**Stereoscopic rendering of 3D point clouds**

**Hayden Marshall**

2106822



A thesis submitted for the degree of

**Master of Science in Computer Engineering**

at the

School of Computer Science and Electronic Engineering Engineering

Supervisor: Dr Adrian Clark

University of Essex

December 2022

**Abstract**

This thesis hopes to tackle the problem of rendering massive point clouds in real-time. It will explore various methods for processing and rendering massive point clouds consisting of between 10 million and 1 billion data points. It will look at ways to optimize performance by methods such as view culling and OpenGL functionality to improve the rendering pipeline. The thesis culminates in designing and implementing a point cloud visualiser from scratch based on the current research in the field. It follows the process from scanning point cloud data all the way to rendering it on your display, discussing and proposing ways to deal with possible problems for point cloud visualisation. A unique PLY parser was developed to parse point cloud PLY data that was 85%+ better than another competing PLY parser for reading point cloud data while other aspects of improving were explored in the form of stereoscopic rendering providing a proof of concept that could be taken further for VR headset viewing.

**Acknowledgements**

I wish to thank Dr Adrian Clark for being my supervisor for this project while providing the LaTeX template for this report and a selection of PLY point cloud files. He has also played a key role in getting myself to this point where I can finally say I will hopefully acquire a degree.

I also wish to thank Dr David Richerby for providing myself with a GLA job to put food on the table and giving up his time to give advice on pursing a PhD.

**Contents**

[**1 Introduction** **1**](#_Toc60576)

[1.1 Problem Definition 1](#_Toc60577)

[1.2 Scope 2](#_Toc60578)

[1.3 Work undertaken 3](#_Toc60579)

[1.4 Structure of work 4](#_Toc60580)

[**2 Literature review** **5**](#_Toc60581)

[2.1 Handling massive point cloud data 5](#_Toc60582)

[2.1.1 Rendering issues caused from poor data quality 5](#_Toc60583)

[2.1.2 3D scanners 7](#_Toc60584)

[2.1.3 Structure from motion 8](#_Toc60585)

[2.1.4 Data storage 9](#_Toc60586)

[2.1.5 Pre-processing point cloud data 10](#_Toc60587)

[2.2 Point cloud rendering 11](#_Toc60588)

[2.2.1 Triangle vs point based rendering 12](#_Toc60589)

[2.2.2 Data structures 13](#_Toc60590)

[2.2.3 Octree 13](#_Toc60591)

[2.2.4 Point scaling 14](#_Toc60592)

[2.3 Rendering optimizations 14](#_Toc60593)

[2.3.1 Level of Detail (LOD) 14](#_Toc60594)

[2.3.2 Subsampling 14](#_Toc60595)

[2.4 Stereoscopic rendering 15](#_Toc60596)

[**3 Methodology** **19**](#_Toc60597)

[3.1 Transforming vertex data 26](#_Toc60598)

[3.2 Point cloud data parsing 27](#_Toc60599)

[3.2.1 Octree frustum culling 28](#_Toc60600)

[3.3 Project management 29](#_Toc60601)

[**4 Testing** **31**](#_Toc60602)

[4.1 PLY point cloud parsers 32](#_Toc60603)

[4.1.1 Test scenarios 32](#_Toc60604)

[4.1.2 Results 34](#_Toc60605)

[4.2 Rendering performance 37](#_Toc60606)

[4.2.1 Mono vs Stereo vision 37](#_Toc60607)

[4.2.2 Frustum culling 38](#_Toc60608)

[4.2.3 Octree effect on performance 39](#_Toc60609)

[4.2.4 Point size 41](#_Toc60610)

[**5 Conclusion** **48**](#_Toc60611)

[5.1 Future work 48](#_Toc60612)

[**A Appendix** **54**](#_Toc60613)

[A.1 Test 4 PLY parser graphs 54](#_Toc60614)

**Chapter 1**

# Introduction

One of the oldest problems in the field of computer graphics is how to represent and display virtual objects in the most efficient and realistic manner possible. Polygons have been the cornerstone to construct three-dimensional objects since the beginning of real-time computer graphics. However, they are by no means the only geometric primitive that can be used to do so. Point-based geometry has often been overlooked as a viable alternative for surface representation, both for efficient rendering and flexible geometric processing of complex 3D-models. This thesis hopes to explore the potential benefits of point-based rendering for visualising these 3D surface models.

3D Point-based surface models often consist of large collections of points known as point clouds. Point clouds are three-dimensional models that consist of a collection of unconnected data points in three-dimensional space. They are most often created by using a scanning method, such as laser scanners or photogrammetry techniques to map the surrounding physical world. Point clouds are being used in a wide range of areas such as fast object detection for convolutional neural networks [1], autonomous driving [2] [3], heritage site preservation [4] [5] and mapping changes to forests and other geographical landscapes [6].

However, the point cloud models produced by this process can vary a lot. They can range from a few thousand up to billions or trillions of points, depending on the scan resolution and scanned area. For example, one of the largest point clouds publicly available is published by the United States Geological Survey (USGS) [7]. This survey produced a point cloud covering the entire breadth of the north american continent which resulted in having around 27 trillion points [8]. As you can imagine, this requires roughly 540 terabytes of storage space [8] and poses a heavy burden on the rendering process to visualise the entire data set.

This thesis hopes to tackle this problem of rendering massive point clouds in the range of 10 million to 1 billion points in size.

## 1.1 Problem Definition

Real-time rendering of large point clouds is notoriously difficult due to the large amount of data needed to be visualised within a heavily constrained amount of time. Current methods of visualising point clouds consisting of 500 billion points or more involve a process called pre-rendering. Pre-rendering is the process in which video footage is not rendered in realtime. This means that the video footage is rendered and recorded but not displayed on any display until it has rendered the entire scene. After the pre-rendering process is complete, the

1

*CHAPTER 1. INTRODUCTION*

recorded footage can be played back. The main appeal of pre-rendering is that you can render scenes that have extreme performance requirements such as rendering trillions of points of data a frame which are not able to be viewed in real-time.

However, there are a couple of drawbacks with this method. A user has no control over a scenes projected view after the rendering process has started. In order for a person to move around in a virtual scene, the movement and camera position has to be planned in advance. If a mistake is made, it will only be known after the process has completed and the footage viewed back. This can be very problematic as scenes can take days or weeks to render and making a mistake could set someone back a significant amount of time. The process would have to be restarted from scratch with amendments made to the defined camera movement and positions. Therefore, pre-rendering takes up a lot of time and resources to offer a very rigid method of rendering a scene.

This is where real-time rendering comes into play. Real-time rendering, as the name suggests, renders a scene in real-time. This is commonly between 33.3ms to 16.6ms per a frame (30-60 frames per a second [9]). Unlike pre-rendering, real-time rendering has the advantage of allowing for quick inspection of point clouds. A point cloud can be viewed on demand within a couple of minutes and if the view point is not how the user would like, they are able to change it on the fly. This helps to significantly reduce the amount of time to view a point cloud and allows for greater interactivity and control over the scene’s projection providing a greater sense of immersion and user satisfaction.

The issue with real-time rendering is that it imposes a time constraint on the real-time rendering process. For example, 60 frames per a second (FPS) is a very common goal for many gamers to run video games at. If 60 frames are produces in a second then that means that each frame is being produced within 16.6ms which compared to pre-rendering which can take days or weeks to complete is a drastic difference in time scale.

There is an increasing demand for quick and accessible point cloud creation and visualisation in a variety of fields such as indoor mapping, surveillance, robotics, and forensics to capture movement and poses [10]. One key advantage of visualising point cloud data is that it is much easier and quicker to see if the collected data is correct e.g if a robot is seeing the world around it correctly).

This thesis hopes to tackle this problem by exploring current research and literature to produce a unique custom-built solution to visualise point clouds in real-time. Interactivity and immersive techniques such as stereo will be looked into as well to improve user satisfaction.

## 1.2 Scope

The main goals of the thesis were:

* To build an OpenGL render capable of rendering point clouds in the range of 10,000 to 1 million points
* To allow a point cloud in PLY format to be visualised on demand
* To implement at least one optimization algorithms (frustum culling)
* To develop a stereoscopic capable visualiser to render point clouds in the range of 10,000 to 1 million points

## *1.3. WORK UNDERTAKEN* 3

* To develop a point cloud visualiser capable of rendering massive point cloud data sets of up to 100 million points in real-time
* To create an visualiser with interactive controls that allow a point cloud to be viewed from different view points

The limitations of the thesis were: It does not cover the acquisition of point cloud data through LiDAR or structure from motion. It does talk in depth about how these methods collect data as it is vitally important to understand how it effects the data used for the visualisation process and can explain why certain images may not look the way they are expected too.

The thesis is open ended in scope as the rendering performance can always be improved to some degree as long as there are advancements in the field of point cloud rendering. The main limitation therefore is the time available to do the thesis. This has meant certain desirable features such as filtering, registration or further optimization techniques would have to be left to the future work section so to not run out of time before the deadline. Another limitation is that the thesis has to be built from scratch so not pre-existing visualisers or libraries can be used as a base to visualise point clouds. This is the main challenge posed by the thesis to implement my own ideas and research from literature to analyse and research what is currently available in the field.

There is also the limitation of getting access to the necessary hardware to parse point clouds in the hundreds of millions of points. For example, if a point cloud consists of 300 million points with each vertex taking up 27 bytes of storage, to parse all the points into RAM would roughly require 7.5GB of memory. This isn’t always available or there may be other applications taking up significant memory space as happens to be the case with windows so therefore can be a limiting factor. It is also possible to transfer the vertex data to the GPU which VRAM is far rarer to find available on most consumer grade PCs. Low end GPU’s (Integrated graphics) do not have sufficient memory to store point clouds in VRAM which can affect the rendering pipeline. This will be discussed later in the thesis.

**1.3 Work undertaken**

Over the course of the thesis, a number of distinct achievements were made. These were:

* Creating a custom PLY parser for the PLY files Dr Adrian Clark supplied that was between 85-90% faster than the [hapPLY](https://github.com/nmwsharp/happly) PLY parser on GitHub.
* Created a point cloud visualiser from scratch using C++ and OpenGL.
* Implemented a number of optimization techniques were implemented to improve performance low end systems such as frustum culling with an octree based approach, level of detail and exploiting the vertex buffer memory management system.
* Created a proof of concept stereo implementation with a split view approach to further expand towards VR compatibility.
* Implemented an interactive camera to view point cloud scenes from different vantage points.

*CHAPTER 1. INTRODUCTION*

* Achieved real-time performance (10+ frames per a second) for point clouds in excess of 95 million points!

The end result was a point cloud visualiser that was built from scratch capable of rendering point clouds of up to 100 million points in real-time.

## 1.4 Structure of work

The rest of the paper is structured as follows: Chapter 2 provides a literature review covering areas such as similar point cloud visualisers and the current approaches to handling and rendering massive point cloud data sets. Chapter 3 covers the main methodology comprised of the reasoning for the design and implementation of the proposed point cloud visualiser. Chapter 4 covers the testing of the thesis to ensure projects goals were met and the visualiser functions as intended. Lastly, chapter 5 provides an evaluation of the work undertaken and suggests future work for the reader.

**Chapter 2**

# Literature review

This chapter will cover all the fundamental background knowledge required to understand the research topic and appreciate the previous and ongoing research within the field. This includes research and projects involving the rendering of massive point cloud data sets, as well as methods for improving the quality of point-based rendering.

## 2.1 Handling massive point cloud data

Before considering how to approach rendering point clouds, we first need to consider the origin of the data that’s going to be processed and rendered to the screen. Point clouds are most often obtained by either using 3D scanners or structure from motion [11] to reconstruct areas of the real world. 3D scanners, such as LiDAR or the Kinect, generate point data by measuring the time of flight of a projected laser, whereas structure from motion estimates the 3D structure of a scene based on an array of overlapping 2D images. The type of capture method chosen and the manner in which it is carried out can have a huge impact on the quality of the data generated. Poor data quality can negatively affect the pre-processing and rendering pipeline introducing irregularities and inaccuracies into the final rendered scene that does not reflect the original scanned 3D model. Therefore, it is critical to be aware of the effects poor data quality can have on the final rendered scene to be able to verify if what we are viewing is an accurate representation of the captured point cloud model. The effects of poor data quality will be discussed thoroughly for the next few pages and throughout the remaining literature review where relevant.

### 2.1.1 Rendering issues caused from poor data quality

The following paragraphs describe common issues caused by poor data collection for visualising point cloud data.

**Non-uniform point cloud distributions** are point clouds that do not have an even distribution of points along the surface area of a model. We can see an example and effect of a non-uniform point cloud distribution in figure 2.1 image (f) where the point cloud data is lacking information in certain areas such as the bottom left and right hand corners of the image while having significantly more points mapping the road intersection ahead. A lack of information can cause surfaces like the side of the road in image (f) to have large holes in the visualised data or to have so few data points that the renderer is unable to display

5

enough points to accurately visualise the original surface at all. This creates large holes or dark patches in the visualised images reducing the overall perceived image quality and ability to make out the surface detail of a model of the image. However, there are several hole filling algorithms [12] [13] [14] available that hope to fix this visual problem by altering the rendered point sizes which will be discussed later in the literature review.

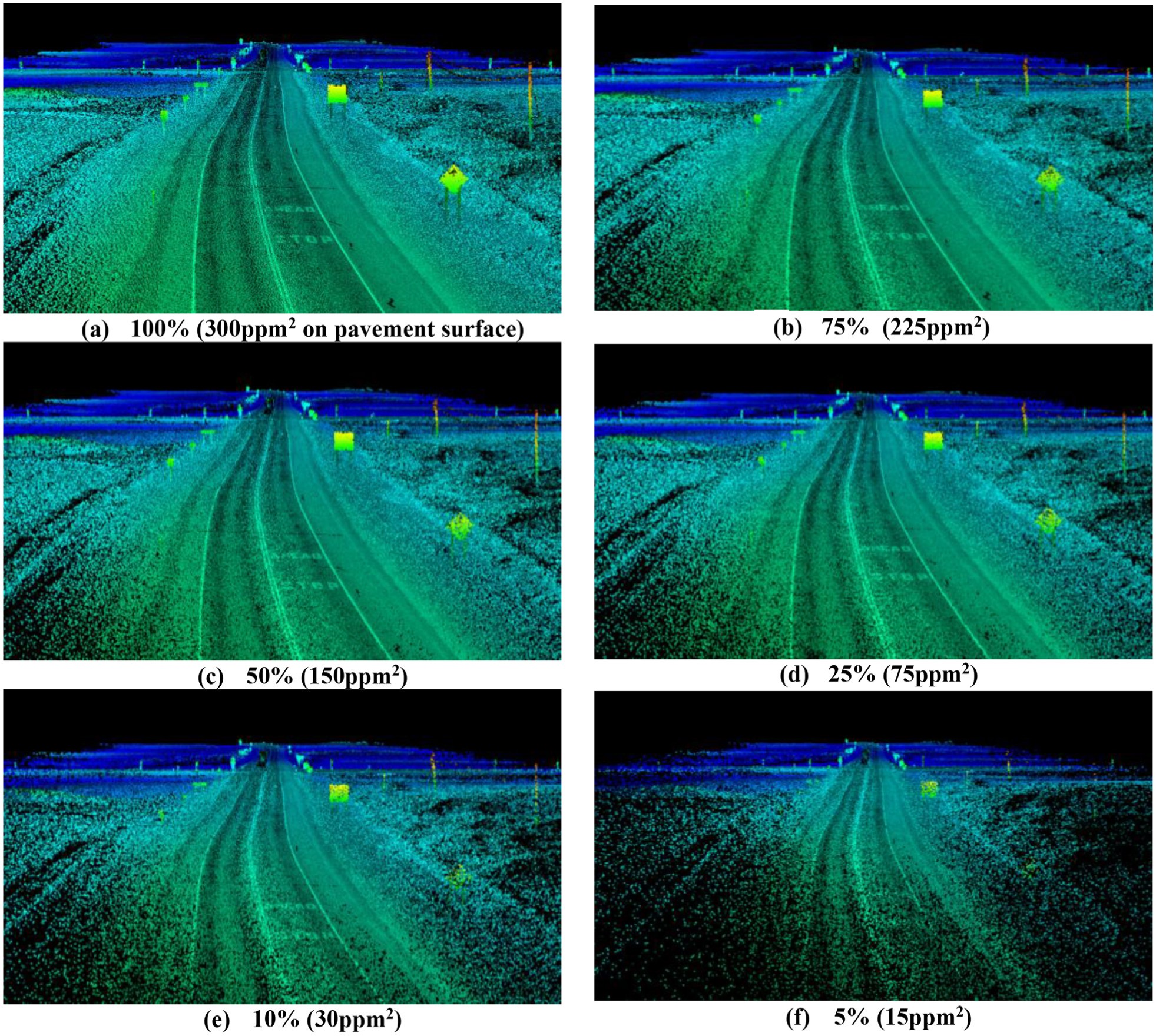


Figure 2.1: [15] Visualised point clouds taken at different LiDAR data resolutions

Mapping more points in low density regions would improve the visual quality of the images as we would have more information for all visible surfaces. Ideally, the points will have an even distribution and spacing however this generally never happens in the real world because of the variability in the collection methods (e.g. noise from poor equipment, scans taken only in a limited number of positions) and environment conditions (e.g. weather, reflective surfaces).

While mapping more points in general correlates with a more detailed and fuller surface model, it doesn’t guarantee it. For example, images (a) and (b) of figure 2.1 show an increase in LiDAR data resolution from 225*ppm*2 to 300*ppm*2 which has very little effect on the overall image quality and surface information from the given viewing angle. The most notable difference between the two images is the low-density dark patch in the bottom left part of image (b) as it becomes more filled out due to the increase in point density around that area in image (a). Overall, it is extremely hard to notice any other difference between the two

##### 2.1. HANDLING MASSIVE POINT CLOUD DATA

images. Therefore, it can theorised that while increasing the total number of mapped points produces a better surface model and image quality shown by images (f) to (d), it better to try and fill in the low-density patches to create a more uniform surface distribution creating a better visual image. Either way, the images do provide a strong case for visualising data to check the quality of the collected data and to make sure it looks how we expect it to as it is much easier to visualise data like in the images of figure 2.1 then to look through millions of numbers.

### 2.1.2 3D scanners

3D scanners have been used since the 1960s to analyse real-world objects and environments to collect data about their shape and sometimes appearance (i.e. colour) [16]. Laser scanners such as LiDAR have become very popular for generating point cloud models in a wide range of applications such as environmental surveys, ground-surface monitoring, disaster prevention, forest resource inventory and urban planning and management [6]. LiDAR generates point clouds by measuring the distance from the scanner to surrounding surfaces, and then transforming the distance values and the known orientation of the scanner into 3D coordinates [17]. However, standard LiDAR scanners are not able to detect colour attributes of the point data collected. They have to use another method such as cross-referencing image data to colourise the data or equip laser diodes to the scanner to detect visible colour [18].

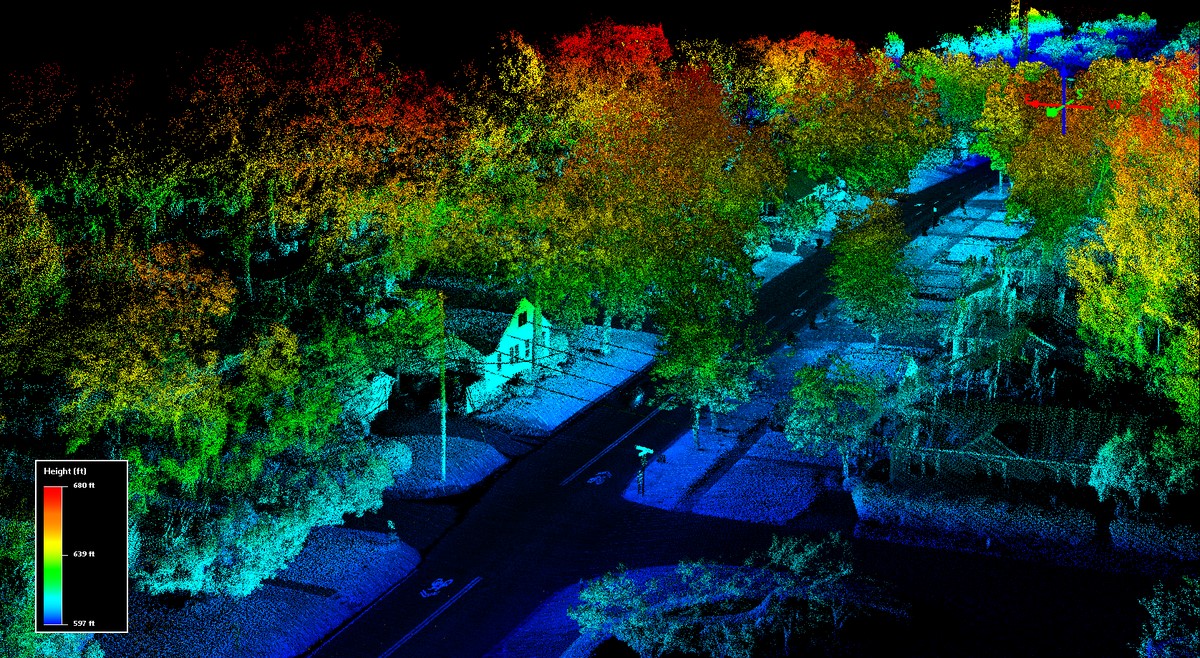


Figure 2.2: A LiDAR point cloud visualised using RGB colour based on the elevation of points [19]

LiDAR has a high efficiency of collecting high-density and large-scale point clouds [20] and is able to penetrate cover like forests and water to map the surface below. However, it can struggle to map reflective surfaces because the projected laser reflects off the surface with only a small portion of scattered light making its way back to the sensor to generate the distance data. This may cause a gap in the generated point cloud as the sensor may not be able to gather enough distance data from the surface due to the scattered laser readings. Despite this issue, LiDAR is still widely used today in many research fields like robotics.

