Visualizing Point Cloud Data in Virtual Reality with LOD Techniques

Rami Shehadeh
University of Maryland

Roger D. Eastman
University of Maryland

Daniel Lopez
University of Maryland

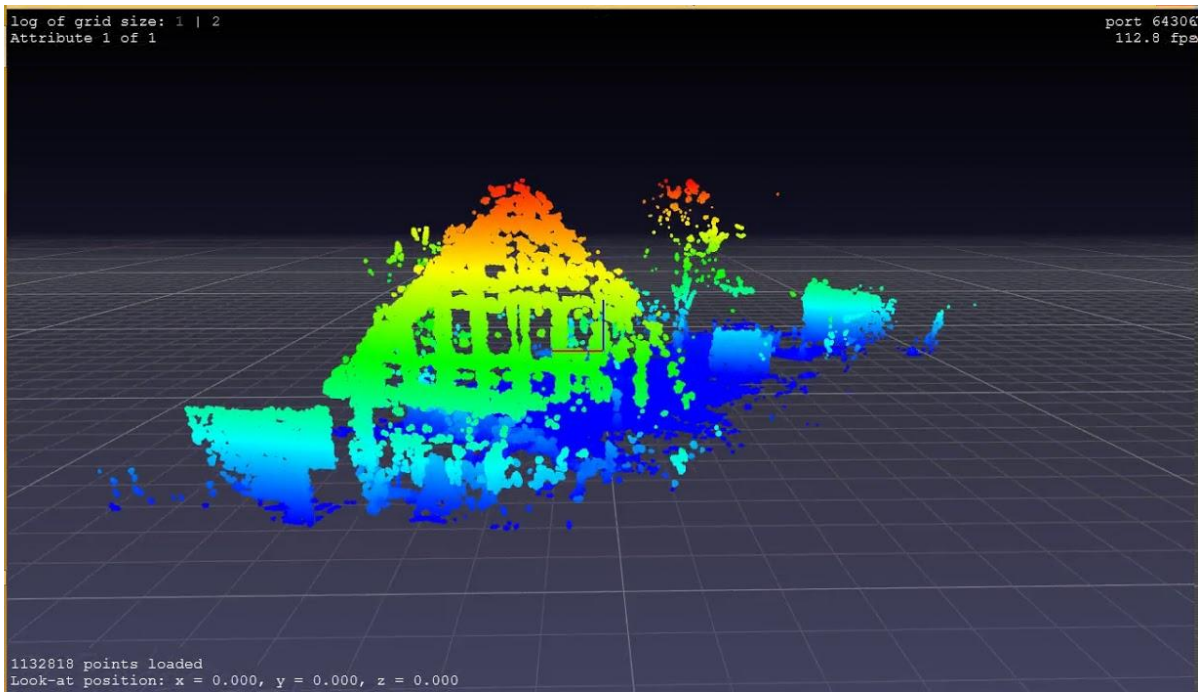

Figure 1: *PPTK Viewer, a Python library used to visualize and render point cloud data.*

Abstract

In this research project, we explore a method to visualize large weather data from the NASA Goddard Earth Observing System in virtual reality using Unity. Our goal is to render 300 million sequences of point cloud weather data by partitioning the data and streaming it as it becomes visible from the user's gaze. We achieve this by using Point Processing Toolkit (PPTK) Point Cloud Viewer, an external program, to calculate visible points and render them as a 2D texture and mask that is composited as a layer with textural and label data in Unity.

1 Introduction

The NASA Goddard Earth Observing System (GEOS) numeric model is a valuable tool for understanding and predicting weather patterns on Earth. However, with the massive amount of data generated by this model, it can be difficult to effectively visualize and interpret the data in a meaningful way. This is where virtual reality comes in as a promising solution.

