Compression of Point Clouds

Authors

Lin Zhang

CSCI 599: Special Topics
Instructor: Dr. Shahram Ghandeharizadeh

December 5, 2022
University of Southern California, Los Angeles, California 90007

Compression of Point Clouds

(ABSTRACT)

Nowadays, the point cloud is widely used in 3D visualization, VR, AR, and HoloDesk systems. If we render a 3D model from a point cloud perspective. A large number of vertices will be used to form a 3D model. Since the number of vertices is large, we need to find a way to minimize the number of vertices used. Then, the compressed result can be easily transferred and re-rendered. This is especially important in the Flying Light Specks system [3]. It will significantly reduce the number of active (turning on) FLSs, save a lot of energy, and even provided a better outline of the 3D model. The compression algorithm provided in this report will mainly focus on the human view perspective. Specifically, it will achieve the compression result from a certain direction for every single run.
To sum up, the main purpose of this project is to implement an algorithm that can reduce the number of points being used in a 3D point cloud and ensures a clear 3D display. Furthermore, as an extension, this solution also supports a simple version of eye-tracking so that the compression results can be easily visualized.

Contents

List of Figures

Chapter 1

Introduction

Before implementing the compression algorithm and eye tracking, it is essential to ask why the point cloud compression matter. Nowadays, with the continuous expansion topic of the network and metaverse. VR, AR, and HoloDesk gradually become a technology that is close to people's life. We all have seen some fancy display devices in the movies and shows like Star Trek. Those devices can show 3D models and even have some interactions with humans. All these movies and shows are pointing us to a new way of human interaction.

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 AR field.

To better achieve a compressed result with a point-view perspective, there are several potential scenarios we need to discuss first. One is about using some 3d projectors. A user can easily view the models in 3D mode by using several projectors, but the downside is that it is hard to have some human interaction with the model. All objects are virtual. Then we have seen some AR and HoloDesk technologies. In the current AR and Holodesk systems, There are a lot of requirements for the environment and the interactive objects in order to produce some interactions.

For our project, we are mainly focused on a certain point to view to achieve a compressed result. For example, when a person stands in a particular position, his eyes can view a certain surface of a 3D model (point cloud). Correspondingly, all points on the opposite side of this point cloud can be ignored. This is easy to understand since the human eye cannot look through a real object. It can be said that all the points on the back of this point cloud are meaningless and can be eliminated. If we find a way to calculate these points, then we can reduce the number of points in the point cloud and achieve faster-rendering speed. On the other hand, the shape of the point cloud will be clearer than the original point cloud, as shown in the Figure 1.1 and 1.2 We can see that the shape of this shoe is not very intuitive,

and many points on the back side of the model are also shown in front of the point cloud through the gap between the two points. So if we eliminate the points on the back side, the shape of the object will be clearer.

