A tool for manipulating huge point clouds
Technical report

Zsolt Jankó

SZTAKI, Budapest, Hungary
janko@sztaki.hu

Abstract. By the evolution of 3D scanning techniques, creating 3D models of real world objects is getting much easier. Beyond the human-sized objects one can easily scan complete buildings, roads, squares, or even towns and countries as well. The raw data that scanning technologies, such that LIDARs or photogrammetry based applications can provide are point clouds. The size of such a point cloud can be enormous, with billions of points, and processing and converting it to another format is very costly. Due to this it is important to efficiently visualise large point clouds and make it possible to modify them. In this paper we give a brief overview of point cloud visualisation techniques and describe our system that provides tools to manipulate point clouds by selecting, annotating, deleting, cleaning necessary parts of them.

1 Introduction

The geometry of digital 3D models are traditionally represented by polygons, majorly by triangles in the form of a mesh. This kind of representation helps efficiently visualise the model. However, real-world objects are rarely built of triangles in the nature, but they consist of little particles. Techniques that are intended to scan the 3D world, either active laser scanners or passive multi-camera based photogrammetry algorithms produce a huge set of such particles in the form of 3D points. Scanning objects and generating 3D assets is important from many aspects: entertainment (games and movies), cultural heritage, health care, architecture, etc.

In order to be able to visualise such data, the standard way is to convert them to triangular meshes, which is a rather complicated and time consuming task. Additionally, this conversion is a kind of lossy compression, which involves significant data loss. To avoid these drawbacks, during the last years it became popular to visualise the raw point data, instead of converting them to other format.

The difficulty of rendering resulting 3D point clouds of scanning technologies originates from the fact that such a point cloud usually consists of a huge number of points, typically hundreds of millions or billions. Even for a simple model a high number of points is necessary to present smoothly. If a 3D point is considered to be represented by 3 floating point numbers, as its position, and 3 bytes for RGB colors, then we need at least $3 \times 4 + 3 = 15$ bytes per points.

That is, for one billion points we need approximately 15 GB memory space to load and use the complete data.

One of the biggest challenge of processing and visualising 3D point clouds is to work with data that does not fit into memory. Methods that can handle this situation are usually named as *out-of-core* algorithms. These algorithms analyse the scene together with the viewers position and orientation and instead of loading the complete data set only those parts are kept in memory that are necessary for current visualisation. The set of necessary parts, frequently named as chunks, changes dynamically, loading new and removing obsoleted ones, as the viewpoint changes. To make visualisation quick and smooth, both data processing (loading, removing) and rendering should be performed in real time, that is also a challenging task.

1.1 Previous work

Starting from the pioneer work of Levoy and Whitted [1], many works have been appeared in the field of point cloud rendering. The QSplat method of Rusinkiewicz and Levoy [2] was the first multiresolution technique that could interactively visualise hundreds of millions of points using hardware acceleration. Their system transforms the input point cloud into a bounding-sphere hierarchy. Carefully traversing this hierarchy until a leaf node with its stored 3D point information is reached makes it possible to compute visibility and control rendering. Overlapping points are blended by using two rendering passes.

In their paper Wimmer and Scheiblauer [3] introduced a new hierarchy called nested octree that is efficient for rendering large point clouds using the GPU. Their octree consists of an outer octree, that is used for view-frustum culling and out-of-core rendering, while nested inner octrees are memory optimized sequential point trees, which allows rendering by the GPU using sequential processing of stored data.

Kovač and Žalik [4] presented another approach to render large LIDAR datasets. Their typical input data was a LAS file containing 3D points aligned sequentially along multiple scan pathes. Due to this, points that are close in 3D space can appear rather far in the LAS file. To speed up loading spatially continuous parts, LAS data is organized into a quadtree structure and bounding rectangles of point groups are assigned to the leaf nodes. Points are generally not inserted into the quadtree, but their location, with following states:

- unloaded, i.e. points are in the file,
- requests loading, they need to be loaded from LAS file,
- loaded into RAM,
- loaded into GPU RAM.

In order to store only important points in the limited memory, they use an asynchronous dynamic loading strategy that is based on the current viewpoint and frustum. For visualising a GPU implemented algorithm is used that render oriented splats calculated from the viewer's distance and orientation.

Beyond quadtree or octree one can use other hierarchical structures as well, such that the multi-way kd-tree used in the work of Goswami et al. [5]. Contrary to octree, kd-tree divides data equally among all nodes, which significantly speeds up searching in the tree. This technique is capable of high quality and quick rendering of massive point clouds.

In their papers [6,7] Richter and Döllner follow a similar strategy to visualise massive (even of 5 billion points) point clouds. After a preprocessing step, where raw data is converted into a multiresolution tree structure, rendering is performed by the following iterative steps:

1. load tree structure into memory,
2. evaluate structure and select visible points,
3. render points,
4. user interaction, and go back to point 2.

The main advantage of their solution is that they can achieve high frame-rate and render large point clouds in real time, due to the rendering front strategy that adapt automatically to available memory and resources.