While LiDAR is by no means the only 3D scanner out there, it is widely used for good reason. Other scanners such as ultra sound or radar can be used to map an environment and therefore create a point cloud. The problem is that devices such as ultra sound are not very applicable to situations such as mapping environments like forests or urban environments where the sound will be reflected introduce a lot of noise into the data. Therefore, it best not to use these devices unless you need to use them for a very specific reason.

In the early days of 3D scanners, it was a very time consuming process to accurately scan objects due to the limitations of the equipment. However, due to the rapid development of low-cost sensors such as the Kinect and time of flight cameras, 3D scanners have become cheaper and more accessible than ever [21]. On October 23, 2020, Apple Inc. unveiled the iPhone 12 Pro with an integrated LiDAR scanner making LiDAR widely accessible on mobile phones for the first time [22]. Despite its limited range, the iPhone proved that an easy-touse and cost-effective alternative to established remote sensing techniques was possible, with potential applications in a wide range of geo-scientific fields and teaching [22]. However, the problem with cheaper devices is they often have a number of shortcomings such as producing more noise and artifacts [23] or are ill suited to map densely populated environments such as forests and complex urban environments. This means that while devices such as the Kinect and iPhone are suitable for hobbyist projects such as surveying a small room, a more accurate and expensive LiDAR device is a necessity for large-scale projects such as geological surveys and urban planning.

When choosing a scanner, there are three main criteria to consider: the accuracy and data resolution of the device, the time it takes to capture points and the price of the unit. LiDAR is the most popular and widely used 3D scanner thanks to high level of accuracy and wide versatility. Some LiDAR’s cost tens of thousands of dollars to use but sometimes there is no alternative due to the situational need to acquire very accurate data of a highly complex environment. This is where structure from motion can prove to be a viable alternative.

### 2.1.3 Structure from motion

Structure from motion is a photogrammetry technique that uses multiple images with overlapping points of reference to create 3D models as shown in figure 2.3. It provides a very low cost alternative to 3D scanners with a reduced need for user supervision and experience. The main determining factors for how accurate structure from motion can be are: the quality of the camera, environmental issues (heavy rain, fog, exposure) and inadequate number of images taken. Almost everyone these days has access to a camera either on their phone or in the form of a dedicated camera.

The ability to extract high resolution and accurate spatial data using cheap consumer grade digital cameras has meant more people than ever can now create large point clouds at a relatively cheap price. It also involves minimal user experience where all the user has to do is ensure that there is a sufficient amount of points of references between the images they take. This could manifest itself simply in a video, as long as it is taken in good quality, where each frame is treated as an image in its own right. The images collected can then be put through the structure from motion process where it estimates the relative distance between points and generates a 3D model of the surface area captured. Because coloured images are used to extract data, structure from motion is also capable of mapping colour which makes it very useful for visual rendering.

One drawback of structure from motion is that the process is more complex than using 3D

##### 2.1. HANDLING MASSIVE POINT CLOUD DATA

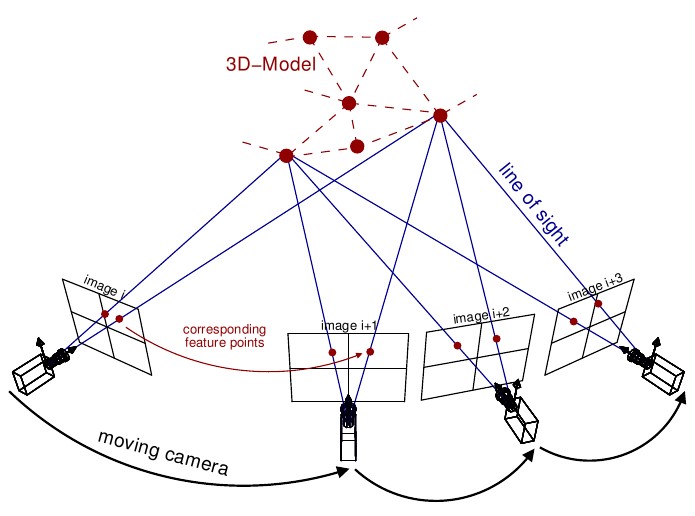


Figure 2.3: [24] This figure shows how a number of images are taken from a variety of different positions and orientations for structure from motion. Each image has a number of feature points that appear in each image which can be used to estimate the points in 3D space.

scanners. Structure from motion is relatively easy to capture data but involves a much more complex procedure to generate the point data from the capture images compared to the 3D scanners data. It also has similar problems to 3D scanners such as reflective surfaces are hard to detect and if there is an absence of light, structure from motion will not be able to capture anything because there’s no light to create images. However, these are generally easily fixed issues such as turning on a camera light or taking the image from a different position that’s less reflective. On the whole, structure from motion is an easy to use and highly accessible point cloud capturing method that is finding wide spread use today. Ultimately however, both 3D scanners such as LiDAR and the photogrammetry technique structure from motion can be both used to good effect with neither being decisively better than the other.

### 2.1.4 Data storage

There is currently no universally accepted point cloud file format for storing point cloud data. The type of capture method and intended use plays a significant part in choosing the appropriate file format to use. There are many type of formats used to store point clouds such as LAS (LASer) [25], E57 [26], PLY [27] and more. Some tools and libraries have their own format such as BIN for cloud compare [28] and PCD [29] for the point cloud library (PCL) [30]. For more information on file formats, Cloud compare provides a comprehensive list of [file formats](https://www.cloudcompare.org/doc/wiki/index.php/FILE_I/O) and key characteristics to look over.

For this project, the main file format used for handling point cloud data is the PLY file format. This was because Dr Adrian Clark supplied a range of PLY files for testing purposes which provided a suitable collection of data to work with.

A PLY file, also known as Polygon File Format, is comprised of two main sections, a header at the top and the stored data following the end\_header statement. The Fenwick-sparse.ply file header is shown below as an example.

Listing 2.1: Header of Fenwick-sparse file

|  |
| --- |
| ply  format binary\_little\_endian 1.0 element vertex 64597 property float x property float y property float z property float nx property float ny property float nz property uchar red property uchar green property uchar blue end\_header  // Data follows on here . . |

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

The PLY file format was originally developed to store three-dimensional data from 3D scanners, usually in the form of vertex data and faces use to create triangle meshes. Each of the PLY elements in the file (vertex’s, faces) can store a range of properties such as the colour and transparency of a vertex. However, it is perfectly suitable for storing any threedimensional data or simply just vertex data on its own. How PLY data is processed will be discussed further in the methodology PLY parser section 3.2.

### 2.1.5 Pre-processing point cloud data

With all this data captured and stored, it now needs to be processed in order to be suitable for rendering. Pre-processing is the key stage between mapping the point cloud data and passing it to the graphics rendering pipeline to render. This can involve algorithms such as filtering which try to remove irregularities such as overlapping points, outlier data and noise while preserving geometric features of the three-dimensional model [31]. If this is not done, the quality of the final rendered scene can be negatively impacted. Regardless of the capture methods employed, there will nearly always be some degree of undesirable irregularities such as strong variability of local point density, missing data, overlapping points or noise caused by scattering characteristics of the environment [23] [31]. These usually occur due to measurement errors of the capture device.

In the context of rendering however, these irregularities often manifest themselves as either holes in the point cloud model or large number of out of place points which I refer to as distortions in the final image. Figure 2.4 highlights the latter displaying the roof of a church with strange white and blue clouds of points. These are not clouds but are irregularities from the capture method used that have found their way into the final rendered image. The renderer being used is displaying the point cloud correctly however it may not look like it because of the irregularities in the original point cloud data supplied to the renderer. Thus, may be a necessary process for some point cloud visualisers to help improve image quality and possibly performance from reducing the number of erroneous points in the model.

There is currently a significant focus within research to develop quick and robust filtering algorithms for point cloud data [21] [31]. K-D trees [32] have particular usage in this area as many of the algorithms employ numerous nearest neighbour checks which K-D trees are especially suited for [31]. Other data structures such as octrees can be used but are notably slower and more cumbersome depending on the point clouds density and distribution of points.

## *2.2. POINT CLOUD RENDERING*



Figure 2.4: Image taken of Elmstead-dense.ply point cloud file using point cloud visualiser implementation. The red box highlights the area of white/blue irregularities which distort the visual image of the model due to errors in the original point cloud data

**2.2 Point cloud rendering**

Real time rendering of massive point clouds is notoriously difficult. Over the years, the methods and equipment for capturing point clouds has become increasingly detailed and advanced. This has meant an increasingly amount of detail is able to be captured to render more realistic and accurate environments. The size of the point clouds has therefore increased putting a higher performance requirement on the hardware and software needed to process and visualise the point clouds. With real time rendering, there is always a careful balance between accuracy and speed. This can be viewed as the total number of points being viewable at any given time vs the frame rate to display it. Ideally, it would be great to have all the points displayed at any given time which are within the field of view and to run at very high frame rates to give a smooth experience but this is not possible. Therefore, various optimization methods have been developed to improve the efficiency of visualising the point cloud to render. This can involve only rendering points which are in view or ignoring the data that doesn’t affect the users perceived interpretation of the environment.

Point bases rendering was first suggested by Marc Levoy and Turner Whitted [33] where they claimed a sufficiently dense number of points could be used to represent a continuous three-dimensional surfaces. The fundamental idea is if you take a sufficient number of scans or images to create a dense point cloud of the desire surface model, then it will be viewed as if it was a solid face.

Most methods use variations of hierarchical space-partitioning structures aka multiresolution structures, such as k-d trees and octrees. These structures are populated with data allowing the original model to be represented at different levels of resolutions. They are also good for frustum culling and other algorithms.

Several desktop-Based massive point cloud renderers exist such as Scanopy [34], Cloud Compare [28] and Potree [8] that are designed with rendering point clouds in the billions. However, some of these either don’t prioritise or forgo completely improving visual quality. A point cloud renderers aim should not be to solely render as many point as possible although this is a hard requirement to later facilitate rendering point clouds in better quality as the resource demands often increase along with it.

Some popular point cloud viewing software/libraries/tools are: **The point cloud library** (PCL) [30] is an open source, standalone library for 2D/3D image and point cloud processing. It has a range of features such as filtering, feature estimation, surface reconstruction, registration, model fitting and segmentation. It is built using modern C++ template programming and uses a number of libraries such as Eigen [35] FLANN (Fast Library for Approximate Nearest Neighbour) [36] to accomplish these tasks. Its a very well established and respected library that many people use today.

**Cloud Compare** [37] is an open source, Qt-based application for point cloud and mesh manipulation, visualization and processing. It was originally designed to perform comparisons between two 3D point clouds. It takes an octree approach to optimizing the performance of processing and rendering point clouds. It can take many different file formats ranging from ( ASCII, LAS, E57, etc ) to manufacturers, triangular and polygon formats. It can load almost any point cloud format making it appeal to a wide audience. It does have stereo capabilities but this is not suitable for the NIL setup. This could be used as the base of the project if things do not go as planned. **Potree**

### 2.2.1 Triangle vs point based rendering

Triangle-based rendering is the dominant rendering method used today. GPUs are highly optimized towards doing this very quickly. GL\_POINTS has fallen out of favour for a number of reason such as it requires a large number of points to represent models like flat surfaces. Converting a point cloud model to a simpler triangle mesh with textures isn’t always possible or desirable. low resolution polygon meshes lose a lot of information of the original model. With all these alternatives it may not be possible due to poor scan quality (low scan density for complex objects/surfaces) and hugely time consuming.

Low-resolution triangle meshes lose information as they smooth out or just ignore certain points of information to simplify the mesh structure. Ideally we would use infinite points to represent surfaces. We cannot do this because this would cause huge computation demands and be impossible to render a single frame.

When rendering point clouds compared to 3D mesh alternatives, there is no need for mesh reconstruction and is not required to store or maintain polygonal-mesh connectivity. [21] This means that it allows a lower overhead compared to its 3D mesh counter parts and offer better performance when its comes to processing and manipulating model data. While there can be advantages in processing, its important to consider the real world applicability of the data. 3D meshes can be created from point clouds although there is currently no reliable way to do this without producing unwanted noise or reducing accuracy to some extent. This is still an on going research interest area in the field.

One of the most widespread solutions for rendering a given set of points is by using the native point primitive that modern graphics APIs provide in addition to lines and triangles, such as GL\_POINTS for OpenGL, D3D11\_PRIMITIVE\_TOPOLOGY\_POINTLIST for DirectX, VK\_PRIMITIVE\_TOPOLOGY\_POINT\_LIST for Vulkan. [17]

One key advantage point primitives have compared to triangles is that they only require a single coordinate vector as input. This helps to keep the vertex buffer and bandwidth usage to a minimum. For instance, points don’t have any connectivity information.

A **quadtree** is a two-dimensional tree data structure in which are partitioned into 4 equal sized quadrants called nodes. Each internal node has exactly 4 children which can be further

##### 2.2. POINT CLOUD RENDERING

subdivided into four more nodes. The data associated with each node varies based on the application. For point clouds, this usually manifests itself as split up groups of points into smaller nodes of the quadtree up until either a maximum depth is reached or a set amount of points are allocated to a node. When the maximum capacity of a node is reached, the node is further subdivided again unless a max depth has been reached. Quadtrees are great at subdivided areas of space in a 2D and even 3D space with other algorithms.

### 2.2.2 Data structures

### 2.2.3 Octree

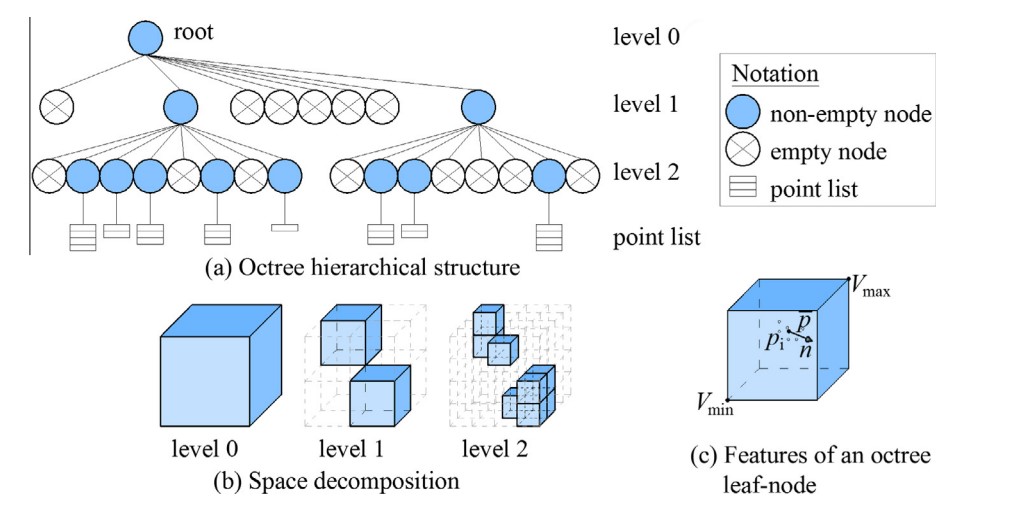


Figure 2.5: Example of an octree and how its nodes are broken down [38]

An octree can be viewed as a quadtree but applied to 3 dimensional space. Octrees are tree data structure used to partition a three-dimensional space by recursively subdividing it into eight octants. An octant is a node.

##### **Out of core optimizations**

Rendering point clouds of a few thousand or million is a fairly trivial task that can be done using the basic graphics API point primitive to render to the display. However, as point clouds escalate into the hundreds of millions or billions of points, they run the risk of not fitting into main memory. These types of points clouds require out-of-core algorithms that stream in, process and render only a small subset of the whole data. [8] Most of the predominant methods in literature employ variations of hierarchical space-partitioning structures, also known as multi resolution structures, such as kd-trees, octrees, or quadtrees where a model can be represented at different resolutions of detail based on the depth of the space-partitioning structures traversed. These structures are encoded into different files so that they can be loaded when needed. This frees up memory on the computer to load only the files it requires to render within its given view. The files typically have some ordering system so easily identify if a set of points falls within a given viewing region and the extends of the bounds which can be read without having to read through the entire file. This helps to quickly check a files bounds and continue loading the data if it needs to be loaded, otherwise gets discarded. Out of core processing is rather slow but necessary because not many people have large amount of RAM (128GB+) and it can be quite expensive. For hundreds of billions which is possible by Potree [8], its loads these files over the web and is one of the most impressive visualisers out there. It relies on out of core optimizations to load point clouds that do not fit into memory.

### 2.2.4 Point scaling

By definition, a point has zero dimensions. It has no height, width or depth so in order for a user to see a point we have to give it a point size. In OpenGL this is done by using glPointSize function call to display a point in a spherical shape of a given radius.

The size of a point can have a significant impact on speed and visual quality. A two point size can improve performance and reduce occlusion between points at the cost of leading to holes in the rendered image [8]. A larger point size helps to reduce the size holes between points however puts a greater burden on the graphics pipeline and increases occlusion artifacts thus reducing performance [8]. It is a careful balancing act between choosing a low enough point size that improves performance and reduces occlusion between points while a large enough point size to help reduce holes between points.

A fixed point size means the same pixel size is used for all points. This can be suitable for many cases however it can cause holes to appear when zoomed in to view points at close range and overdraws when the user zooms out. The points will start to overlap and put a greater burden on the rasterization stage of the graphics pipeline reducing performance.

Potree suggest an adaptive point size that adjusted point sizes to the level of detail [8].

To read more about this, you can find it available at [8].

## 2.3 Rendering optimizations

### 2.3.1 Level of Detail (LOD)

Level of detail

For example, in figure 2.6, two images are rendered one using a brute force approach of rendering all vertices in the scene and the other employing a discrete LOD approach to reduce the number of vertices rendered. As you can see, the images are indistinguishable however, the number of vertices rendered for each scene and render time per a frame are drastically different. The brute force method renders 2,328,480 vertices while the discrete LOD approach only renders 109,440 vertices. This allows the dynamic LOD scene to be rendered far quicker and still retains the visual quality. This is the essence of level of detail where we try to reduce work load put on the graphics pipeline while maintaining the visual quality of the scene. This allows the scenes to be rendered quicker and spare resources for other tasks.

There are multiple types of LOD implementations such as discrete, continuous and view based LOD techniques that are actively researched and developed with point clouds in mind [8].+-

### 2.3.2 Subsampling

Shown above are some of the most popular sampling algorithms for point distribution which you can see how these methods effect the point distributions and bias them towards certain

## *2.4. STEREOSCOPIC RENDERING*

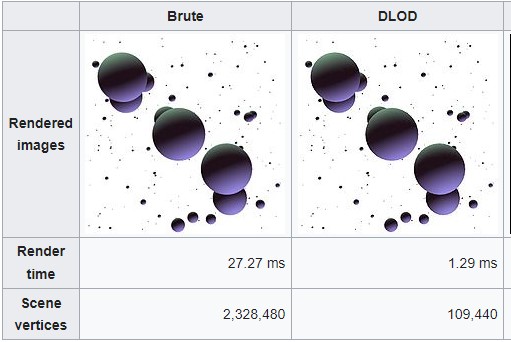


Figure 2.6: Example of LOD: image looks the same however the rendering times are drastically different for brute force vs the LOD implementation [39]

distributions.

###### Poisson disk subsampling

Point cloud density needs to be considered in order to retain as much image quality as possible. For example, if we only sampled points from the densest regions then there will be very little or no information about the vast area covered by less dense regions therefore using a uniform-random selection algorithm for sampling points is not good for this as can be seen in the figure below. We need to sample points with a bias towards points that come from less dense regions of space so the model representation still has a fair representation for lower density regions. If a point is removed from a low density region, that has a far greater information loss and loss of image quality because its a larger percentage of the points making up that region of space.

Poisson-disk sampling is one of the most common sampling schemes in the context of Computer Graphics due to its blue noise properties. [41] It produces a uniformly random distribution where the minimum distance between each point in the distribution does not overlap any other points minimum distance "disk" radius. No two points can be chosen for subsampling if they are within a minimum radius of a point already inserted.

**2.4 Stereoscopic rendering**

Some of the information in this section is taken from the project proposal with a number of edits and additions.

Monoscopic rendering refers to rendering one image to a screen that is directed to both eyes. Both of the users eyes will see the exact same image on the display. This is what most people view everyday on their electronic devices that display regular images or video of things.

