Compression of Point Clouds Report

Authors

Lin Zhang

CSCI 599: Special Topics
Instructor: Dr. Shahram Ghandeharizadeh

December 14, 2022
University of Southern California, Los Angeles, California 90007

Compression of Point Clouds Report

(ABSTRACT)

Nowadays, point clouds are widely used in 3D visualization, VR, AR, and HoloDesk systems. When rendering a 3D model from a point cloud perspective, a large number of vertices are used to form the model. Since the number of vertices can be quite large, it is important to find a way to minimize their number to make the compressed result easier to transfer and re-render. This is particularly important in the Flying Light Specks (FLS) system [3], as it can significantly reduce the number of active (turned on) FLSs, save a lot of energy, and provide a better outline of a 3D model. The compression algorithm presented in this report focuses on the human view perspective and achieves the compression result from a specific direction for each run.

In summary, the main goal of this project is to develop an algorithm that can reduce the number of points used in a 3D point cloud while maintaining a clear 3D display. As an added feature, this solution also includes a simple implementation of eye-tracking to make it easier to visualize the compression results.

Contents

List of Figures

Chapter 1

Introduction

Before starting to implement a compression algorithm and eye tracking, it is important to consider the reasons why compressing point clouds is important. With the growing popularity of virtual and augmented reality, as well as the emergence of the metaverse, technologies like VR, AR, and HoloDesk are becoming more and more integrated into people's lives. In movies and shows like Star Trek, we have seen advanced display devices that can show 3D models and even interact with humans. These examples hint at a potential future where human interaction is fundamentally different.

From a research perspective, since decades ago, the research on graphics compression has never stopped. So far, there are a lot of amazing articles and algorithms dealing with compression technology. In this survey [5], they mentioned over 100 graphics compression algorithms. These algorithms are mainly focused on how to compress and restore a 3D model on a computer screen. All these technologies are already being widely used in the current graphics rendering product. In the past few decades, due to the improvement of these rendering technologies, we start to see some impressive products in the VR and AR field.

In this project, I am focusing on using a specific point of view to compress point clouds. For example, when a person is standing in a certain position, their eyes can only see one surface of a 3D model (point cloud). Therefore, all the points on the opposite side of the point cloud can be ignored, since the human eye cannot see through objects. If we can find a way to calculate these points, we can reduce the number of points in the point cloud and make it easier to render. Additionally, the shape of the point cloud will be clearer, as shown in the figure 1.1 and 1.2. By eliminating the points on the back side of the model, the shape of the object becomes more intuitive.

