

Visualization of Large Point Cloud in Unity

J.M. Santana¹ , A. Trujillo¹ and S. Ortega¹

¹Imaging Technology Center (CTIM), ULPGC, Spain

Abstract

Large point cloud rendering has become a very relevant topic on 3D graphics as scanners and other sources of 3D point data are nowadays available to companies and the general public. In this project, we propose an implementation of a point cloud viewer, designed for the full-detail visualization of virtually unlimited point clouds for their inspection on short ranges. This work presents the data structure and the LoD technique to achieve a real-time rendering of the model, making emphasis on the details of an initial prototype based on Unity.

CCS Concepts

• **Human-centered computing** → Scientific visualization; • **Hardware** → Scanners;

1. Introduction

Currently, we can find 3D scanners in all sort of novel applications such as drones, autonomous vehicles, the game industry (Microsoft Kinect), etc. The scientific and technical importance of datasets generated by 3D scanners cannot be overstated as they provide faithful models that do not rely on spatial interpolations. The data produced by laser scanning hardware consists normally on a set of XYZ points along with other punctual properties, such as reflection, intensity and return number. The set of points does not warrant its locality, continuity, or, more generally, any other characteristic that eases the rendering process. The LAS file format was introduced to store datasets with these raw scanner data along with per-point classification, which segments the objects seen in the scene.

Several alternatives already exist for the display of large classified point clouds, being three of the most notable open-source examples Potree [Sch16], Plas.Io [†] and the 3D tiles of the Cesium world-engine, aimed at the visualization of point clouds on the web. In that regard, a previous work [SWT*17] already used georeferenced point clouds to display simulation results on a GIS environment. In the realm of proprietary applications, we find alternatives like Fugro (analysis and visualization tool for geospatial data) [‡] or the Point cloud scene layers of ArcGIS Pro for Desktop [§]. However, these tools normally focus on the visual appearance of the model, relying on decimated versions of the point clouds or on

a view-dependent LoD strategy that shows coarser models of the same, which can be problematic during a thorough inspection of the model.

In contrast with these specialized viewers, many other scientific and technical visualization tools rely on general purpose game engines like Unity or Unreal. This trend is not only due to the relative ease of use of these frameworks compared with low-level libraries, but their capacity to deploy our visual experiences on multiple platforms. In this work-in-progress, we are focusing on Unity as it is commonly used by the technical community, that might be interested in point-cloud inspection, due to its ease of use and wide community and support. In the first use case of this tool, our intent is to obtain a point cloud viewer which can perform real-time rendering of airborne LiDAR terrain scans on Unity. The developed tool allows combining the complexity of 3D point cloud models with the advantages and features of a state-of-the-art game engine.

2. Background and Motivation

The particular use case that motivated this project has been the visual inspection of airborne scans of long power line corridors, from which automatic and manual classification has been produced. The longitude of these corridors varies within a range of 100 - 200 km., with an average width of ~100 m. and a point density up to 50 p/m². Hence, the multi-corridor models are usually encoded in several LAS files, adding up to hundreds of GBs.

The final intention of the project is to accurately display the point cloud within the range of view, enabling a seamless navigation across the model, while allowing the inclusion of other 3D elements and the use of tools provided by the Unity framework, as seen in Figure 1. The use case imposes a local but holistic render-

[†] <http://plas.io>

[‡] <https://www.fugro.com/your-industry/power/transmission-and-distribution>

[§] <https://pro.arcgis.com/en/pro-app/help/mapping/layer-properties/point-cloud-scene-layer-in-arcgis-pro.htm>

ing of the point cloud, facilitating the spotting of undesired noise generated by the LiDAR scanner, as well as misclassified points.

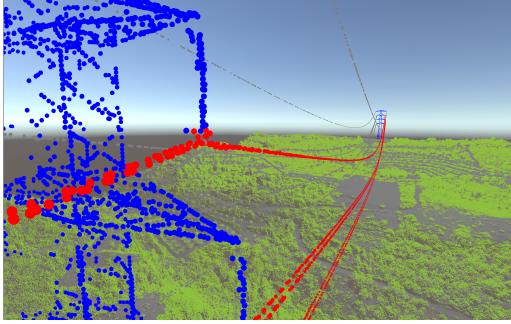


Figure 1: Final view of the point cloud rendering. Other components as the Sky-Box or the camera navigation are provided by the Unity engine.

