Rendering Large Point Clouds in Web Browsers

Markus Schütz*
Supervised by: Michael Wimmer†

Institute of Computer Graphics
Vienna University of Technology
Vienna / Austria

Abstract

We present a method to display large point data sets in web browsers. Such data sets can consist of hundreds of millions to billions of points and are therefore too large to be loaded and rendered at once. The idea of this method is that only points inside the view frustum and up to a certain level of detail are loaded and rendered. Using adjustable point-count limits, even mobile and low-end desktop hardware is able to display these point clouds in real time or at least interactively. The implementation is based on standard web technologies and requires no additional plugins to be installed. This allows developers to combine it with other web applications, like mapping software, in order to synchronize georeferenced point clouds with world map overlays. We also show how to choose point sizes in order to avoid holes and hide varying point densities caused by different levels of detail.

Keywords: WebGL, Point Cloud, LIDAR

1 Introduction

Various 3D scanning technologies such as laser scanners and photogrammetry produce enormous amounts of point cloud data. Datasets with billions of points are not uncommon anymore. Processing and rendering these datasets is a challenging task that neither the memory nor the speed of today's hardware can handle in real time unless broken down into smaller parts. The simplest approach is to tile datasets in small regional chunks and handle one or a few at a time. Another one is to subsample data down to a manageable size. A third one, which is gaining more and more popularity, is the combination of both by creating a multi-resolution hierarchy. Such a hierarchy consists of multiple levels of some sort of tree with a low-resolution model stored at the top and increasing resolutions stored in all descendants.

This paper is based on the multi resolution octree of Instant Points [1]. The concept of inner and outer octree was simplified to one octree where all nodes, including root, inner and leaf nodes, store various resolution subsets of the original point cloud data. Unlike the inner octree approach, we do a uniform point selection with a user-defined point spacing. This approach has higher preprocessing costs but improves visual quality at lower levels of detail. This is especially useful with slow internet connections, where viewers have to wait longer times until higher levels of detail are loaded, but also for low-end hardware with a low point budget. The predefined point spacing also allows for a new adaptive point-size mode that changes the point size according to the level of detail in order to avoid holes. However, points that are close together will be discarded due to the spacing requirements. Like the Instant Points system, we store one file for every node, which makes it possible for the client to load the necessary data without relying on server-side applications.

Our implementation, Potree [2], and some of the examples presented in this paper are available online at http://potree.org

2 Related Work

QSplat [3] is a multi-resolution algorithm that traverses a bounding-sphere hierarchy and builds the rendered point cloud point by point. It adjusts level of detail to the rendering duration. The fine granularity of the hierarchy, each point is represented by a leaf or inner node, allows to render low-detail images during user navigation and progressively higher levels of detail once movement has stopped. Hierarchy traversal, on the other hand, is very costly, and the point-wise assembly of the visible dataset makes it hard to efficiently use the GPU.

Instead of associating each node with just one point, Layered Point Clouds (LPC) [4] store M points per node, where M can be chosen freely. This approach is much more GPU friendly, as points can be stored on the GPU in blocks of M points each, and the application only has to tell the GPU which blocks to render. Our approach also stores blocks of points in each node. The difference to LPC is that points are chosen differently and it is not required to have exactly M points in each inner node. LPC also uses a binary tree that continuously splits along the longest axis, while our approach uses an octree instead.

Plas.io [5] is a web-based viewer for LAS and LAZ files, and coupled with the point-streaming back-end grey-

*mschuetz@potree.org
†wimmer@cg.tuwien.ac.at

hound. The main differences to our implementation are that greyhound uses a quadtree instead of an octree, points are chosen according to the distance to a grid center during indexing, and the client does not depend on the tree hierarchy since the server takes care of it. The quadtree approach works well for LIDAR datasets with large length and width but relatively low height. General-case point clouds with larger height are problematic, though. Indexing, on the other hand, is faster due to simpler point-selection methods.

glob3mobile [6] is a mobile mapping framework with point cloud support. Points can be stored either in a quadtree or in an octree, depending on the extent of the dataset. Data is stored in leaf nodes only, which decreases potential overhead in high zoom levels and top down views, where few leaf nodes are visible. On the other hand, at low zoom levels with a high amount of visible nodes, the overhead increases.