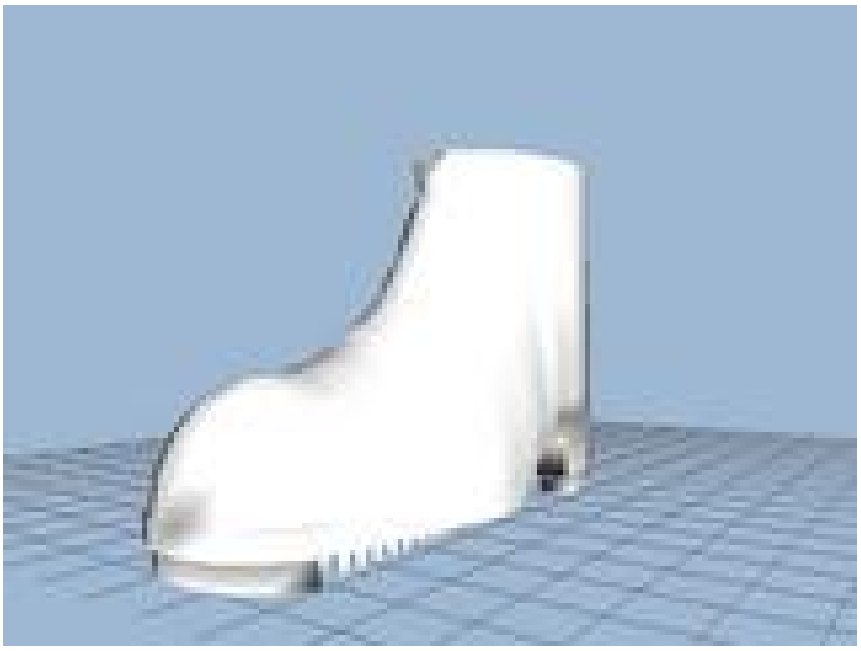


Figure 1.1: Shoe Figure

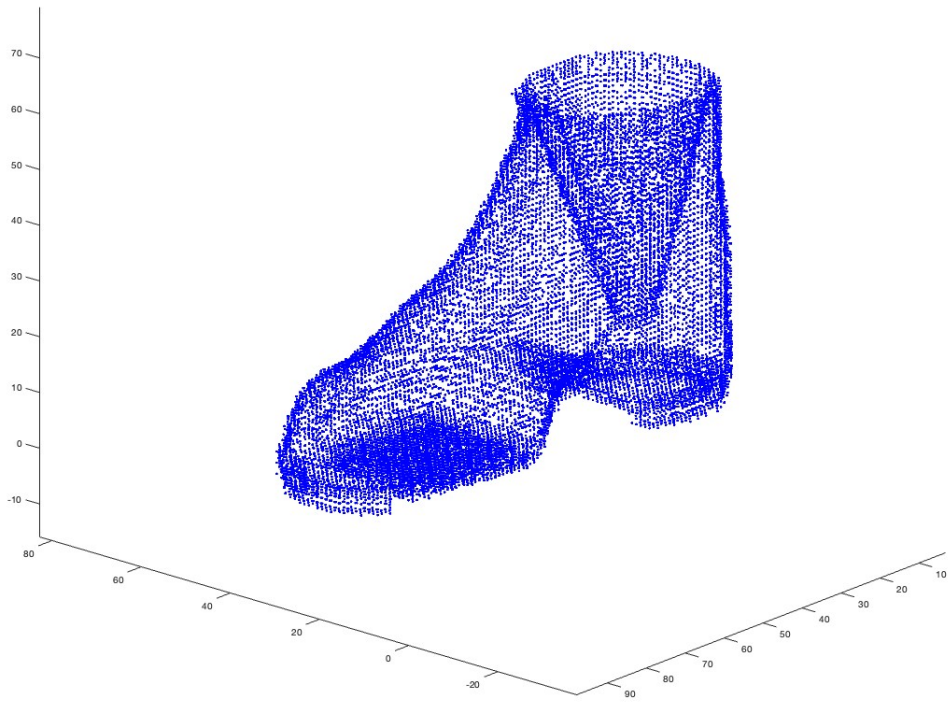


Figure 1.2: Point Cloud Overview

Chapter 2

Requirements

The goal of this project is to reduce the number of points in a point cloud.

1. Matlab with version R2022b (9.13.0.2049777) August 24, 2022.
2. Open source code at GitHub:
   https://github.com/LIN251/CompressionOfPointClouds.
3. The point cloud database is Princeton Shape Benchmark [1].
4. Requires Matlab Computer Vision Toolbox
   (To run this project, you will need to install the Computer Vision Toolbox. Once you run the project, a link will pop up in the Matlab command window asking you to click and install the toolbox.).

Chapter 3

Design

3.1 Implementation

3.1.1 Point cloud

This project uses the dataset reading and generation code from Dr. Ghandeharizadeh's paper [4]. The compression algorithm is built on top of this point cloud generator.

3.1.2 Axis aligned bounding box boundary(AABB Boundry)

In this project, I use the axis-aligned bounding box ray tracing algorithm (AABB ray tracing) to compress the point cloud. This involves creating an ideal box that contains the entire point cloud. As shown in Figure 3.1, 3.2. Calculating and judging the intersection of rays with a complex model can be time-consuming, so using a bounding box to surround the model allows us to reduce the time cost significantly. As shown in the Figure 3.3, Calculate the boundary points of a point cloud using the coordinates of all the points. I only need to keep two points which are located at the lower left and upper right corners of the ideal box to represent its size.
Finding the edge vertex in the ideal box:

1. minX Y Z.
2. X minY Z.
3. X Y minZ.
4. maxX Y Z.
5. X maxY Z.
6. X Y maxZ.

Min(lower left): [minX, minY, minZ] or [minX-1, minY-1, minZ-1]
Max(upper right): [maxX, maxY, maxZ] or [maxX+1, maxY+1, maxZ+1]

