Revista Minelor – Mining Revue
ISSN-L 1220-2053 / ISSN 2247-8590
vol. 30, issue 4 / 2024, pp. 25-31

PROFESSIONAL WEB APPLICATION FOR LIDAR DATA VISUALIZATION AND METRIC INSPECTION

Cristiana GLONȚ[1], Octavian Laurențiu BALOTĂ[2], Csaba BALASZ[3]*

[1]Ph.D student, University of Petrosani, Petroșani, Romania
[2]Tehnogis Grup SRL, Romania, octavian.balota@tehnogis.ro
[3]Ph.D student, University of Petrosani, Petroșani, Romania, balaszcsaby@yahoo.com

DOI: 10.2478/minrv-2024-0059

Abstract: *Lidar data is difficult to access for regular users due to the very large memory volume required and the need to handle it with powerful computers and quite expensive specialized software. The presented web application is offered for free in the LidarTools version and at a low cost in the LidarMap version, which combines shape-type vector data with the Lidar point cloud. The application uses a POTREE structure adapted for rapid visualization functions but enhanced with functions for measuring in the point cloud (distances, areas), visualization functions, profile extraction, statistical functions, semi-automatic vectorization functions of linear objects (power lines), coordinate transformation functions, as well as other functions specific to users of such data. It is also presented here the project for stereo vectoring module in LidarMap application.*
Keywords: *Lidar, change detection, application, cloud points, data visualization*

1. Introduction

The most efficient solutions for developing complex scientific applications with point cloud data are based on the development environment around the open-source web application, POTREE. Following an in-depth analysis of the Potree working environment, a basic program was built on the Potree framework and using its function libraries, where we developed specific applications based on our own algorithms. Potree files are partitioned in an **octree** structure. All octree nodes, both intermediate and branches, contain a subsample of sparse points. The spacing defines the minimum distance between points in the root node. With each level, the distance is halved; for example, if the distance in the root is 1.0, then the distance in its children is 0.5. Rendering at a lower level of smaller nodes is done through a coarse representation of the point cloud.

This hierarchical structure helps manage and efficiently render very large datasets by subdividing the point cloud into a series of nested cubes.

Root Node: The entire point cloud is initially represented by a single root node, which covers the entire spatial extent of the data.

Subdivisions: The root node is recursively subdivided into eight smaller cubes (octants), each representing a smaller portion of the point cloud. This subdivision continues down the hierarchy, creating nodes at various levels of detail.

Leaf Nodes: The octree structure ends at the leaf nodes, which contain the actual point data. Each leaf node holds a subset of the point cloud, and these subsets are smaller and denser compared to nodes at higher levels. The more nodes loaded, the higher the quality. The figure below shows the content of the root and its children, as well as how the level of detail increases when rendered together

This structure is perfectly suited for visualizing large Lidar datasets, but data processing operations such as classification, filtering, and even data selection are difficult or impossible on such structures. Therefore, for manipulating, visualizing, and processing Lidar data, combined data structures should be used, and if we want to develop complex applications, we must be careful to choose programming environments, languages, and function libraries that are adapted to the objectives we aim to achieve. An application that we

* Corresponding author: Balasz Csaba, eng. Ph.D. stud., University of Petrosani, Petrosani, Romania, Contact details: University of Petrosani, 20 University Street, balaszcsaby@yahoo.com

consider extremely useful for Lidar data users is one that allows us to extract 3D linear details useful for infrastructure project designers, for example. 3D mapping from Lidar data is accessible through complex and very expensive programs such as ORBIT GT (Orbit Geospatial Technologies) from Belgium, VisionLidar produced by Geo-Plus from Canada, or the most widespread on the market of applications that work with Lidar data, the set of applications from TerraSolid.

