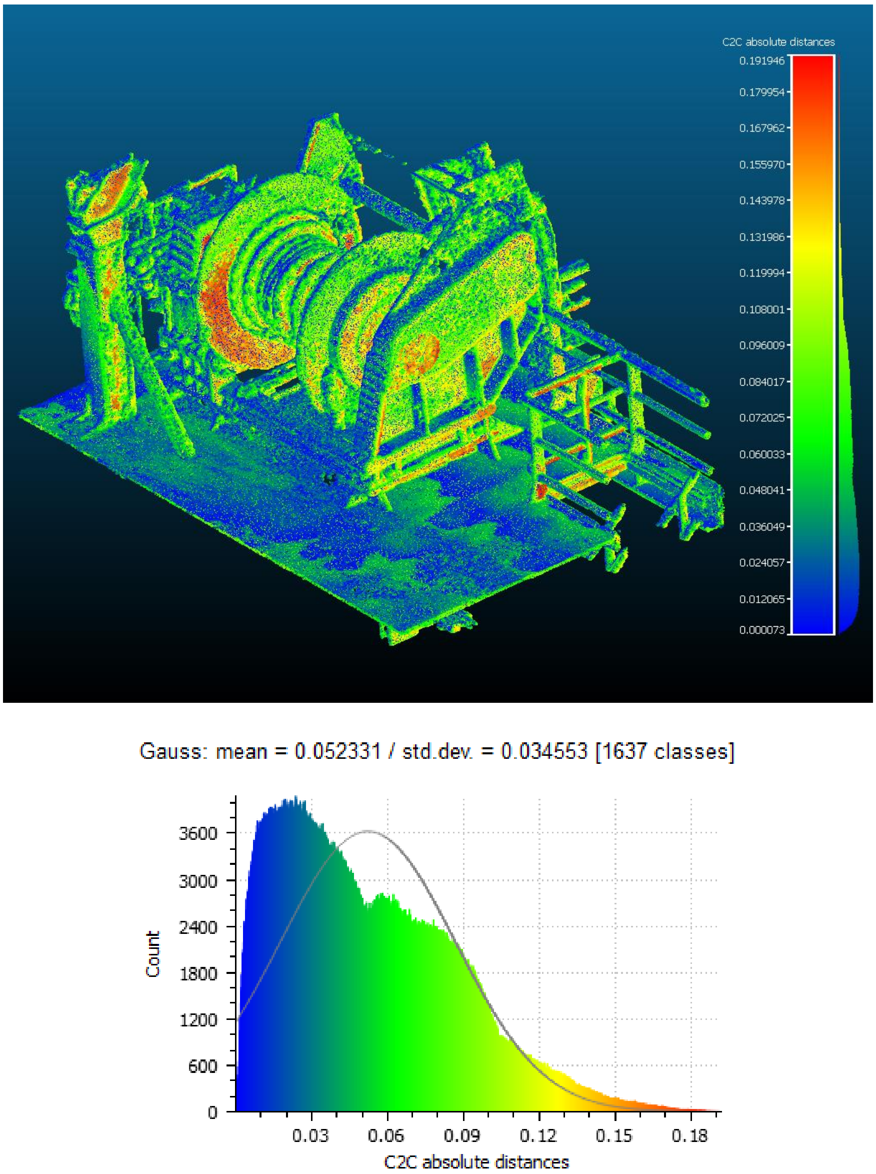


Figure 71: Cable spool distances



Finally, a 4.6GB point cloud representing the complete support vessel was passed through the pipeline. Figure 72 shows the original point and Figure 73 shows the resulting mesh from the pipeline. For this point cloud, a 256×512×256 grid was used, resulting in an 80.9MB file. Notice the non-uniform and larger dimensions of the grid used to reflect the elongated shape of the vessel's deck and larger volume. This more complex shape and file took indeed longer to be processed, taking 2.44 seconds to finalize. Figure 74 shows the calculated distances from the original point cloud with an average distance of 0.081.