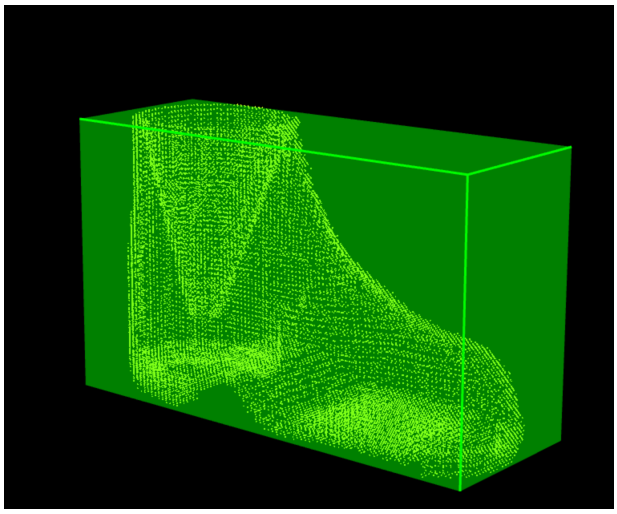


Figure 3.1: Ideal box for a shoe point cloud

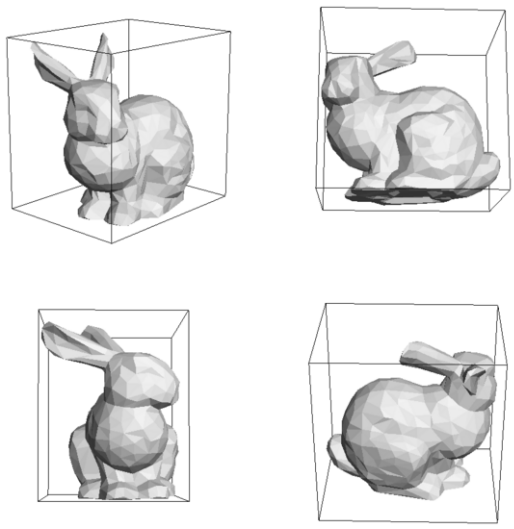


Figure 3.2: ideal box
An example of how an ideal box surrounds a complex model. The rabbit will perform as a point cloud and the cube is the ideal box.
Image Reference: https://mathimages.swarthmore.edu/index.php/Bounding_Volumes

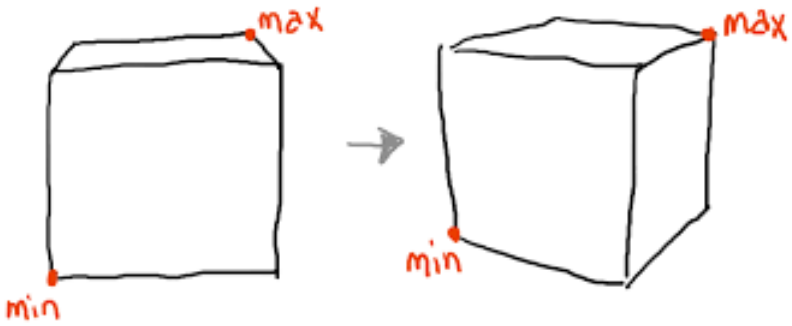


Figure 3.3: Boundary points
In Figure 3.3, a cube is shown, and its boundaries are represented by two points, “min” and “max”. These points are located at the opposite corners of the cube and are used to represent its boundaries..
Image Reference: http://raytracerchallenge.com/bonus/bounding-boxes.html

3.1.3 Point cloud cutting

In the first version of the compression algorithm, ray tracing was used to monitor each point in the point cloud, which resulted in a slow rendering speed due to a large number of points. For example, a point cloud in Figure 1.2 contains 16231 points would require nearly 16000 rays to be sent from the eye position.

In the second version, I used vector quantization for data compression, which divides a set of points into groups and represents each group by its centroid, as in k-means and some other clustering algorithms. The point cloud is first divided into small blocks based on the point cloud’s boundary. The cutting is performed according to the parameters in the “cells” array, which is a three-element vector in the form [numBlocksX, numBlocksY, numBlocksZ]. These elements represent the number of blocks that will be split in each dimension. Figure 3.4 provides a 2D example.

While dividing the point cloud into blocks, we also need to record the boundaries of each block. Because each block is a cuboid, we can record its boundaries using two points located at opposite corners, as shown in Figure 3.3. These two points are represented by [x1 y1 z1] in the lower left corner and [x2 y2 z2] in the upper right corner, so the boundary of each block is stored as [x1 y1 z1; x2 y2 z2].

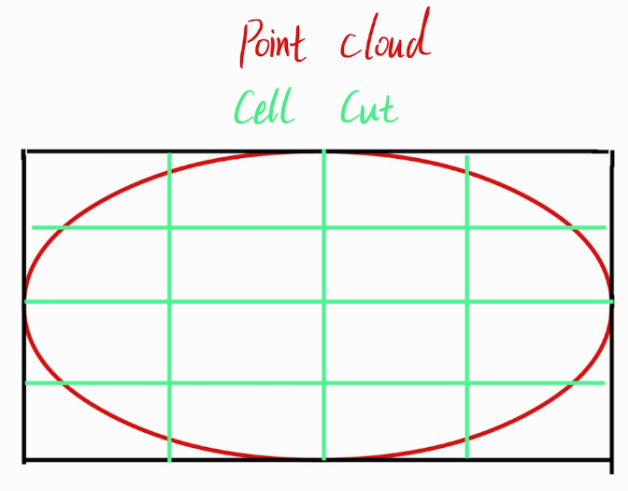


Figure 3.4: 2D example for point cloud blocks
In this figure, the black rectangle represents a 2D ideal box, while the red oval is a point cloud. The point cloud has been divided into small green rectangles by cutting it along the x-axis and y-axis into 4 sections each.