By visualizing the weather data in a virtual reality environment, we can create a more immersive and interactive experience for users. This allows them to explore the data in real-time and gain a deeper understanding of the patterns

and trends that are present. However, rendering such large datasets in virtual reality is a challenging task, especially with Unity, a popular 3D game engine, that due to being single-threaded cannot load and render large amounts of data at once.

To address this challenge, we propose a method that involves partitioning the data and streaming it based on the user's gaze direction and positional data. This allows us to take advantage of other open-source software and plugins that are more specifically equipped to handle large amounts of point cloud data. Our goal is to develop proof-of-concept software that can be passed on to the NASA team for further development and use in the open-source Reality Engineering Toolkit (MRET), which is built with Unity.

This research project is being carried out under the NASA Goddard AIST grant “GEOS Visualization And Lagrangian Dynamics Immersive eXtended Reality Tool (VALIXR) for Scientific Discovery”. The project is supervised by Professors Matthias Zwicker and Roger Eastman, with Daniel Brown as the supervising graduate student, along with contributions from several graduate and undergraduate students including Amir Hajiabbasi, Rod Rahmjoo, and Arnan Huang. In this paper, we will present our methodology, results, and analysis of the performance of the framework, and discuss its implications for future research.

2 Background

Our research objective is to develop a method to render 300 million sequences of point cloud weather data in virtual reality using Unity. Since Unity is single-threaded, it cannot load and render such a massive amount of data all at once in the scene. Also, Unity doesn’t support any point cloud file that is too large.

The current method of displaying a point cloud natively within Unity is using the built-in point renderer for mesh vertices, which requires the user to initially create a mesh or a point cache from the point cloud data and then display the vertices using a visual effects graph (Erfani, 2022). Alternatively, the pcx plug-in developed by Keijiro provides support for the popular PLY point cloud format, however, it fails to handle massive point clouds effectively as the size rendered is limited by the computer’s RAM and GPU (Takahashi, 2017/2023). The shader used by pcx also is limited by GPU memory as the entire point cloud is loaded regardless of the current view of the camera.

Therefore, our research is focused on exploring other methodologies to visualize data as it becomes visible to the user's gaze. Our plan is to leverage an external point cloud renderer by passing in live VR transform data while sending the rendered display to Unity as a 2D mask. The following paragraphs will discuss the techniques of different point cloud software to efficiently process data which will inform our methodology.

2.1 Exploration of Point Cloud Processing Software

Within many 3D point cloud processors and visualizers, one of the common ways to efficiently render massive point cloud data is by partitioning the data into an octree data structure. This could allow for only the relevant portions of the data needing to be loaded into the RAM or GPU for rendering. We can now easily cull, or not display any data, that lies outside of the camera view frustum or that would be occluded by other points. Generating an octree is

also crucial to optimize the rendering of level-of-detail (LOD). The octree structure consists of nodes partitioned into areas in world space, where lower levels of detail are parents of nodes containing higher levels of detail. In real-time, we are able to quickly transition between lower and higher LOD levels based on user position in order to give the impression that the whole data set is visible when really only a portion of it is rendered. When targeting external software to test with our methodology, we will prioritize applications that use these rendering methods in order to handle massive point cloud data.

One of the most popular point cloud renderers that uses the previously mentioned methods is Potree (potree, 2012/2023; Schütz, 2016). In order to parse the massive data, the .LAS files, which typically hold LiDAR data, must go through a preprocessing phase handled by the Potree converter (*PotreeConverter*, n.d.). This tool generates an out-of-core octree by chunking (sorting points), indexing each chunk in parallel, and merging all local trees into one overarching octree. This gets output in a unique Potree format that can be handled by Potree, an open-source, web-based renderer (Schütz et al., 2020). There have been developments in a method to support the Potree format in Unity (Fraiss, 2017). However, the lengthy preprocessing step is required in order to handle .LAS files by using an external converter tool as well as not supporting the .PLY file format, which is another common file format for point clouds, leads us to seek alternative solutions to handling point clouds in Unity.