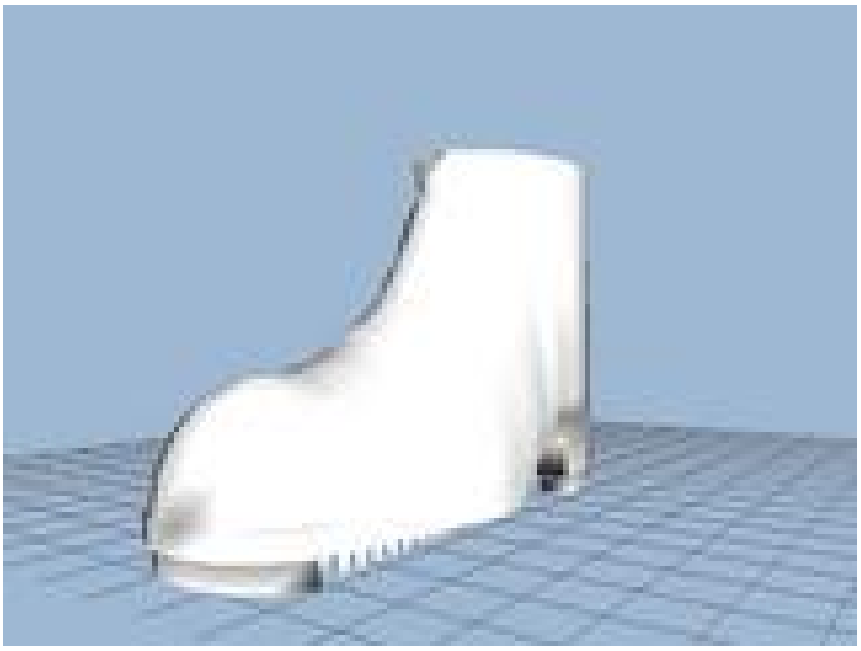


Figure 1.1: Shoe Figure

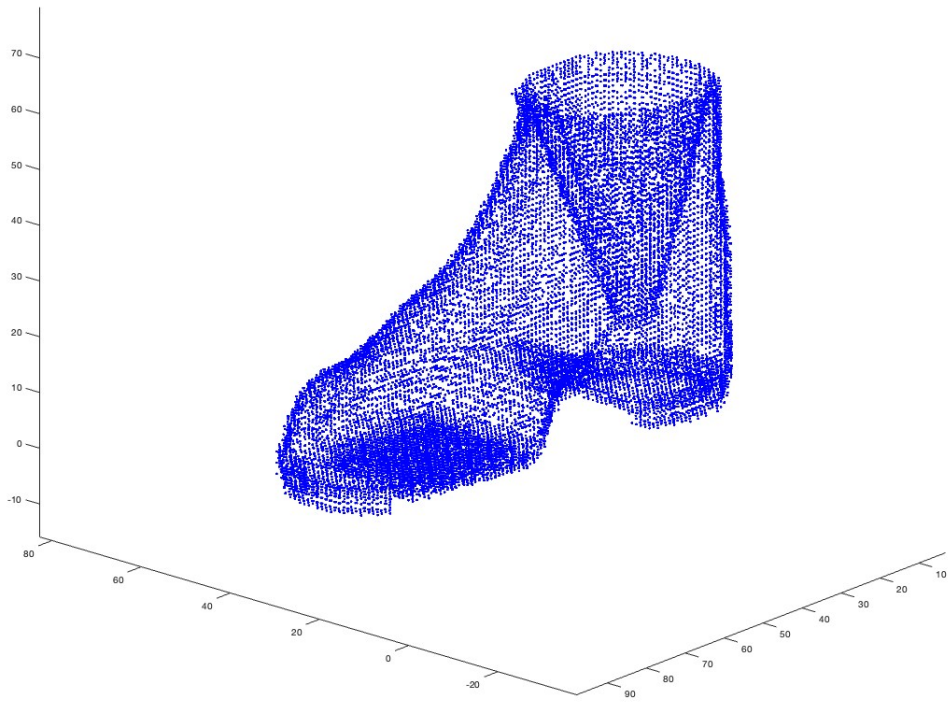


Figure 1.2: Point Cloud Overview

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].

Chapter 3

Design

3.1 Mode

This project provides two visualization modes. The first is that the point cloud will be compressed according to the position of the mouse. The position and viewing angle that focuses on the point cloud is fixed. The second mode is by clicking, the user can click any position in the Matlab figure, and then the program will modify the observation position and angle of the camera automatically, so as to achieve the effect that people are watching the compressed point cloud.

3.2 Implementation

3.2.1 Point cloud

For dataset reading and generating, it uses the source code from Dr. Ghandeharizadeh's paper [4]. The compression algorithm is based on this point cloud generator.

3.2.2 axis aligned bounding box boundary(AABB Boundry)

Here we use the axis-aligned bounding box ray tracing algorithm (AABB ray tracing). First, we need to create an ideal box Figure 3.1, which will contain the entire point cloud. It is time-consuming of calculating and judge the intersection of rays with a complex model, so if we can surround the complex model with a bounding box, then calculating the intersection of the ray and the ideal box will reduce the time cost significantly. As in the example in Figure 3.2, we can calculate two boundary points through the coordinates of all points in the point cloud. We only need to keep the points in the lower left and upper right corners of the ideal box to represent its size.