3.1.4 Point cloud classification

To determine which block a point belongs to in a point cloud, we first need to find the boundary of each block. Once we have this information, we can calculate which block each point belongs to by checking the x, y, and z coordinates of the point. If the coordinates of a point fall within the boundary of a particular block, then that point belongs to that block. This process only needs to be performed once, as the boundary of each block does not change.

3.1.5 Block represents points

In order to increase the accuracy of our calculation, I want to use multiple points to represent each small block, rather than using one point in the vector quantization. To do this, I first use the k-mean algorithm to calculate the center point of the block and then add six other boundary points to represent the block. This gives us a total of seven points that represent each small block, which helps to increase the accuracy of our calculations. All of these points are shown below.:

1. CenterX CenterY CenterZ.
2. minX Y Z.
3. X minY Z.
4. X Y minZ.
5. maxX Y Z.
6. X maxY Z.
7. X Y maxZ.

Figure 3.5 shows a 2D example of a center point inside a block. If a block contains less than seven points but more than zero points, then I use all of the points in that block to represent it. If a block contains zero points, then ignore it. Once I have calculated the representative points for each block, I can move on to the next step in the ray-tracing process.

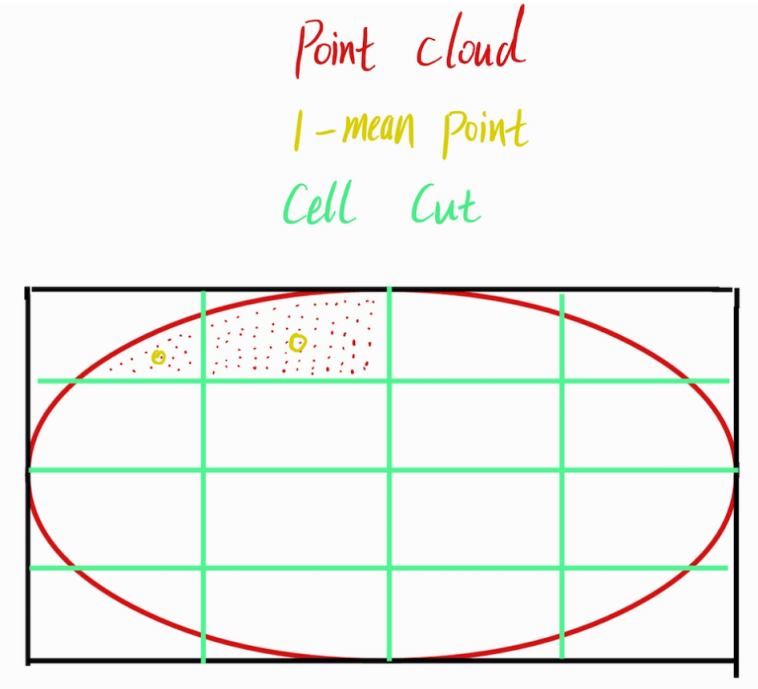


Figure 3.5: 2D example for block center point

In Figure 3.5, the black rectangle represents a two-dimensional ideal box, and the red oval represents a point cloud. The x-axis and y-axis have been divided into four equal parts, resulting in a grid of small green rectangles, each representing a block. The yellow circle represents the center point of each block, as calculated using the k-mean algorithm.

3.1.6 Unit vector calculation

In the previous step, we calculated the representative points and the boundaries of all the small blocks. To begin the ray tracing process, we need to calculate the direction of each ray. We first need to determine the position of the "eyes" from which the rays will originate. In this project, I will use the position of the cursor in the figure as the position of the eyes, so that when the cursor is moved, the position of the corresponding eyes will change as well. Once I have the position of the eyes, I can calculate the unit vector from the eyes to each representative point, ensuring that the direction is correct (from the eyes to the representative point). This will allow us to trace the rays from the eyes to the points in the point cloud.
Representative point:$[R_x, R_y, R_z]$;
Eye:$[E_x, E_y, E_z]$;
Ordered triple: $[R_x\text{-}E_x, R_y\text{-}E_y, R_z\text{-}E_z] = [O_x, O_y, O_z]$;
Magnitude: $\sqrt{O_x^2 + O_y^2 + O_z^2} = \mathrm{M}$;
Unit vector: $[O_x/\mathrm{M}, O_y/\mathrm{M}, O_z/\mathrm{M}] = [U_x, U_y, U_z]$;

3.1.7 Ray tracing

Now that I have the boundaries of the ideal box, the boundaries of each small block, and the unit vectors from the eye to each representative point, I can start to calculate the intersections of the rays with the point cloud. It's important to note that except for the unit vectors, all other values will be reused in the next iteration to improve performance.
For each unit vector, I need to calculate the first intersection point of the ray with the ideal box. This is the point "t" where the ray first touches the ideal box. We need to consider three possible cases, as shown in Figure 3.6:

R1: The ray does not intersect with the ideal box, we need to return.
R2: The ray intersects with the ideal box, but the intersecting ray does not hit any effective small blocks (i.e., there are no points in the block that the ray will hit).
R3: The ray hits the ideal box, and the path of the ray does hit a block that contains points.