ShareLiDAR [7] also uses a multi-resolution model approach and it is currently the only web viewer listed in this paper that supports normals and therefore illumination as well. According to their description, they have a preprocessing throughput of 40kb of LAS files which is roughly 1500 points per second. For large datasets with billions of points, this throughput is problematic.

3 Multi-Resolution Octree

In order to be able to render point clouds in real time and to keep load times low, we first create a multi-resolution octree hierarchy of the point cloud. The first level of this octree, the root node, contains a coarse representation of the whole point cloud. The resolution is defined by the *spacing* parameter, which specifies the minimum distance between points at root level. The default spacing is set to

$$spacing = \frac{boundingBoxDiagonal}{250} \tag{1}$$

but users may define other values if required. Each subsequent level halves the spacing and stores models with increased resolution.

The spacing influences the number of points in each node and therefore affects download times, the total number of nodes, the number of rendered nodes for a certain point budget and the efficiency of frustum culling. The default value was chosen as a trade-off between fewer points per node to improve download times but enough points to avoid generating a large amount of mostly empty nodes.

Since hierarchy traversal is done on client side, the client has to know about the hierarchy. Depending on the size of the input dataset and the indexing parameters, the octree can grow up to millions of nodes. Downloading the full hierarchy at once increases the initial load times. To keep load times low, the hierarchy is split into smaller chunks, and only parts that are needed will be loaded. A chunk contains a node and its descendants for the next *chunkDepth* levels. Assuming *chunkDepth* = 5, a chunk is generated for the root node, containing the hierarchy from root to descendants at level 5. The same is repeated for all nodes at level 5, resulting in a multitude of chunks that contain the hierarchy from level 5 up to level 10 in the respective regions.

4 Point Cloud Indexing

This section describes how a multi-resolution hierarchy is created from an input point cloud.

First, the bounding cube of the input data is calculated and the spacing and the depth of the octree are defined. The default value for spacing is given by Equation 1 but may be set to any other value by the user. The octree depth has to be defined by the user. In the next step, points of the input dataset are subsequently added to the octree. If the distance to any other point inside the root is larger than the spacing, the point is added to the root node. If there is already another point in close proximity, it is passed to the next level and the same test is repeated with half the spacing. This process is repeated until the point has been added to a node or the octree depth has been exceeded. In the latter case, the point is discarded and will not be added to any node. No duplicate points are generated.

Each node may contain thousands of points. To reduce the amount of necessary distance tests while adding a new point, points in a node are stored in a 3D grid. The length, width and height of each cell is equal to the *spacing* at the nodes level. During the distance test, only points in the same cell and neighbouring cells have to be tested.

The hierarchy and nodes are written to disk regularly, e.g., for every millionth point processed. The results up to the last write can be viewed at any time, giving the user the possibility to immediately cancel the conversion process if, for example, the user decides to adjust conversion parameters based on the current results.

Each node of the octree is given an ID that also represents its exact position in the hierarchy. The numbers in the ID stand for indices ranging from 0 to 7. The root node is named r. The first child of the root node has index 0 and therefore its ID is r0, while the last child of the root has index 7 and ID r7. The same is repeated for each level, always concatenating the ID of the parent and the index of the child to calculate the child's ID. The second child of r0 has index 1 and thus the id r01.

5 Data Streaming and Disposal

This section describes how point data is loaded and unloaded.

The indexing process splits the point data into small nodes and stores each of them in its own file. The only task of the server is to host these files and send them to the client upon request.