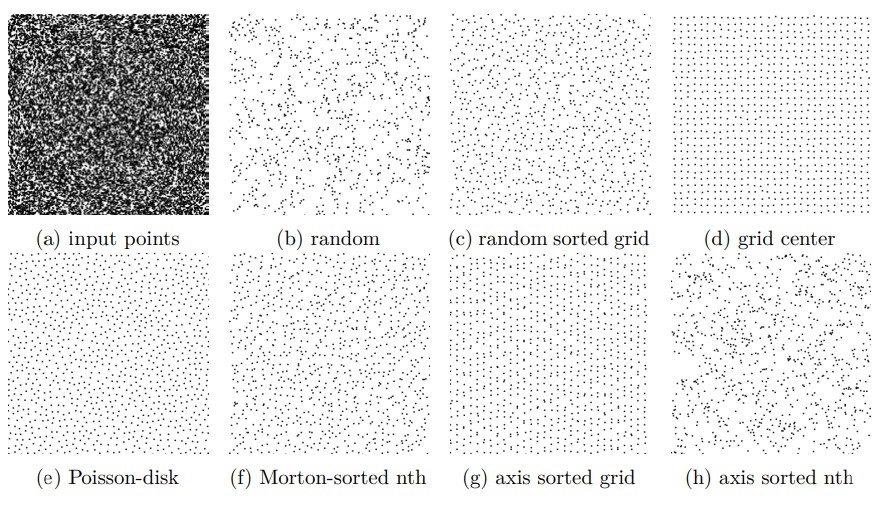


Figure 2.7: Various sampling strategies used to sample a given number of points selected from an parsed input set. [8]

Stereoscopic rendering refers to the use of two images to give an impression of depth to the user. These images are projected onto a screen from two different sources in the case of this project. Each projector projects the same image but through the use of colour filters and careful field of view alignment and special glasses, it provides a sense of depth to the user. In the illustration below, the chess piece is at the focal point of the users eyes. However, due to the interocular distance between the persons eyes, they have slightly different fields of view and angles to view the chess piece. When they are wearing glasses with lens that are aimed towards that eye, they are able to see two different images like on the sheet of paper. A person mind tries to comprehend this and merges them together to give the impression of a 3D image with depth.

The most common and widely used method for stereoscopic rendering is to render a scene for the left and right eye views separately. [44] Rendering two view points compared to one effectively doubles the rendering complexity. This is still the model in use in graphics APIs such as OpenGL [45] Vulkan can leverage better performance thanks to its improved scalability on multi-core CPUs due to the modernized threading architecture [46]. The alternative approaches can be categorized as pipeline-based solutions aiming to improve the rendering of the rasterization stage of the rendering pipeline; and image-base solutions, where one view is rendered using the graphics rendering pipeline, and the other view is generated from this image, using the correspondences of the two views and lastly a perceptually-based solution utilizing suppression theory of binocular vision [44]. Stereo has a number of problems such as only 15% of people can see it, may cause some people to become nausea’s and requires special glasses to see depth. While it would be interesting to explore the different approaches of stereoscopic rendering to improve stereo performance, it is not the main problem targeted by this thesis as discussed in the introductions scope.

#### *2.4. STEREOSCOPIC RENDERING*

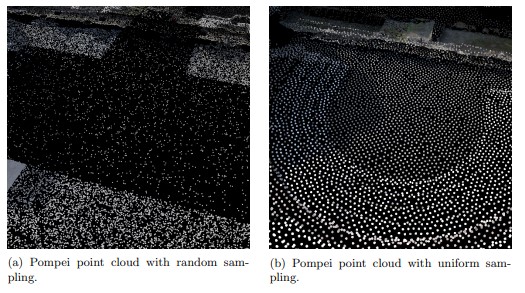


Figure 2.8: Uniform selection vs poisson-disk subsampling [40]

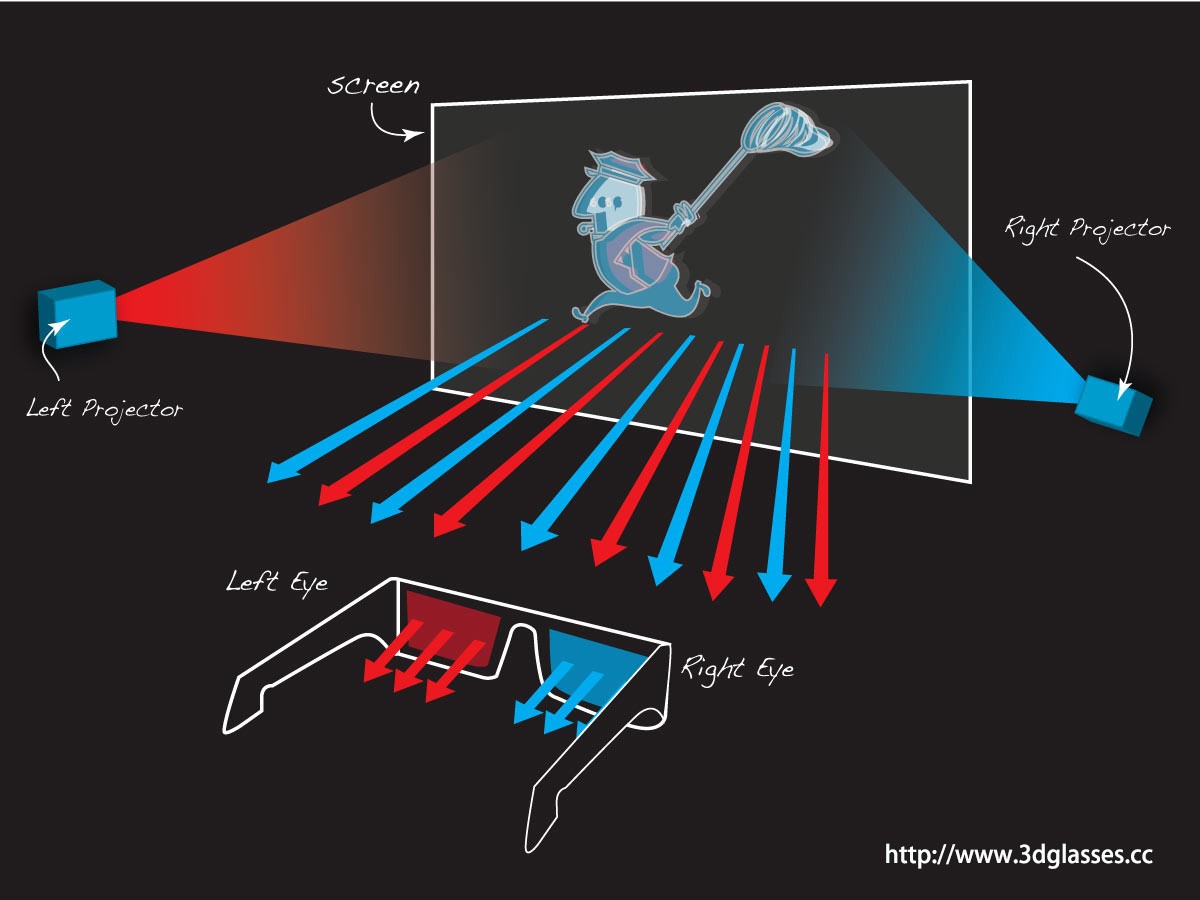


Figure 2.9: Stereo projection example [42]

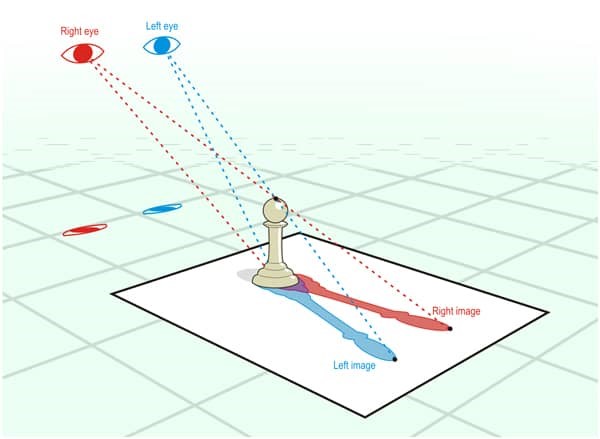


Figure 2.10: Depth perception illustration [43]

**Chapter 3**

# Methodology

The first thing that needs to be done before rendering any graphics to screen is to create an OpenGL context and window to draw the graphics to. An OpenGL rendering context is a port through which all OpenGL commands pass. Rendering contexts link OpenGL to the Windows windowing systems. All actions and commands must be directed through an OpenGL context to have an effect. OpenGL [47] does not handle window creation, context creation or user input events so a library will be used to manage these things. There are a number of libraries available such as GLUT [48], FreeGLUT [49], and GLFW [50]. For this project, GLFW was chosen as it provides finer control over context creation and window resource sharing and allows the event loop to be in the control of the programmer. This allows for more explicit control of the timing and execution of the rendering process.

With the context creation, window management and I/O handling going to be managed by GLFW, the next step was to consider what loading library was needed to load and access OpenGL functionality. A loading library is a library that loads pointers to OpenGL functions at run time to make them easily accessible across multiple platforms. Every GPU has its own implementation of OpenGL based on the unique architecture of the product line which is defined by the drivers of the manufacturer. A loading library tries to latch onto the drivers OpenGL implementation to make it easier to access the functionality OpenGL has to offer. While a loading library is not strictly needed because you can manually access the drivers on a PC and therefore the implementation of OpenGL for that hardware, it is very difficult to do so and has practically no performance benefit by doing so. Therefore a loading library is strongly recommended to help improve development time.

There are a range of loading libraries such as GLEW [51], gl3w [52] and glad [53]. This is by no means an exhaustive list however they are all loaders that are commonly used today. Choosing either one of them would be suitable for this project however GLEW was chosen for the following reasons.

GLEW is simple to use and compile, comes with both the core and compact profiles for accessing functionality of the core profile and the depreciates glfunctions for the compatibility profile. This allows functions such as glBegin() and glEnd() and glVertex3f() to be used for testing the older functionality of OpenGL vs the more modern versions (3.3+ core profile). By having the both the compatibility profile and core profile, the whole breadth of OpenGL functionality can be accessed. This is good for comparing older functionality from earlier versions to the more modern core profiles implementations to compare how the performance methods have changed over time. One possible downside of GLEW is that it takes longer

19

to initialise than other libraries such as gl3w which only has the core profile. However, this can be improved by setting glewExperimental = GL\_TRUE. As the focus of this project is on the rendering performance and less about the initialisation of the application, this is a minor concern.

The programming language chosen to implement the main code of the visualiser was C++. This is because C++ offers better performance than other languages such as Python because it is a compiled language that compiles the source code into or close to machine code. OpenGL as mentioned previously natively support C++ with their bindings being written in C. This means that there is no intermediate software layer between the bindings of OpenGL and the source code. This remove a layer of overhead from the application and improves the processing speed. Python could be used but it is much slower as it is an interpreter based language and rendering speed is the larger priority when it comes to rendering massive point clouds than improving development time. When compared to other languages, C/C++ offers the a suitable level of memory management and performance while being high level enough to make development not too difficult.

Before graphics can be rendered to the screen, we need to tell OpenGL the position and desired space for which to render graphics to. This usually is the size of the rendering window however as will be later discussed for stereo, two view ports can be used to split the screen projecting multiple views.

To render graphics to the screen, GLFW and GLEW must be initialised before any windows and graphics can be created. Once this is done, optional initialisation stages such as setting up a GUI or enable OpenGL features such as depth testing, multi sampling or point smoothing before the main rendering loop begins.

The main rendering loop is where all the repeated graphics rendering calls are made to the graphics pipeline. This consists of clearing buffers such as the colour and depth buffer, updating the viewport size in case the window is resized, swapping out the colour buffer to display what has been rendered to the window viewports and poll for new I/O events that may have happened during the rendering process. A simple implementation of this looks like this:

|  |
| --- |
| // The window variable is a GLFWwindow context that has been // previously created and ini ti al ise d along with GLEW. while (!glfwWindowShouldClose(window)) { glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT) ;  int bufferWidth, bufferHeight;  glfwGetFramebufferSize(window, &bufferWidth, &bufferHeight) ; glViewport(0 , 0, bufferWidth, bufferHeight) ;  // Render stuff here  glfwSwapBuffers(window) ; glfwPollEvents() ; } |

1

2

3

4

5

6

7

8

9

10

11

12

13

14

The main graphics pipeline can be seen as

Listing 3.1: Graphics pipeline summarised

|  |
| --- |
| read PLY files create/insert into octree create shaders/vertex vertex buffer objects |

1

2

3

|  |
| --- |
| start main render loop clear buffers update view port in case of window resizing set view port render the scene to the view swap buffer and poll for I/O events  Continue looping until exit |

4

5

6

7

8

9

10

The main challenge for rendering computer graphics to a screen is how to approach converting 3D vertex data into the 2D array of pixels that fit the window on your display. The process of transforming 3D coordinates to 2D pixels is called the graphics pipeline. Each stage of the OpenGL pipeline is shown below:



Figure 3.1: [54] OpenGL rendering pipeline

The idea behind a pipeline is that each stage is specifically towards doing a certain task which can be parallelised. However, OpenGL is more akin to a state machine and does not support asynchronous behaviour by default. For example, all the commands to OpenGL are executed sequentially however, the return of these statements are not guaranteed to be returned by the time the rendering has finished for that loop. The OpenGL specification defines commands such that they will execute sequentially, and that every command must treat previous commands as if they have been completed and become visible [47].

Graphics cards (GPUs) have thousands of small processing cores to allow for quick parallelised data processing. These cores are specifically designed towards rendering graphics as fast as possible and processing different stages of the graphics rendering pipeline. The processing cores run shaders (small programs) on the GPU for each step of the rendering pipeline. Shaders are a great way to shift more processing over to the GPU which is designed to process data relating to high resolution images and geometry far quicker than the generalised process of a CPU. This helps to free up the CPU for other tasks.

As shown in figure 3.1, connect together one after the other, transforming vertex data to pretty graphics on the screen. The blue stages (rasterization, clipping and primitive set up) are stages handled by OpenGL where the programmer has no control over how they are implemented. The Red stages such as vertex shader and fragment shader processing are mandatory in OpenGL core profile versions 3.3+ which the programmer implements themselves. The purple stages of the figure are optional stages that do not have to be implemented by the programmer or used at all. They do useful things but mainly directed towards polygon rendering. They will not be used for this project.

But first, we need to define what the vertex data is. Vertex data is simply a collection of vertices we would like to draw to the screen. A vertex is a 3D coordinate for now which we will do basic processing to.

The first stage of the pipeline is to apply a vertex shader to the vertex data. A shader is a program that is applied to every vertex rendered to the screen. A vertex is taken as input and a set of operations are carried out based on what’s programmed in the shader. The main purpose of the vertex shader is to transform 3D coordinates to coordinates relative to a camera or different view space for other useful processing tasks. An example vertex shader looks something this:

Listing 3.2: Vertex shader example

|  |
| --- |
| #version 330 core layout (location = 0) in vec3 aPos;  void main()  {  gl\_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0) ;  } |

1

2

3

4

5

6

7

Following this, the primitive assembly takes all the transformed vertex data and organises the vertices into their associated geometric primitives in preparation for clipping and rasterization. Once this is done, clipping of the viewport is carried out by OpenGL where any primitives outside of the viewport will not be rendered or make it to the rasterization stage. Rasterization happens straight after clipping where the primitives are sent to the rasterizer for fragment generation. Fragments can be interpreted a "candidate" or "potential" pixel on the screen. The rasterizer tries to determine which screen locations are covered by a particularly piece of geometry such as a point or triangle. Through the use of linear interpolation of the previous data values of the pipeline, it generates fragments to the fragment shader can colour pixels the correct colour. The fragment shader stage is used to apply colour to the pixels mapped to your screen and is usually where all the advanced effects such as lighting and shadows would occur. Finally, the output is passed through the final OpenGL enabled per-fragment Operations such as alpha testing, depth testing or blending to be applied to the final image. After all of this, if a fragment successfully makes it way through all the stages it will be written to the frame buffer to be drawn to the screen when the buffers are swapped in the rendering loop.

So how do we go about programming this?

First, we must define at least one vertex buffer object and vertex array object as its mandatory in the OpenGL specification core profile for version 3.3 on wards. A vertex buffer object, also known as a VBO, is a buffer that is used to store data like vertex positions, colour data, normal’ s etc on the GPUs (Graphics processing unit) memory. A vertex array object, also known as a VAO, is an object used to store attribute pointers pointing to the VBO. This will be explained in more depth later. For now, VBOs will be the focus.

A VBO is used to manage memory on the GPU to define buffers to store vertex data. The main advantage of doing so is that large write operations can be carried out to store vertex data on the GPU in batches or keep them there permanently for the duration of the program. If all the vertex data can fit on the GPU then a buffer could be used to send it to the GPU and store the vertex data in the VRAM (video RAM) which effectively removes the need and overhead of writing data repeatedly to the GPU. This is one of the main advantages of "modern" OpenGL which forces you to employ vertex buffer objects.

Small VBOs are often easier to use as you can use a new VBO for each object to render. However, creating a lot of VBOs that only contain a very vertex points do not offer much benefit and increase the overhead of the application due to the increased number of calls to switch buffers and render the vertex data. Larger VBOs help to reduce the overhead of switching buffers which is rather small anyway but can become significant if its being called millions of times. The only drawback is an excessively large VBO may not be able to fit inside the GPUs VRAM. This is one of the OpenGL aspects I exploit to improve performance by settings up vertex buffer objects to move the data over to the GPU to render. It removes the need to transfer data from RAM to VRAM and exploits the significantly higher access speeds of dedicated VRAM. VRAM for cards like the GTX TITAN can read data at 288.4GBps. This is blisteringly fast compared to RAM which was measured on my home PC to be in the region of 1.8GBs.

A VBO is first generated using the command glGenBuffers() which created a unique id to refer to that buffer on the GPU. To transfer data over to the GPU, we then need to bind the buffer to the OpenGL context and declare the buffer target e.g GL\_ARRAY\_BUFFER. The reason we do this is that the GL\_ARRAY\_BUFFER target can be used to configure the currently bound VBO and send data through buffer called targeting that target. glBufferData() can be used to copy the data over by declaring the target, size of data, data address and draw method. Once this is done, the vertex data is transferred to the GPU memory.

After this or before shaders can be compiled ready to use. The following shader programs were used to transform vertex data to clip space and colour the respective fragments produced.

Listing 3.3: Vertex shader

|  |
| --- |
| #version 330 core layout (location = 0) in vec3 aPos; layout (location = 1) in vec3 aNormal; layout (location = 2) in vec3 aColour; uniform mat4 mvp;  out vec4 vertexColor; out vec3 Normal;  void main()  {  gl\_Position = mvp \* vec4(aPos, 1.0) ; vertexColor = vec4(aColour, 1.0) ; Normal = aNormal;  } |

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

Listing 3.4: Vertex shader

|  |
| --- |
| #version 330 core  in vec4 vertexColor; in vec3 Normal;  //uniform vec3 lightPos ; out vec4 FragColor;  void main()  {  FragColor = vertexColor;  } |

1

2

3

4

5

6

7

8

9

10

11

12

13

The shader programs wont be covered in detail but the overall application of them will later in the methodology.

But now, we need to go back to the vertex array object we have yet to create. We currently have some shader programs which we can assume have been compiled and are ready to use and the vertex data has been shipped to the GPU memory. However, we will need to define how this vertex data is going to be read from memory. The vertex data might have several attributes such as a position, normal and colour which need to be carefully read in the correct order to pass through the vertex shader.

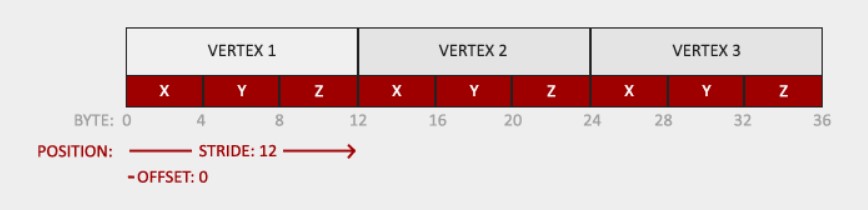


Figure 3.2: Vertex buffer memory example [55]

For example, figure 3.2 shows how three vertex’s are being stored in memory. They are all tightly packed together due to them being floats so there is no padding added between them. Each vertex has three coordinates x, y and z of type float. To read the data from memory we can setup a pointer to begin at a certain position in this case, byte 0 or the beginning address of the memory and tell the OpenGL how many values we are reading and the stride length to the next vertex attribute. This is done using glVertexAttribPointer() function where it takes 4 arguments. The first arguments defines the vertex attribute location. The location was previously declared in the vertex shader of listing 3 in the second line of code. The location is essentially like a port which vertex attributes can be passed through in this case to be declared as a vector called aPos in the shader for each vertex. So the first parameter, refers to the location of the position vertex attribute within the vertex shader. The next argument tells OpenGL how many values the vertex attribute has, in this case 3 (x,y and z). The following argument defines the vertex attributes types (GL\_FLOAT). The next argument indicates to OpenGL if we want OpenGL to convert the passed vertex data into normal device coordinates. The 4th argument specifies the stride length which defines the number of bytes between vertex attributes, in this case on a window 64 bit PC that would be 12 bytes! Finally, the last argument is a void pointer which is used to give an offset to the attribute pointer to point a set number of bytes further along in memory. This is pivotal to get right for adding multiple attributes as we will get to next.

If we had two vertex attributes such as vertex position and colour which were represented by 3 float and 3 uchar data type respectively, then the following code shader and C++ code outlines an example of how to go about reading that data to the vertex shader with comments.