All three cases are the end conditions for each ray when running the tracing algorithm.

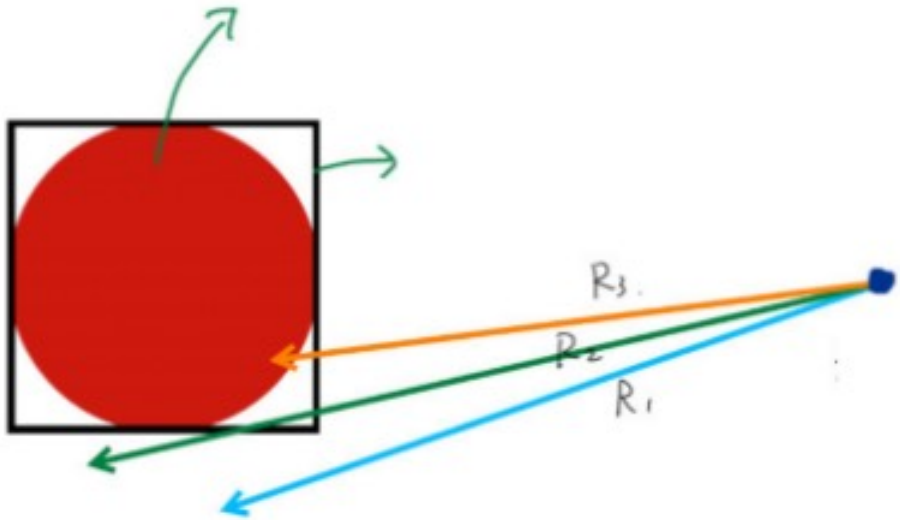


Figure 3.6: Hit conditions
This is a two-dimensional example. The blue dot represents the position of the eye. The three lines labeled R1, R2, and R3 represent the three possible cases when a ray interacts with the ideal box. The square represents a two-dimensional ideal box, and the red circle represents a two-dimensional point cloud in the shape of a circle. This figure shows how the different cases can occur when a ray is traced through the point cloud.

Next, in the AABB ray tracing algorithm, I need to calculate the intersection of the ray equation and the plane equation to find a "t" value. This "t" value represents the point at which the ray first hits the box. To do this, I need to know that the light equation is R(t) = O + t * Dir, where "O" represents the position of the eye and "Dir" represents the direction of the ray. In this project, "Dir" is the unit vector. After transforming the equations, I get t = (R(t) - O) / Dir. Then, the plane equation is "aX + bY + cZ + d = 0". Since the six faces of the ideal box are parallel to the XY, XZ, and YZ planes, so the equation I can use for the parallel plane is :
x_1=d_1,
x_2=d_2,
y_1=d_3,
y_2=d_4,
z_1=d_5,
z_2=d_6.
When the ray intersects two faces perpendicular to the x-axis,
t_x = (d - O_x) / Dir.x
When the ray intersects two faces perpendicular to the y-axis,

t_y = (d - O_y) / Dir.y
When the ray intersects two faces perpendicular to the z-axis,
t_z = (d - O_z) / Dir.z
Put all equations together then we can easily calculate the corresponding t_{min} and t_{max} with [t_x, t_y, t_z]. Shown in the Figure 3.7

First hit tmin: $t_{min} = (t_{x0}, t_{y0}, t_{z0})$;
Second hit tmax: $t_{max} = (t_{x1}, t_{y1}, t_{z1})$;