Initially, the client loads metadata such as bounding box, spacing and point attributes from the server. In the next step, the first hierarchy chunk, containing the hierarchy for the first few levels, is loaded. At this point, the client starts to calculate the visible nodes and determines nodes that are visible but have not yet been loaded. Unloaded nodes with the largest screen-projected size are then requested from the server until at most *maxParallelRequests* are loaded at the same time. A value of 4 for *maxParallelRequests* has proven to work well in practice. If a node has a hierarchy chunk associated with it, that chunk will be loaded as well, thus expanding the currently loaded hierarchy by another few levels in the respective region.

The client cannot load and store an infinite amount of data in memory. It is therefore necessary to remove nodes which are no longer or rarely used. This is done by keeping track of the least recently used (LRU) nodes. After a certain threshold on the number of loaded points has been reached, the client starts to remove least recently used nodes from memory before loading new ones. Previously disposed data is often loaded faster on future http requests since web browsers usually cache data by themselves.

6 Rendering

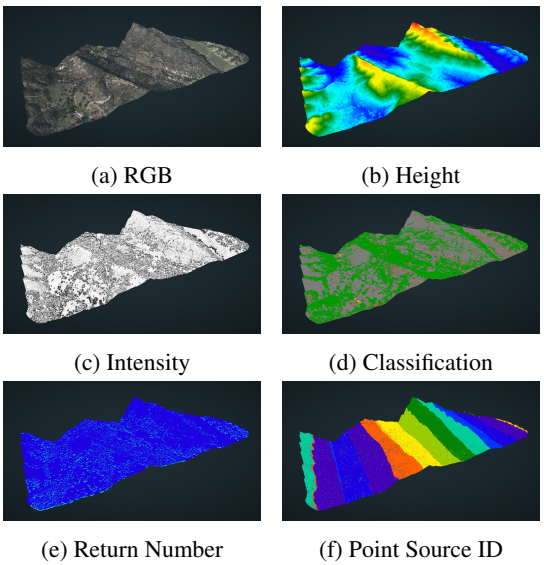
This section describes the rendering process, including hierarchy traversal to calculate visible nodes, coloring, calculating point sizes and different point-rendering qualities.

6.1 Octree Traversal and Visible Node Determination

Octree traversal fulfils 4 main tasks:

- Discard nodes outside the visible area (Frustum Culling)
- Prioritize nodes with large screen-projected size
- Enforce point budget
- Discard nodes with a small screen-projected size

The traversal is done in a largest to smallest screen projected size order since nodes with a larger projected size tend to have a higher impact on visual quality. This is done by adding the children of each traversed node into a priority queue, and then visiting the node with the highest priority. The priority is given by the screen-projected size. A child always has a lower priority than its parent, but distant high-level nodes may have a lower priority than low level nodes that are close to the viewer. During traversal, the *visiblePoints* and *visibleNodes* variables keep track of the amount of points and nodes that were found to be visible. Traversal stops if there are no more nodes to visit or if a user-defined point budget has been reached.



(a) RGB (b) Height (c) Intensity (d) Classification (e) Return Number (f) Point Source ID

Figure 1: Coloring different point cloud attributes; CA13 point cloud courtesy of [8]

Frustum culling is done using box frustum intersection tests. A node that is not inside or does not intersect the view frustum will not be rendered and is omitted from further processing.

Nodes with a small projected size have a smaller impact on quality. They are discarded if their size is lower than a user-defined threshold.

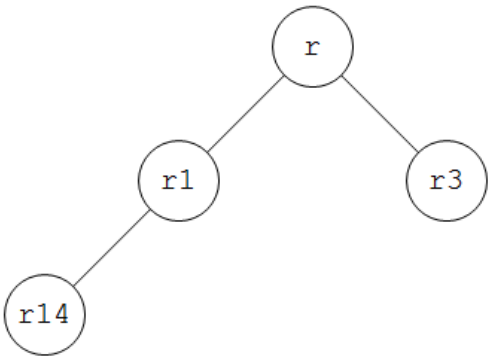
The visible hierarchy is kept in a list for use with other features such as ray casting, point picking and adaptive point size, which is explained in detail in subsection 6.3.