Another popular software for handling point cloud data is CloudCompare, which is an open-source 3D point cloud processing software and a versatile tool that provides a variety of features for processing, analyzing, and visualizing 3D point cloud data (*CloudCompare*, n.d.). It supports many common point cloud file formats, including .LAS and .PLY. The software is capable of creating an efficient octree structure for point cloud data, which is beneficial for managing and manipulating large datasets. However, despite these useful features, we encountered a key limitation when trying to integrate CloudCompare into our research methodology. Our primary aim was to render the point cloud and have it update in real-time according to the movements of the VR headset. Unfortunately, we found that CloudCompare could not accommodate this particular requirement. Although CloudCompare offers robust functionalities for static point cloud data processing and visualization, its current design doesn't support real-time updates efficiently. The real-time rendering and updating of point cloud data is a significant component of our research. After attempting to adapt CloudCompare to our requirements and realizing its limitations in this regard, we decided not to pursue further usage of CloudCompare in our project.

PPTK is a Python package for visualizing and processing 3D point clouds. Using a similar octree-based LOD renderer with frustum and occlusion culling, PPTK can effectively handle up to 100 million points in a scene (*Pptk - Point Processing Toolkit*, 2018/2023). PPTK also accepts most popular point cloud formats, as long as it can be represented as a 3-column numpy array, including .LAS and .PLY files. Importantly, we can easily install this package within our Python environment to simplify the development of our end-to-end proof of concept. With useful methods that allow us to change the camera position of the viewer, save images of the viewer to the disk, and change the properties of the viewer such as the background color, we ultimately decided to move forward using PPTK as our external PPTK renderer.

3 Methodology

3.1 Selection of Tools and Initial Attempts

By using PPTK, which proved to be more promising than the previously mentioned alternatives, we could send data using UDP packets and have the point cloud change in real-time according to the VR headset's movement.

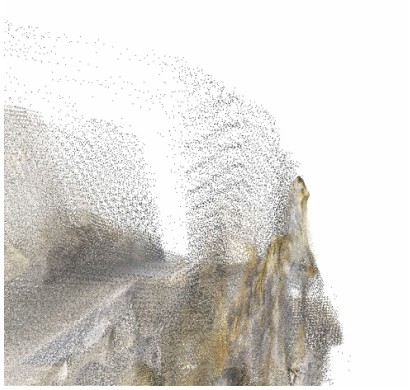

Figure 2: *Real-time output of our test point cloud from the PPTK Viewer (10 million points).*

Our initial methodology involved outputting the user's positional data and gaze direction from a VR headset and Unity into an external software program such as CloudCompare or MeshLab. However, we found that there are better choices for our purposes since CloudCompare and MeshLab do not support updating point cloud positions in real-time, despite successfully sending position and rotation data from Unity using a UDP socket.

In order to render the point cloud as a 2D texture and mask to be used in Unity, we experimented with various approaches for sending the image data back to Unity. Initially, we attempted to use PPTK's built-in ***capture()*** function to save images as .png files and send them to Unity as bytes. However, this approach was unsuccessful as Unity was unable to convert the bytes to a texture, likely due to data loss during transmission.

Our next attempt involved saving the images resulting from the ***capture()*** function directly to Unity's assets folder, with the intention of overwriting the same image file to avoid using up excessive storage space. While we were able to display this image as a mask in Unity, we encountered issues with updating the image in real-time and file corruption due to overwriting.

Another option we explored was to fork the existing PPTK GitHub repository and make a new package that implements an alternate version of the ***capture(),*** instead returning image data as bytes. After receiving the data, this package would send the raw image bytes to our Unity session without saving a file to disk, which would then load the image bytes as a 2D texture to display as intended. As PPTK is currently no longer being maintained, setting up a working PPTK build has proved difficult, specifically due to most of the dependencies being outdated and in some cases deprecated.