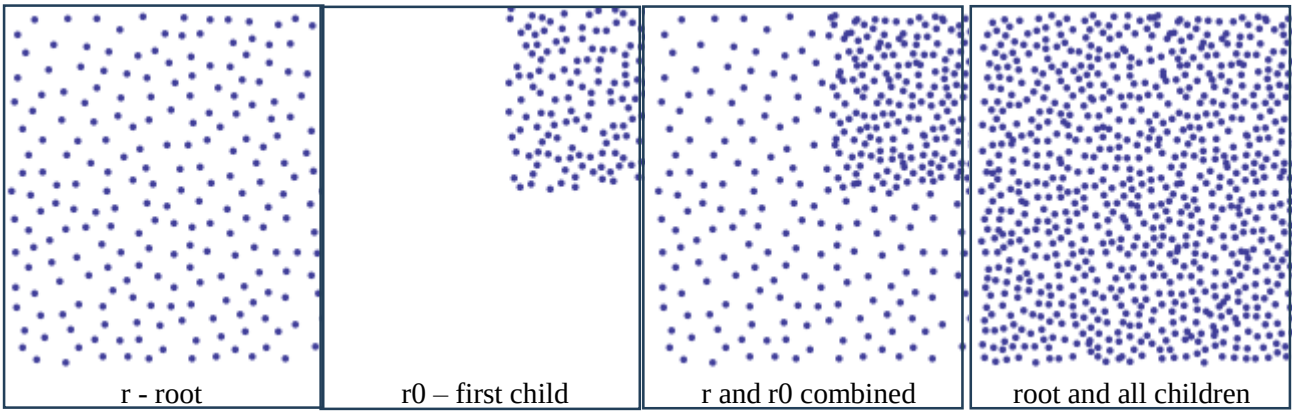

Fig. 1. OCTREE structure

Vectorization on the point cloud in these software tools is conceived as a semi-automatic procedure for drawing lines of slope change, for example, curb corners for sidewalks, or eaves edges for a roof. It involves drawing on a 3D model but viewed on a 2D screen. It should be emphasized that such vectorization cannot be more accurate than vectorization on a pair of stereoscopic images because the likelihood that the Lidar point cloud on which the vectorization is snapped to a point contains points exactly on the roof corner is almost zero. Figure 2 shows how, in a roof corner, the distribution of points is so random that there is no point that falls exactly at the mathematical corner of the roof.