Figure 72: Support vessel deck point cloud

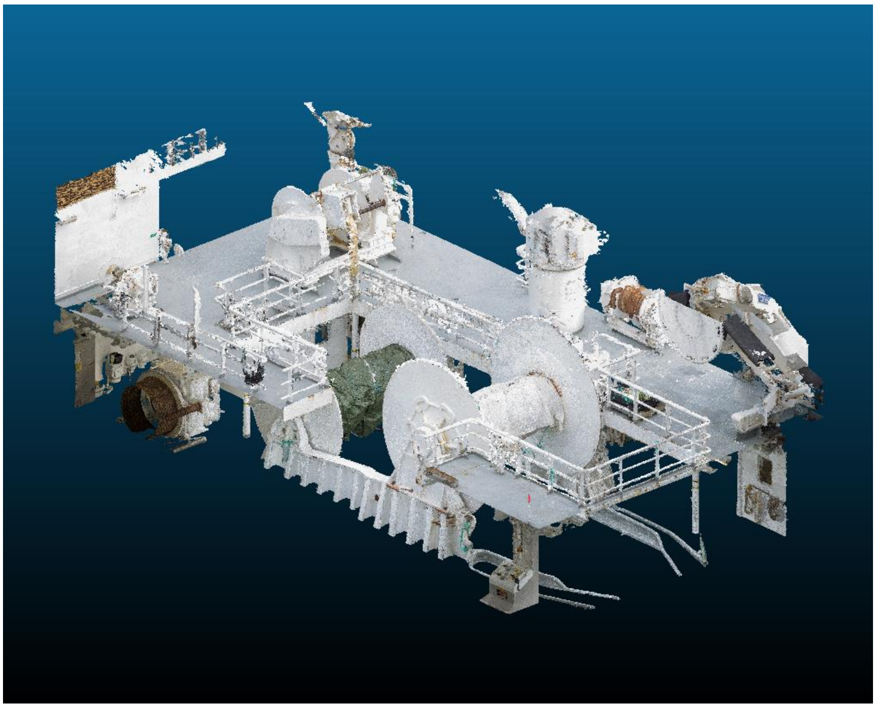


Figure 73: Support vessel deck mesh

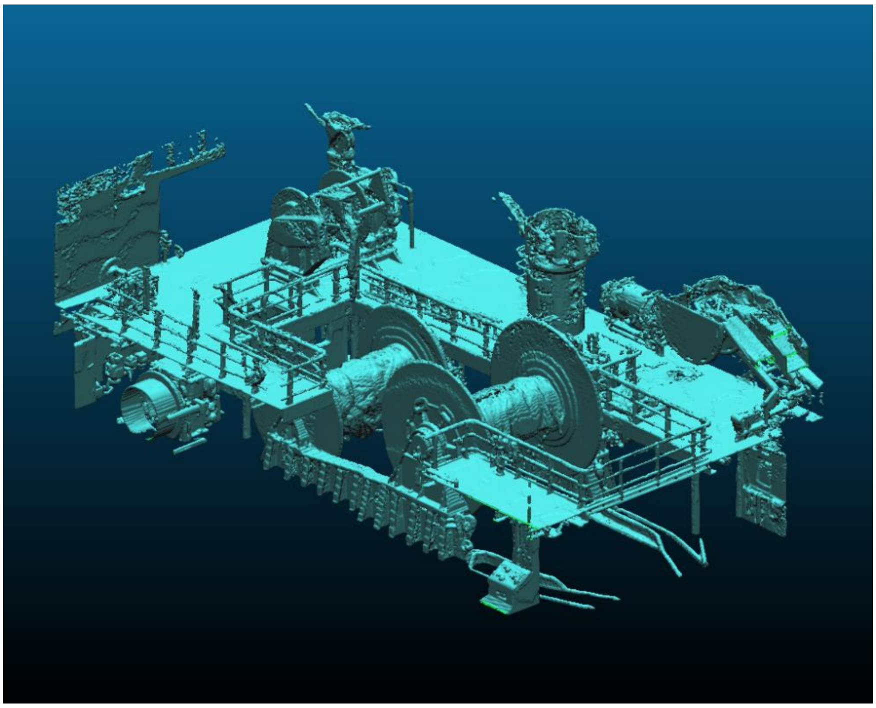
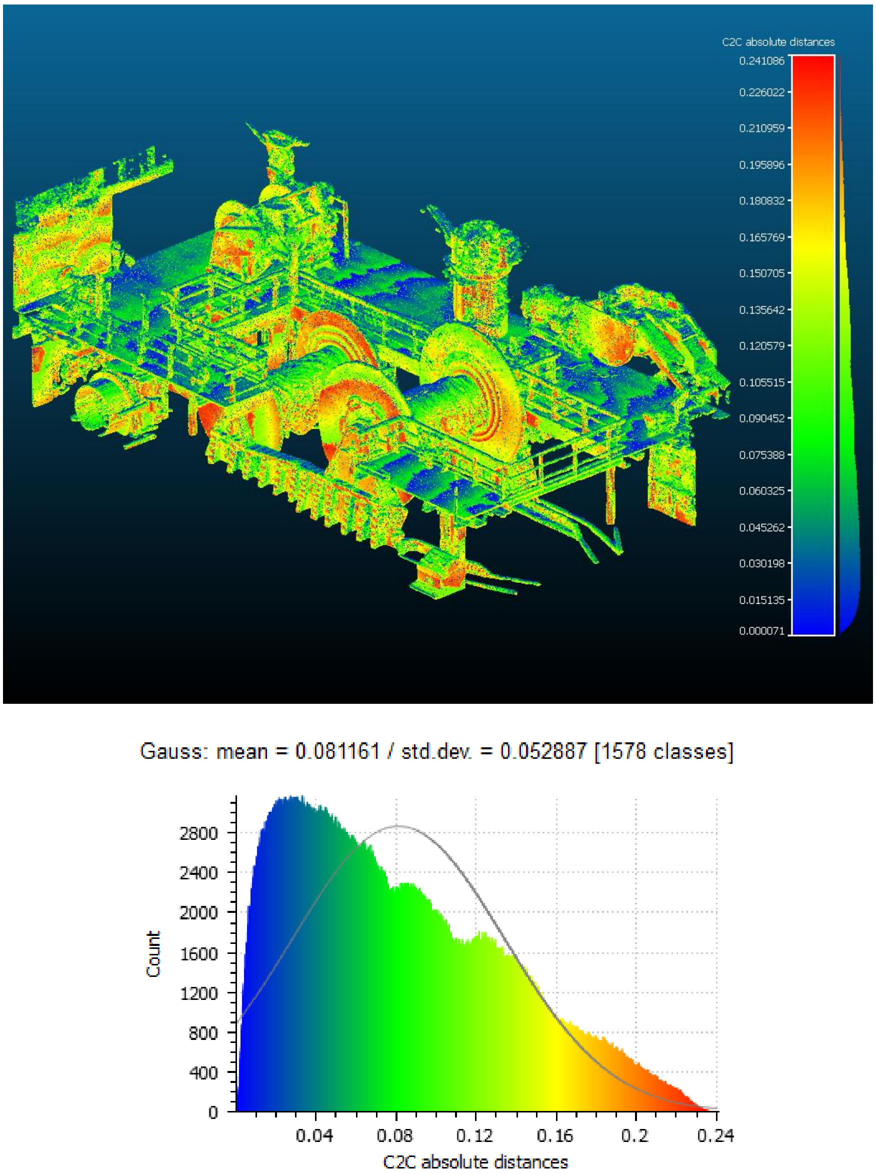


Figure 74: Support vessel deck distances



8 Further work

From the techniques analyzed in this thesis, we were able to identify the best solution to answer the questions proposed in section 1.3. However, it fell short on some aspects, which will be discussed in this section.