One of the main drawbacks of point cloud rendering compared to triangulated meshes originates from its discrete nature. Points are small particles that compose a continuous surface only if they are viewed from a proper distance. If they are too close to the viewpoint, gaps will appear among them yielding a visually unpleasing result. To avoid it, Dobrev et al. [8] presented a solution based on image-space operations. Points are first projected onto the image plane, and filters are used to eliminate visual artefacts. Their operations are:

- Fill empty background pixels based on neighbours.
- Fill holes caused by occluded surface parts using colour, normal and depth values of the surface.
- Apply a low-pass filter to smooth the result.
- Anti-aliasing to remove staircase effects around the silhouettes.
- Depth peeling to group the scene into sorted layers and apply it for rendering transparent surfaces by blending front to back with given opacity values.
- Point cloud shadow texture generation to render shadow effects.

Using the steps above, their solution is capable of providing visually pleasing results at interactive frame rates.

A similar technique is presented in the paper of Pintus et al. [9]. Unstructured raw point clouds are first projected into the image space, where screen-space operators are applied to enhance the quality. Even for very noisy datasets their method can provide visually pleasant output. Their multi-pass GPU-based rendering pipeline consists of a visibility pass, a multi-pass surface reconstruction to fill gaps among points and a multi-pass deferred shading part that enhance photorealism of the rendering.

Point sets can be visualised not only by point rendering, but also using a volumetric, voxel based representation. In this case the 3D space is hierarchically divided into small uniform cubes, the so named voxels. If there exists at least

one point that falls into the cube of the given voxel, then that voxel should be visualised, otherwise not. The colour of the voxel can be determined by the colour of the corresponding points, e.g. taking their average value. The main advantage of the voxel based rendering compared to the point based one is its speed and efficiency. However, this kind of representation is usually a lossy compression, not all the original 3D points are visualised, but only an approximation of them. A nice solution of efficiently rendering large voxel models by using sparse voxel octrees can be found in the work of Laine and Karras [10]. Similarly, the paper of Museth [11] presents a detailed and thorough solution to efficiently store, represent and visualise sparse, time-varying volumetric data.

When the size of a point cloud starts to grow enormously, memory consumption becomes a critical problem. In their paper Elseberg et al. [12] present an octree based format that is very efficient in compressing and storing point data without significant loss of precision. Their solution is capable of storing one billion points in less than 8 GB of memory. Its importance is shown by the fact that to represent a 3D point one needs to store at least the coordinates (3 floats, i.e. 12 bytes), but usually other attributes are added, too, e.g. the RGB colors (3 bytes), or at least a reflectance value (1 byte). Storing only the raw data of one billion (say one giga-) points then would cost at least 12-15 GB. Additionally, the octree structure itself requires memory allocation as well.

The idea of Elseberg et al. was to drastically decrease the memory footprint of a 3D point by storing the coordinates using only two bytes instead of four. It can be done without significant loss of precision due to the fact that each points are grouped into rectangular cuboids of size 5-10 cm, and coordinates are expressed as relative values according to the box corners. With this representation 3D points can be expressed with a precision of 1 micrometer, which is usually more than enough. The correspondence between the points and their surrounding cuboids is stored in the octree structure. This visualisation technique has been implemented in a freely available software, named *3DTK – The 3D Toolkit* at `http://slam6d.sourceforge.net`.

One of the best open source solution for visualising massive point clouds that is actively maintained and used is Potree. Potree is a WebGL based point cloud renderer developed by Markus Schütz at the Institute of Computer Graphics and Algorithms, TU Wien. The theory and algorithms are best described in Schütz's thesis [13]. Further development of his work can be found in [14].

The main contribution of Potree is that it is a web based solution. Presenting a large 3D point cloud does not require any special software, but as WebGL became natively supported by all major web browsers, even on mobile devices, visualisation can be easily transported. Potree makes it possible to upload, view and share 3D point clouds very easily, even if they contain billions of points. As the software itself is open source, it is freely available to use and modify under the FreeBSD license. Some examples of massive point clouds visualised in Potree are visible in Figs. 1–6.

Fig. 1: Example point cloud from potree.org.

Fig. 2: Example point cloud from potree.org.

Fig. 3: Example point cloud from potree.org.

Fig. 4: Example point cloud from potree.org.

Fig. 5: Example point cloud from potree.org.

Fig. 6: Example point cloud from potree.org.

2 Point cloud manipulator

The industry standard format of storing point clouds obtained by laser scanners and LIDARs is LAS, an open binary file format. LAS is efficient for archiving and interchanging point cloud data, but not good enough for visualisation, as no structural information is stored in it. Therefore, in order to be able to present massive point clouds in real time, we need to convert LAS to a proper file format.

Similarly to others, we have chosen an octree based format in our system to store structural point data. There are two kind of representations that our system supports: point based and voxel based representations. Both of them have advantages and disadvantages compared to the other; it depends on the concrete application which one of them to choose.