Ultimately, the methodology we relied on was the use of a webcam texture in Unity, leveraging Open Broadcaster Software's (OBS) virtual camera to record the PPTK viewer window, which updates in real-time. We have demonstrated that live webcam footage can be displayed in Unity using the WebCamTexture feature. This gives us the right tools to start with our implementation.

The two point clouds we used for testing were as follows:
Data set **(A)**, is a relatively small data set of about 16 million points.
Data set **(B)**, is a large data set of about 82 million points.

We believed this difference in scale would give us a good measure of the performance of our implementation.

3.2 Detailed Process

The process consisted of several crucial stages of data processing and interaction.

Firstly, the position and rotation data from Unity were transmitted via a UDP/TCP socket. The selection between UDP and TCP was based on a trade-off between speed and reliability. UDP, though faster, may result in data loss, while TCP provides more reliable data transfer, albeit at a slower pace. We decided that UDP would be more suitable in this case.

```
Viewer: received message type
[[-2.03, 0.93, 46.51, 1.216657791755297, 1.2280886873061554, 5]]
[[-2.03, 0.93, 46.51, 1.2572872236424264, 1.238083197396645, 5]]
Viewer: received message type
Viewer: received message type
Viewer: received message type
Viewer: received message type
[[-2.04, 0.93, 46.51, 1.2411378438248493, 1.2531558924828727, 5]]
Viewer: received message type
Viewer: received message type
[[-2.04, 0.93, 46.52, 1.1855052278050469, 1.2627558386925601, 5]]
[[-2.04, 0.93, 46.52, 1.0924433084441045, 1.276409108049586, 5]]
Viewer: received message type
Viewer: received message type
Viewer: received message type
Viewer: received message type
[[-2.05, 0.92, 46.53, 1.029030693628321, 1.2820079766379815, 5]]
[[-2.06, 0.92, 46.53, 0.9214298280942234, 1.2927052060536854, 5]]
Viewer: received message type
Viewer: received message type
Viewer: received message type
Viewer: received message type
[[-2.06, 0.91, 46.54, 0.7359883866334337, 1.3100552168251949, 5]]
[[-2.07, 0.91, 46.54, 0.501564763897015, 1.32113803052939, 5]]
Viewer: received message type
Viewer: received message type
Viewer: received message type
Viewer: received message type
[[-2.09, 0.9, 46.55, 0.26240235696287345, 1.3113376281529587, 5]]
Viewer: received message type
```

Figure 3: *Position and rotation data transferring in real-time from Unity to PPTK.*

Once the transmission phase was concluded, our Python script, equipped with the PPTK library, would begin the second step which involved receiving the transform data from Unity. This data interchange capability was crucial in achieving real-time interaction between the two platforms.

Following the reception of data, the third phase of our methodology consisted of parsing this data. Here, we translated the incoming stream into distinct x, y, z coordinates, and the camera's azimuthal (phi), and elevation (theta) angles, along with the camera's distance to the look-at point (r). This parsing was crucial to creating the set of parameters that the PPTK library could utilize effectively.

In the fourth stage, we used PPTK's ***play()*** function, which accepted the poses array composed of the parsed position and rotation data. This function was primarily responsible for rendering the point cloud data in the PPTK viewer, enabling real-time updates in sync with the VR headset's movements.

Our fifth step was to configure the viewer's background to a monochromatic green color. This seemingly trivial setup was pivotal for our chroma key filter application, providing the means to distinguish the point cloud data from the background. This is due to PPTK not allowing a proper transparent background for the point cloud viewer. Changing the alpha value in PPTK will also reduce point visibility.