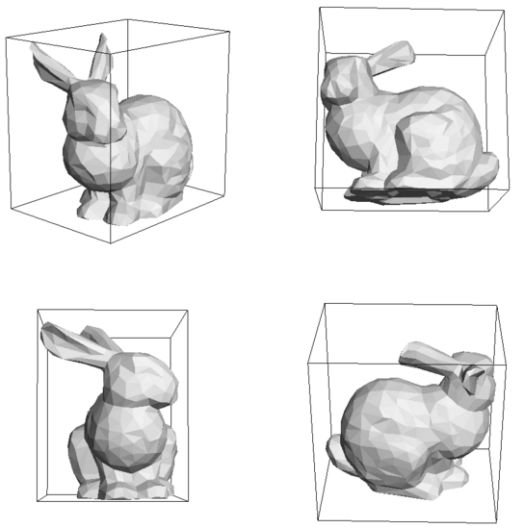


Figure 3.1: ideal box

3.2.3 Point cloud cutting

In the first version of the compression algorithm, I used ray tracing to monitor every single point in the point cloud. Since there are so many points in a point cloud, the rendering speed of a new point cloud is very slow. For example, the point cloud in Figure 1.2 contains 16000 points. If the ray tracing is performed directly on each point in the point cloud, then there are nearly 16000 rays sent from the eye position onto the point cloud. The time cost of rendering a new point cloud for a certain eye position in Matlab is about 3 to 10 minutes. Then we adopted a new compression method (version two), which is using Vector quantization. To summarize, vector quantization (VQ) was used for data compression. It works by dividing a large set of points (vectors) into groups. Each group is represented by its centroid point, as in k-means and some other clustering algorithms. Here we first cut the point cloud into small blocks (cubes). The cut will be performed based on the boundary of the point cloud. From the previous step, we already know the boundary of the ideal box, we will use some equilateral blocks, and we can cut according to demand in the parameters "cells" (will explain in the developer's manual section). While doing this, we also need to record the boundaries of all small blocks, because each block is a cube, so we can record the boundaries of the block through two points as how the ideal box works. These two points are located in the diagonally opposite corner of the block as Figure 3.2 shown. Here we will use the point [x1 y1 z1] in the lower left corner of the block and the point [x2 y2 z2] in the upper right corner, so the boundary of each block is stored as [x1 y1 z1;x2 y2 z2].

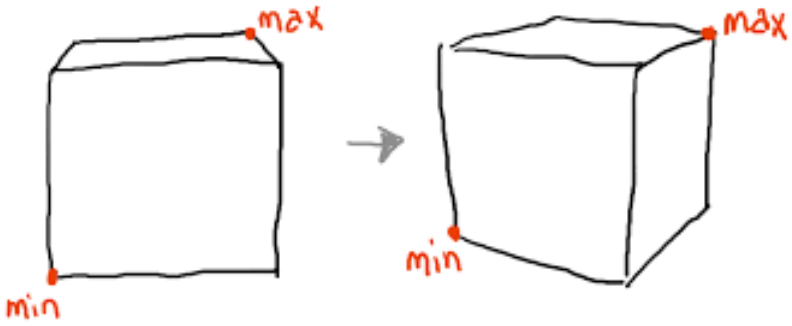


Figure 3.2: Boundary points

3.2.4 Point cloud classification

When we have the boundary of each block, then we are able to calculate which block each point belongs to. As long as the x y z of a single point is within the block boundary, then this point belongs to the current block. We need to do this for all points in the point cloud. This action only runs once.

3.2.5 Block represents points

We want to calculate the representative point of each small block. Unlike vector quantization mentioned before, in order to increase the accuracy, multiple points are used here to represent a block. First, we use the k-mean algorithm (1-mean) to calculate the center point of the block and add 6 other boundary points to represent the current block. All 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.

These points represent the corresponding maximum and minimum values of x, y, and z in the current block. If the number of points contained in this block is less than 7 and greater than 0, that is, less than the total number of representative points, then we use all the points to represent this block. If points in the block are equal to 0, then we ignore this block. When we have all these representative points, we can proceed to the next step of the ray-tracing process.