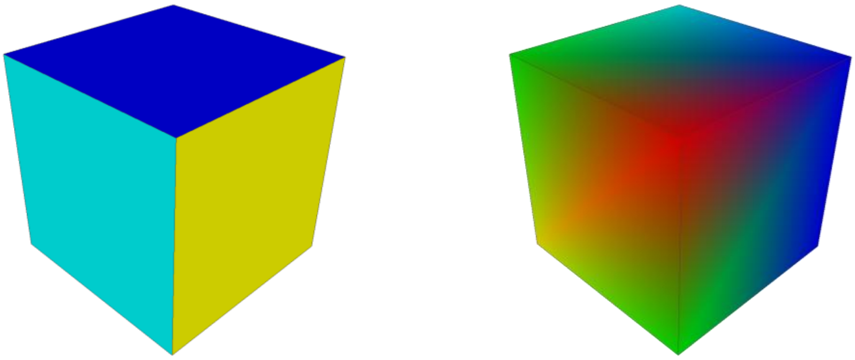
8.1 Color Information

Considering the importance of color information to the final result and how it could potentially affect engineering decisions, the main goal for the future is to integrate this into the MC algorithm. Considering that the meshes are represented in PLY format, color information can be added to the output file easily in terms of encoding, since it is described in the specification. As such, according to the specification, color can optionally be associated with each vertex of the mesh.

To retain the color information, the process has to be modified from the gridifcation phase, described in section 4.3.1.1, and the vertex generation phase, described in section 4.3.1.2. That means that, when calculating the density of a cell, the average color of all the points within the cell would also be computed and stored in that cell. Once the vertex generation phase begins, the color associated with it would then be the interpolated values of the grid cells according to its location.

By encoding the color information with the vertices instead of the faces, the rendering software is able to interpolate the color of the faces and therefore obtain a smoother coloring and a more reliable mesh. When the color is encoded in the face, the whole face will contain the same color, giving raise to sharp edges in the coloring. Figure 75 shows a comparison of the two methods; on the left the colors were encoded with the face, and on the right they were encoded with the vertices.

Figure 75: Comparison between per face and per vertex coloring



8.2 Parallelization

To further increase the speed and efficiency of the algorithm, it would be desirable to port all the phases of MC into OpenCL. So far, only the gridification was implemented in OpenCL, with a snippet seen in Code 2 on page 35, but according to section 2.7, there is much to gain with a complete pipeline executed in the GPGPU.

The main challenge with the porting is memory management. The movement to and from the GPGPU memory is costly and there is usually much less memory to work with than the main CPU. This coupled with the addition of color information can quickly consume all the available memory and force loading and offloading of sections of the data set. Hence the importance of proper memory management to effectively harness the power of the GPGPU.

8.3 Graphical User Interface

Finally, a GUI (graphical user interface) would increase the ease of use of the software immensely. So far, the only interface is CLI (command-line interface) and it is prone to mistyping and misconfiguration. A GUI would provide the user with all the options and parameters together with a more intuitive way of loading files.

Given that the software is written in C++, QT [28] would be a good solution. Not only providing GUI widgets, it also provides load and save dialogs as well as a wrapper for OpenGL, allowing for the user to visualize the output, and choosing to change a parameter before saving

9 References

[1] G. Pang, R. Qiu, J. Huang, S. You and U. Neumann, "Automatic 3D industrial point cloud modeling and recognition," in *IAPR International Conference on Machine Vision Applications*, Tokyo, 2015.

[2] T. Rabbani Shah, "Automatic reconstruction of industrial installations: Using point clouds and images," Nederlandse Commissie voor Geodesie, 2006.

[3] M. Berger, A. Tagliasacchi, L. M. Seversky, P. Alliez, J. A. Levine, A. Sharf and C. T. Silva, "State of the Art in Surface Reconstruction from Point Clouds," *Eurographics,* 2014.

[4] T. A. S. f. P. &. R. Sensing, LAS Specification, Bethesda: The American Society for Photogrammetry & Remote Sensing, 2013.

[5] Y. Lipman, D. Cohen-Or, D. Levin and H. Tal-Ezer, "Parameterization-free Projection for Geometry Reconstruction," *ACM Transactions on Graphics,* vol. 26, no. 3, 2007.