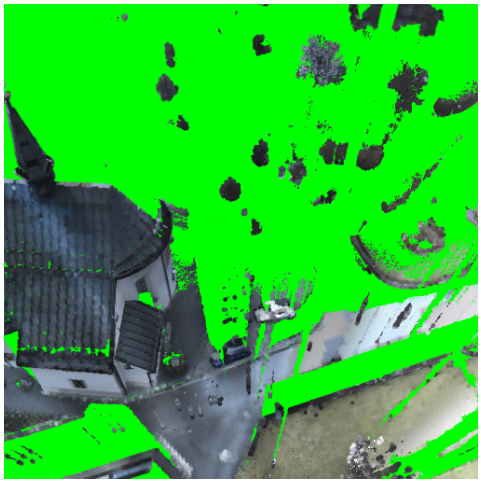

Figure 4: *PPTK viewer output of data set (B)*

As we moved into the sixth stage, we employed the OBS virtual camera to capture the output window displaying the PPTK viewer. This ensured a real-time, live feed of our point cloud visualization could be channeled into the subsequent stages of our methodology.

The seventh step involved receiving the OBS virtual camera's output via a Unity script utilizing the WebCamTexture feature. This output was subsequently applied to a raw image object, which is nested within the Canvas object in our Unity scene. This was a significant step towards integrating our point cloud visualization into the Unity environment.

Once we initiated the scene in the eighth stage, our Unity scene began to display the real-time, rendered PPTK point cloud. However, at this stage, the green background was still evident, creating a need for further processing.

To resolve this issue, our ninth step involved the creation of a script to apply a chroma key filter in Unity. This script effectively removed the green screen background, improving the point clouds visibility by adding a transparent background. Since neither PPTK nor OBS provided this functionality in a way that would work well within Unity, this step was crucial for achieving a clean visualization in Unity.

3.3 Final Rendering

Finally, we achieved the desired outcome—a point cloud rendered in PPTK that updates in real-time within our Unity scene according to the VR headset's movements. The resultant point cloud not only moved in real-time but also appeared on a transparent background, ensuring that only the points were visible. This culminating step marked the successful execution of our methodology, demonstrating a seamless integration of PPTK and Unity through a carefully designed and executed process.

4 Results

4.1 Overview

Over the course of this study, we have delved into numerous external libraries, such as CloudCompare and PPTK, and progressively honed our solutions for real-time rendering of large point cloud weather data in Virtual Reality (VR) using Unity. The challenges were manifold, given the necessity to identify an appropriate software capable of managing real-time modifications to point cloud positions and transmitting rendered images back to Unity.

Our culminating breakthrough arrived with the successful utilization of Open Broadcaster Software's (OBS) virtual camera in combination with Unity's Webcam texture. This innovative approach enabled us to render the point cloud data in Unity as an overlay in the Canvas game object. Remarkably, the overlay, despite its inherent 2D texture format, projected a compelling illusion of three-dimensionality that responded in real-time to the movement of the VR headset. As the user navigated the VR space, the overlay updated dynamically, synchronously adapting the representation of the point cloud. The careful handling of perspective, scale, and positional data created a convincing sense of depth and spatial relation in the visual output.

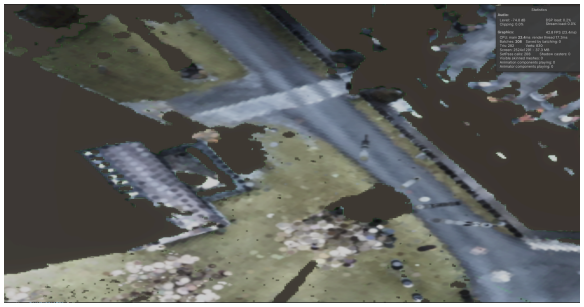

Figure 5a: *The output in the Unity game view.*

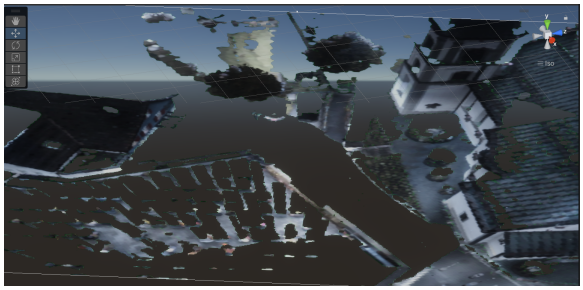

Figure 5b: *The output in the Unity scene view.*

4.2 Performance

This result was further enhanced by the exceptional performance of the system, which maintained a frame rate between 70-90 frames per second (fps) in Unity for data set **(A)**. Generally, this number of points rendered in VR would result in about 40-50 fps in Unity.