The point based representation contains all the available points that are originally stored in the raw LAS file. The root of the octree belongs to the bounding box of the points, more specifically to the minimal axis aligned cube that contains all points. This cube is then subdivided into eight smaller ones, equally dividing along each axes. These smaller cubes are assigned to the eight inner nodes of the next level of the octree. If a cube contains points of which number is over a threshold, say 100, then it is also subdivided into eight smaller cubes, and so on, until the point number falls below the threshold. In this case, a leaf node is created which contains the list of the corresponding 3D points. This kind of point cloud representation is lossless and is useful in cases where precision and original point data is crucial.

At point based representation the length of a branch in the octree is determined by the number of points in the corresponding parts of the 3D space. When storing the original points is not so important, one can eliminate the leaf nodes from the octree and set a fixed maximal level for the octree that branches are not allowed to exceed. In this case the nodes at the latest level do not contain points, but only the information if the corresponding voxel is occupied or not, and if so, the corresponding colour (the average colour of the points belonging to), similarly to the inner nodes. This is called as the voxel based representation (Fig. 7). Being a lossy compression, the main advantage of it is the reduced size both on disk and in memory, and due to this the increased speed. This is useful if visualisation is more important than storing the original raw point data. Difference between point based and voxel based representation is illustrated in Fig. 8.

With modern LIDARs it is possible to capture large outdoor scenes, even cities or complete countries as well. Beyond visualisation, this kind of data is frequently used for AI, to train deep learning systems to detect and classify real-world objects. To this end it is necessary to select distinct small parts of the enormous point set and identify them if they represent a car, a tree, a building, a walker, etc. This kind of selection is usually referred to as annotation. See Fig. 9.

Another important pre-processing task for point clouds obtained by LIDARs or other laser scanners is data cleaning (Fig. 10). Due to their technology, these scanners are sensitive for reflective surfaces, that means reflectance can cause

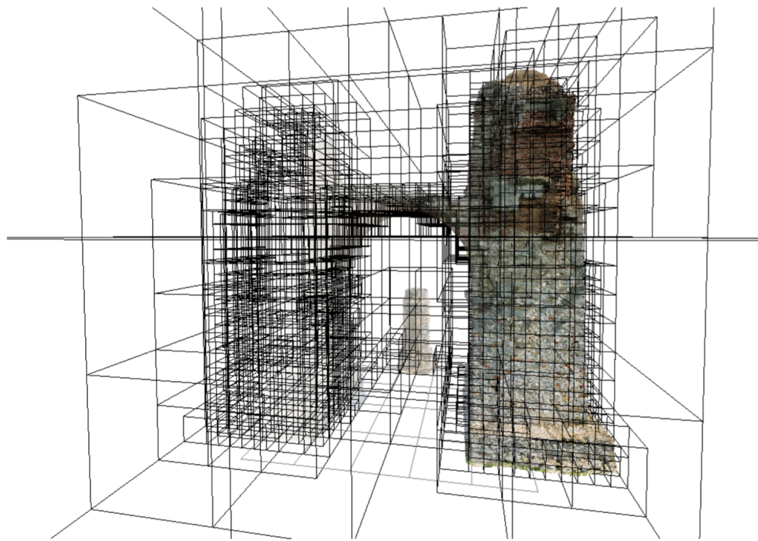


Fig. 7: Voxel based octree. From [13].

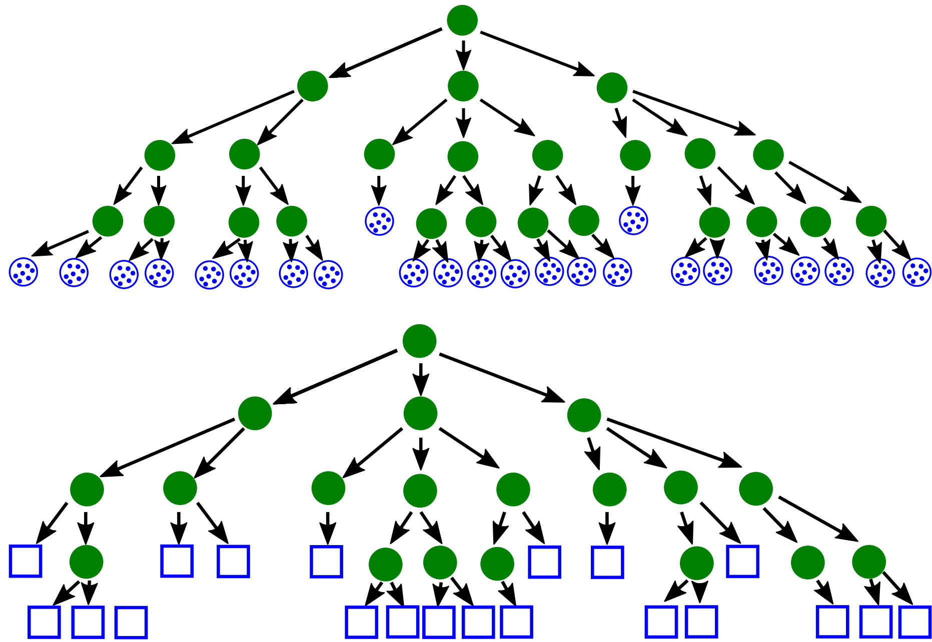


Fig. 8: Difference between point based (top) and voxel based (bottom) octrees. In point based octrees leaf nodes contain lists of points, while in voxel based octrees the leaf nodes are voxels.