3.2.6 Unit vector calculation

In the previous step, we have all the representative points and the boundaries of all the blocks. For ray tracing, the first thing we need to do is to calculate the direction of each ray. We first need to calculate the position of the eyes. In this project, we will obtain the position of the mouse in the figure as the position of the eyes, so that when the mouse moves in the figure, the position of the corresponding eyes will change. When we have the position of the eye, we need to calculate the unit vector from the eye to each representative point, here we need to calculate the correct direction, that is, from the eye to the representative point.
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: $[O_x/\mathrm{M}, O_y/\mathrm{M}, O_z/\mathrm{M}] = [U_x, U_y, U_z]$;

3.2.7 Ray tracing

Now that we have the boundaries of the ideal box, the boundaries of each smaller block, and the unit vectors from the eye to each representative point, we can start to calculate the ray intersections. One thing to be pointed out is that except for unit vectors, all other values will be reused in the next iteration to improve the run time performance.
First, for each unit vector, we need to calculate the first intersection point of the ray with the ideal box, that is, the point "t" where the ray first touches the ideal box. Here we need to consider three cases in Figure 3.3: R1. If the ray does not intersect with the ideal box, then we need to return. R2. The ray intersects with the ideal box, but the intersecting ray does not hit any effective small blocks, in other words, there are no points in the block that the ray will hit. Noted, the ideal box is larger than the 3D model, so there are some blocks with zero points. R3. The ray hit the ideal box, and the path of the ray does hit a block that contains points. All three cases are the end conditions for every single ray when running the tracing algorithm.

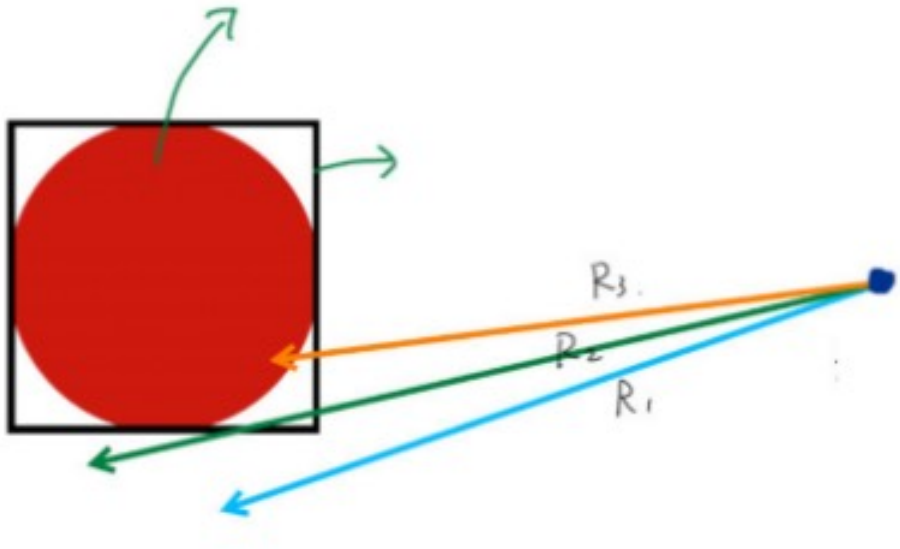


Figure 3.3: Hit conditions

Next, for the AABB ray tracing algorithm, it is actually equivalent to calculating the intersection of the ray equation and the cube equation, combining the two equations to find a "t" value. This "t" value represents the point at which the ray first hits the box. First, we need to know that the light equation is R(t)=O+t∗Dir. Among the equation, "O" represents the position of the eye, and "Dir" represents the direction of the ray. In this project, "Dir" means the unit vector. After transforming the equations, we will 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 can use the parallel plane:
x_1=d_1, x_2=d_2,
y_1=d_3, y_2=d_4,
z_1=d_5, z_2=d_6.
and reduce the equations to the following:
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} with [t_x, t_y, t_z] in Figure 3.4
First hit tmin: t_{min} = (t_{x0}, t_{y0}, t_{z0}) ;
Second hit tmax: t_{max} = (t_{x1}, t_{y1}, t_{z1}) ;