6.2 Coloring

Point clouds can have a variety of different attributes. LIDAR data, for example, often contains intensity, return number, point source ID and classification, but not necessarily color (RGB). Photogrammetry-based point clouds, on the other hand, have color and possibly normals, but no intensity or return number. In our system, octree nodes are stored as files on the disk, and the format of these files can be chosen freely. For point clouds with only color data, points are stored in binary files with coordinates and color. For point clouds with color, intensity, classification or return number, points are stored in LAS or compressed LAZ files, instead. Color data is rendered as is, while intensity is rendered using a grayscale gradient, classification is rendered using a look up table, and other attributes are rendered using rainbow color gradients, as shown in Figure 1.

6.3 Point Size

One problem of level of detail using multi-resolution hierarchies is the noticeable difference in point density in



ID	index	children	offset
r	0	00001010	1
r1	1	00010000	2
r3	2	00000000	0
r14	3	00000000	0

Figure 2: Hierarchy encoded in a 1D array containing children masks and relative offsets to a node's first child.

regions with different level of detail. To overcome this problem, an adaptive point-size mode was implemented. This mode sets the point size based on both, the visible octree depth at the point position and the distance to the camera. Calculating the visible octree depth for all visible points and sending this data to the GPU each frame is a relatively slow process. Instead, this calculation is done directly on the GPU. The visible hierarchy is encoded in a 1D array, stored in a texture and sent to the GPU. Inside the vertex shader, the hierarchy is traversed towards the vertex position and the *localOctreeDepth* variable is incremented along the way.

Nodes are stored in breadth-first order and are encoded in 3 bytes, each. The bits of the first byte indicate which children are visible. The second byte contains the relative offset to the node's first child inside the array. These 2 properties are sufficient to traverse the octree from top to bottom. The third byte is empty. Figure 2 shows an example of a hierarchy and its encoding. Each traversed level requires one texture lookup, counting how many bits have been set in a byte and whether a certain bit has been set or not. This puts additional overhead on the vertex shader, but it also reduces the number of vertices and fragments required to fill holes.

The point-size is calculated by taking the spacing of the octree root, then halving the size for each visible node at that location and finally computing the screen projected size, as shown in Equation 2 and 3.

$$worldSpaceSize = \frac{spacing}{2^{localOctreeDepth}} \tag{2}$$

$$pointSize = project(worldSpaceSize) \tag{3}$$

This algorithm only works properly if the *worldSpaceSize* is higher than the sampling density of the original point cloud data. In regions where the size is lower than the original sampling density, holes will appear because there are not enough points available to make up for the decreased point size. For a good utilization of adaptive point size it is therefore important that the octree depth is not too high.

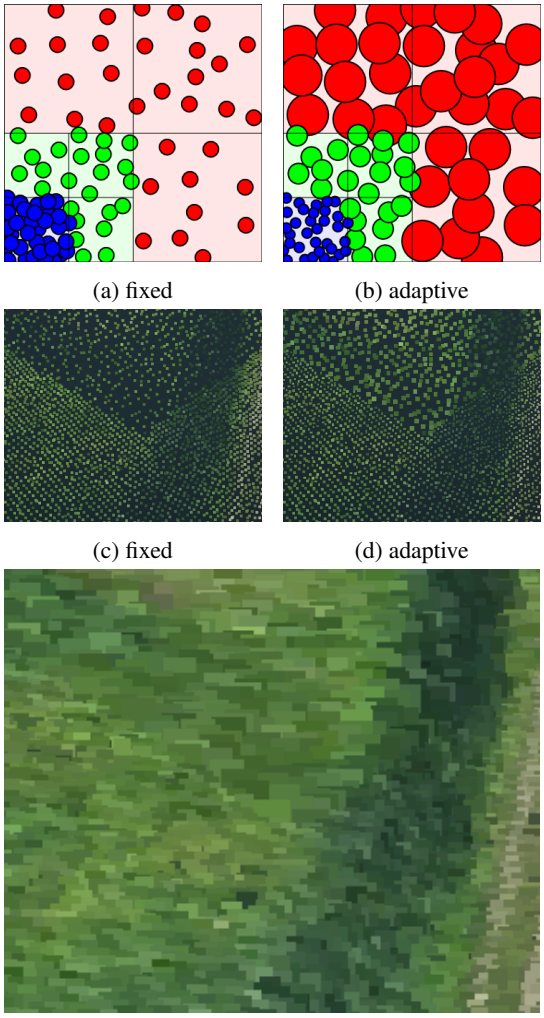
Figure 3 shows the difference between fixed and adaptive point size.