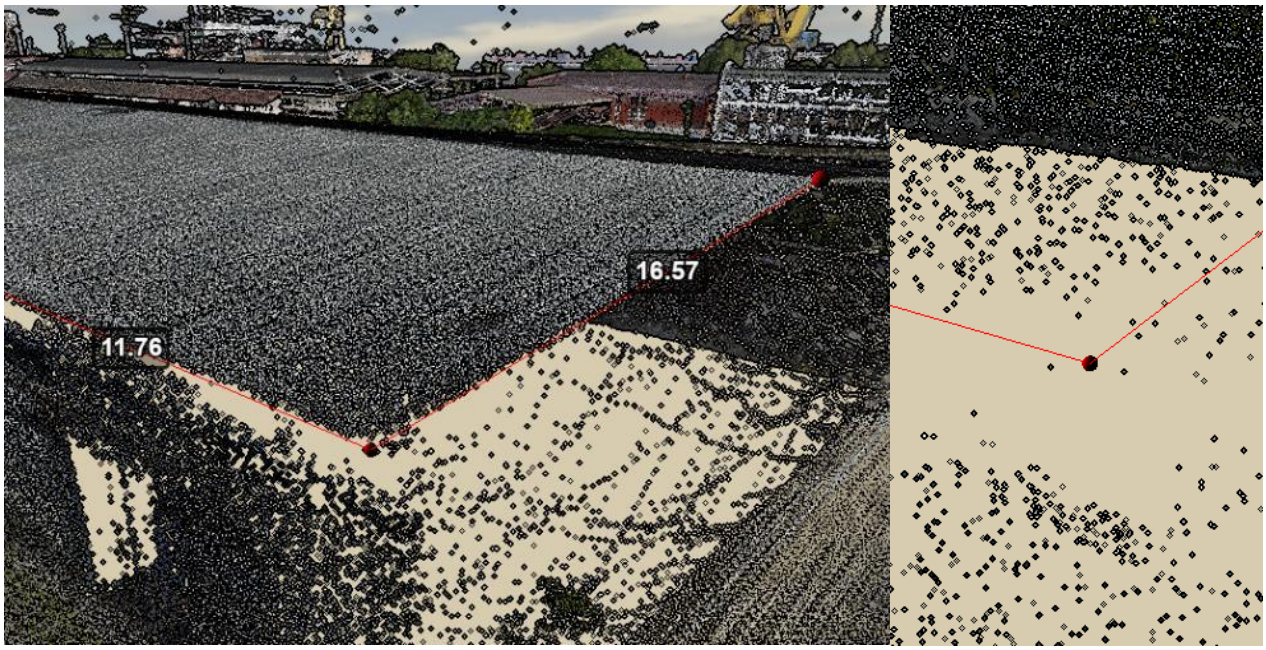

Fig. 2. 3D Vectoring using 2D screen with our LidarTools web application

In our ongoing efforts to extract linear information from Lidar point clouds, we have recurrently encountered a significant challenge: selecting the most optimally positioned point at a corner necessitated the exhaustive rotation of the point cloud in all possible orientations to ascertain the best solution. To mitigate the substantial time investment required for these vectorization processes, we identified the imperative need to develop a straightforward yet effective application, which we have designated as LidarMap. This tool facilitates stereoscopic vectorization within the point cloud, thereby achieving a level of precision commensurate with that of image-based vectorization. Moreover, stereoscopic vectorization within the point

cloud affords the capability to dynamically alter the perspective, enabling the identification of the most precise point placement for details, including the building's footprint. This process is markedly more arduous when constrained to images, which are limited to a fixed nadir perspective.

For creating 3D stereo applications, OpenGL and the associated GLUT library is needed. It is assumed that the reader is both familiar with how to create the appropriate eye positions for comfortable stereo viewing and the reader has an OpenGL card and any associated hardware (eg: stereo monitors and associated 3D glasses).

2. Stereo viewing principle

We will explain here the basic principles of stereoscopic vision implemented through OpenGL and the associated libraries that we used in the development of the LidarMap application. It is good to know that the options are diverse and depend on the programming environment and the hardware used, but the principle of approaching stereoscopic vision is largely based on the similarity with human vision.

The most frequently used principle is based on the use of a monitor at least 120Hz frequency and which allows the alternative display of two perspectives corresponding to the 2 eyes at the frequency of 60Hz. Thus, the perspective of the left eye will alternate with the perspective of the right eye, creating the effect of depth, of stereoscopy on the screen for the viewer. This technique is known in OpenGL terminology as Quad Buffering. Another hard implementation is based on the use of a 4K monitor and the alternate display of perspectives on odd and even horizontal lines, the technology being called "interlaced". But the programmer working with OpenGL no longer has to take into account these technological aspects, the OpenGL routines and libraries recognize and apply the specific procedures transparently for the programmer.

The best implementation of the stereo view using OpenGL is the so called **off-axis method**, the correct one as is described by Paul Bourke in [1].

Camera (the eye) is defined by its position, view direction, up vector, eye separation, distance to zero parallax (figure 3) and the near and far cutting planes. Position, eye separation, zero parallax distance, and cutting planes are most conveniently specified in model coordinates, direction and up vector are orthonormal vectors. With regard to parameters for adjusting stereoscopic viewing we would argue that the distance to zero parallax is the most natural, not only does it relate directly to the scale of the model and the relative position of the camera, it also has a direct bearing on the stereoscopic result ... namely that objects at that distance will appear to be at the depth of the screen. In order not to burden the operators with multiple stereo controls one can usually just internally set the eye separation to 1/30 of the zero parallax distance (camera.eyesep = camera.fo / 30), this will give acceptable stereoscopic viewing in almost all situations and is independent of model scale.