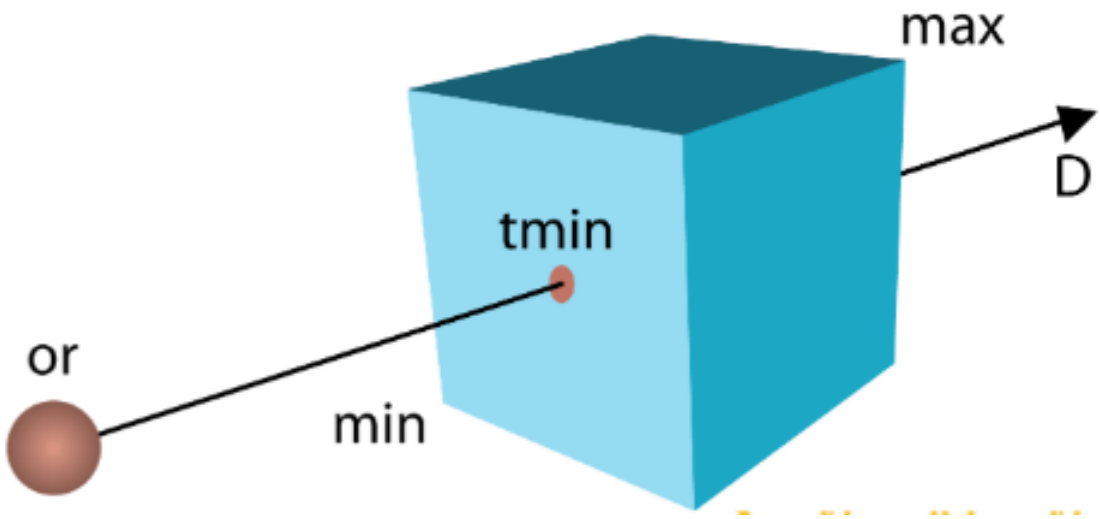
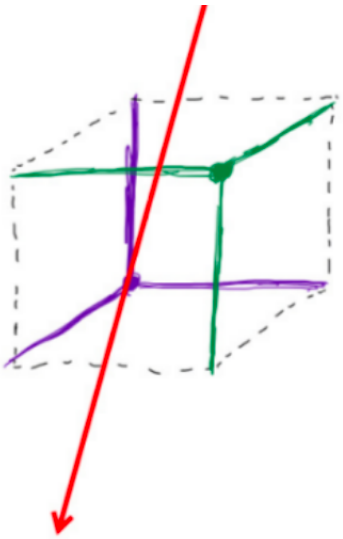


Figure 3.7: Ray tracing
In this Figure, the "or" represents the position of the eye, and the arrow labeled "D" represents a ray with direction "D". The blue cube represents the ideal box that contains the entire point cloud. "tmin" (also labeled as "min") is the first hit when the ray intersects the ideal box, and "tmax" (also labeled as "max") is the second hit when the ray exits the ideal box. These values are used to calculate the intersections of the ray with the faces of the ideal box.
Image Reference: https://ww2.mathworks.cn/matlabcentral/fileexchange/49671-hardware-accelerated-ray-box-intersection

Next, I need to calculate whether the light ray intersects with the ideal box. This paper [2] explains a method for calculating whether a light ray intersects with an ideal box. In short, when the light ray misses the ideal box, it will look like the scenario shown in Figure 3.8. This figure shows how the ray can miss the ideal box entirely, without intersecting it at any point. In this case, I can simply return and move on to the next ray.
The Figure 3.8 shown:
In X-axis direction $t_{min} = t_{x0}$, $t_{max} = t_{x1}$

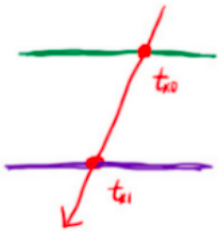
In Y-axis direction $t_{min} = t_{y0}$, $t_{max} = t_{x1}$
$t_{min} > t_{max}$, so no intersection

When the light ray hits the ideal box, it will look like the scenario shown in Figure 3.9.
In X-axis direction $t_{min} = t_{x0}$, $t_{max} = t_{x1}$
In Y-axis direction $t_{min} = t_{x0}$, $t_{max} = t_{x1}$
In Z-axis direction $t_{min} = t_{x0}$, $t_{max} = t_{x1}$