3. Model preparation

The main concern when dealing with these datasets is their massive sizes, where literally millions of points are required to represent a scenario or object with an acceptable degree of fidelity. The complexity of these models imposes a hierarchical partitioning that enables out-of-core rendering. The literature [Fra17] covers a series of possible subdivisions of the point cloud model in order to serve manageable chunks to the GPU. In this project, we have opted to use a binary tree partition of the space, similar to the one proposed by Gobbetti and Marton [GM04]. However, as our model must be rendered with all its points at any distance, no coarser levels of detail have been generated in upper levels of the tree.

All the nodes of the tree are contained within a tight axis-aligned box, which is precomputed and serves the LoD test and point-picking strategies. Different bipartition strategies were implemented and tested. Dividing the nodes at the mean value along their longest axis offered the best results, minimizing the overall bounding volume of the nodes. The final model is stored in an uncompressed folder which contains a JSON index of the tree, pointing to binary files. Each node file stores the offset relative to the node center and the class of each point, all encoded in single precision. Our current Matlab implementation generates models at a rate of ~2.15 sec. per million points on desktop hardware. In Unity, the whole binary tree forms a hierarchy of *GameObjects*, and the LoD test uses the precomputed bounding boxes (as *Bounds* instances).

4. Discrete Level Of Detail Strategy

In order to keep a high-performance rendering without removing points from the visible model, a strategy for the rendering at different distances was devised. At a short distance, the goal is to show the points as rounded objects with a fixed physical size. At long distances, points must preserve a minimum screen-space size to remain visible and to avoid undesirable aliasing problems. However, Unity does not enable the user to establish a screen point size to preserve DirectX 11 as a target platform.

Our solution consists in using two different materials that are interchanged depending on the distance of the node to the viewer.

- Far Distance Material (FDM): The mesh is rendered by setting *Points* as the mesh topology. This generates pixel size points during rasterization and the vertex color is used as fragment output.
- Near Distance Material (NDM): When points need to maintain a physical size on screen, we make use of the geometry stage, which requires at least OpenGL 4.1 or DirectX 11.0 Shader Level 5 on desktop (the *OpenGL Core Rendering Platform*). In our implementation, this shader stage takes each one of the vertices V in our mesh as input and outputs a single equilateral triangle. The screen-aligned triangles are generated based on the camera vector (Cam_{Up} and Cam_{Left}), including texture coordinates T_C that allow the fragment shader to cut off the embedded circle (as depicted in Figure 2), following the Expressions 1.

$$\begin{aligned} V_1 &= V + 2S \cdot \overrightarrow{Cam_{Up}} & T_{C1} &= \left\{ 0, \frac{1}{\sin(\pi/6)} \right\} \\ V_2 &= V + S \cdot (\overrightarrow{Cam_{Up}} + \overrightarrow{Cam_{Left}}) & T_{C2} &= \left\{ \frac{1}{\tan(\pi/6)}, -1 \right\} \\ V_3 &= V + S \cdot (\overrightarrow{Cam_{Up}} - \overrightarrow{Cam_{Left}}) & T_{C3} &= \left\{ -\frac{1}{\tan(\pi/6)}, -1 \right\} \end{aligned} \quad (1)$$

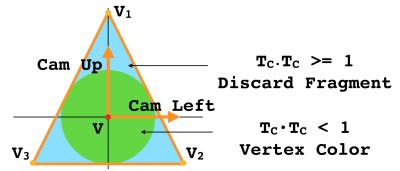


Figure 2: Circle symbol of radius S generated from the texture coordinates of the GPU-generated triangles.

By considering the vertical screen resolution W and the camera field of view F_V provided by Unity, we establish the distance threshold $D_T = \frac{S}{\tan(F_V/W)}$ at which the projected size of a NDM point equals one pixel. For any given node i we can compute the distance to the furthest and closest point of its bounding box, which constrains the distance to any point within it. Our rendering algorithm applies the FDM to any node in which $D_{Min} > D_T$ and the NDM if $D_{Max} < D_T$. For any other node in between, it performs a double pass rendering with both materials, ensuring a smooth transition between levels of detail.

References

- [Fra17] FRAISS S. M.: Rendering large point clouds in unity, 2017. URL: <https://www.cg.tuwien.ac.at/research/publications/2017/FRAISS-2017-PCU/>.
- [GM04] GOBBETTI E., MARTON F.: Layered point clouds: a simple and efficient multiresolution structure for distributing and rendering gigantic point-sampled models. *Computers & Graphics* 28, 6 (2004), 815–826.
- [Sch16] SCHÜTZ M.: Potree: Rendering large point clouds in web browsers. *Technische Universität Wien, Wieden* (2016).
- [SWT*17] SANTANA J., WENDEL J., TRUJILLO A., SUÁREZ J., SIMONS A., KOCH A.: Multimodal location based services - Semantic 3D city data as virtual and augmented reality. 2017.