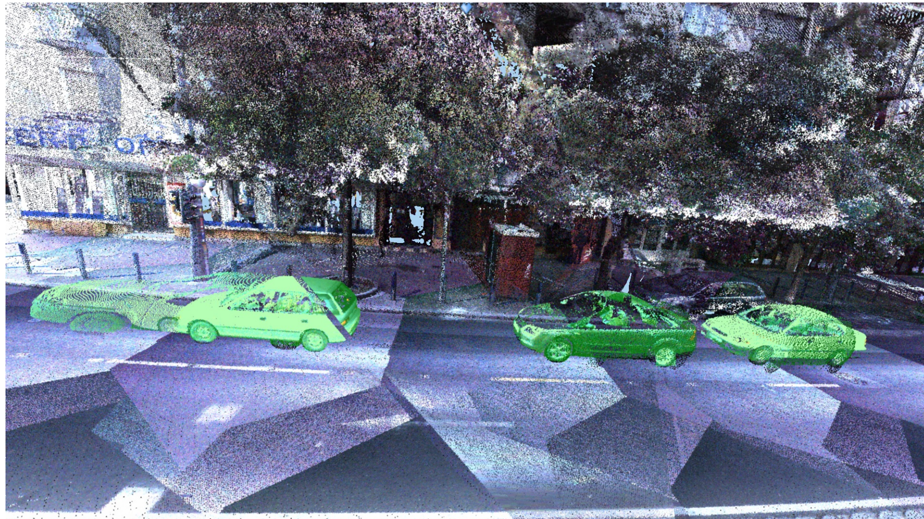


Fig. 9: Example for annotated moving cars.

invalid, mispositioned points in the cloud. Similarly, objects' motion increases the noise as well, as moving objects become blurred and ghosting artefacts appear in the scene. These artefacts yield visually unpleasing results, thus selecting and removing them from the point cloud is an important problem.

Our system has been built to help both annotation and cleaning. The following functions are supported:

- Add to selection. Select a continuous part that is added to the list of selected parts. By applying this tool multiple times one can select multiple distinct areas as well. Coloured by red. See Fig. 11 a).
- Remove from selection. A selected area can be pruned by marking out and removing some parts of it from the selection. See Fig. 11 b).
- Intersect selection. Keep those points as selected that fall within the intersection of the original selected part and the new selection. See Fig. 11 c).
- Select all.
- Select none.
- Finalise selection, extract object. This function is for annotation. When a selection is finished, e.g. a complete tree is already selected, we can finalize the selection and assign a label to it, e.g. "tree". Coloured by green. See Fig. 11 d).
- Convert all finalised objects to selected. Finalized objects cannot modified any longer. However, we can convert them back to "selected" and continue modification.
- Delete selection. This function is for cleaning. Unnecessary parts can be removed from the point cloud. See Fig. 11 e).
- Delete finalised objects. The same as the previous function, but applied on finalised objects instead of selected.

Marking in the 3D space using only 2D tools, namely the screen and the mouse or touchbar, is not self-evident. The selection tool we use in the system makes it available to the user to select an axis aligned rectangular region on the screen, by selecting the starting corner and finishing by the opposite corner along the diagonal. The marked rectangle is in the screen space that we need to transform to the world space. For this we create a 3D cone starting from the camera center and going through the image plane at the selected rectangle. All 3D points that fall into this cone are marked as selected. See algorithms 1 and 2.

Selection, deselection and intersection are expressed by the combination of multiple 3D cones, as seen in Fig. 12. The full selection consists of *parts*. Each part is built from a single selection cone, multiple deselection cones and multiple intersection cones. We say that a point is selected based on the part, if it falls within the corresponding selection cone and all intersection cones, but does not fall into any of the deselection cones. See formula (1), where $\mathcal{P}$ is the part, $\mathcal{S}$ the selection cone, $\mathcal{I}_i$ the intersection cones, $\mathcal{D}_j$ the deselection cones, respectively.

$$\mathcal{P} = \mathcal{S} \cap \left(\bigcap_{i=1}^{n} \mathcal{I}_i \right) \setminus \left(\bigcup_{j=1}^{m} \mathcal{D}_j \right) . \tag{1}$$

Fig. 10: Example for cleaning. Top: before; bottom: after; center: selected points.

(a) Function: add to selection.

(b) Function: remove from selection.

(c) Function: intersect with selection.

(d) Function: finalize.

(e) Function: delete selected.

(f) Function: mark selected for deletion.

Fig. 11: Examples for functions.