light miss

X axis direction



$$t_{min} = t_{x0}$$
$$t_{max} = t_{x1}$$

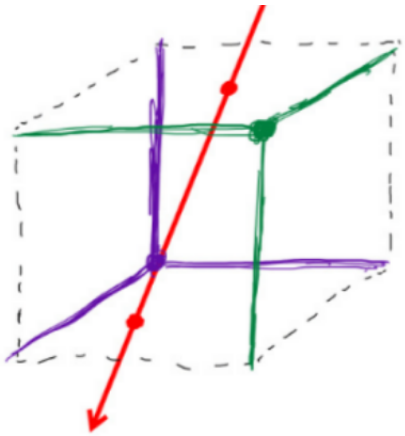
Y axis direction

$$t_{min} = \max(t_{x0}, t_{y0})$$
$$t_{max} = \min(t_{x1}, t_{y1})$$

t_min > t_max, so no intersection

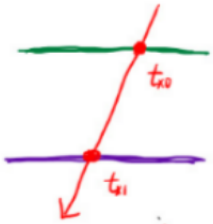
Z axis direction

Figure 3.8: Light miss



light hits

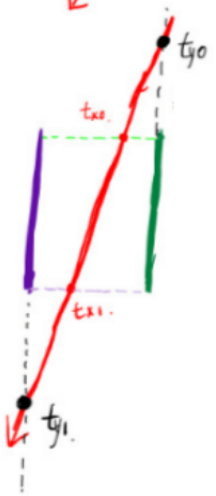
X axis direction



$t_{min} = t_{x0}$

$t_{max} = t_{x1}$

Y axis direction



$t_{min} = \max(t_{x0}, t_{y0})$

$t_{max} = \min(t_{x1}, t_{y1})$

Z axis direction

$t_{min} = t_{z0}$

$t_{max} = t_{z1}$

Figure 3.9: Light hits

3.1.8 Eye tracking

After we have calculated the intersections of a single ray with the ideal box, we need to extend the ray in the correct direction to find any additional intersections. If the ray hits a block that contains points, we will keep track of that block.
In order to improve the accuracy of the results, we can allow the ray to hit multiple blocks before returning. We can then add all of these blocks together to get a complete set of blocks that can be seen from a given position. This will allow us to accurately render the point cloud from a specific perspective.

Camera view:
To achieve the desired camera view, there are two tasks that need to be completed.
Task 1: Set the Matlab camera position to the Eye position (Cursor position).
Task 2: Set the Matlab camera direction to the center of the point cloud.

The vector formed by the camera position and direction is the direction of the optical axis. This vector can be used to control the camera coordinates and view.
The eye (cursor) position can be obtained from the Matlab callback function. The direction can be calculated by using the center of the point cloud (found with k-means clustering) and the eye position. This direction will be a unit vector.

3.2 Mode

This project provides two visualization modes:

Mode 1: In this mode, the point cloud will be compressed according to the position of the cursor. The position and viewing angle that focuses on the point cloud is fixed.

Mode 2: In this mode, the user can click any position in the Matlab figure, and the program will automatically modify the camera's observation position and angle to create the effect of looking at the compressed point cloud.

For comparison purposes, "Mode 2" uses the "pcshowpair()" function from the Computer Vision Toolbox to plot two point clouds together. One for the original point cloud with **Yellow points** and one for the compressed point cloud with **Blue points**. Shown in Figure 3.10

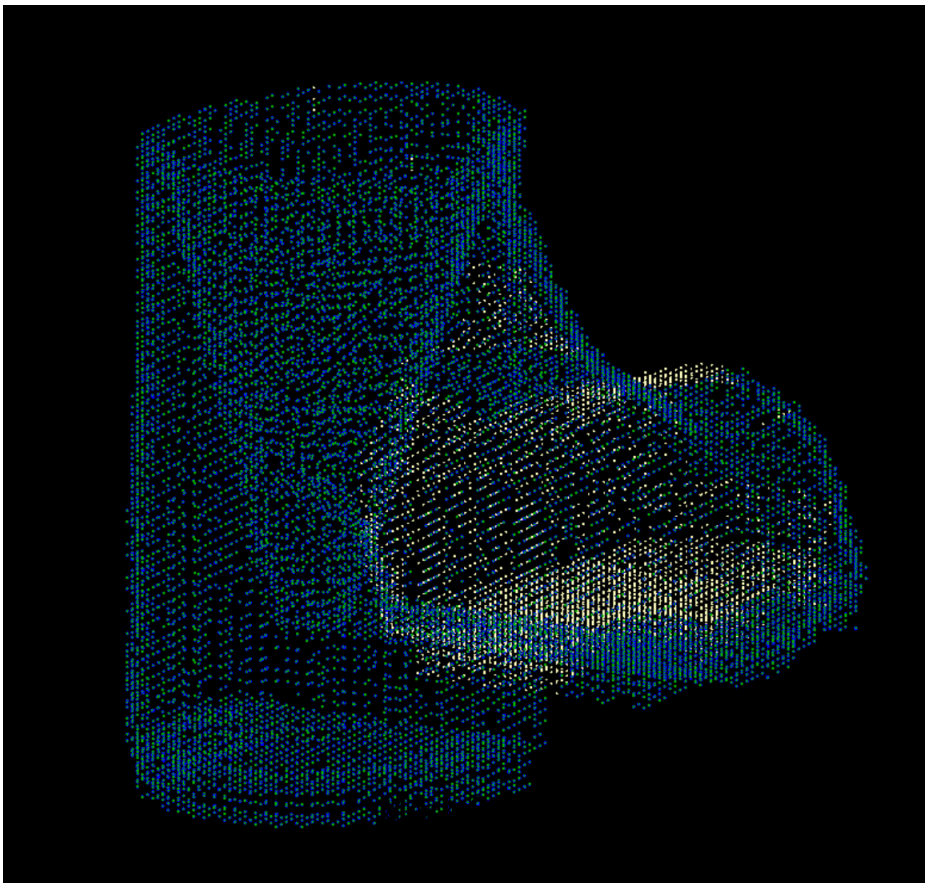


Figure 3.10: Mode2 example
The original point cloud is plotted using yellow points and the new compressed point cloud is plotted using blue points. The yellow points are located on the opposite side, indicating that they are the points that were eliminated during the compression process.
Note: To show all the yellow points, I rotate the point cloud, so the camera position and direction are manually changed in this figure.