6.4 Rendering Quality

Most point cloud viewers render points either as squares or circles. Increasing the size will cause these primitives to overlap and reduces the readability of high-frequency features such as text and fine details. In order to improve readability, the high-quality splatting [10] algorithm using screen-space aligned circles was implemented. Instead of rendering only the fragments closest to the camera, fragments within a certain distance are blended together. This algorithm requires at least 3 rendering passes. In the *depth pass*, a linear depth map with an additional linear offset, for example 1cm, is rendered. In the next step, the *attribute pass*, all fragments that pass the depth test, i.e., all fragments closest to the camera as well as fragments at most 1cm behind them, will be blended together. The fragments are weighted according to their distance to the center of the point, and the weighted value as well as the weight will be summed up. The last step is to normalize the buffers by dividing the weighted sum of the attributes by the sum of their weights. Both, blend depth and weight function can be made dynamic to adapt to different needs. Large-scale point clouds will need a different blend depth than point clouds of small objects. Changing the weight function can result in very smooth or blurry but also very sharp images. High-quality splatting gives very good results, but the downside is the need for at least 3 rendering passes.

In order to achieve good results in just one pass, we implemented another point-rendering algorithm with a similar idea. High-quality splatting assigns weights to fragments based on their distance to the point center and then blends them together. Instead of summing up weights and attributes, we use a weight function as an offset to the fragment depth. Essentially, this means that instead of rendering screen-aligned squares or circles, each point is rendered as a three-dimensional object. Fragments far from a point's center are more likely to be occluded due to their high depth offset. The result is a nearest-neighbour-like interpolation with similarities to a Voronoi diagram.

Figure 4 shows a comparison of the different rendering modes.



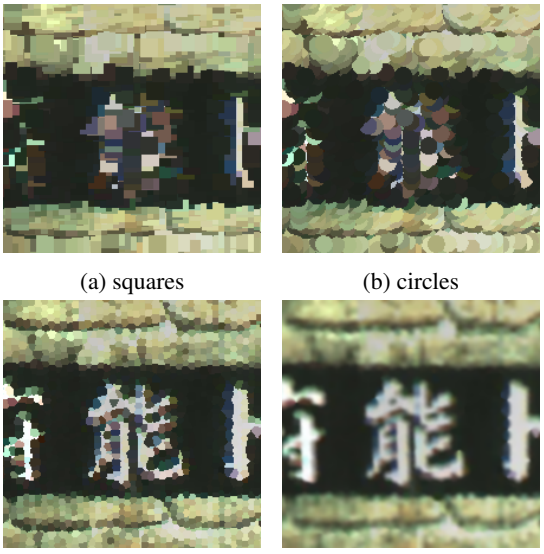
(a) fixed (b) adaptive

(c) fixed (d) adaptive

(e) adaptive optimal

Figure 3: With fixed or projected point size, noticeable holes appear in regions with a lower level of detail. In figure (b) and (d), adaptive mode is used to increase point size in order to avoid holes. In figure (e), point size was chosen to avoid holes completely. Highway construction point cloud courtesy of [9]

.