Algorithm 1 Algorithm to compute cone transformation that projects and transforms the 3D cone into the $[0,1]x[0,1]$ 2D box

```
Float4x4 computeConeTransform(Int2 selectStartMousePos, Int2
    selectEndMousePos)
{
  int box_x1 = min(selectStartMousePos.x, selectEndMousePos.x);
  int box_x2 = max(selectStartMousePos.x, selectEndMousePos.x);
  int box_y1 = min(selectStartMousePos.y, selectEndMousePos.y);
  int box_y2 = max(selectStartMousePos.y, selectEndMousePos.y);

  // 4x4 world to camera matrix:
  Float4x4 W2C = getWorldToCamera();

  // OpenGL type projection matrix (camera space to clip space):
      projects into the [-1,1]^3 cube
  Float4x4 P = getProjectionMatrix();

  // Matrix to transform [-1,1] to [0,1]
  Float4x4 T = getTranslationMatrix(Float3(1, 1, 0)) *
      getScaleMatrix(Float3(0.5, 0.5, 1));

  // Matrix to scale [0,1] to viewport size
  Float4x4 S1 = getScaleMatrix(Float3(viewWidth, viewHeight, 1));

  // Matrix to get coordinate relative to the top-left of the box
  Float4x4 TB = getTranslationMatrix(Float3(-box_x1, -box_y1, 0));

  // Matrix to scale box size to [0,1]
  Float4x4 S2 = getScaleMatrix(Float3(1 / (box_x2 - box_x1), 1 / (box_y2
      - box_y1), 1));

  // compose final matrix
  return S2 * TB * S1 * T * P * W2C;
}
```

Algorithm 2 Algorithm to check if a 3D point falls within the cone

```
bool isPointWithinCone(const Float3& point3DInWorld, const Float4x4&
    coneTransform)
{
  Float4 p = coneTransform * Float4(point3DInWorld, 1);

  if (p[3] == 0.0) return false;

  p *= 1 / p[3];

  return (0 <= p[0] && p[0] <= 1 && 0 <= p[1] && p[1] <= 1 && -1 <= p[2]
      && p[2] <= 1);
}
```

The connection among the parts is given by an *OR* type relation, that is a point is marked as selected if it is selected by any of the parts. See formula (2).

$$\text{All} = \mathcal{P}_1 \cup \cdots \cup \mathcal{P}_k. \tag{2}$$

With this formalism we can easily express all selection functions discussed above.

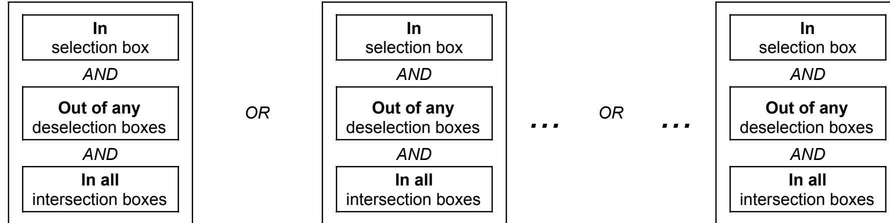


Fig. 12: A point or voxel is selected if it matches to the given rules.

3 Discussion

We have noted that selected or finalized 3D points are recoloured to red and green, respectively, which is implemented in the shaders. However, deleting 3D points and visualise the point cloud as the deleted points do not exist is a more complicated problem. Note that at this stage the points are not physically removed from the octree, as modifying the octree on-the-fly is extremely costly. Instead we mark these points as deleted and simulate as they do not exists. The difficulty arise from the occlusion, that is if a point is deleted, we should see what is behind that. To this end we apply ray tracing to solve occlusion problems.

However ray tracing is costly again, after deleting 10-20 regions from the point cloud makes the visualisation very slow. Due to this we have decided to simulate deletion only until a new selection is not started, and recolour deleted part to dark gray after that. With this solution one is enabled to check if the deletion is successful, each necessary parts are deleted, and if so, continue the work on other parts and leaving the deleted part as *marked for deletion*, see Fig. 11 f). Due to this the visualisation remains real time, without any significant loss in speed.

After cleaning the point cloud by deleting unnecessary sets of points, one would need the possibility to save the result. In our system it is not supported to save the result directly as an octree. Instead, we can export all remaining (i.e. not deleted) points or voxels into a binary or text file. After that a converter program is available that takes as input the original LAS file (from which the octree was generated) together with the exported data file and from them creates another LAS file as output that contains only the remaining points.

At this point there is a significant difference between the two cases, namely when a point based or voxel based octree is used for cleaning. When voxels are stored in the octree, only voxels, that is small 3D boxes can be exported. The converter program reads all original points and verify if they are within any of the voxels or not, which can take a long time. Contrary, when the octree contains the 3D points as well, they can be exported, and hence their conversion to LAS is much faster, computing point-box intersections is not required. Due to this, if our system is used only for cleaning, it is suggested to use point-based octree for that. For annotation both point-based and voxel-based octrees are convenient.

Further examples can be seen in Figs. 13–16.