Listing 3.5: Vertex shader

|  |
| --- |
| #version 330 core layout (location = 0) in vec3 aPos; // position attribute assigned location 0 layout (location = 1) in vec3 aColour; // colour attribute assigned location 1 uniform mat4 mvp; out vec4 vertexColor;  void main()  {  gl\_Position = mvp \* vec4(aPos, 1.0) ; vertexColor = vec4(aColour, 1.0) ;  } |

1

2

3

4

5

6

7

8

9

10

11

12

13

Listing 3.6: C++ OpenGL code

|  |
| --- |
| unsigned int VBO;  glGenBuffers(1 , &VBO) ;  glBindBuffer(GL\_ARRAY\_BUFFER, VBO) ;  glBufferData(GL\_ARRAY\_BUFFER, sizeof (vertices) , vertices, GL\_STATIC\_DRAW) ; // Sends ←vertices over to the GPU  glVertexAttribPointer(0 , 3, GL\_FLOAT, GL\_FALSE, 3 \* sizeof ( float ) , ( void \*)0) ; // This was ←explained above same as \ ref {Vertex Attributes } example .  glVertexAttribPointer(1 , 3, GL\_UNSIGNED\_BYTE, GL\_TRUE, 3 \* sizeof ( float ) , ( void \*) (3 \* ←sizeof ( float )) ; // Explanation given below  glEnableVertexAttribArray(0) ; glEnableVertexAttribArray(1) ; |

1

2

3

4

5

6

7

8

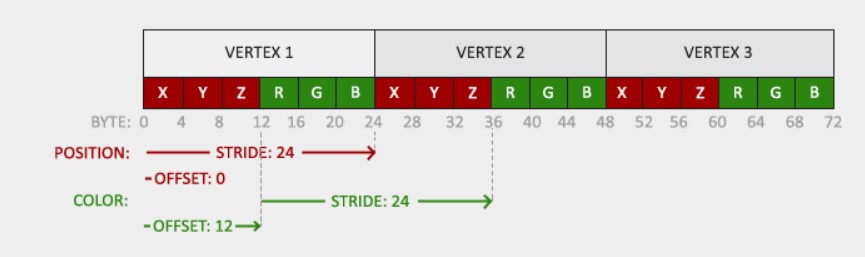


Figure 3.3: Vertex buffer memory with 2 vertex attributes [55]

For line 6, same process as before. The vertex attribute pointer signifies that it points to colour attribute data and specifies the location of the colour vertex attribute in the vertex shader (location = 1). However, this time the data is uchar so the data type has been changed to GL\_UNSIGNED\_BYTE which is the size of a uchar value on a windows 64 bit system I work on. GL\_TRUE normalises the values to fit between 0 and 1 which is very useful as OpenGL colours are float values which produce the wrong colour (white) if not converted properly. Lastly, the offset is 12 bytes along in memory as that’s the end of the vertex position attribute data. This can be seen in figure 3.3. OpenGL can then repeatedly read over this data at the defined intervals for any DrawArray() calls. With what has been covered so far, a set of points could be drawn to the screen if they were in normalised device coordinates. However, if they are not then the vertex position data has to be converted to screen space also known as your display.

## 3.1 Transforming vertex data

The main goal of the graphics rendering process is to convert vertex data in 3D space to normal device coordinates on your screen. There are 5 key coordinates systems that we have to be aware of before we get to the process and they are:

1. Local space/object space
2. World space
3. View space
4. Clip space
5. Screen space

We start off with the vertex data of a point cloud in local space. A point by default is at the original of its own coordinate system because the centre point of the point is the point itself. This simplifies things and means we can take this value and apply a model matrix to it to convert the vertex coordinates from local space to world space.

World space is basically the coordinate system where all the objects/models are arranged in the same coordinate system. The coordinates for all models are relative to the worlds origin and so are their vertex’s. The model matrix used for this is really simple:

1 0 0 0

0 1 0 0

 

0 0 1 0

0 0 0 1

Its an identity matrix! This is because there are no transforms applied to point cloud data as its technically already in world space unless there is another model to arrange the point relative to. If there is, the model matrix can be used for things such as rotating, translating or scaling your object to place it in the desired way in world space. Therefore, I make the assumption for the point cloud data that all vertex coordinates are relative to the world origin. You may be thinking I could get rid of this matrix in the end calculation and I could but if i wanted to apply transforms to the local space of a point then it would be best to do that to the identity matrix instead of the other matrix’s (view and projection) as it reduces the potential problems that it could cause by doing so.

With the coordinates in world space, we can apply a view matrix to convert vertex coordinates from world space to view space. View space is the coordinate system relative to what many people consider the "camera" or view of the world. All the coordinates are translated relative to the position and direction of the camera. The view matrix is stores all the relevant translations/rotations to translate objects relative to the view of the world.

The next matrix we use is called a projection matrix. This projects view space coordinates in clip space with either a orthographic or perspective projection which is shown in figure

## *3.2. POINT CLOUD DATA PARSING*

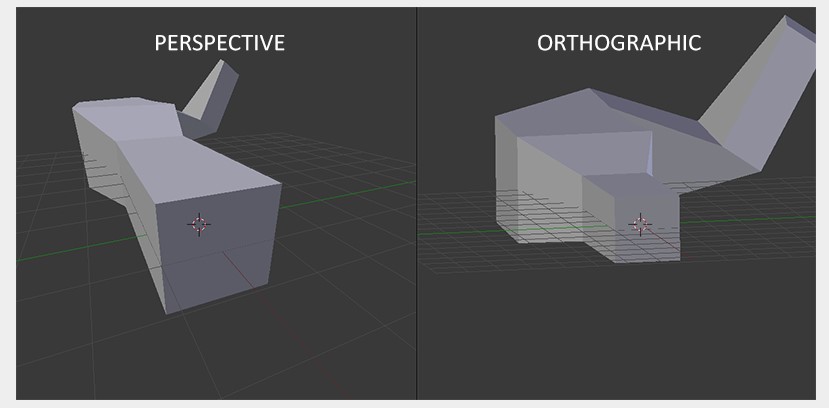


Figure 3.4: Example of perspective and orthographic projection views [55]

3.4. The coordinates in clip space are scaled between -1.0 and 1.0 and determines which vertices/fragments are visible on the screen. I chose to implement perspective projection as that is the normal viewing experience humans experience everyday. We naturally can see things smaller further away and larger close up. To implement this, I use a perspective projection matrix which maps a given frustum range to clip space while manipulating the w component of the vertex position. This scales the w component between -w and w which is later used for apply perspective division to the clip space coordinates. This takes the form of each vertex coordinate being divided by the w component making the coordinates values become smaller the further the further away a vertex is from the camera.

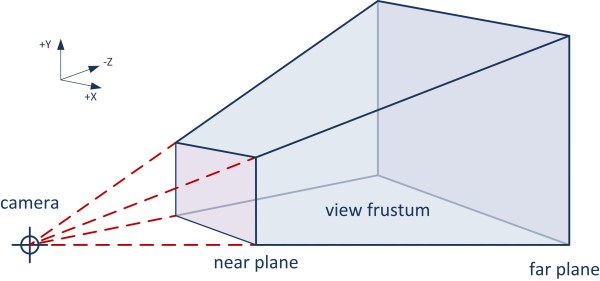


Figure 3.5: Example of a frustum [55]

An example of a frustum is shown above. A frustum is a geometric shape comprising of 6 sides that can be used to cull points and define the view of the camera.

To define a perspective in OpenGL, I use the glm library to simplify the frustum matrix produced. The perspective projection matrix can be produced in one line of code:

Listing 3.7: Vertex shader

|  |
| --- |
| glm::mat4 proj = glm::perspective(glm::radians(45.0f) , ( float )width/( float )height, 0.1f, ←*-*  100.0f) ; |

1

Following this the points are clipped to the viewpoint size to produce the graphics to your screen.

**3.2 Point cloud data parsing**

Now that we can successfully render points to the screen, we can start parsing points from a point cloud. This involves using a PLY parser to parse Dr Adrian Clarks PLY files through the graphics rendering pipeline described above.

MiniPLY is by far the fastest ply file reader for when it comes to reading based on two sets of benchmarks available on GitHub covered in the testing section 4.1 However, the problem with miniply along with many other ply parsers is they are aimed at reading triangle mesh data stored in a PLY file format. Miniply does not make it easy to access the read ply data through function calls which means there would have to be alterations to the library. This would lead to wasted development time and would be like fitting a square object into a roung hole. Still, some of the most popular PLY parsers that are publicly available are:

* [hapPLY](https://github.com/nmwsharp/happly) - A C++ header-only parser for the PLY file format. Parse .ply happily!
* [miniply](https://github.com/vilya/miniply) - A fast and easy-to-use library for parsing PLY files, in a single c++11 header and cpp file with no external dependencies, ready to drop into your project.
* [msh\_ply](https://github.com/mhalber/msh) - A range of libraries used by the user [Maciej Halber](https://github.com/mhalber) for their daily development, written in C.
* [tinyply 2.3](https://github.com/ddiakopoulos/tinyply) - Tinyply supports exporting and importing PLY files in both binary and ascii formats. Allows for filesizes >= 4gb.
* [RPly](https://w3.impa.br/~diego/software/rply/) - RPly is a library that lets applications read and write PLY files. The PLY file format is widely used to store geometric information, such as 3D models, but is general enough to be useful for other purposes.

Why create my own PLY parser? - its customizable and a lot faster for point cloud parsing.

The ply parser is the first port of call to load and initialise point cloud data in the application. Creating a new ply parsers allows for developers to customize the operations that go into it such as finding the min and max vertex points as the vertex’s data is read line by line avoiding having to iterate over the data again. This means that only one pass over the vertex data is needed as its being read. float and uchar are used to store data. This is to help reduce the total memory required at run time. Point clouds can be very large and size of vertex \* number. we don’t have much control over the number but we can definitely reduce the data size of the vertex but information can be lost by down casting the data types eg double to float but this can be acceptable or not noticeable in the final results. The accuracy vs memory usage is trade off that has to be balanced carefully for the needs of the application.

### 3.2.1 Octree frustum culling

Octree frustum culling works by storing a frustums dimensions in a struct with a six planes. These six planes each have a normal for the direction and an offset which together can be used to place the planes in line with each side of the viewing frustum of the camera. The main idea behind the method is as follows:

Calculate the current position of the frustum based off the camera location. Pass frustum to octree to iterate over the first level to tell if a nodes extents are within the view frustum If they are, that nodes children are then evaluated. This process keeps happening up until a certain depth threshold is reached to prevent the frustum from doing hundreds of thousand checks for high resolution octrees. For any octree nodes that pass the check and are a leaf node get set to active. If a node is no a leaf or a leaf child of a non leaf node is not within the frustum, they are set to inactive. An illustration of this is provided in figure 3.6 below.

Each of the nodes in the octree are axis aligned and therefore each of the frustum bound checks become AABB vs frustum comparison checks which simplifies the later equations.

## *3.3. PROJECT MANAGEMENT*

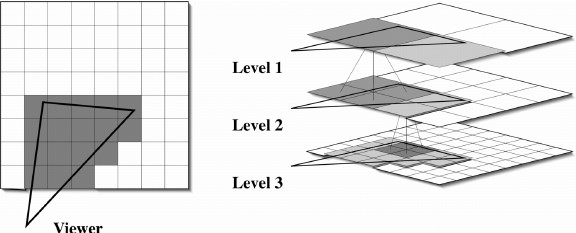


Figure 3.6: Example of how frustum vs octree node detection works. Light grey = inactive nodes, Dark grey = active/nodes within view [56]

**3.3 Project management**

The project was conducted in an agile manner with ad hoc testing used predominately as and when it was needed. Towards the end of the project where more work was focused towards the implementation, GitHub was used instead of GitLab because I plan on releasing this project on there when I am finished with my masters so makes sense to keep the work in one place. One huge problem during the project was that I had originally intended to deploy this project in stereo to The Networked Interaction Laboratory at the University of Essex. However, due to numerous hardware issues in the lab and not being able to get access to it. The project goals had to change to accommodate for it. This meant that the operating system it was developed on was windows because it was the main operating system I had at home and it was much more comfortable and easy for me to work on and deploy to. The performance of the application became a bigger priority as improving that was a much greater challenge than deploying to the lab would have been to achieve a higher level of difficulty and achievement. The stereo implementation was changed to not be on a projected display but instead two view ports that could be used similarly as a proof of concept for VR. This meant it had far greater applicability to real world situations and accessibility as VR is getting more popular. If it was taken further to deploy to VR, it could be viewed anywhere however for the NIL, it would most likely only work on that system or a similar cave system which most people don’t have. Either way, the project was adapted despite the change in direction to still be applicable to real world applications like virtual reality and to keep it relevant to the original idea at hand.

The following text was taken from the project proposal but still holds true to this project for the legal and health issues facing point cloud data and visualisation.

The main ethical issue surrounding point clouds come from the acquisition of data. With the increased accessibility, people can acquire data of nearly any 3D structure or location from land or air. It is not illegal to acquire photographs of peoples property or person in the UK but it does pose a privacy issue. As it stands, it is possible for any person to acquire images via drones or mobile phones to acquire data of private areas which people may not want to disclose.

Another ethical issue is how point clouds can be used for cheating in sports. Reverse engineering have been used in formula one to create identical designs of competitors cars. With this data, they are able to generate 3D models designs that can be used to copy the cars aerodynamics. This creates an ethical issue as it creates an unfair advantage and undermines the efforts of the other manufacturers research and development. This has lead to the Fédération Internationale de l’Automobile, the FIA for short, to rule no Competitor may design its LTCs based on “reverse engineering” of another Competitor’s LTC through the use of photographs and software to generate point clouds or similar CAD models. [57] Therefore, its worth considering the application of point clouds in the given context to be sure it is not crossing any boundaries within this proposal.

One legal issue is using other people point clouds. This is a very common problem summarised as using other peoples data in general. Permission should be sort to use other people data to make sure no legal boundaries are crossed.

The major health concern of the project is that stereoscopic displays and virtual reality headsets can induce nauseous effects. This is caused by having a significant disconnect between your brain thinking you are moving in space but your body is actually being still which triggers motion sickness in some users. This was covered in the risk register of the project but its worth reiterating that it should always be considered for when people are viewing the display/headset.

**Chapter 4**

# Testing

For all the tests mentioned in the testing chapter, unless otherwise specified, were carried out using the system specs below:

**Processor** Intel(R) Core(TM) i7-4790K CPU @ 4.00GHz, 4001 Mhz, 4 Core(s), 8 Logical

Processor(s)

**Memory** 32GB RAM DDR3 @1600 MHz

**Graphics** NVIDIA GeForce TITAN X 12GB

**Storage** Seagate ST2000DM006 Barracuda 2TB Hard Drive

**OS** Windows 10

The following PLY files were used for testing purposes:

1. Elmstead
   * Sparse - 337,284
   * Dense - 95,977,687
   * Ultra Dense - 331,678,561
2. Fenwick
   * Sparse - 64,597
   * Dense - 58,853,540

Each point cloud has a different number of points which needs to be kept in mind for when it comes evaluating the performance. More points puts a greater burden on the processing and rendering pipeline so it should be expected that more points equates to worse performance. The main goal however is to see how we can minimise the negative effect increasing point cloud size has on the performance.

Before running the test cases below, all user controlled programs were closed on the computer to free up as much resources as possible and to minimise the impact other applications could have on the applications performance. For example, running an application which takes up half of the available RAM or VRAM would significantly skew the results when disk thrashing occurs for parsing the larger points cloud into memory. This may occur for the largest point cloud however, that may be unavoidable due to the size of it.

31

## 4.1 PLY point cloud parsers

The first set of tests carried out were for the custom PLY parser created for the point cloud visualiser to see how it compared to other PLY parsers. As noted in the methodology chapter, the majority of PLY parsers online are not suitable for the PLY files that Dr Adrian Clark supplied. As a result, I was only able to compare the custom PLY parser against one publicly available parser online called [hapPLY.](https://github.com/nmwsharp/happly) I did consider altering other libraries such as miniply to extract the data from the library by directly access one of the arrays that stores PLY data. However, this didn’t work out as intended due to the unique storage method is used was rather difficult to find a way to access other data in the library. This is one of the main problems of using third party libraries is they are usually built for a specific task and if your task differs slightly from it, you are going to have to change the library yourself to accommodate your needs. Sometimes this can be easy but at other times impossible.

Miniply and many other parsers are designed to read and sometimes write triangle mesh data. These triangle meshes could be treated as point clouds by just reading their vertex data. However, as I mentioned previously, this is not an easy thing to do with an unfamiliar library that wasn’t designed to do it. Therefore, these other libraries that fall under this condition are not going to be evaluated as it would take too much time away from the development of the project which has no direct impact on the rendering pipeline,

There is also the another argument that if I changed the library to accommodate my task, it would not be an accurate or optimal solution and therefore not be faithful to the original libraries performance. It would be unfair and inaccurate to count a modified version of miniply or any other library against my own when I would have introduced some bias and uncertainty into the results. Therefore, I would direct the reader to look over the publicly available benchmarks of the most common PLY parsers for parsing triangle mesh data that can be found at [ply-parsing-perf](https://github.com/vilya/ply-parsing-perf/) and [Maciej Halber’s ply\_io\_benchmark!](https://github.com/mhalber/ply_io_benchmark)

Even though triangle meshes are different, it would not be unreasonable to assume the benchmarks would be roughly the same for only vertex data. The reason being is that most of the execution time would be in the reading and allocation method of data rather than the limiting factors regarding the libraries specific data structure and method for storing data in.

According to the benchmarks listed above, miniply is the fastest library for reading PLY files. If this could be applied to this project, it would have been the preferred option. However, the benchmarks do offer some results indicating how fast ply parsers are compared to each other. HapPLY was recorded to have a read time of 589.435 milliseconds for the test case of the [ply\_io\_benchmark](https://github.com/mhalber/ply_io_benchmark) set while miniply was the fastest with 35.935 milliseconds. The times don’t mean much for the upcoming PLY parser tests below but the percentage difference does. HapPLY was found to take 16.4x longer than miniply to pass their tests which can potentially be extrapolated to some of the test results below. I must stress that this is not to be taken as a accurate way to test but it does pose an interesting thought to the reader for the potential crossover between reading triangle meshes vs point data for future work and to provide a general indication for how much faster the ply parsers are to each other.

### 4.1.1 Test scenarios

For all test case scenarios the results were run at least 10 times and the average taken to reduce effect of erroneous results and issues relating to the operating system.

##### 4.1. PLY POINT CLOUD PARSERS

###### Test case 1

Test case 1 involved reading data using a PLY parser and directly accessing and add values to std::vector’s. This is not the most efficient way to store values but the stored values by hapPLY cannot be access any other way than by using the get methods of the library and assigning the returned vector to another. For the custom parser, there can be more efficient options such as an array with fixed length due to the flexibility to program the parser ourselves and use the header information to our advantage.

Listing 4.1: Struct used to store data for test case 1

|  |
| --- |
| struct data { std::vector<float> x; std::vector<float> y; std::vector<float> z; std::vector<float> nx; std::vector<float> ny; std::vector<float> nz; std::vector<unsigned char> red; std::vector<unsigned char> green;  std::vector<unsigned char> blue;  }; |

1

2

3

4

5

6

7

8

9

10

11

Test case 1 was executed on another PC with identical specs however, only had 16GB of RAM. This caused disk thrashing to occur when trying to load the Elmstead-Dense point cloud using the hapPLY parser. This was evident by figure 4.1 graph and statistics.

To help identify at what point the disk thrashing was occurring during the read/allocation process, the read method was altered like below where I would assign the read x coordinates vector to a structs vector. Each time I ran this I measured the time and would assign another vertex attributes eg y. Eventually, I got to seven vertex attributes being added and the time it took to assign the vector became non-linear. This was the point where disk thrashing occurred. This is shown in figure 4.2

Listing 4.2: Struct used to store data for test case 1

|  |
| --- |
| happly::PLYData plyIn(filePath) ;  pointData.x = plyIn.getElement( " vertex " ) .getProperty<float >("x" ) ; |

1

2

3

It should also be considered in the following test results that the point data struct uses vectors to store x y z data which favours happily and the compiler optimizations where as the custom read does not. If I was to change the test to an array of fixed length, it may produce fairer results as they both would have to converted or read into a different data structure but it would go against the principle that you would tailor your solution to the implementation eg vectors for happily to make use of the compiler optimizations.

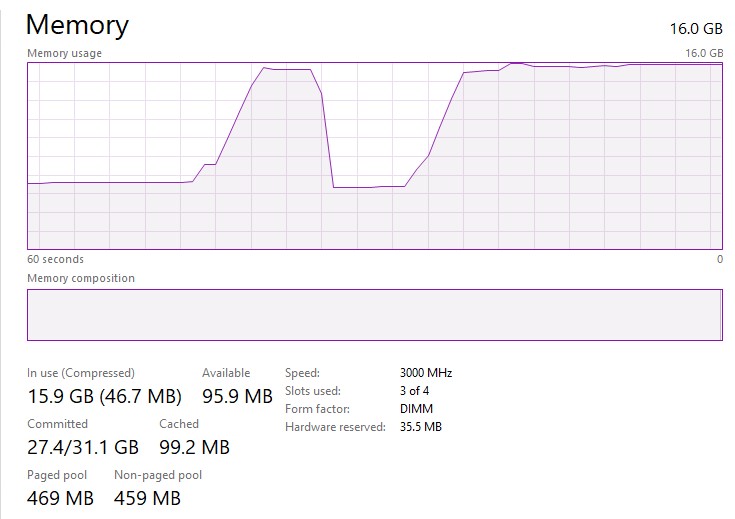


Figure 4.1: Disk thrashing occurring as RAM is at max capacity. Viewed using windows task manager.