(a) squares (b) circles

(c) interpolation (d) splats

Figure 4: 4 different rendering modes. Squares and circles have issues with overlapping points. Interpolation and splats provide considerable improvements in readability of high frequency features. Point cloud courtesy of [11]

.

7 Georeferencing

One of the largest use cases of point clouds is the capturing of landscapes with LIDAR or photogrammetry. These kinds of point cloud scans can be georeferenced, which means they can be assigned coordinates that refer to exact positions on the planet. A variety of projections exists for different tasks and locations. These projections are usually planar and do not account for the earth's curvature, but they work as a good approximation in small regions. Additionally, coordinates are often stored as 32bit floats or integers which have enough precision for a given area but not the whole planet. Most projections are therefore only valid for small regions or countries and not meant to be used outside.

One of the challenges of working with georeferenced data are the huge values of point coordinates. According to Spatial Reference [12], x coordinates in EPSG:21781 (Swiss) projection lie between 485869.5728 and 837076.5648. Floating-point types have high precision for values near 0 but cannot accurately handle such high values. For this reason, the point cloud will be transformed to local scene coordinates by translating all points to the origin.

We use georeferencing to provide a side-by-side view of a 3D point cloud scene and a web map such as OpenStreetMap [13], as shown in Figure 5. In order to display camera position and direction in a web map, the camera scene coordinates are first transformed to the projected point cloud coordinates and then to the map coordinates. The reverse can be done as well, for example to draw a

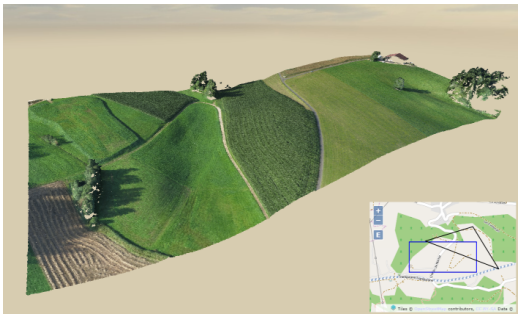
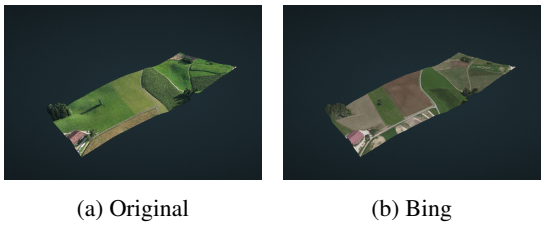
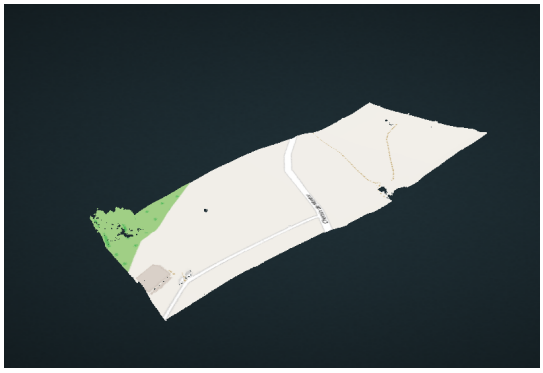


Figure 5: Map overlay showing camera position and point cloud extent. Point cloud courtesy of [9]

.



(a) Original (b) Bing



(c) OpenStreetMap

Figure 6: Bing and OpenStreetMap projected on point cloud data. Point cloud courtesy of [9]

.

line on the map and then display the same line in the 3D scene.

We also use georeferencing to project web maps, e.g., OpenStreetMap or Bing Aerial [14], onto point cloud data, as shown in Figure 6. First, the bounds of the desired region are calculated and then transformed to map coordinates. The map inside these coordinate bounds is used as a texture that is projected onto the point cloud data. Map projections may be used for colorless LIDAR data or to compare map data with point cloud data.