4 Conclusion

In this paper we gave an overview of methods and techniques to represent and visualise huge point clouds, as well as a detailed description of our system that is capable of manipulating (annotating, selecting, cleaning, etc.) such data. The difficulty of such techniques arises from the size of the data, that can easily exceed the size of the RAM and GPU memory available in the processing computer. The significance of these visualisation and manipulation methods is getting bigger as scanning technologies evolve and become smarter, faster and cheaper.

Acknowledgement. This work was supported by the National Research, Development and Innovation Fund, under the grant NKFIA K-120233.

References

1. Levoy, M., Whitted, T.: The use of points as display primitives. Technical Report 85-022. Computer Science Department, University of North Carolina, Chapel Hill, NC, USA, 19pp. 1985.

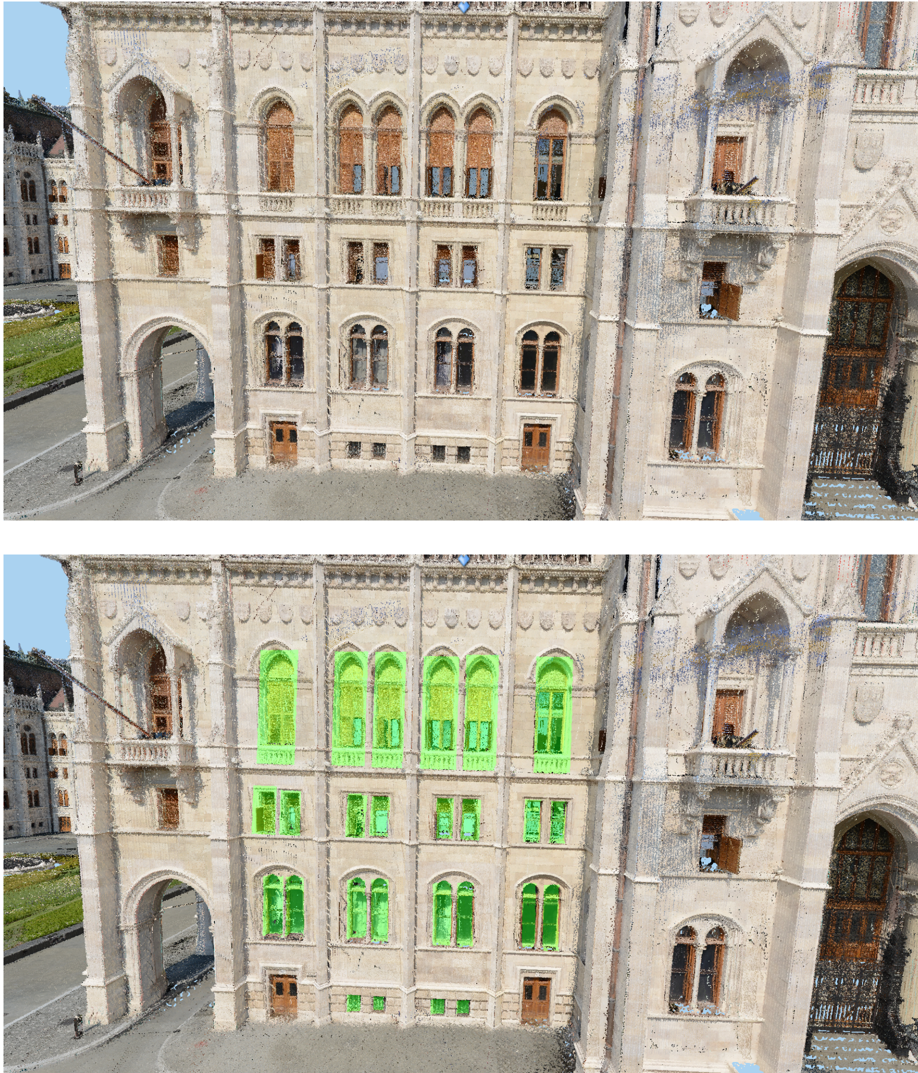
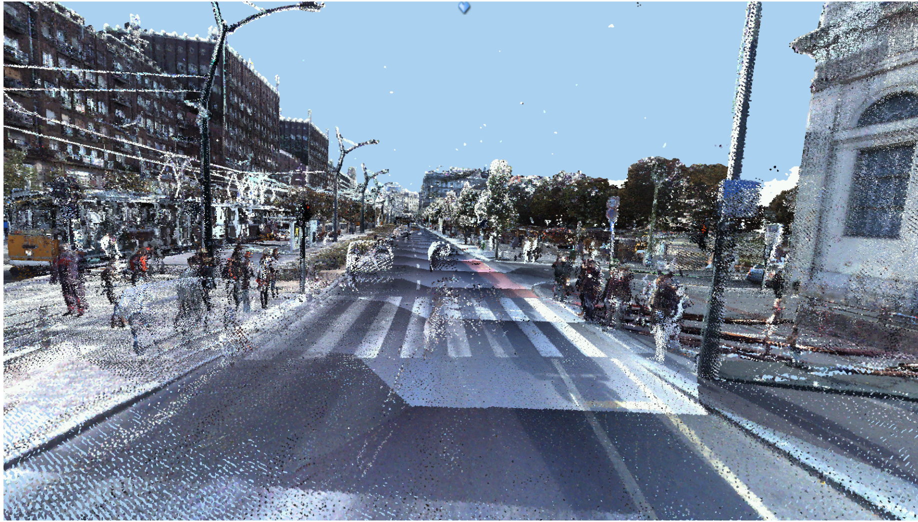


Fig. 13: Example for annotated windows.