### 4.1.2 Results

|  |  |  |  |
| --- | --- | --- | --- |
| Test 1 | Happily | Custom | Difference |
| Fenwick-Sparse | 0.0722s | 0.00735s | 89.819% |
| Elmsted-PhotoScan-Sparse | 0.250s | 0.0368s | 85.274% |
| Fenwick-Dense | 44.254s | 6.5087s | 85.234% |
| Elmsted-PhotoScan-Dense | 70.405s | 10.568s | 84.499% |
| Elmsted-PhotoScan-UltraDense | 246.195s | 35.320s | 85.653% |

The remaining test cases were: test case 2 – read point cloud data from PLY file into a point cloud object. test case 3 - read point cloud data from PLY file into a point cloud object and create a AABB to define an octree size. Test case 4 – Compares directly reading data into an octree vs reading data into a point cloud object and then inserting that data into an octree. This highlights the benefit of reading and inserting data directly into an octree for storage vs storing ait in another object to insert into the octree.

##### 4.1. PLY POINT CLOUD PARSERS

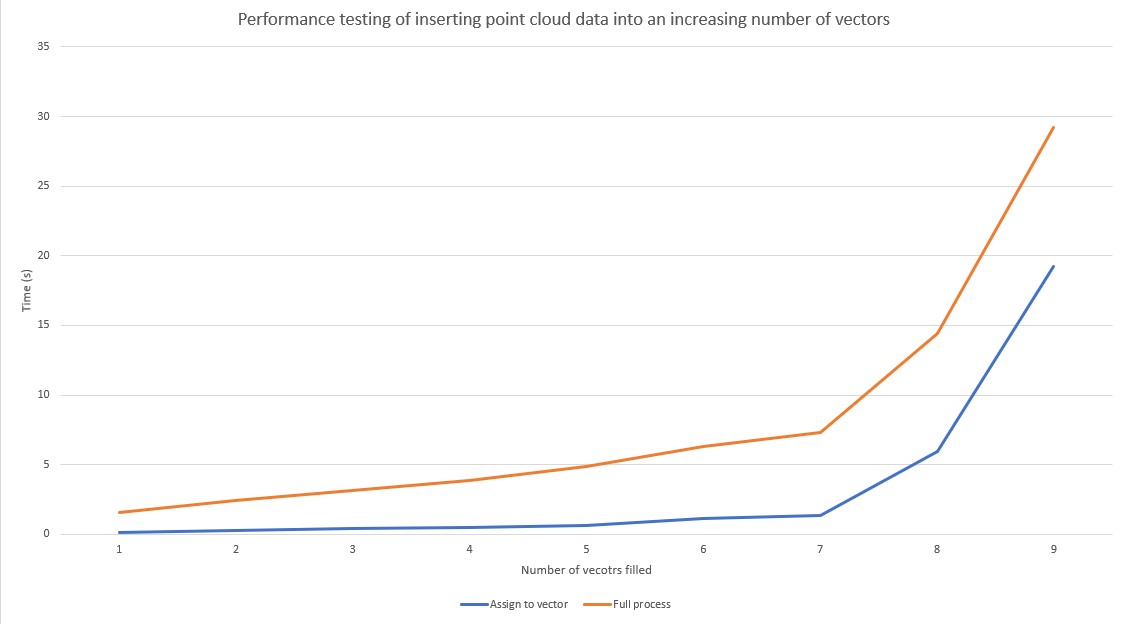


Figure 4.2: Linear relationship up until 8 vector attributes have been assigned then the time became exponential indicating disk thrashing.

|  |  |  |  |
| --- | --- | --- | --- |
| Test 2 | Happily | Custom | Difference |
| Fenwick-Sparse | 0.0513s | 0.00492s | 90.410% |
| Elmsted-PhotoScan-Sparse | 0.265s | 0.0273s | 89.686% |
| Fenwick-Dense | 46.908s | 4.344s | 90.738% |
| Elmsted-PhotoScan-Dense | 77.129s | 7.431s | 90.346% |
| Elmsted-PhotoScan-UltraDense | 303.350s | 27.569s | 90.912% |
| Test 3 | Happily | Custom | Difference |
| Fenwick-Sparse | 0.0524s | 0.00547s | 89.563% |
| Elmsted-PhotoScan-Sparse | 0.272s | 0.0307s | 88.667% |
| Fenwick-Dense | 47.934s | 4.878s | 89.424% |
| Elmsted-PhotoScan-Dense | 78.714s | 8.325s | 89.424% |
| Elmsted-PhotoScan-UltraDense | 300.385s | 31.348s | 89.564% |

Test 1 and 2 caused disk thrashing because they would store PLY data into an object then insert that data into the octree essentially doubling the data stored in RAM. The custom reader I made does not have this issue in later tests because it reads data line by line and insert the data directly into the octree for test case 4. This helps to reduce overhead and reduce the number of parses over the point cloud data. However, it doesn’t necessarily mean its faster to build the octree. I suspect that’s because the compiler struggles to optimize it effectively or my insertion method is inefficient. HapPLY stores the data into vectors and you have to copy the vectors which potentially can be optimized by the compiler but are still slower than directly accessing the vectors stored by hapPLY. For test cases 1 and 2,

For test cases 1, 2 and 3, all the graphs plotting the data for happily and the custom made reader all looked closely the same. This is because all of the readings are a linear relationship between the size of the point cloud and the time it takes. For example, test cases 2 and 3 all have a difference of roughly 90%. Test case 1 has a greater difference for reading FenwickSparse compared to the other files and test cases. I believe this is due to test case 1 being significantly different to the others involving adding data to vectors in a struct and because the total time it takes to do so is a fraction of a second, this can be put down to variance of the compiler optimization.

All the other results of test case 1 follow the same linear pattern except one of the results. This is probably due to a unique optimization issue for that test. The test runs so quicly that other things can impact it. Overall still follow the same trend.

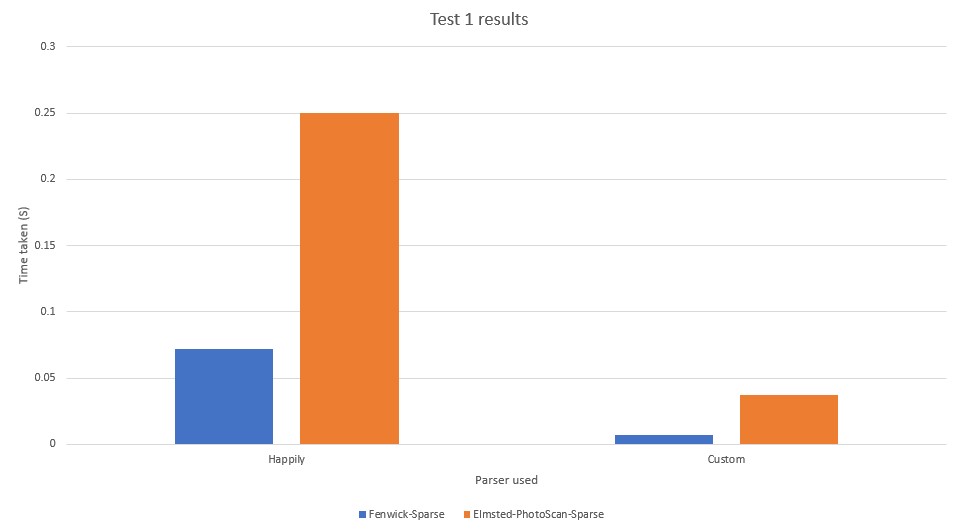


Figure 4.3: Test 1 performance graphs comparing the time taken for happly my own custom reader to read a file

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Test 4 10K | | Happily | | Custom | Difference | |
| Fenwick-Sparse | | 0.0391s | | 0.0311s | 20.554% | |
| Elmsted-PhotoScan-Sparse | | 0.0.0868s | | 0.0644s | 25.650% | |
| Fenwick-Dense | | 31.463s | | 31.777s | -0.995% | |
| Elmsted-PhotoScan-Dense | | 27.237s | | 37.027s | -35.945% | |
| Elmsted-PhotoScan-UltraDense | | N/A | | N/A | N/A | |
| Test 4 100K | | Happily | | Custom | | | Difference | |
| Fenwick-Sparse | | 0.0122s | | 0.00786s | | | 35.370% | |
| Elmsted-PhotoScan-Sparse | | 0.0604s | | 0.0419s | | | 30.504% | |
| Fenwick-Dense | | 14.489s | | 9.750s | | | 32.710% | |
| Elmsted-PhotoScan-Dense | | 19.337s | | 15.792s | | | 18.336% | |
| Elmsted-PhotoScan-UltraDense | | 102.276s | | 118.241s | | | -15.610% | |

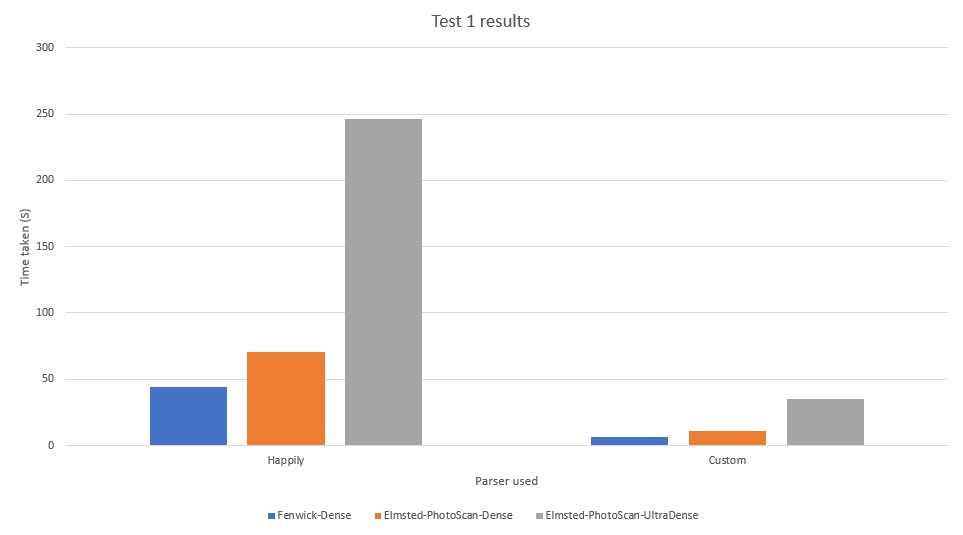


Figure 4.4: More Test 1 performance graphs comparing the time taken for happly my own custom reader to read a file

|  |  |  |  |
| --- | --- | --- | --- |
| Test 4 1 Million | Happily | Custom | Difference |
| Fenwick-Sparse | 0.0214s | 0.00938s | 56.057% |
| Elmsted-PhotoScan-Sparse | 0.0452s | 0.0340s | 24.806% |
| Fenwick-Dense | 13.238s | 9.685s | 26.844% |
| Elmsted-PhotoScan-Dense | 19.078s | 16.804s | 11.919% |
| Elmsted-PhotoScan-UltraDense | 122.638s | 126.159s | -2.871% |

Test 4 indicated that it was better to read and insert directly into my octree implementation as each line was read for point clouds up to 300 million depending on number of octree nodes produced. If there are a lot of octree nodes produced then the insertion time can take longer and thus in the case of having a 10K point limit it is shown in the larger point clouds that require more nodes take the biggest performance hit. Therefore, larger octree node capacity favours inserting points as they are read. The graphs for the tables are in the appendix which may make it clearer to see A.1.

## 4.2 Rendering performance

FPS performance metrics visible at the top of each window. A table will display the values in a clearer and more consider manner for the reader.

### 4.2.1 Mono vs Stereo vision

Rendering the Fenwick-Dense point cloud from a far distance in monoscopic view had an fps value of 13.01 shown in figure 4.5. When rendering in stereo, the fps value was 5.574 shown in figure 4.6. A 57.5% decrease in frames per a second. This is to be expected because the stereoscopic render renders two images with a slight offset drawing points to one view port and then going through the same process again for the other. This effectively doubles the complexity of the operation which is evident by the frames per a second halving. This further supports state in the literature review in section 2.4 of the chosen stereo technique would lead to doubling the complexity and therefore, halving performance.

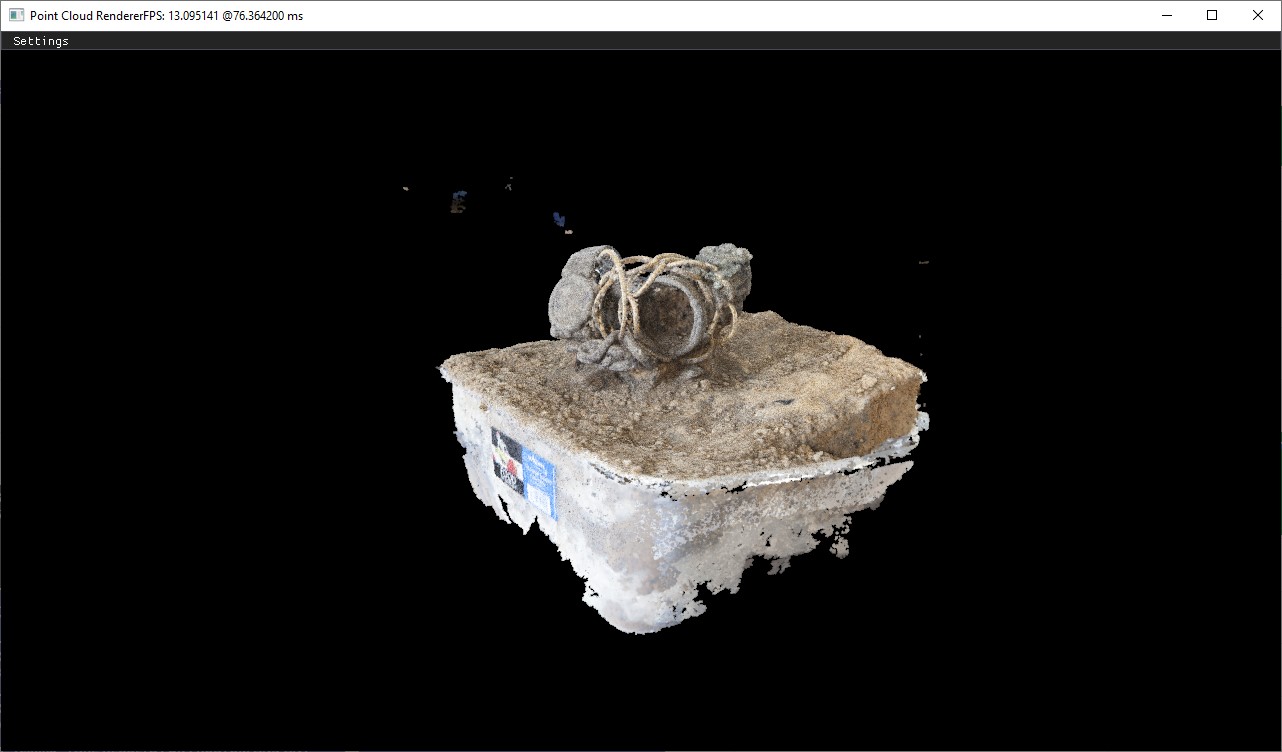


Figure 4.5: Monoscopic view of Fenwick-Dense point cloud

### 4.2.2 Frustum culling

Frustum culling was applied to the rendered Elmstead-Dense scene to really highly the potential benefits of culling points via a frustum. The more points that are culled, the more notable the difference in performance will be. Therefore, it is expected for the tests to have no noticeable effects at long distance but when the view is moved closer where the clipping can be taken advantage of, there should be a significant improvement in fps value.

A far view was chosen for testing initially to provide a base fps value to work with. All points are in view so the frustum has no effect on the performance as all are in view and no points are culled.

A medium distance shot was used to add another comparison between a far and close distance to see how the fps value changes based on distance. It would be nice to record the total number of points rendered each frame however this could impact performance slightly so was left out. The fps improved significantly by 14824%. I suspect this is due to over half the points being culled by the frustum thus reducing the amount of points parsed through the graphics pipeline. If I could do this test again, I would record the number of points being rendered at that time and compare either in a table or graph to see how the number of points culled correlates with the fps performance of the application.

Finally, a close distance was used to maximise the performance of the frustum culling to exploit it as much as possible. This lead to a 50% increase in speed. The increase is not

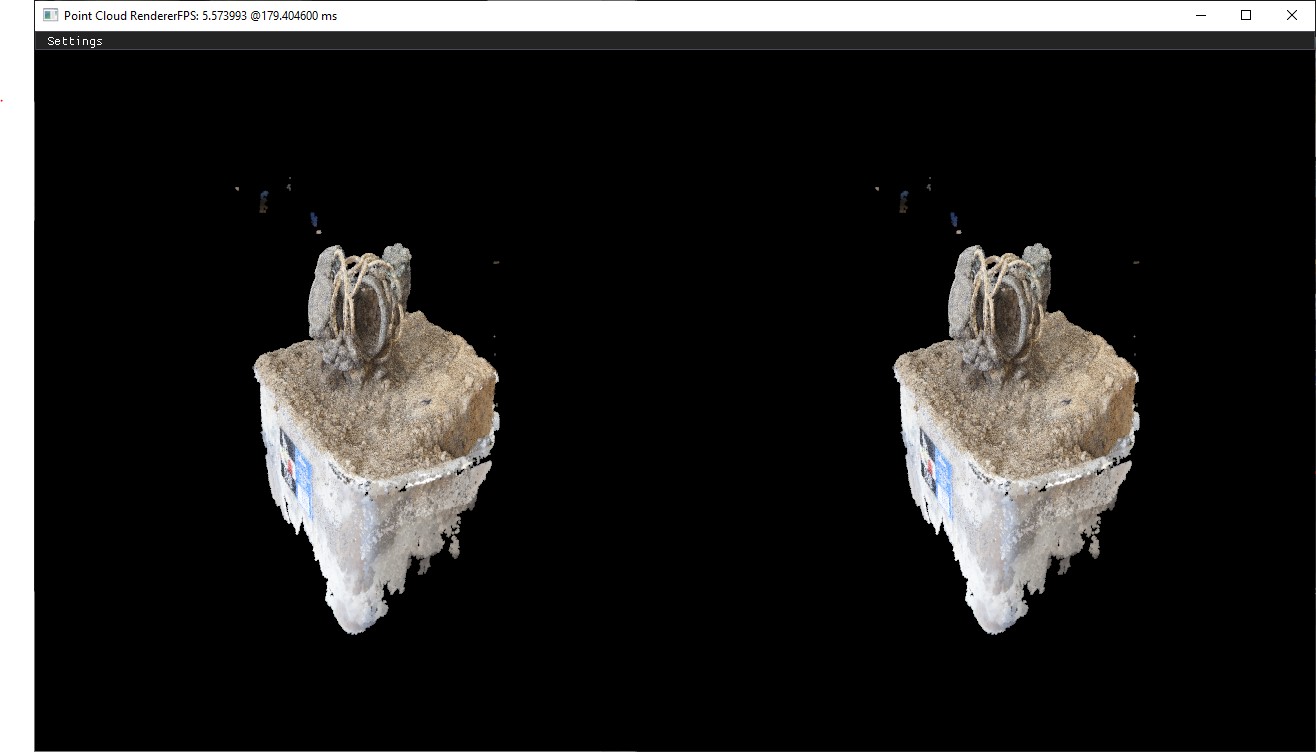


Figure 4.6: Stereoscopic view of Fenwick-Dense point cloud

as much because the initial fps value for figure 4.8 is already a large value (around 1000) therefore percentage wise the increase is smaller but the total fps gain is around the same (around 1000).

Frustum culling was not yet implemented for stereo perspectives and will be left as a future work to help improve stereo performance.