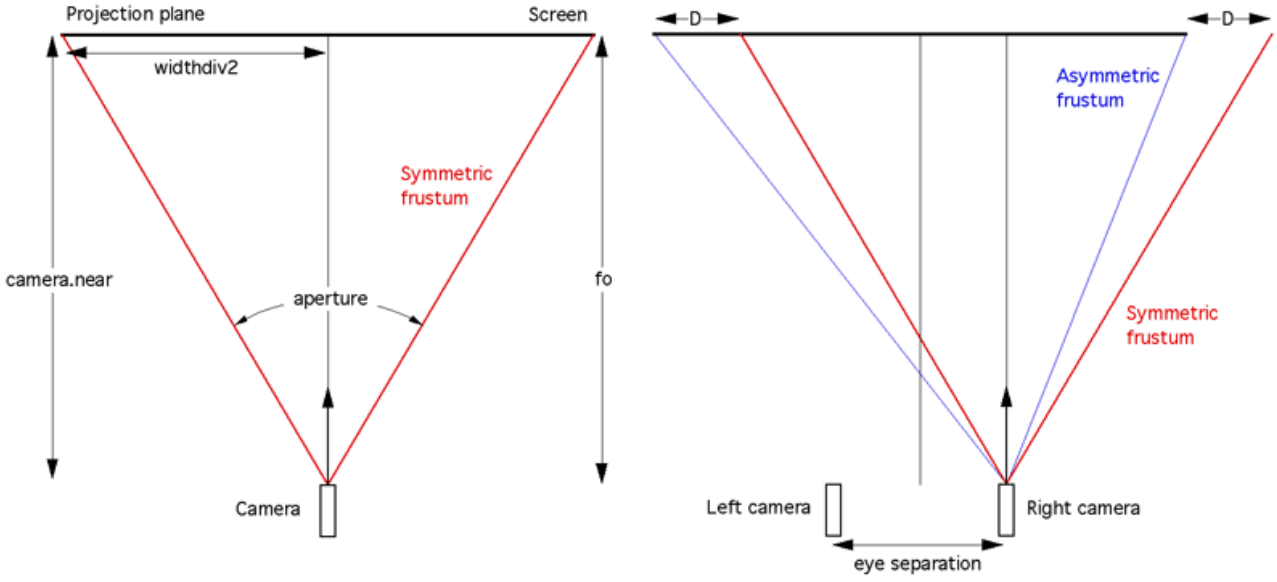

Fig. 3. The off-axis frustum method

The above diagram (view from above the two cameras) is intended to illustrate how the amount by which to offset the frustums is calculated. Note there is only horizontal parallax. This is intended to be a guide for OpenGL programmers; as such there are some assumptions that relate to OpenGL that may not be appropriate to other APIs. The eye separation is exaggerated in order to make the diagram clearer.

Another approach is Toe-in method (figure 4) where the camera has a fixed and symmetric aperture, each camera is pointed at a single focal point. Images created using the "toe-in" method will still appear stereoscopic but the vertical parallax it introduces will cause increased discomfort levels. To implement this, gluPerspective() function is used comparing with off-axis method where glFrustum() is used.

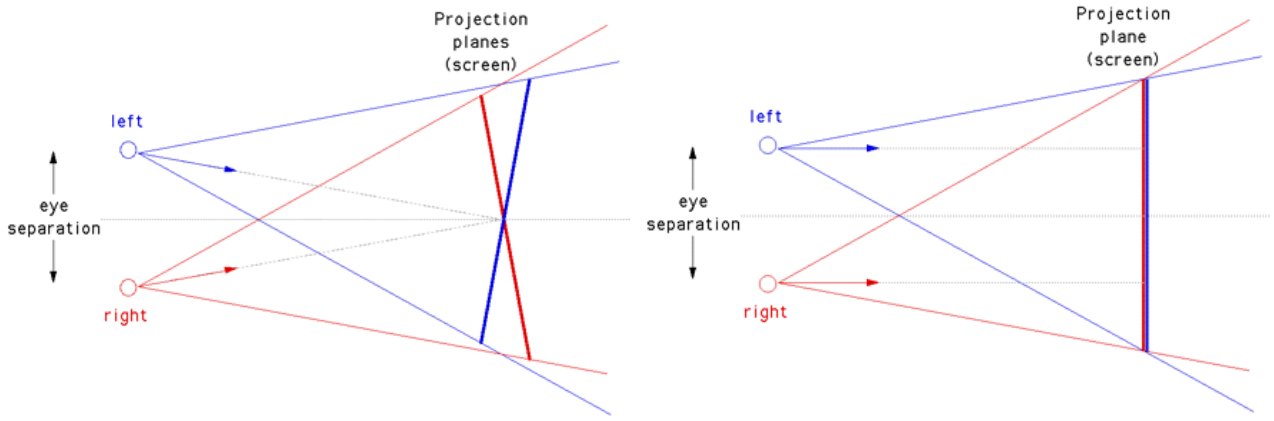

Fig. 4. The Toe-in method (left) versus off-axis frustum method (right)