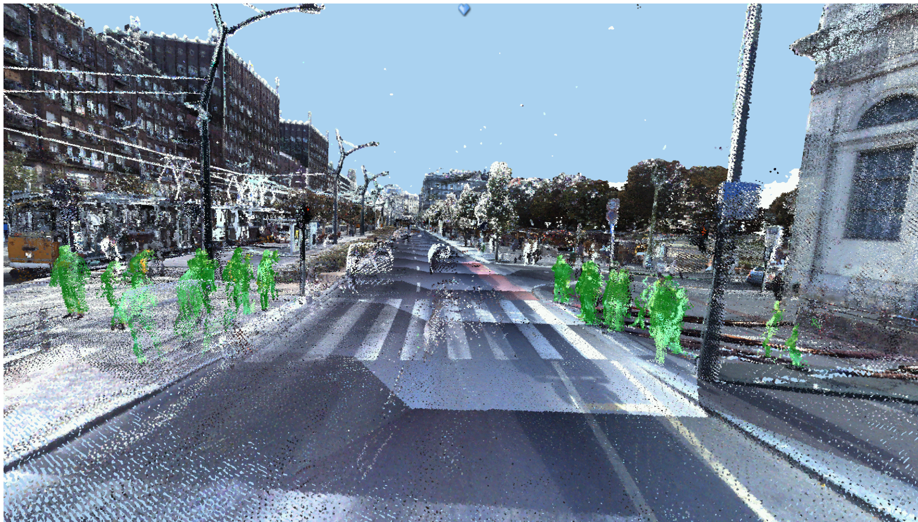
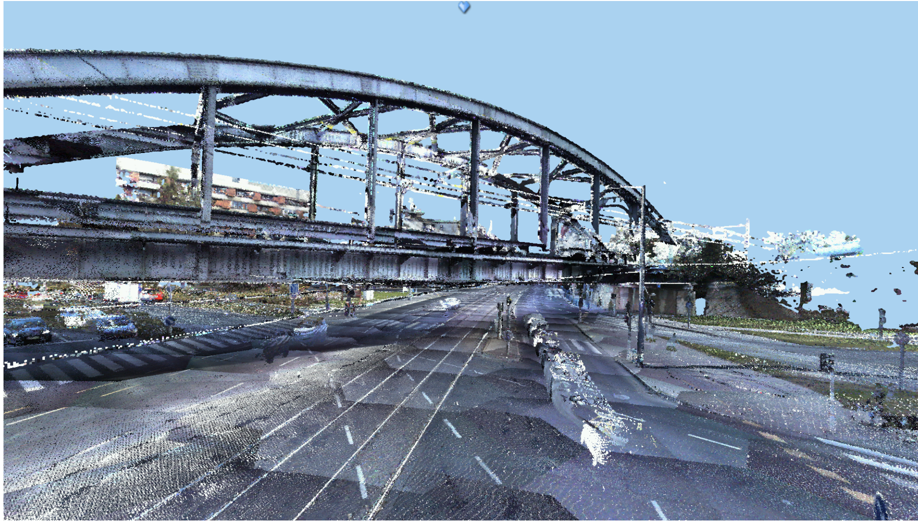
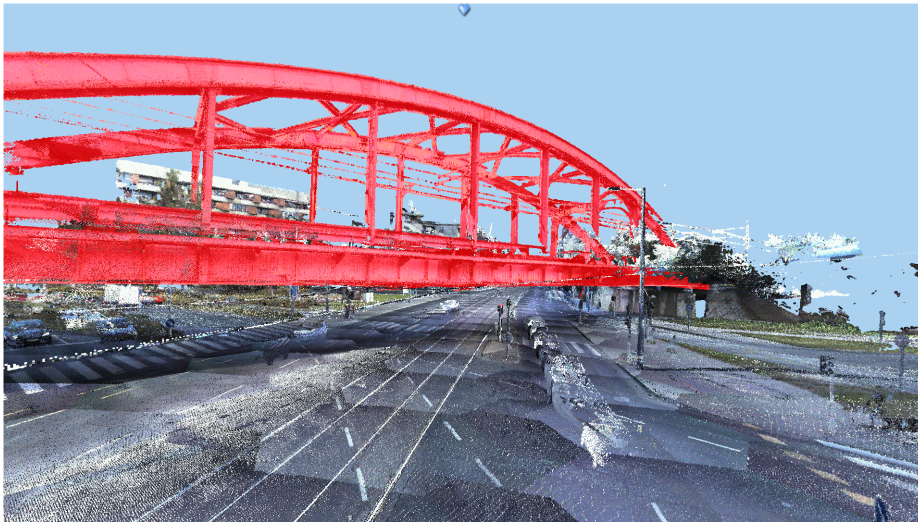


Fig. 14: Example for annotated pedestrians.