### 4.2.3 Octree effect on performance

The number of nodes an octree has can significantly effect the amount of memory used to store it as well as the traversal time of the tree. As the rendering is integrated into the octree in the form of vertex buffer objects that store sections of vertex data on the GPU for each node, it was hypothesised that the more nodes present in the octree for the given octree implementation would lead to a decrease in rendering performance. An octree was created using the Elmstead-Dense point cloud and the octree nodes has a maximum capacity of 10000 before being subdivided. These octree information is shown below:

Points per level:

Level 0: 0

Level 1: 0

Level 2: 17,220

Level 3: 40,798

Level 4: 33,0475

Level 5: 25,36247

Level 6: 15,582847

Level 7: 40,345953 Nodes per level:

Level 0: 1

Level 1: 8

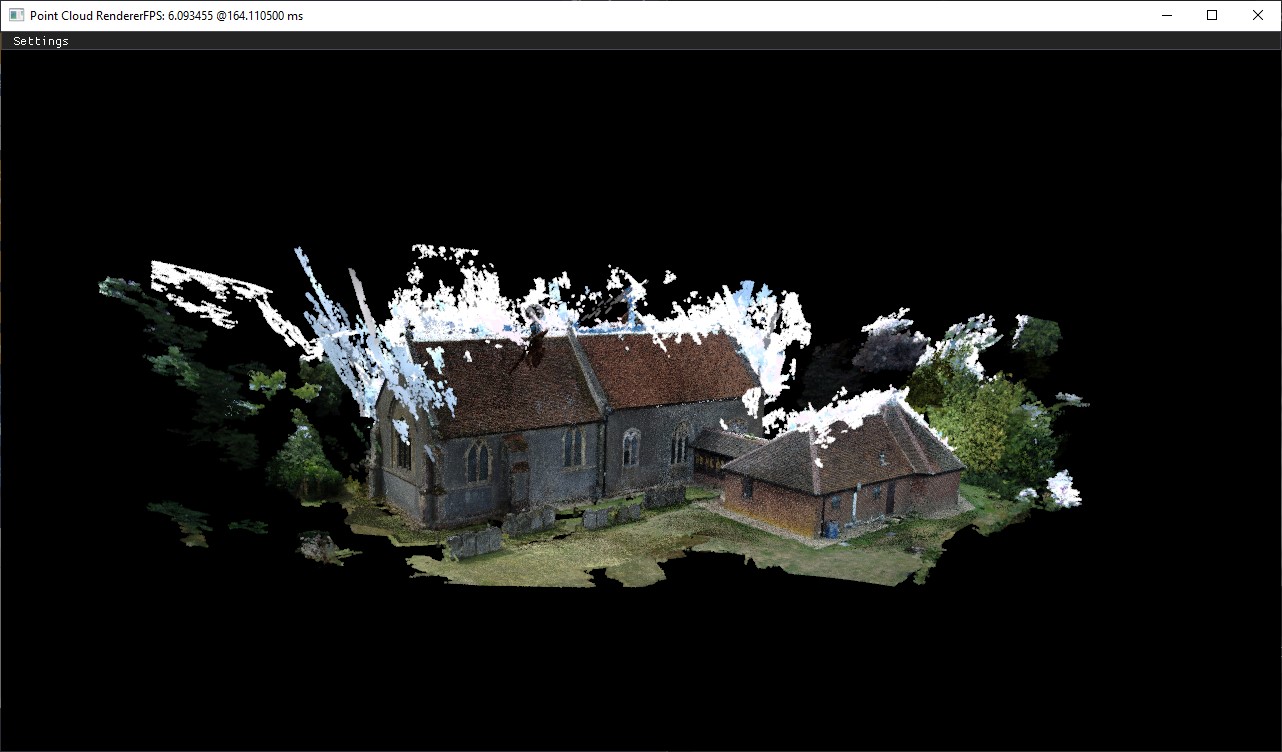


Figure 4.7: Far view of model using frustum - no performance benefit noticed

Level 2: 40

Level 3: 147

Level 4: 534

Level 5: 2080

Level 6: 6966

Level 7: 14119

Total points: 58853540 Total nodes: 23895

The stages of the octree can be rendered progressively by incrementally increasing the depth of the octree rendered. This is shown in figures 4.10 4.11 4.12

As you can see the octree is successfully created and rendered. However, the fps values in figures 4.10 4.11 4.12 cannot be used because the debug octree drawn ontop of the point impacts fps significantly. Therefore the following table was produced to measure the performance of different octree node capacity and how it effects performance:

|  |  |  |
| --- | --- | --- |
| Max node capacity | FPS | Improvement |
| 1k | 10.02s | N/A% |
| 10k | 9.783s | 2.365% |
| 100k | 9.534s | 2.365% |
| 1 Million | 9.232s | 3.167% |

Based on the results table above, it shows a direct correlation between the size of nodes and the FPS of the renderer. It can also by hypothesized that the main difference in performance is from the number and size of the vertex buffer objects being rendered. There is an inherent overhead for draw calls to OpenGL that can effect performance. The 1k max point capacity octree has significantly more nodes than the others and therefore it takes more calls to render the octree’s vertex buffer objects leading to worse performance. There is a 3% increase seen between decreasing the number of octree nodes and therefore vertex buffer objects that lead to a performance increase due to the lower overhead from draw calls.



Figure 4.8: Medium distance view of model using frustum - notable difference

### 4.2.4 Point size

Point size refers to the glPointSize() function value set for each point primitive. Increasing it makes the points appear larger and vise versa. The same view point was take for all tests for point size to isolate the variable to measure the performance. This was mentioned back in section 2.2.4 where the point size was mentioned to effect the performance of the model. The basic idea is that for points that are close to the camera will appear more spaced out and holes will appear thus reducing the burden on the rasterizer due to less overlapping points. Shown in figure 4.13 below is the Elmstead-Sparse point cloud with a relatively close up view of the model. Due to the projection matrix, points are shown to get further apart the close they are relative to the field of view of the user. The point positions are not actually changing but the perspective of them is. You can also see in figure 4.13 that having the same point size ruins the perspective and immersion as points that are further away that can be seen due to the low point density are shown the same size which isn’t normally what happens with perspective. Objects further away appear smaller which gives the impression of distance and perspective.

The remaining images taken for testing are using the Elmstead-Dense point cloud.

The following test compare how changing the point sizes (1 vs 2 vs 4 vs 8) can have an impact on performance. The test was carried out from the same view point to ensure the only variable that could impact performance was the altered point size.

In test cases 1 and 2, there was not much difference between the visual quality of the image however the fps was lowered by 25%. In test cases 2 and 3, the fps was reduced by 29% and for the final comparison between sizes 4 and 8, the fps was reduced by 33%. That is a consistent 4% increase as the point size is doubled. This further support the problem statement above about how the overlapping points which can be clearly seen in figure 4.17 can heavily impact performance if they are overlap. However, changing point size is not the only way points can overlap. For example, when the view point is changing by zooming or moving

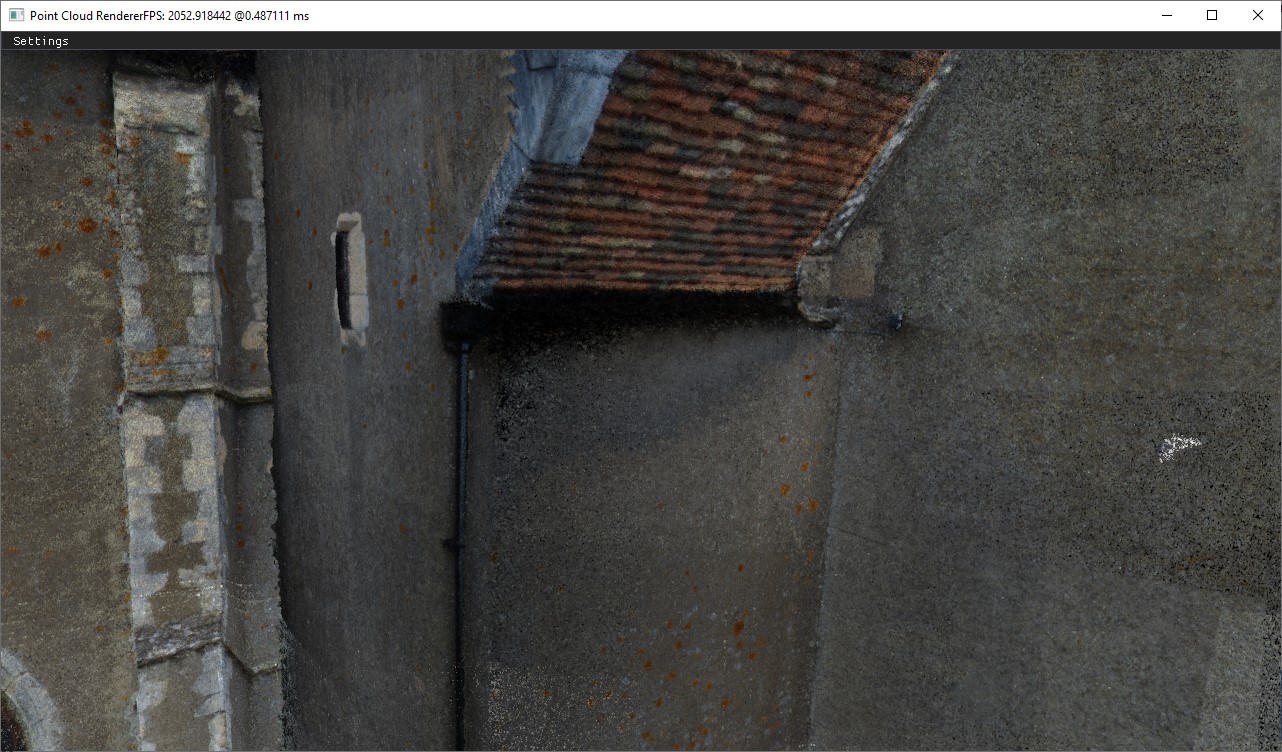


Figure 4.9: Close distance view of model using frustum - significant difference best fps results achieved

towards or away from the model, the perspective will change and the points will appear to get close together. This has the same effect as above due to the change in perspective causing the points to overlap and put a greater burden on the rasterizer.

Again this supports the test results above by showing the fps value decreases by roughly 20%. The reason I didn’t do further tests by moving the model further away was that the perspective projection matrix used has a far frustum plane of a 1000 units and this would cause the points to get culled and therefore not be visible on screen. The points would still impact performance as they would still be rendered up until the perspective projection matrix is applied to the graphics pipeline process where the view points will get culled.

Therefore, these tests support the idea that resizing points based on the projected view distance and position can be very influential on performance and visual quality of a point cloud.

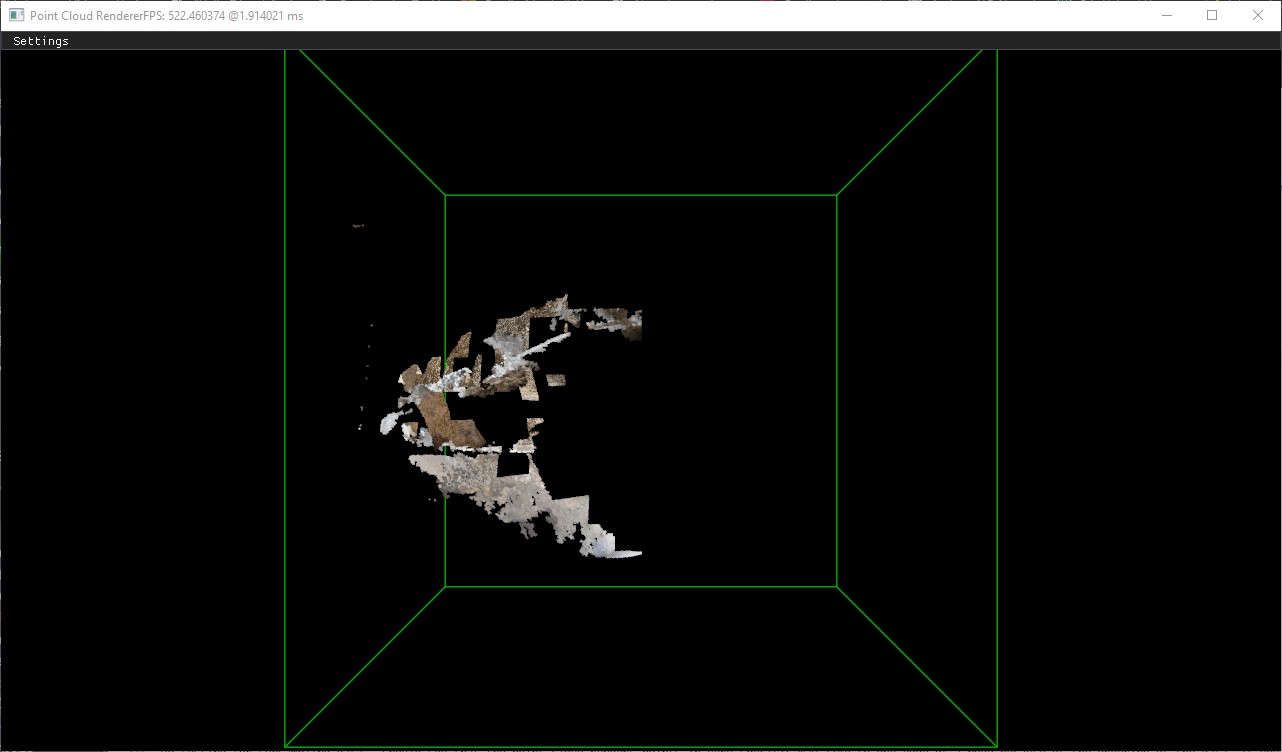


Figure 4.10: Octree rendered to depth 0

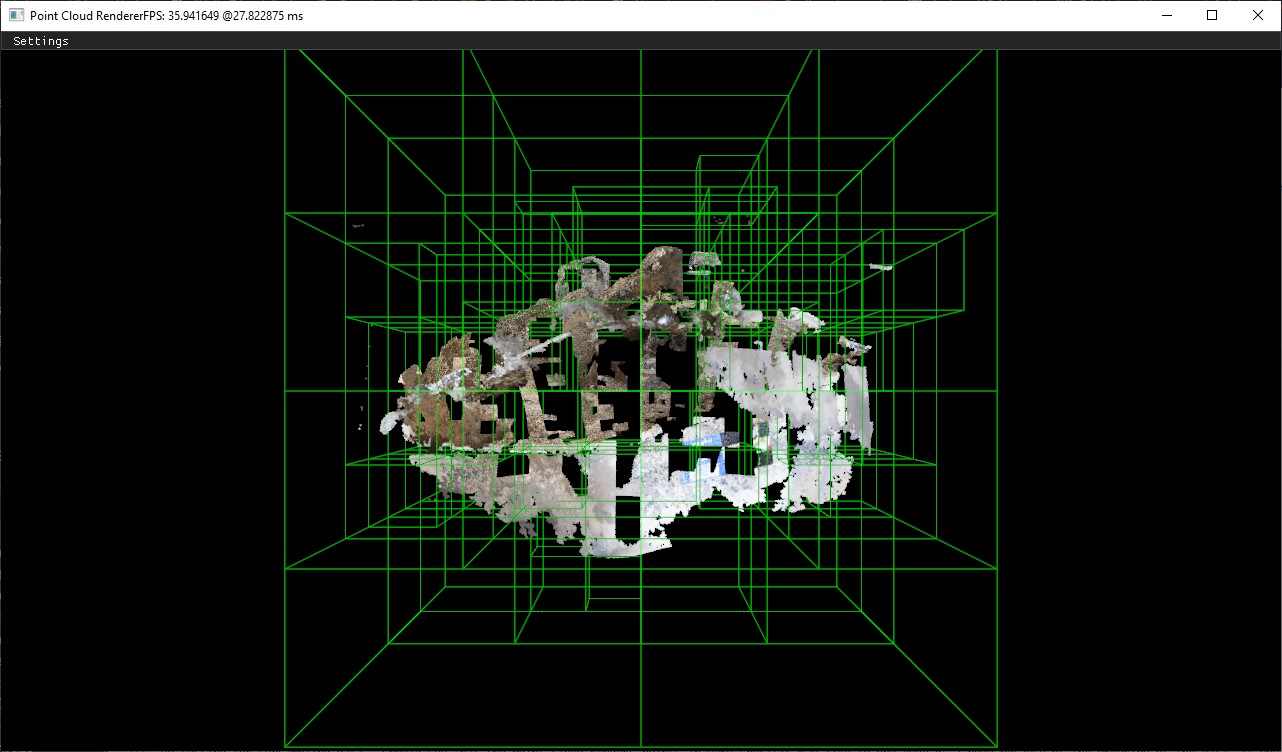


Figure 4.11: Octree rendered to depth 3

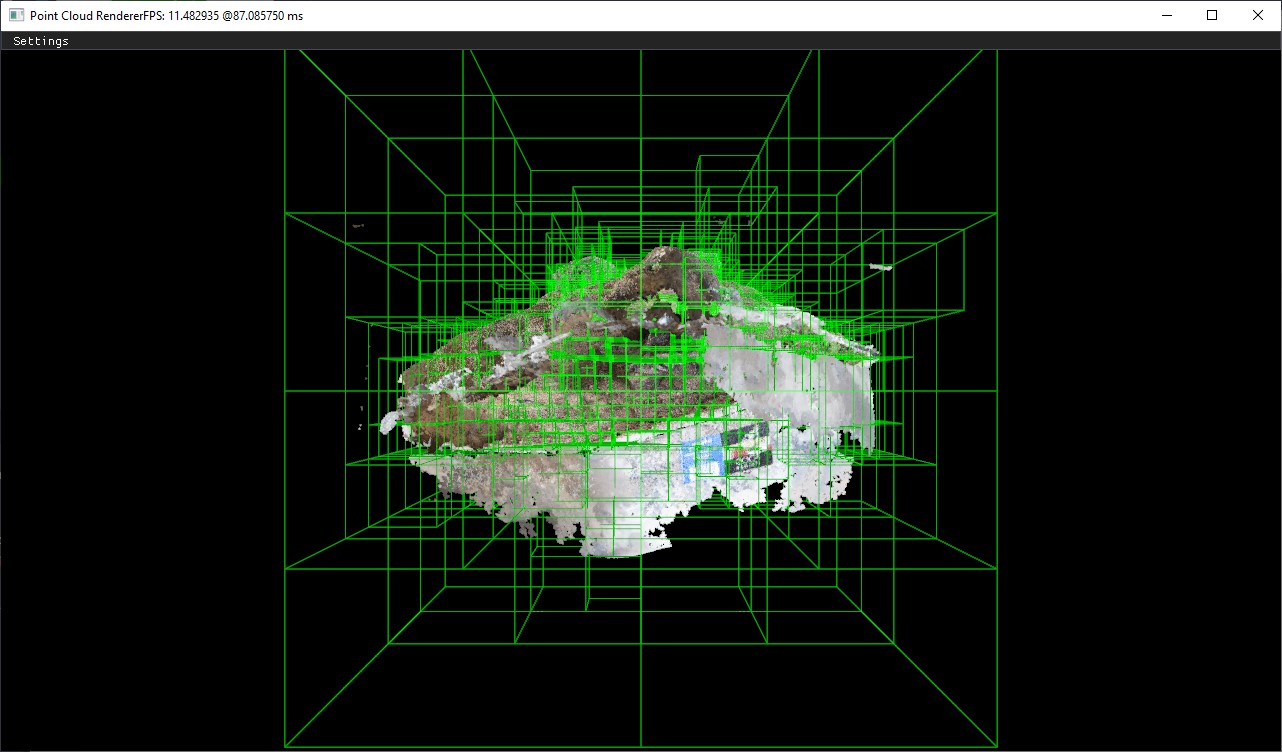


Figure 4.12: Octree rendered to depth 7

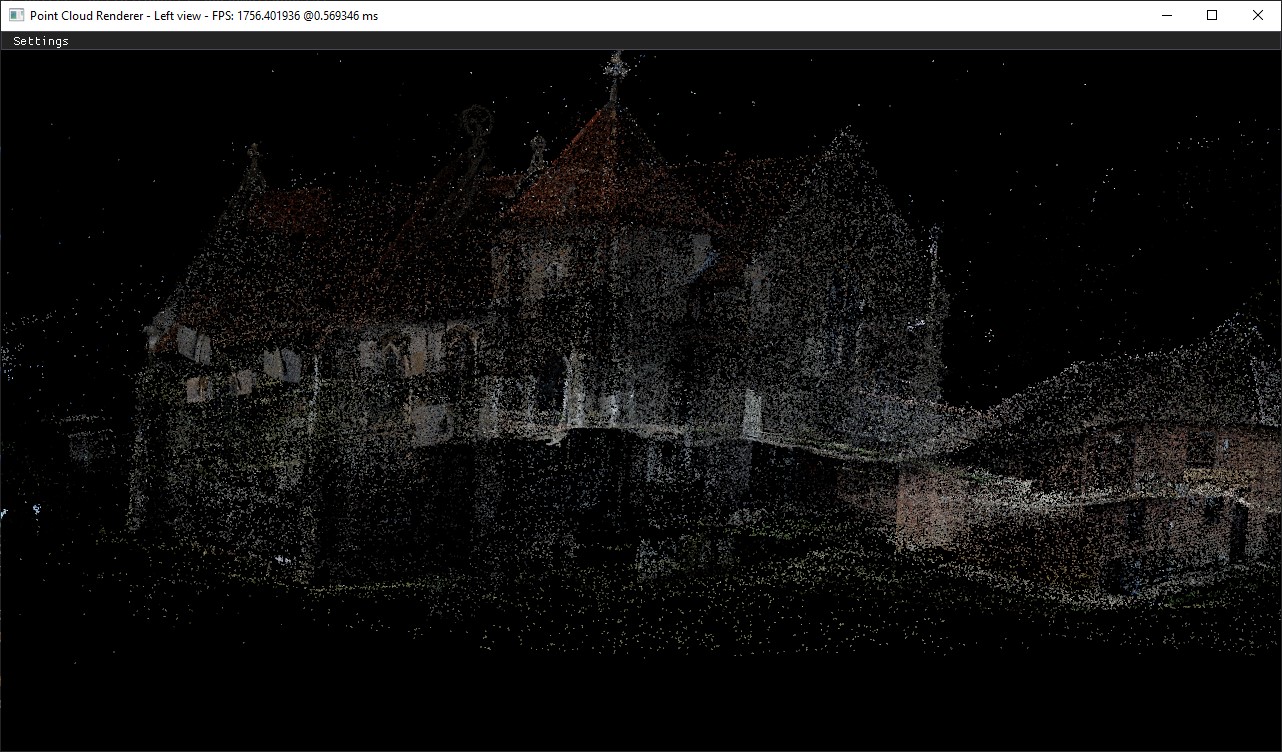


Figure 4.13: Sparse point cloud with holes allows for points to be seen far away leading. Points cause a break in immersion due the points being a fixed size and being able to be seen through other points. Gives a lack of perspective.