This is also impressive when tested with the very large point cloud data set **(B)**, which consists of about 82 million points. This resulted in a consistent frame rate of 40-60 fps.

This high frame rate ensured smooth and responsive interactions, crucial for the immersive and intuitive experience required in VR applications. We should note that a fully immersive VR implementation would require the rendering of two different images, one for each eye, in order for the user to perceive depth.

These achievements were accomplished parallel to our explorations into various code bases, data structures, LOD strategies, User Interface (UI) limitations on PC visualization, and other methodologies to augment the performance of our framework.

In summary, we have successfully developed and documented a proof-of-concept software that demonstrates the effective real-time rendering of large point cloud weather data in VR using Unity.

5 Conclusion and Future Work

In conclusion, our research project has successfully rendered large point cloud data in virtual reality using Unity. Our methodology, which began with an exploration of various external libraries and software, has evolved to the successful implementation of OBS Virtual Camera and Unity's Webcam texture for real-time rendering and transmission of point cloud data. This has resulted in a dynamic overlay that updates in real-time according to the movement of the VR headset, all the while maintaining an impressive frame rate between 70-90 fps for 10 million points, and 40-50 fps for 82 million points.

While we have made significant strides in our research, the work continues. Future directions of this project may involve further optimization of the rendering pipeline to enhance the software's efficiency and responsiveness. For example, by building a forked version of PPTK, we could integrate key optimizations in the saving and sending of image data. This is also due to a key limitation of our current implementation, which requires 3 separate software to be running at the same time in order to function. We believe that by having software that can properly send image data to Unity in real-time, we can eliminate the need for OBS.

Additionally, we will explore different methods for sending image data between our current system and Unity, with the aim of circumventing any remaining limitations in the software. For example, as an alternative to sending positional data via UDP protocol, we considered using named pipes for inter-process communication as we are running all programs on the same machine, meaning we would be able to prevent data loss.

We will also consider the integration of our solution with other external libraries and software, especially those that use alternative methods to optimize point cloud rendering. Outside of the LOD-based approaches we were targeting, (Schütz et al., 2021) proposes several methods that rely on computer shader-based methods of rendering that outperform previous methods that rely on the provided point primitives of popular graphics APIs such as OpenGL. This paper also proposes an optimized vertex ordering method for storing point cloud information in memory that makes existing graphics APIs more efficient. With the adoption of these methods for rendering in the OpenGL standard, we could see an improvement in rendering from software that uses their APIs such as in (Ruiz, 2021/2023).

As for other external libraries and plug-ins, while we initially disregarded Open3D, a free open-source, actively maintained Python library with point cloud visualization features, as they lacked octree generation which disabled them from handling large point cloud data. However, we must note that the package is actively improving its point cloud support, and have recently implemented point cloud partitioning using PCA (*Open3D – A Modern Library for 3D Data Processing*, n.d.; Zhou et al., 2018). There also exist priced Unity plug-ins which claim to handle massive amounts of point clouds data as well (mika, 2016/2023). We would have to purchase in order to test its performance, which we are unable to within the financial constraints of this project.