[6] H. Huang, D. Li, H. Zhang, U. Ascher and D. Cohen-Or, "Consolidation of Unorganized Point Clouds for Surface Reconstruction," *ACM Transactions on Graphics,* vol. 28, no. 5, 2009.

[7] M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, D. Levin and C. T. Silva, "Computing and Rendering Point Set Surfaces," *IEEE TRANSACTIONS ON VISUALIZATION AND COMPUTER GRAPHICS,* vol. 9, no. 1, March 2003.

[8] D. Levin, "Mesh-independent Surface Interpolation," *Geometric modeling for scientific visualization,* pp. 37-49, 2004.

[9] L. S. Tekumalla and E. Cohen, "Smoothing Space Curves with the MLS Projection," in *Proceedings of the 1999 symposium on Interactive 3D graphics*, 1999.

[10] G. Tavares, R. Santos, H. Lopes, T. Lewiner and A. Viera, "Topological reconstruction of oil reservoirs from seismic surfaces," *International Association for Mathematical Geology,* September 2003.

[11] Jam, J. Davis, S. R. Marschener, M. Garr and M. Levoy, "Filling Holes in Complex Surfaces using Volumetric Diffusion," *3D Data Processing Visualization and Transmission,* pp. 428-441, 19 June 2002.

[12] W. E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3D surface construction algorithm," *SIGGRAPH,* pp. 163-169, 1987.

[13] S. Scott and J. Warren, "Dual marching cubes: Primal contouring of dual grids," *Computer Graphics and Applications,* pp. 70-76, 6 October 2004.

[14] T. Lewiner, H. Lopes, A. Vieira and G. Tavares, "Efficient Implementation of Marching Cubes Cases with Topological Guarantees," *Journal of Graphics Tools,* vol. 8, no. 2, pp. 1-15, 2003.

[15] M. Kazhdan, M. Bolitho and H. Hoppe, "Poisson Surface Reconstruction," in *Eurographics Symposium on Geometry Processing*, Cagliari, Sardinia, Italy, 2006.

[16] M. M. Kazhdan, "Reconstruction of solid models from oriented point sets," *Symposium on Geometry Processing,* pp. 73-82, 4 July 2005.

[17] W. Sierpinski, "Sur une courbe dont tout point est un point de ramification," *Comptes rendus hebdomadaires des séances de l'Académie des Sciences,* vol. 160, pp. 302-305, 1915.

[18 T. Hammervoll, *Efficient rendering of parametric CAD*
] *data using geometry instancing,* Bergen: Høgskolen I Bergen, 2016.

[19 R. Qiu, Q.-Y. Zhou and U. Neumann, "Pipe-run Extraction
] and Reconstruction from Point Clouds," in *European Conference on Computer Vision*, Zurich, 2014.

[20] P. Cignoni, M. Callieri, M. Corsini, M. Dellepiane, F. Ganovelli and G. Ranzuglia, "MeshLab: an Open-Source Mesh Processing Tool," *Sixth Eurographics Italian Chapter Conference,* pp. 129-136, 2008.

[21] D. Girardeau-Montaut, "CloudCompare," [Online]. Available: http://cloudcompare.org/.

[22] C. E. Board, "CGAL," CGAL Editorial Board, November 2016. [Online]. Available: http://www.cgal.org/.

[23] P. Lancaster and K. Salkauskas, "Surface Generated by Moving Least Squares Methods," *Mathematics of Computation,* vol. 37, no. 155, pp. 141-158, 1981.

[24] E. Smistad, *Real-Time Surface Extraction and Visualization of Medical Images using OpenCL and GPUs,* 2012, pp. 141-152.

[25] K. O. W. Group, The OpenCL Specification, 2012.

[26] W. Technologies, *Advanced Visualizer Manual,* p. Appendix B1.

[27] P. Bourke, *Ply-polygon file format,* 2009.

[28] Trolltech, Nokia and QT Project, "QT," [Online]. Available: https://www.qt.io/.

[29] Various, "Wikipedia - Normal Distribution," [Online]. Available: https://en.wikipedia.org/wiki/Normal_distribution. [Accessed 15 May 2017].