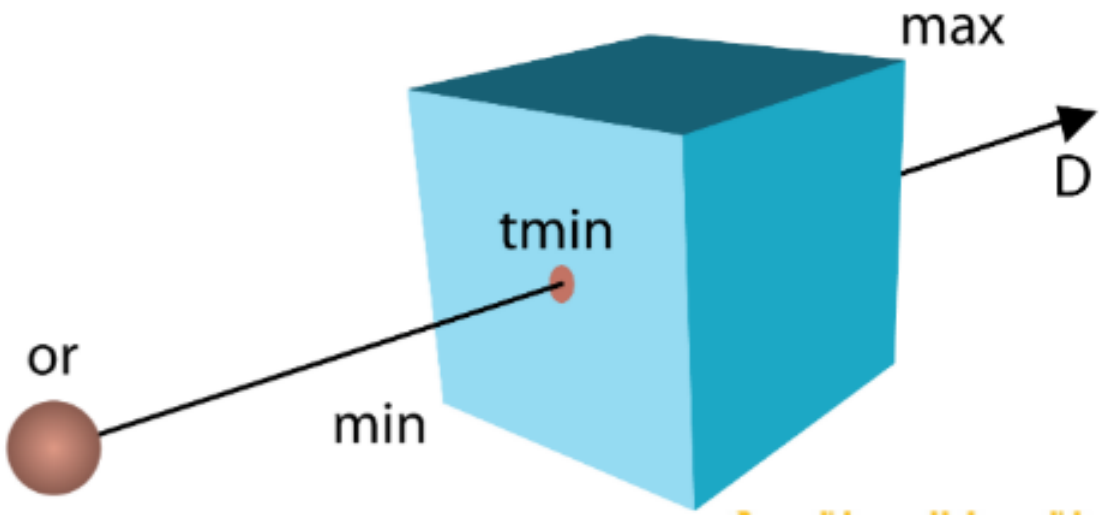
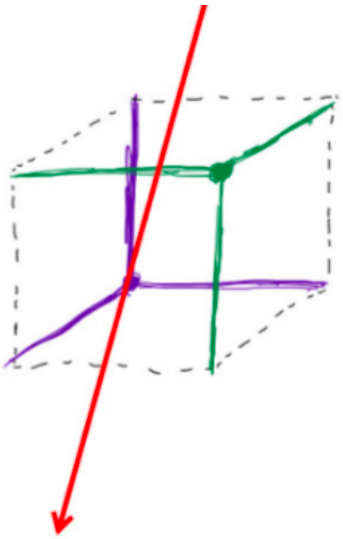


Figure 3.4: Ray tracing

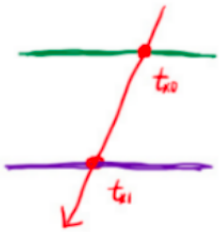
Following, we need to calculate whether the light intersects with the ideal box. This paper [2] explains the method of calculating whether it intersects with the cube. In short,
when the light misses the cube, Figure 3.5
X-axis direction $t_{min} = t_{x0}$, $t_{max} = t_{x1}$
Y-axis direction $t_{min} = t_{y0}$, $t_{max} = t_{x1}$
$t_{min} > t_{max}$, so no intersection

When light hits the cube Figure 3.6
X-axis direction $t_{min} = t_{x0}$, $t_{max} = t_{x1}$
Y-axis direction $t_{min} = t_{x0}$, $t_{max} = t_{x1}$
Z-axis direction $t_{min} = t_{x0}$, $t_{max} = t_{x1}$