We also recognize that our techniques, which involve sending multiple messages between sessions for each rendered frame, might be cumbersome and inefficient but work as a proof of concept. However, during the course of our project, other novel advancements in point cloud rendering built into Unity have been made, which may prove to be faster for VR development. In particular, FastPoints was released as a state-of-the-art point cloud renderer plug-in for Unity, which aims to support standard point cloud formats files such as .LAS and .PLY. FastPoints prioritizes ease-of-use and drag-and-drop support by displaying a decimated point cloud in the Unity Editor while in parallel, generating a LOD octree in alignment with the Potree Converter methods. These methods could prove to be more useful to rendering point clouds directly in Unity as it uses the existing, popular Potree hierarchical structure, however, the rendering performance for VR remains to be tested.

As we seek to study other methods of point cloud rendering, we can not only create a more robust and versatile framework but also expand the capacity of our system to render even larger point cloud datasets in VR environments.

6 References

6.1 Bibliography

CloudCompare. (n.d.). [3D point cloud and mesh processing software. Open Source Project]. Retrieved May 20, 2023, from https://www.danielgm.net/cc/

Erfani, A. (2022, September 20). *Point cloud rendering with Unity*. Medium. https://bootcamp.uxdesign.cc/point-cloud-rendering-with-unity-1a07345eb27a

Fraiss, S. (2017). *Rendering Large Point Clouds in Unity*. https://www.cg.tuwien.ac.at/research/publications/2017/FRAISS-2017-PCU/

mika. (2023). *Unity PointCloud-Viewer* [ShaderLab]. https://github.com/unitycoder/UnityPointCloudViewer (Original work published 2016)

Open3D – A Modern Library for 3D Data Processing. (n.d.). Retrieved May 20, 2023, from http://www.open3d.org/

potree. (2023). *Potree* [JavaScript].

https://github.com/potree/potree (Original work published 2012)
PotreeConverter. (n.d.). Retrieved May 20, 2023, from https://github.com/potree/PotreeConverter
pptk—Point Processing Toolkit. (2023). [C++]. HERE Technologies. https://github.com/heremaps/pptk (Original work published 2018)
Ruiz, A. L. (2023). *Large Point Cloud Rendering* [C++]. https://github.com/AlfonsoLRz/PointCloudRendering (Original work published 2021)
Schütz, M. (2016). *Potree: Rendering Large Point Clouds in Web Browsers*.
Schütz, M., Kerbl, B., & Wimmer, M. (2021). Rendering Point Clouds with Compute Shaders and Vertex Order Optimization. *Computer Graphics Forum*, *40*(4), 115–126. https://doi.org/10.1111/cgf.14345
Schütz, M., Ohrhallinger, S., & Wimmer, M. (2020). Fast Out‐of‐Core Octree Generation for Massive Point Clouds. *Computer Graphics Forum*, *39*(7), 155–167. https://doi.org/10.1111/cgf.14134
Takahashi, K. (2023). *Pcx—Point Cloud Importer/Renderer for Unity* [C#]. https://github.com/keijiro/Pcx (Original work published 2017)
Zhou, Q.-Y., Park, J., & Koltun, V. (2018). *Open3D: A Modern Library for 3D Data Processing* (arXiv:1801.09847). arXiv. https://doi.org/10.48550/arXiv.1801.09847

6.2 Point Cloud Sources

- Small point cloud data (**data set A**): untermaederbrunnen_station1_xyz_intensity_rgb.txt http://semantic3d.net/view_dbase.php?chl=1
- Large Point cloud data (**data set B**): Bildstein_station1_xyz_intensity_rgb.txt http://semantic3d.net/view_dbase.php?chl=1

7 Appendix

7.1 Source Code & Supplementary Materials

This following Google Drive folder includes:

1. The unitypackage file for our work in Unity (the display might have to be changed when running the scene from Display 2 to 1 or vise versa in order to properly see the point cloud)
2. The python script used to run PPTK and receive position and rotation data from Unity.

https://drive.google.com/drive/folders/1qLzGwXtYaJPmF-Jkje5YMeT9j4_i9CQ0?usp=sharing