Fig. 15: Example for removing ghosting artefacts. Top: before; bottom: after; center: selected points.





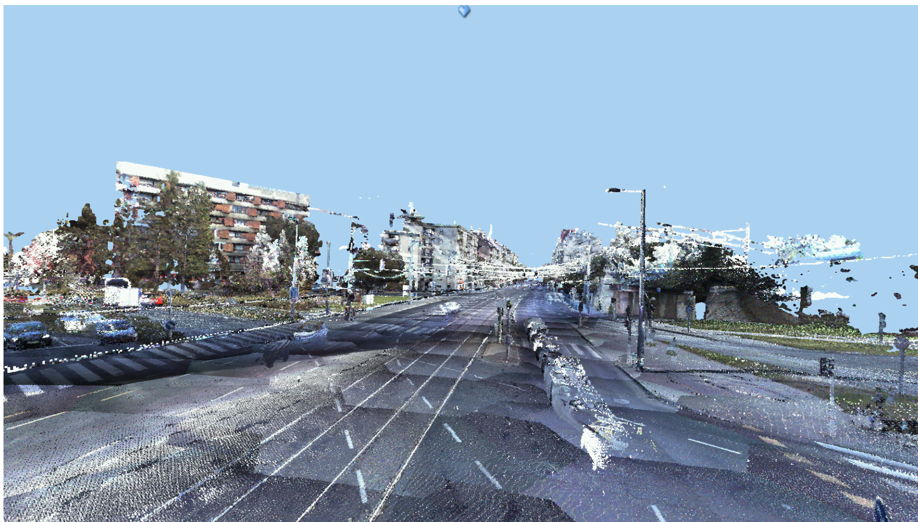


Fig. 16: Example for removing large occluding object. Top: before; bottom: after; center: selected points.

2. Rusinkiewicz, S., Levoy, M.: Qsplat: a multiresolution point rendering system for large meshes. In: SIGGRAPH'00: Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques. ACM Press, Addison-Wesley Publishing Co., New York, NY, USA, pp. 343–352. 2000.
3. Wimmer, Michael, Scheiblauer, Claus: Instant Points: Fast Rendering of Unprocessed Point Clouds. In: Proc. Symposium on Point Based Graphics, Boston, Massachusetts, USA, 2006. pp. 129–136.
4. Kovač, Bostjan, Žalik, Borut: Visualization of LIDAR datasets using point-based rendering technique. In: Computers & Geosciences. 36. 1443–1450. 2010.
5. Goswami, Prashant, Erol, Fatih, Mukhi, Rahul, Pajarola, Renato, Gobbetti, Enrico: An efficient multi-resolution framework for high quality interactive rendering of massive point clouds using multi-way kd-trees. In: The Visual Computer. 29. 2012.
6. Richter, Rico, Döllner, Jürgen: Out-of-core real-time visualization of massive 3D point clouds. In: Proc. 7th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa, Afrigraph 2010, Franschhoek, South Africa, 2010. pp. 121–128.
7. Richter, Rico, Döllner, Jürgen: Concepts and techniques for integration, analysis and visualization of massive 3D point clouds. In: Computers, Environment and Urban Systems. 2013.
8. Dobrev, Petar, Rosenthal, Paul, Linsen, Lars: An Image-space Approach to Interactive Point Cloud Rendering Including Shadows and Transparency. In: Computer Graphics and Geometry. 12. pp. 2–25. 2010.
9. Pintus, Ruggero, Gobbetti, Enrico, Agus, Marco: Real-time Rendering of Massive Unstructured Raw Point Clouds using Screen-space Operators. In: Proc. 12th International Symposium on Virtual Reality, Archaeology and Intelligent Cultural Heritage, Prato, Italy, 2011. pp. 105–112.
10. Laine, Samuli, Karras, Tero: Efficient Sparse Voxel Octrees. IEEE transactions on visualization and computer graphics. 17. 1048–59. 2010.
11. Museth, Ken: VDB: High-Resolution Sparse Volumes with Dynamic Topology. In: ACM Trans. Graph. 32. 27:1–27:22. 2013.
12. Elseberg, Jan, Borrmann, Dorit, Nuchter, Andreas: One billion points in the cloud - An octree for efficient processing of 3D laser scans. In: International Journal of Photogrammetry and Remote Sensing. 76. 76–88. 2013.
13. Schütz, Markus: Potree: Rendering Large Point Clouds in Web Browsers. PhD Thesis. 2016.
14. Schütz, Markus, Mandlburger, Gottfried, Otepka, Johannes, Wimmer, Michael: Progressive Real-Time Rendering of One Billion Points Without Hierarchical Acceleration Structures. 10.13140/RG.2.2.23386.29120. 2019.