As it was mentioned, to implement the correct method, glFrustum was used. To understand better the frustum concept, see figure 5, a perspective representation of the frustum which is in fact a truncated pyramid with the top cut off, creating a shape that has six planes: near, far, left, right, top, and bottom. These planes define the boundaries of what will be projected onto the screen from the 3D scene.

In perspective projection, objects further from the viewer appear smaller, mimicking how human vision works. OpenGL uses the frustum to determine which objects or parts of objects should be rendered based on their positions in 3D space.

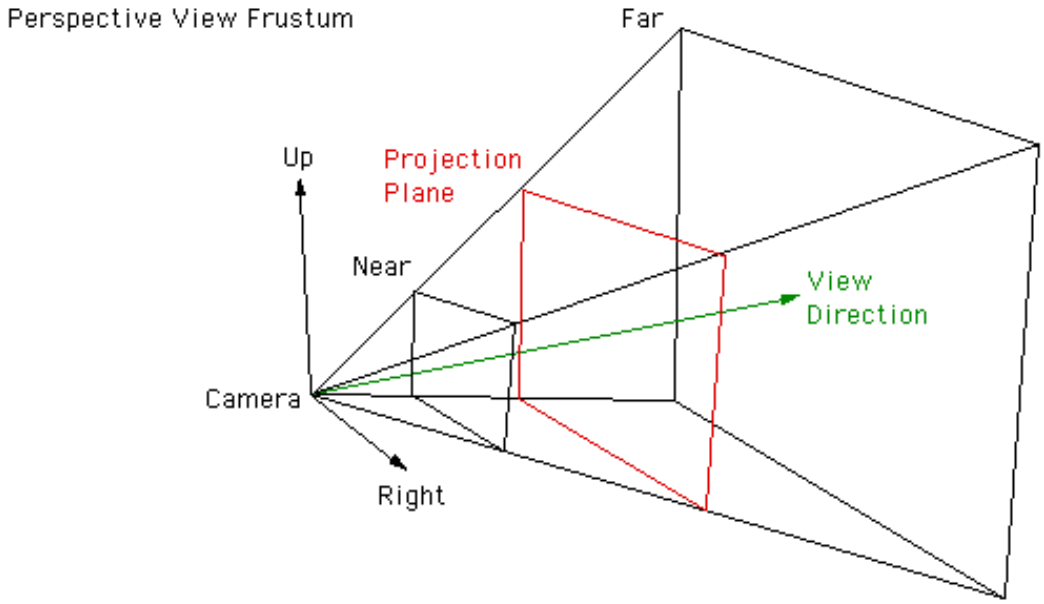

Fig. 5. The Perspective view of the frustum

Parameters of a Frustum

When defining a frustum in OpenGL, the following parameters should be specified:

- **Field of View (FoV):** The angle between the top and bottom planes (vertical FoV).
- **Aspect Ratio:** The ratio of the width to the height of the frustum, often matching the aspect ratio of the display or window.
- **Near Plane Distance:** The distance from the viewer to the near clipping plane.
- **Far Plane Distance:** The distance from the viewer to the far clipping plane.

In OpenGL, is possible to create a perspective frustum using functions like glFrustum (older OpenGL) or more commonly gluPerspective (part of the GLU library), see below:

```
#include <GL/glu.h>
// Set up a perspective projection matrix
void setupPerspective() {
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(fovY, aspectRatio, nearPlane, farPlane);
    glMatrixMode(GL_MODELVIEW); }
}
```

where:

- **fovY:** Vertical field of view angle in degrees.
- **aspectRatio:** Aspect ratio of the viewport (width/height).
- **nearPlane:** Distance to the near clipping plane.
- **farPlane:** Distance to the far clipping plane.