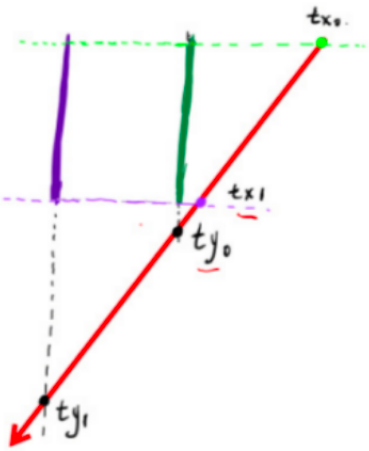
light miss

X axis direction



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

Y axis direction

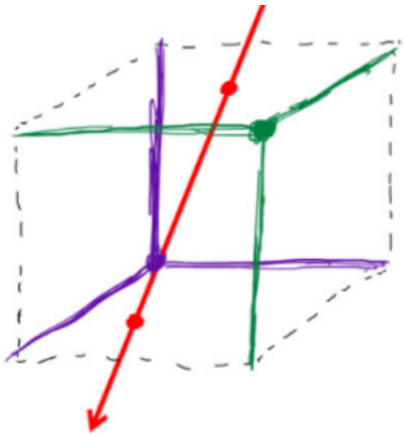


$$t_{min} = \max(t_{x0}, t_{y0})$$
$$t_{max} = \min(t_{x1}, t_{y1})$$

t_min > t_max, so no intersection

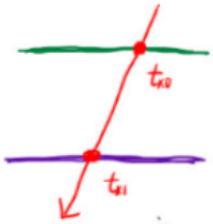
Z axis direction

Figure 3.5: Light miss



light hits

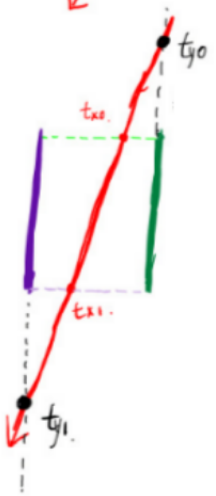
X axis direction



$t_{min} = t_{x0}$

$t_{max} = t_{x1}$

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 3.6: Light hits

3.2.8 Eye tracking

After we calculate the intersection between a single ray and the ideal box, then we need to extend the ray in the correct direction. If the ray hits a certain block that contains points, the block will be retained. In order to improve the accuracy of the displayed results, we can allow this ray to hit multiple blocks before returning. It should be noted that the extension is based on the direction of the current ray direction. Following, we need to add all these blocks together. These are the blocks that can be seen at a certain position.

Chapter 4

Benchmark/Improvement

4.1 Improvement

The first version of the ray tracing algorithm used in this project is to calculate a ray for each point in the point cloud and then calculated whether it will hit a different point before hitting the original one. This algorithm works, but due to a large number of points in the point cloud, the entire algorithm is very time-consuming and cannot be run in real-time. This algorithm takes around $3 \sim 10$ minutes to regenerate the new point cloud from our test data which is the model in Figure 1.2.

Our second algorithm is to perform vector quantization classification. As we mentioned before, this algorithm improves the efficiency of ray tracing from $3 \sim 10$ mins to $10 \sim 20$ seconds for our test data. We classify the point cloud, and the number of rays needed for the algorithm will be markedly reduced. Then we will redraw all the points in the figure. However, this operating efficiency is still very slow, and the execution time is greater than 10 seconds.

Our third optimization is to display and hide the point cloud according to the block. Initially, we need to plot all the points in the figure according to the block during the first iteration. All points are associated with their corresponding blocks. For each subsequent run, we only need to calculate the blocks that need to be displayed and hidden, and then perform the display or hide attributes on the entire block instead of the points. This greatly improves efficiency. For the point cloud of Princeton/db/17/m1740/m1740.off, which is Figure 1.2, when the cells are less than 7, a new point cloud can be rendered in run time. When the cells are greater than 10, each rendering takes about 2 to 4 seconds.

Test data:
Dataset: Princeton/db/17/m1740/m1740.off;
Cells: 6;
Hits: 3;
These parameters will be explained in the developer menu section.

Chapter 5

Lessons learned

This project gave me a deeper understanding of many graphics and point cloud compression algorithms. Knowing about these algorithms made me more aware of the future of AR VR and the Holodesk system. For the study of FLSs, I also understand a new display interaction mode, which can not only provide a more accurate display but also interact with humans (FLSs matter). At the same time, I also came into contact with the algorithm of ray tracing, and deeply understood some algorithms, especially the AABB ray tracing algorithm, and implemented this algorithm. Furthermore, I got more knowledge about how to use Matlab, the camera setting and even worked on some performance improvements.

Chapter 6

Developer's Manual

6.1 System Compatibility

This project supports 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 can find a directory called "pointCloudCompression" which contains all the source codes of this project.
   - In the git repository, you can find a directory called "Princeton" which contains all the Princeton datasets.
3. Run compression algorithm
   - cd pointCloudCompression
   - pointCloudCompression("~/Princeton/db/17/m1740/m1740.off", hits, cells, mode)
     (a) hits: Integer.
     The number of blocks a single ray can hit before returning.
     (b) cells: Integer.
     The number of pieces a point cloud will be divided into. If cells = 6 then the point cloud will be divided into 6 pieces on the x-axis, y-axis, and z-axis.
     (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.