Figure 4.14: Point size set to 1

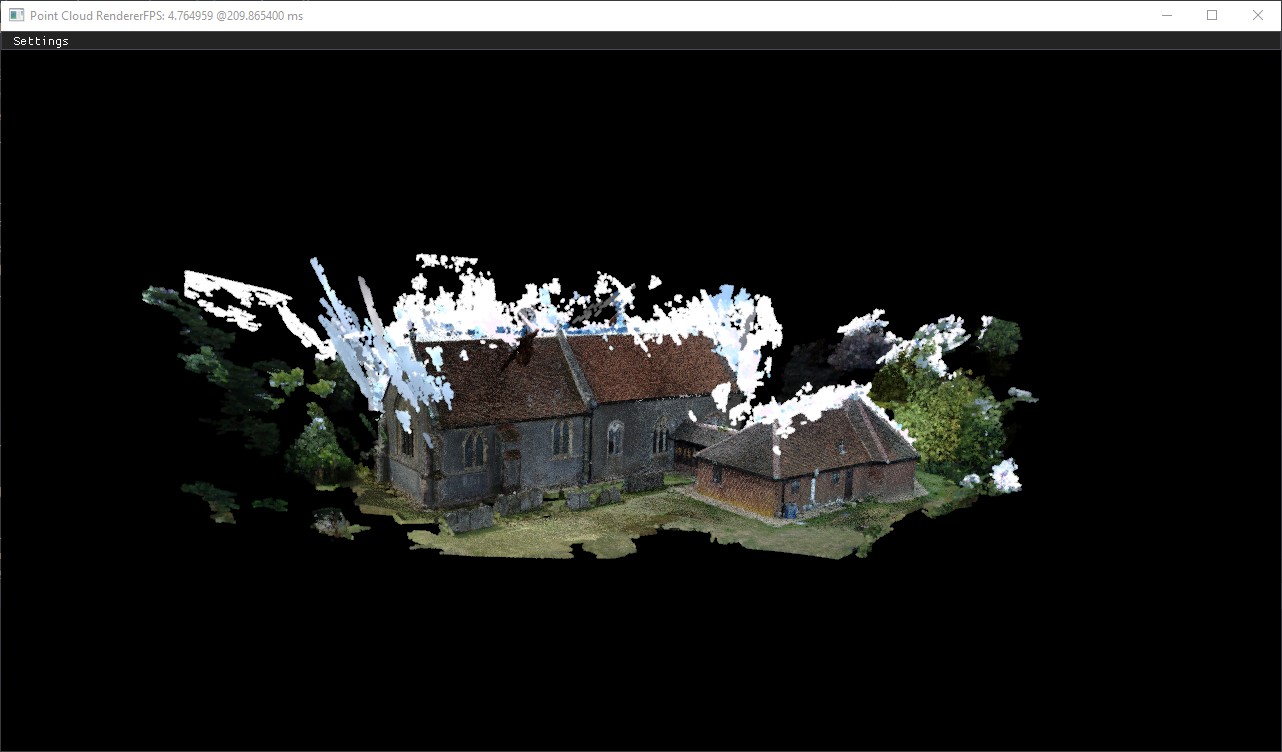


Figure 4.15: Point size set to 2

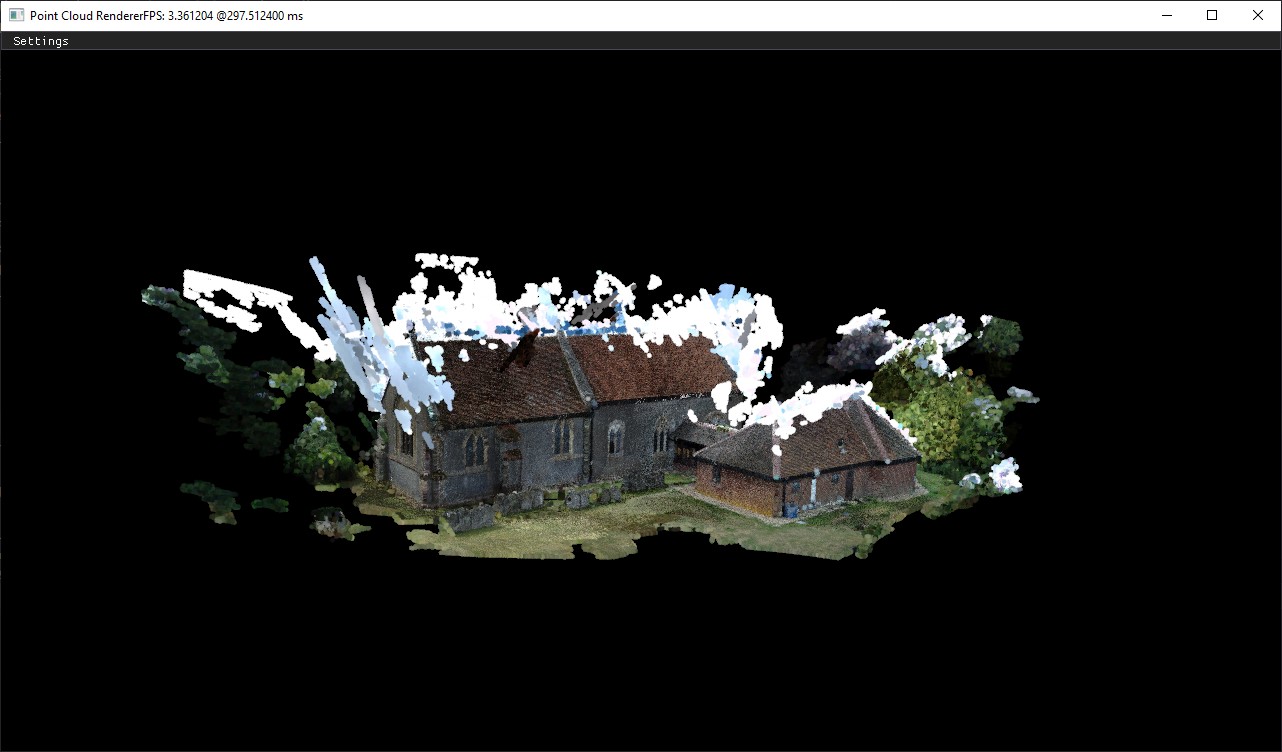


Figure 4.16: Point size set to 4

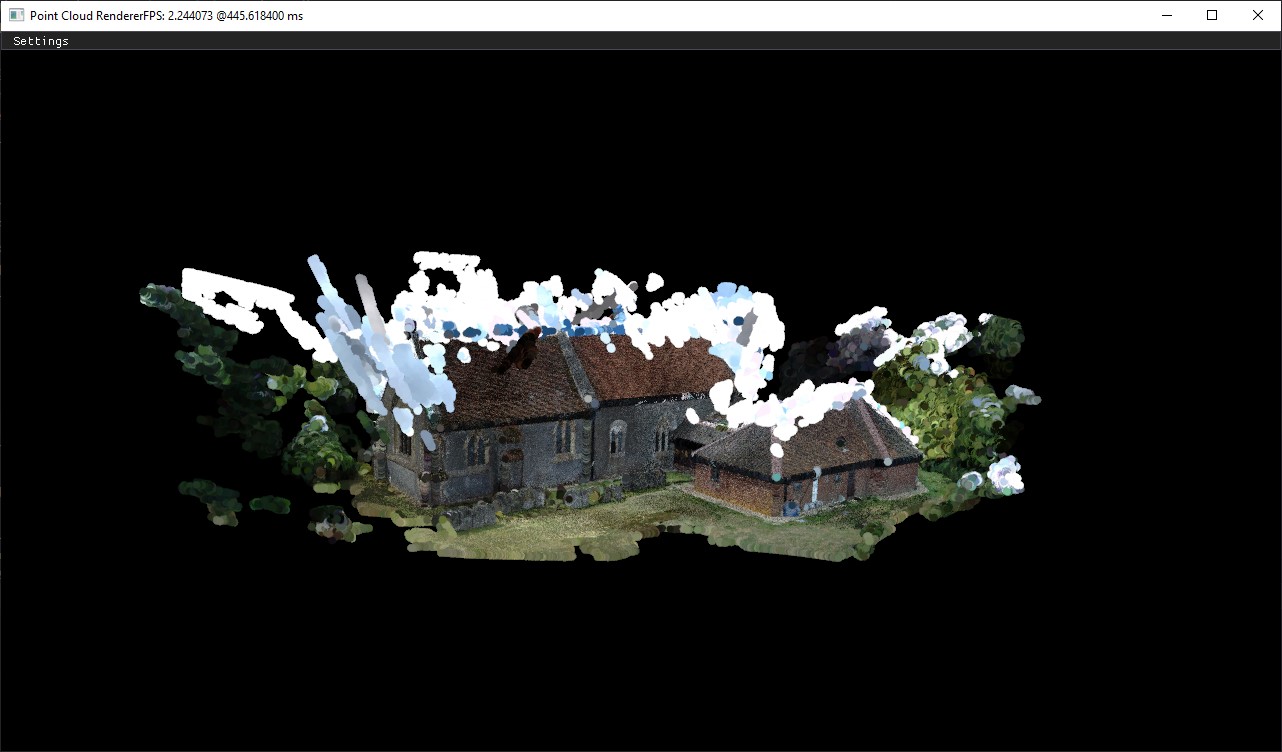


Figure 4.17: Point size set to 8

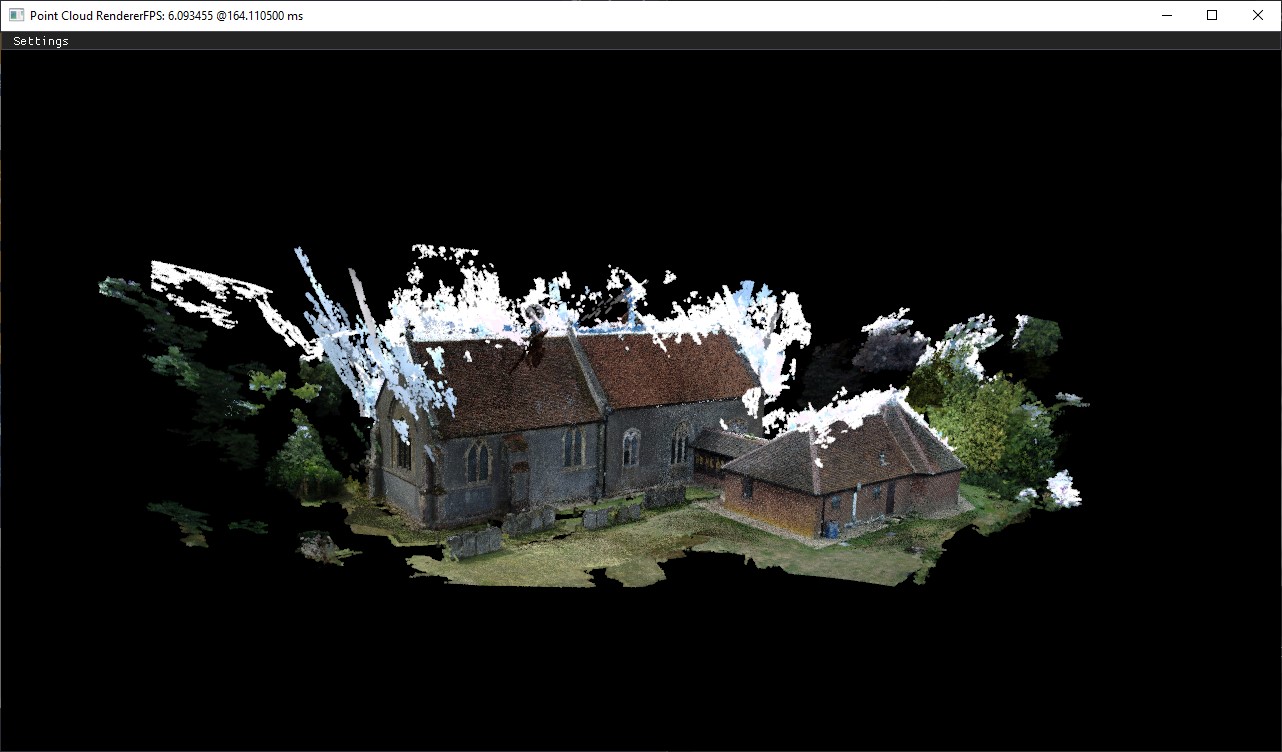


Figure 4.18: Normal view

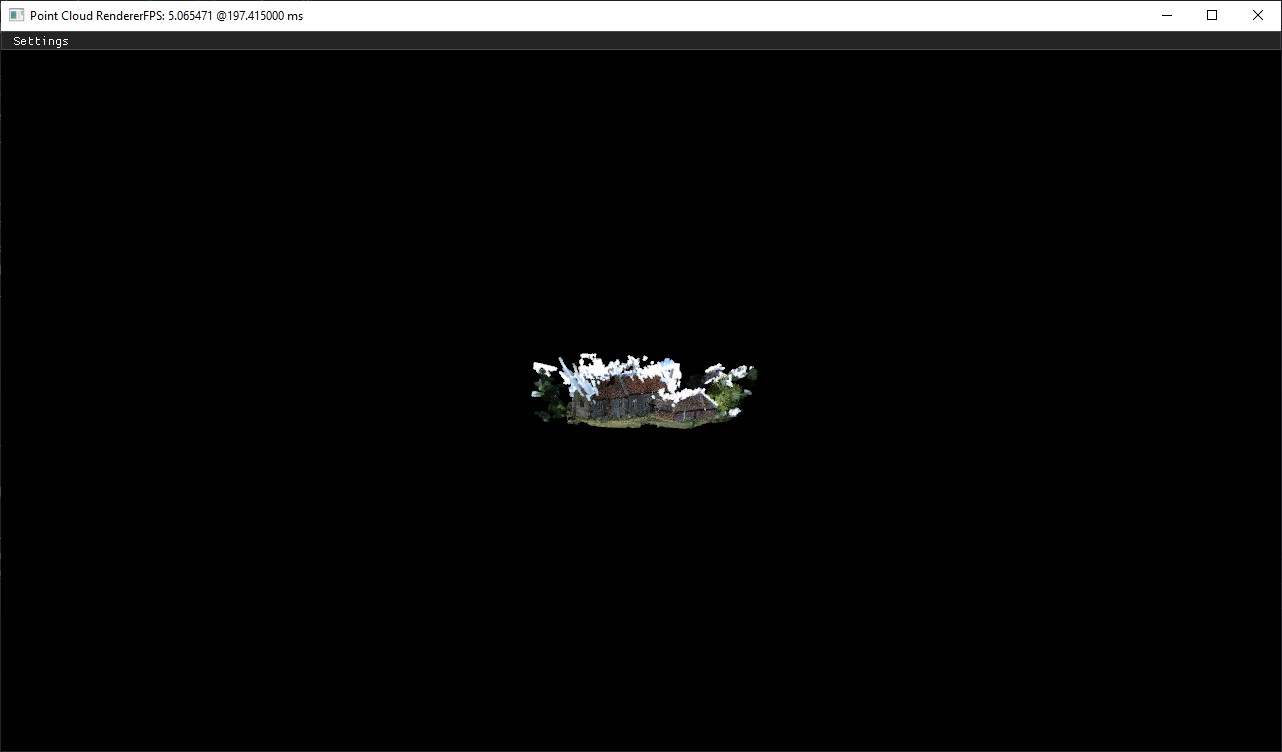


Figure 4.19: Zoomed out view to showcase points can overlap due to the perspective and effect performance

**Chapter 5**

# Conclusion

Overall, the project was successful in producing a point cloud render that could visualise point clouds in real time. A ply parser was created for specifically for reading PLY data that made it possible to view a point cloud in significantly less time than a well established PLY parser on GitHub. It was not the focus of the project but it is a nice edition to the achievements for the overall quality and satisfaction of the point cloud visualiser application built. All the projects are either partially or fully completed. The main goal of real time performance was met along with creating the application from scratch using C++ and OpenGL. The application provides an interactive experience both in mono and stereo providing a base for future expansion to display the point clouds in VR. Frustum culling was effective and successful at optimizing performance for scenes where most of the points are out of view. However, even though I learnt a lot from this project, there are a number of things I would do differently if I was to do it again. I would waste less time reading papers and more time on implementing my own ideas and interests towards rendering OpenGL primitives to the screen as this delayed the project significantly without being able to implement all the ideas I had read about. Another aspect is I would start the writing of this report sooner and focus more on this as it has taken longer than expected to write. Time management would be improved to focus my goals on simpler targets in shorter time frames to make consistent goal hitting a positive target. I think the biggest hurdle I had for this project was learning about OpenGL and computer graphics in general. There is no course besides computer vision that covers anything remotely similar to the work I have conducted and therefore made it very difficult to progress. However, the project overall was still a success with the implementation being particularly successful at reaching the goals of the project.

## 5.1 Future work

It would be interesting to explore illumination techniques such as lighting to be able to use the normal data from the PLY files to more accurately reflect the original 3D point cloud model in the real world. Lighting can have a huge impact on a scenes realism and would be interesting to explore how lighting could be applied to points oppose to triangle meshes. It would also be good to expand the stereo rendering to render to a VR headset to improve the immersion of the point cloud visualiser. People would be able to move around in the point cloud model and get a greater appreciation for the detail of large point cloud surface models.

This would require frustum culling and other optimization algorithms like level of detail to be

48

#### *5.1. FUTURE WORK*

implemented for the stereo rendering to make the experience enjoyable with a low latency. If not, it could cause people to get motion sickness due to the perceived lag. Another algorithm that would be good to implement which I didn’t get round to would be view-base LOD. This could be implemented fairly easily considering the octree and frustum is already created with frustum culling.

One of the interesting topics covered by Potree is point scaling based on the viewing distance and frustum. They were able to scale points in a reasonable manner although not perfect to keep perspective and improve visual quality (fewer holes) by doing so. There is not much research in this area yet or at least not a perfect solution so it could be worth further study to see if there is a better solution for perspective rendering. Expanding the camera to work with quaternions would be useful to avoid gimbal lock and allow a user to view more camera angles without restriction.

Although not related to rendering, improving the time it takes to parse files and insert the data into a data structure would improve satisfaction by reducing the wait time for people wanting to visualise their point cloud in a hurry. An alternative would be to employ out of core optimizations so the loads are reduced because you don’t have to load the entire point cloud data set before viewing. This would be one of the must have goals of the project for anyone wanting to take this further is to implement out of core optimizations to quickly read the necessary files of point clouds in view. An example of this can be seen at [PotreeConverter](https://github.com/potree/PotreeConverter) which implements a very efficient algorithms for streaming and real-time rendering.

Lastly, one of the hottest topics right now for point cloud rendering is utilizing compute shaders to reduce the limitations imposed by hardware. Computer shaders with CUDA can potentially provide for more control and better performance given its an under developed field of study and more specialised towards GPU hardware. [Markus Schütz](https://scholar.google.com/citations?user=BNICw6kAAAAJ&hl=en&oi=sra) has a paper on Rendering Point Clouds with Compute Shaders and Vertex Order Optimization is particularly useful for anyone wanting to explore this avenue for compute based shader in CUDA. It would also be good to explore how different hardware effects the rendering capabilities of points vs triangle meshes to see what optimizations are made for triangles meshes in the architectures and if anything can be optimized for point primitives.

It is worth mentioning on a last note that [Markus Schütz](https://scholar.google.com/citations?user=BNICw6kAAAAJ&hl=en&oi=sra) has come great papers on the topic of rendering and is the creator of Potree so if anyone was to take this further or want to get into it, I would highly recommend looking at his work.

**Bibliography**

1. M. Engelcke, D. Rao, D. Z. Wang, C. H. Tong, and I. Posner, “Vote3deep: Fast object detection in 3d point clouds using efficient convolutional neural networks,” in *2017 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2017, pp. 1355–1361.
2. D. Prokhorov, “A convolutional learning system for object classification in 3-d lidar data,” *IEEE Transactions on neural networks*, vol. 21, no. 5, pp. 858–863, 2010.
3. B. Okumura, M. R. James, Y. Kanzawa, M. Derry, K. Sakai, T. Nishi, and D. Prokhorov, “Challenges in perception and decision making for intelligent automotive vehicles: A case study,” *IEEE Transactions on Intelligent Vehicles*, vol. 1, no. 1, pp. 20–32, 2016.
4. A. Baik, “From point cloud to jeddah heritage bim nasif historical house–case study,” *Digital applications in archaeology and cultural heritage*, vol. 4, pp. 1–18, 2017.
5. E. Canevese and T. De Gottardo, “Beyond point clouds and virtual reality. innovative methods and technologies for the protection and promotion of cultural heritage,” *The International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. 42, p. 685, 2017.
6. W.-C. Hsu, P. T.-Y. Shih, H.-C. Chang, J.-K. Liu *et al.*, “A study on factors affecting airborne lidar penetration,” *Terr. Atmos. Ocean. Sci*, vol. 26, pp. 241–251, 2015.
7. “Lidar point cloud - usgs national map 3dep downloadable data collection,” [https: //www.sciencebase.gov/catalog/item/4f70ab64e4b058caae3f8def,](https://www.sciencebase.gov/catalog/item/4f70ab64e4b058caae3f8def) accessed: 2022-1205.
8. M. Schütz *et al.*, “Potree: Rendering large point clouds in web browsers,” *Technische Universität Wien, Wieden´*, 2016.
9. J.-L. Lugrin, M. Cavazza, F. Charles, M. Le Renard, J. Freeman, and J. Lessiter, “Immersive fps games: user experience and performance,” in *Proceedings of the 2013 ACM international workshop on immersive media experiences*, 2013, pp. 7–12.
10. K. Khoshelham, “Accuracy analysis of kinect depth data,” in *ISPRS workshop laser scanning*, vol. 38, no. 1, 2011.
11. A. Eltner and G. Sofia, “Structure from motion photogrammetric technique,” in *Developments in Earth surface processes*. Elsevier, 2020, vol. 23, pp. 1–24.