The near plane is closer to the viewer, and objects between the near and far planes are rendered.
The far plane is further away, and objects beyond this plane are not rendered.
The left, right, top, and bottom planes define the field of view.

The frustum is crucial for determining visibility and culling. Only objects within the frustum are rendered, which improves performance by avoiding the rendering of objects outside the viewer's view.
Understanding and configuring the frustum correctly ensures that scenes are rendered realistically and efficiently, with the appropriate perspective that matches the intended view of the 3D world.

For the moment we succeed to implement these concepts in a test Python application for stereoviewing the cloud points representing Corvin Castle from Hunedoara town.

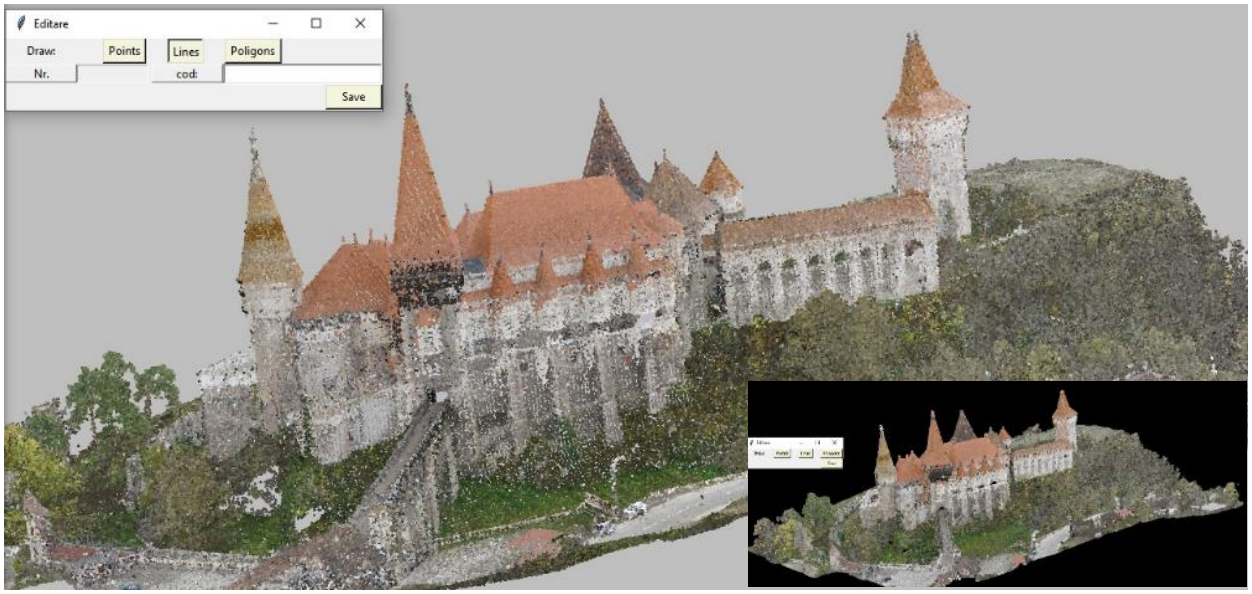

Fig. 6. 3D StereoView using a test Python application

In the image from figure 6, the data contains 20 million points, which Python loads in 4 seconds, then structures according to an octree in another 4 seconds, and subsequently works with only 460 thousand points. Unfortunately, in this this printscreen it is not possible to view both images to understand the stereoscopic view because the images are viewed alternately.

3. Best options for LidarMap development application

In order to create this test we analyze more possibilities to develop the LidarMap application.
First we had to choose the programming language from C++, Python and C#.

C++ is generally faster and offers better memory control compared to Python. And C#. For applications that require handling a large amount of data, such as LAS files with tens or hundreds of millions of points, C++ is a good choice.

C# is a high-level language that offers a balance between performance and ease of use. It provides good performance and better memory management than Python, thanks to its garbage collection mechanism. However, it typically does not match the raw performance and memory control of C++. C# is well-suited for applications that need a compromise between development productivity and performance.