Chapter 4

Benchmark/Improvement

4.1 Improvement

The initial version of the ray tracing algorithm used in this project calculated a ray for each point in the point cloud and then determined whether it would hit a different point before hitting the original one. While this algorithm was very time-consuming due to the large number of points in the point cloud and could not be run in real-time.

The second algorithm uses vector quantization classification to improve the efficiency of ray tracing. As previously mentioned, this algorithm significantly reduced the time required for the ray tracing. By classifying the point cloud, we were able to significantly reduce the number of rays needed for the algorithm, and then we redrew all the points in the figure. However, this approach is still takes more than 10 seconds to execute.

The third optimization is to display and hide the point cloud based on blocks. Initially, we plot all the points in the figure according to their corresponding blocks. For each subsequent run, we only need to calculate the blocks that need to be displayed or hidden, and then perform the display or hide attributes on the entire block instead of the individual points. For the point cloud of Princeton/db/17/m1740/m1740.off, which is shown in Figure 1.2, a new point cloud can be rendered in real-time.

Test data:
Dataset: Princeton/db/17/m1740/m1740.off;
Cells: [6 6 6];
Hits: 3;
For more information about these parameters, please refer to the developer menu section.

Chapter 5

Lessons learned

Through this project, I gained a deeper understanding of many graphics and point cloud compression algorithms. Learning about these algorithms gave me a better appreciation of the future of AR VR and the Holodesk system. In the study of FLSs, I also learned about a new display interaction mode that can provide a more accurate display and interact with humans (FLSs matter). I also had the opportunity to work with the ray tracing algorithm, and I gained a deep understanding of some of these algorithms, particularly the AABB ray tracing algorithm, and implemented it. Additionally, I gained more knowledge about using Matlab, setting up a camera, and working with the Matlab figures.

Chapter 6

Developer's Manual

6.1 System Compatibility

This project is compatible with both Windows and macOS operating systems.

6.2 Installation and Setup

1. Install Matlab with version R2022b (9.13.0.2049777) August 24, 2022 or other compatible versions.
2. Go to GitHub and clone the repository by following the link below
   - https://github.com/LIN251/CompressionOfPointClouds
   - In the git repository, you will find a directory called "pointCloudCompression" that contains all the source code for this project.
   - In the git repository, you will find a directory called "Princeton" that contains all the Princeton datasets used in this project..
3. Run compression algorithm
   - cd pointCloudCompression
   - pointCloudCompression("~/Princeton/db/17/m1740/m1740.off", hits, cells, mode)
     (a) hits: Integer.
     The number of blocks that a single ray can hit before being terminated. This value can be adjusted to improve the accuracy of the displayed results by allowing the ray to hit multiple blocks before returning.
     (b) cells: Array.
     The number of bins, specified as a three-element vector of the form [numBinsX,numBinsY, numBinsZ]. The vector elements indicate the number of bins to use in each dimension, respectively.
     (c) mode: String ~ ["mode1", "mode2"]

Bibliography

[1] The princeton shape benchmark. In *Proceedings of the Shape Modeling International 2004*, SMI '04, page 167–178, USA, 2004. IEEE Computer Society. ISBN 0769520758.

[2] Georgios Chatzianastasiou and George A. Constantinides. An efficient fpga-based axis-aligned box tool for embedded computer graphics. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 343–3437, 2018. doi: 10.1109/FPL.2018.00065.

[3] Shahram Ghandeharizadeh. Holodeck: Immersive 3d displays using swarms of flying light specks. In *ACM Multimedia Asia*, pages 1–7. 2021.

[4] Shahram Ghandeharizadeh. Display of 3d illuminations using flying light specks. *arXiv preprint arXiv:2207.08346*, 2022.

[5] Adrien Maglo, Guillaume Lavoué, Florent Dupont, and Céline Hudelot. 3d mesh compression: Survey, comparisons, and emerging trends. *ACM Comput. Surv.*, 47(3), feb 2015. ISSN 0360-0300. doi: 10.1145/2693443. URL https://doi.org/10.1145/2693443.