50

*BIBLIOGRAPHY*

1. J. P. Grossman and W. J. Dally, “Point sample rendering,” in *Eurographics Workshop on Rendering Techniques*. Springer, 1998, pp. 181–192.
2. R. Pintus, E. Gobbetti, and M. Agus, “Real-time rendering of massive unstructured raw point clouds using screen-space operators,” in *Proceedings of the 12th International conference on Virtual Reality, Archaeology and Cultural Heritage*, 2011, pp. 105–112.
3. P. Rosenthal and L. Linsen, “Image-space point cloud rendering,” in *Proceedings of Computer Graphics International*, 2008, pp. 136–143.
4. S. Gargoum and K. El-Basyouny, “Effects of lidar point density on extraction of traffic signs: A sensitivity study,” *Transportation research record*, vol. 2673, no. 1, pp. 41–51, 2019.
5. M. Edl, M. Mizerák, and J. Trojan, “3d laser scanners: history and applications,” *Acta Simulatio-International Scientific Journal about Simulation*, vol. 4, no. 4, pp. 1–5, 2018.
6. M. Schütz, B. Kerbl, and M. Wimmer, “Rendering point clouds with compute shaders and vertex order optimization,” in *Computer Graphics Forum*, vol. 40, no. 4. Wiley Online Library, 2021, pp. 115–126.
7. T. Ohashi, M. Ishino, K. Yamamoto, H. Fuji, and K. Fujioka, “Simultaneous acquisition of color and distance information by lidar with rgb visible laser diodes,” *Optical Review*, vol. 28, no. 4, pp. 516–523, 2021.
8. V. Novák, “Modern use of lidar is not only in the security field,” 2021, accessed: Nov 27, 2022. [Online]. Available: [https://www.linkedin.com/pulse/ modern-use-lidar-only-security-field-vladimir-novÃa˛k/](https://www.linkedin.com/pulse/modern-use-lidar-only-security-field-vladimir-nov%E1k/)
9. C. Chen, J. Guo, H. Wu, Y. Li, and B. Shi, “Performance comparison of filtering algorithms for high-density airborne lidar point clouds over complex landscapes,” *Remote Sensing*, vol. 13, no. 14, p. 2663, 2021.
10. X.-F. Han, J. S. Jin, M.-J. Wang, W. Jiang, L. Gao, and L. Xiao, “A review of algorithms for filtering the 3d point cloud,” *Signal Processing: Image Communication*, vol. 57, pp. 103–112, 2017.
11. G. Luetzenburg, A. Kroon, and A. A. Bjørk, “Evaluation of the apple iphone 12 pro lidar for an application in geosciences,” *Scientific reports*, vol. 11, no. 1, pp. 1–9, 2021.
12. M. Kulawiak and Z. Lubniewski, “Processing of lidar and multibeam sonar point cloud data for 3d surface and object shape reconstruction,” in *2016 Baltic Geodetic Congress (BGC Geomatics)*. IEEE, 2016, pp. 187–190.
13. S. van Riel, “Exploring the use of 3d gis as an analytical tool in archaeological excavation practice,” Ph.D. dissertation, 06 2016.
14. A.-V. Vo, D. F. Laefer, and M. Bertolotto, “Airborne laser scanning data storage and indexing: state-of-the-art review,” *International journal of remote sensing*, vol. 37, no. 24, pp. 6187–6204, 2016.
15. D. Huber, “A new, open standard for 3d imaging data,” 2011.

52 *BIBLIOGRAPHY*

1. K. McHenry and P. Bajcsy, “An overview of 3d data content, file formats and viewers,” *National Center for Supercomputing Applications*, vol. 1205, p. 22, 2008.
2. Telecom ParisTech & EDF, “Cloudcompare - 3d point cloud and mesh processing software,” 2003, accessed: Nov 28, 2022. [Online]. Available: [https://www.danielgm. net/cc/](https://www.danielgm.net/cc/)
3. “The pcd (point cloud data) file format,” [https://pointclouds.org/documentation/ tutorials/pcd\_file\_format.html,](https://pointclouds.org/documentation/tutorials/pcd_file_format.html) accessed: 2022-12-05.
4. R. B. Rusu and S. Cousins, “3d is here: Point cloud library (pcl),” in *2011 IEEE international conference on robotics and automation*. IEEE, 2011, pp. 1–4.
5. X.-F. Han, J. S. Jin, M.-J. Wang, and W. Jiang, “Guided 3d point cloud filtering,” *Multimedia Tools and Applications*, vol. 77, no. 13, pp. 17 397–17 411, 2018.
6. B. C. Ooi, “Spatial kd-tree: A data structure for geographic database,” in *Datenbanksysteme in Büro, Technik und Wissenschaft*. Springer, 1987, pp. 247–258.
7. M. Levoy and T. Whitted, “The use of points as a display primitive,” 1985.
8. C. Scheiblauer, “Interactions with gigantic point clouds,” Ph.D. dissertation, 2014.
9. G. Guennebaud and B. Jacob, “others. eigen v3,” *URL http://eigen. tuxfamily. org*, 2010.
10. M. Muja, “Approximate nearest neighbors with automatic algorithm configuration,” in *Proc. Int. Conf. on Computer Vision Theory and Applications, Lisbon, 2009*, 2009.
11. J. B˛edkowski, M. Pełka, K. Majek, T. Fitri, and J. Naruniec, “Open source robotic 3d mapping framework with ros—robot operating system, pcl—point cloud library and cloud compare,” in *2015 International Conference on Electrical Engineering and Informatics (ICEEI)*. IEEE, 2015, pp. 644–649.
12. R. Schnabel and R. Klein, “Octree-based point-cloud compression.” *PBG@ SIGGRAPH*, vol. 3, 2006.
13. “Visual impact comparisons and measurements,” [https://en.wikipedia.org/wiki/Level\_ of\_detail\_(computer\_graphics),](https://en.wikipedia.org/wiki/Level_of_detail_(computer_graphics)) accessed: 2022-12-05.
14. M. Schutz, “Potreeconverter-uniform partitioning of point cloud data into an octree,” 2014.
15. M. Corsini, P. Cignoni, and R. Scopigno, “Efficient and flexible sampling with blue noise properties of triangular meshes,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 18, no. 6, pp. 914–924, 2012.
16. V. Barney Britton, “3d video primer, part 1,” 09 2011, accessed March 29, 2022. [Online]. Available: [https://www.dpreview.com/articles/3797744816/ 3d-video-primer-part-1](https://www.dpreview.com/articles/3797744816/3d-video-primer-part-1)
17. “Monoscopic vs stereoscopic 360 vr,” accessed March 29, 2022. [Online]. Available:

<https://immersionvr.co.uk/blog/monoscopic-vs-stereoscopic-360-vr/>

*BIBLIOGRAPHY*

1. A. Bulbul, Z. Cipiloglu, and T. Capin, “A perceptual approach for stereoscopic rendering optimization,” *Computers & Graphics*, vol. 34, no. 2, pp. 145–157, 2010.
2. M. Segal, K. Akeley, C. Frazier, J. Leech, and P. Brown, “The opengl graphics system: A specification (version 2.0). silicon graphics,” *Inc., October*, vol. 22, 2004.
3. A. Blackert, “Evaluation of multi-threading in vulkan,” 2016.
4. M. Segal and K. Akeley, “The opengl graphics system: A specification (version 4.6 (core profile)-october 22, 2019),” *The Khronos Group Inc.[cit. 2020-12-05]. Dostupné z: https://www. khronos. org/registry/0penGL/specs/gl/glspec46. core. pdf*, 2021.
5. “Glut - the opengl utility toolkit,” [https://www.opengl.org/resources/libraries/glut/ glut\_downloads.php,](https://www.opengl.org/resources/libraries/glut/glut_downloads.php) accessed: 2022-12-05.
6. “The freeglut project :: About,” [https://freeglut.sourceforge.net,](https://freeglut.sourceforge.net/) accessed: 2022-12-05. [50] “Glfw: An opengl library,” [https://www.glfw.org,](https://www.glfw.org/) accessed: 2022-12-05.
7. “The opengl extension wrangler library,” [https://glew.sourceforge.net,](https://glew.sourceforge.net/) accessed: 202212-05.
8. “gl3w: Simple opengl core profile loading,” [https://github.com/skaslev/gl3w,](https://github.com/skaslev/gl3w) accessed: 2022-12-05.
9. “glad (multi-language gl/gles/egl/glx/wgl loader-generator),” [https://glad.dav1d.de,](https://glad.dav1d.de/) accessed: 2022-12-05.
10. J. Kessenich, G. Sellers, and D. Shreiner, *OpenGL Programming Guide: The official guide to learning OpenGL, version 4.5 with SPIR-V*. Addison-Wesley Professional, 2016.
11. “Hello triangle,” [https://learnopengl.com/Getting-started/Hello-Triangle,](https://learnopengl.com/Getting-started/Hello-Triangle) accessed: 2022-12-05.
12. A. Nakano, R. K. Kalia, A. Sharma, P. Vashishta, S. Ogata, and F. Shimojo,

“Virtualization-aware application framework for hierarchical multiscale simulations on a grid,” in *Computational Methods In Large Scale Simulation*. World Scientific, 2005, pp. 229–243.

1. F. I. de l’Automobile, “2022 formula 1 technical regulations,” 02 2022, federation Internationale de l’Automobile, Available: [https://www.fia.com/sites/default/files/formula\_ 1\_-\_technical\_regulations\_-\_2022\_-\_iss\_9\_-\_2022-02-18.pdf,](https://www.fia.com/sites/default/files/formula_1_-_technical_regulations_-_2022_-_iss_9_-_2022-02-18.pdf) Accessed: 2022-03-15.

**Appendix A**

# Appendix

## A.1 Test 4 PLY parser graphs

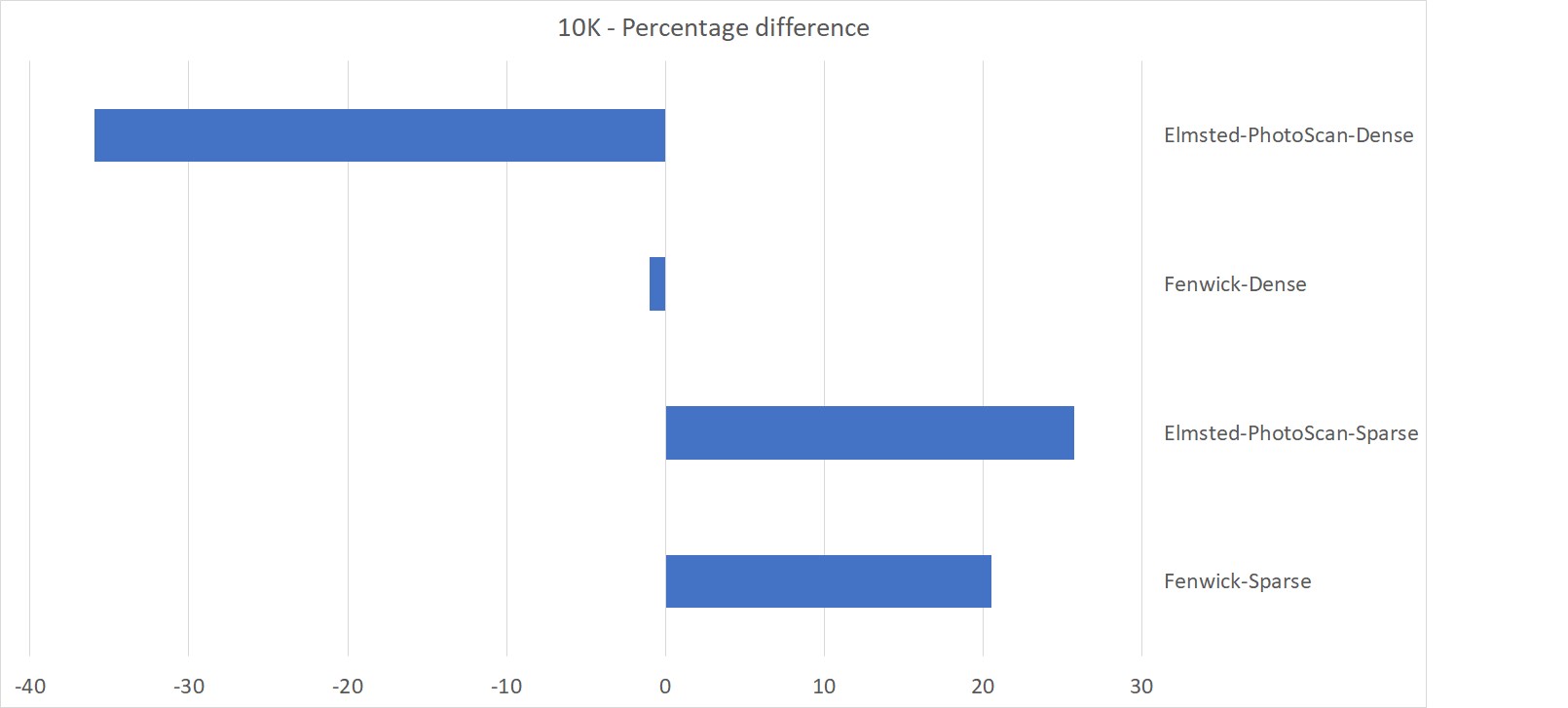


Figure A.1: Test 4 10k

[My code can be found here. If not, it will be there by the time of the presentation.](https://cseegit.essex.ac.uk/21-22-ce869/21-22_CE869_marshall_hayden_a)

54

#### *A.1. TEST 4 PLY PARSER GRAPHS*

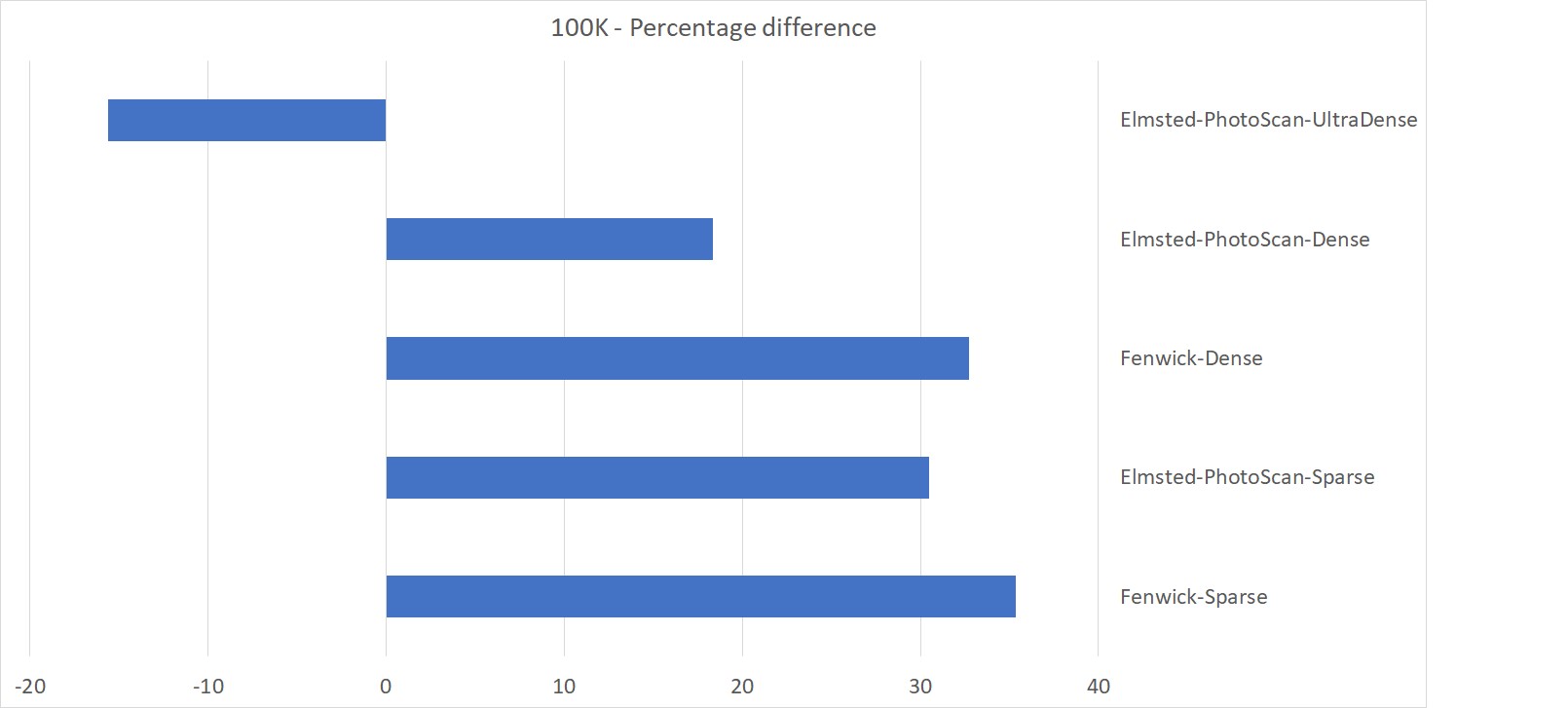


Figure A.2: Test 4 100k

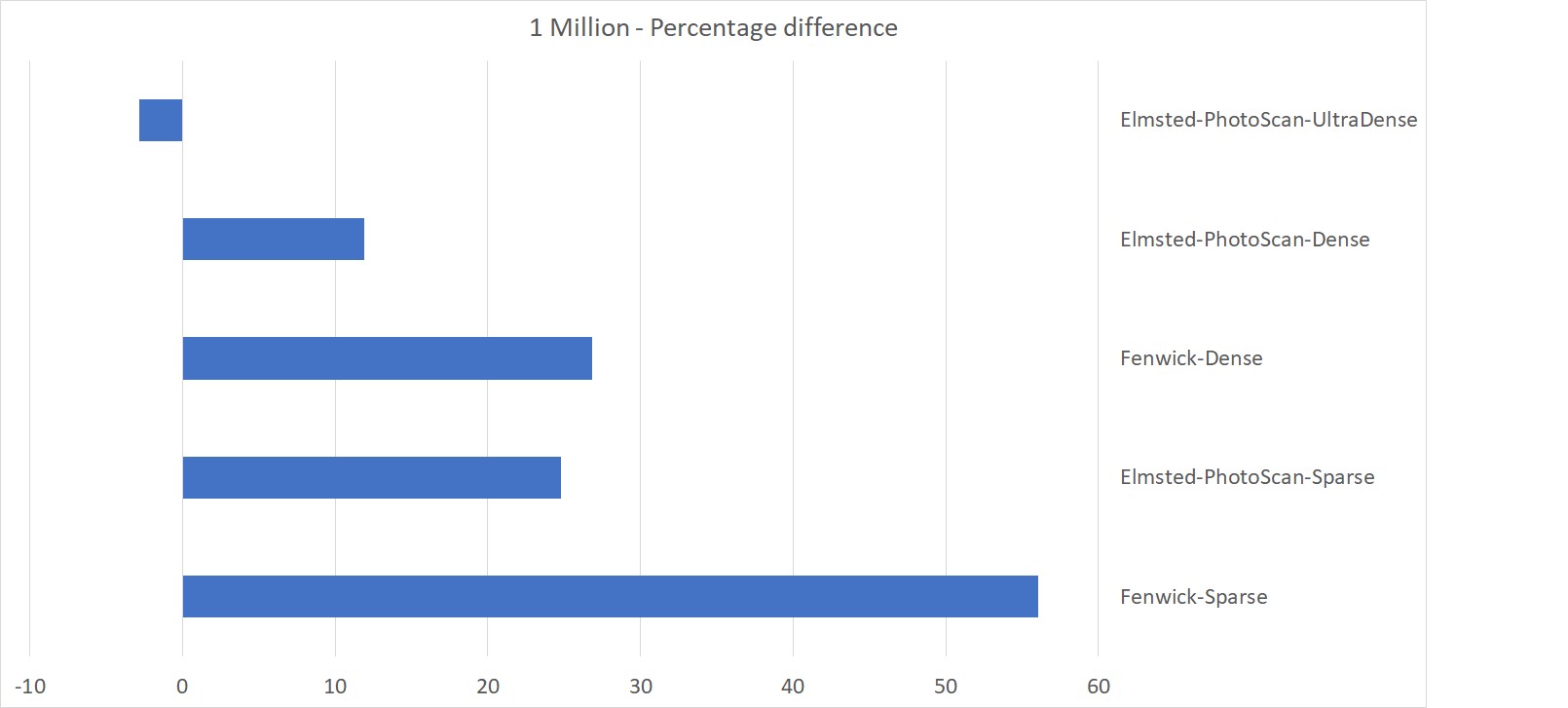


Figure A.3: Test 4 1 Million