Python is easier to use and offers increased productivity, but it may have lower performance for intensive processing and memory-consuming tasks. However, certain libraries such as Numpy and Pandas can help in this regard.

Looking to these characteristics we choose Python for the test application due to the rapid development but the final product will be developed in C++.

Very important in this development, was the choice for supporting Lidar data libraries. The most important are PCL (Point Cloud Library) and PDAL (Point Data Abstraction Library)

PCL provides a robust implementation of the octree, which is used for various operations such as nearest neighbor search, spatial segmentation, and point cloud compression.

PCL includes functionalities like OctreePointCloudSearch, OctreePointCloudCompression, and OctreePointCloudDensity, which can help to perform various operations on point clouds using the octree structure. Using the integrated structures, PCL offers a high level of trust in the implementation, as well as compatibility with other PCL components. It also saves from the complexity of developing and maintaining your own octree.

PDAL is more used for preprocessing and manipulating point clouds, with a focus on data flows and transformations. Although PDAL does not directly provide octree implementations for search or segmentation operations, PDAL may be used to manipulate and preprocess data before sending it to other libraries such as PCL.

PDAL excels in transformations, pipelines, and interoperability with other libraries and file formats. PDAL may be used to process data and then send it to PCL for octree-based operations.

The research for the best environment and libraries was very time consuming with a lot of tests. Here, below there are some results:

Characteristics	Python	C++	C#
Dev. Environment	Anaconda	Visual Studio	Visual Studio, Unity 3D
Support for OpenGL	pyOpenGL, GLFW	GLFW, GLM	OpenTK, GLFW<
Support for Lidar	PDAL, Laspy, PCL,pyproj, Rasterio, scipy, numpy	PCL, PDAL, VTK	PCL, HDF5, Potree
Performance top	3	1	2
Rapid development top	1	3	2

An important aspect for stereo view is that **PCLVisualizer**, part of the Point Cloud Library (PCL), is based on VTK (Visualization Toolkit), which uses OpenGL for 3D rendering. So, although PCLVisualizer uses OpenGL, options for stereoscopic rendering might be limited or require additional configuration, as most PCL implementations focus on traditional 3D rendering.

Stereoscopic Rendering Capability in PCLVisualizer VTK offers support for stereoscopic rendering, which can be enabled and configured for PCLVisualizer. However, the exact details of stereoscopic rendering may depend on the specific hardware and configuration. Stereoscopic rendering can be also useful for VR (Virtual Reality) or AR (Augmented Reality) applications, as well as for advanced 3D visualizations.

4. Conclusions and future development

In according with all these results we created 2 development environments, one for Python and one for C++. For Python, we use Anaconda and we add pygame and we import GLFW. Also, PDAL and PCL were installed through the commands: pip install pdal and pip install python-pcl or conda install -c sirokujira pcl through anaconda. Their use in Windows is done through the classical import pdal and import pcl commands respectively.

For C++ we installed PDAL and PCL through vcpkg preinstalled in Visual Studio. We used the commands: vcpkg install pdal and vcpkg install pcl. For anyone interested in such approach you have to know that the installation could take several tens of minutes up to hours.

Until now we succeed to test the main functionalities of the application, managing the lidar data, read and view in a decent period of time, the read of LAS file in 4,5 seconds for 20 millions of points and movements of the data like rotation, in real time.

Next development will be to implement an intelligent 3D mouse cursor for vectoring [2] and to implement the user interface design presented in the figure no 7.

Fig. 7 LidarMap user interface design

References

[1] **Bourke P.**, 2014
Stereographics: Computation for Stereoscopic Display 1999 – 2014, paulbourke.net

[2] **Schemali L., Eisemann E.**, 2014
Design and evaluation of mouse cursors in a stereoscopic desktop environment, in Symposium on 3D User Interfaces, 3DUI 2014 (TechNote) (pp. 67-70). New York, NY, USA:

This article is an open access article distributed under the Creative Commons BY SA 4.0 license. Authors retain all copyrights and agree to the terms of the above-mentioned CC BY SA 4.0 license.