8 Performance

Indexing performance is measured in points per second. Unlike other methods, the presented method also subsamples the point cloud, so another important measure is the sampling ratio of points read versus points written. Table 1 shows our performance results.

Dataset	spacing	depth	points/s	ratio (%)
CA13	315	10	33k	49
		9	95k	14
		8	121k	4
		7	178k	1
Lion	0.05	4	98k	77
		3	133k	29
		0	333k	1.8

Table 1: Indexing Performance. points/s gives the number of points per second that were processed and ratio the amount of points that were written. Close points are discarded. All tests were done on the same 5400 rpm hard drive.

GPU	points	mode	FPS
860M	2M	fixed	104
	2M	adaptive	100
330M	1M	fixed	14
	1M	adaptive	11

Table 3: Performance of fixed and adaptive point-size modes. In both cases, 1 pixel was rendered per vertex to avoid influence of the fragment shader.

Rendering performance on 2 different notebooks is shown in Table 2.

Performance of adaptive point-size is shown in Table 3.

9 Conclusions and Future Work

We have shown a method to render large amounts of point cloud data in real time in web browsers without the need to download the full dataset. A pre-indexing step is required to sort points into an octree structure, which makes it possible to efficiently load just small parts of the dataset needed for the current view point. For georeferenced point clouds, we also showed how to synchronize them to web maps such as Bing or OpenStreetMap in order to display a map overlay showing current camera position and direction and also how the map can be projected onto point datasets.

Figure 7 shows some screenshots of point clouds that were rendered with our work.

Some important future tasks include improved indexing performance, supporting normals for illumination and avoiding generating a large amount of nodes with a small amount of points in each. The biggest problem with indexing right now is that a relatively slow dart-throwing-type algorithm is used in order to guarantee a minimum distance between points and to sample evenly distributed points without regular repeating patterns.

Additionally, further improvement is needed to make the adaptive point-size mode work with any octree depth

Dataset	#points	#rendered points	#rendered nodes	GT 330M(fps)	GTX 860M(fps)	Adreno 320(fps)
CA13	5500M	400k	77	41	148	12
		991k	115	21	122	4
		1987k	198	10	97	
Lion Statue	0.34M	150k	64	60	136	36
		340k	199	24	93	18
Matterhorn	90M	986k	81	11	120	8

Table 2: Rendering Performance Results. All tests were done in Chrome on 2 notebooks and a mobile phone: A Sony VPCF11C4E(2010) with a Nvidia GT 330M, a custom-built Schenker(2014) with a Nvidia GTX 860M and a Samsung Galaxy S4 Active with a Adreno 320. Tests were done with a point budget of either 1 or 2 million points. We have measured the frames per second (FPS) for different view points and listed the number of points and nodes that were rendered. For all measurements, adaptive point size was used to cover holes. For the notebook tests, a 1920x943 pixel canvas was used.

and to convert the whole dataset without discarding points that are close together.

10 Acknowledgements

The CA13 dataset with a total of 17.7 billion points was taken from OpenTopography [8]. The highway in construction dataset is courtesy of Sigeom SA [9]. The funou dataset used for the point quality section was released online by Anan [11]. The matterhorn point cloud is courtesy of Pix4D [15]. This research was supported by the EU FP7 project HARVEST4D (no. 323567).

References

[1] Michael Wimmer and Claus Scheiblauer. Instant points: Fast rendering of unprocessed point clouds. *Proceedings Symposium on Point-Based Graphics 2006*, 2006.

[2] Markus Schütz. Potree. `http://potree.org`. Accessed: 2015-02-13.

[3] Szymon Rusinkiewicz and Marc Levoy. Qsplat: A multiresolution point rendering system for large meshes. *SIGGRAPH '00 Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, 2000.

[4] Enrico Gobbetti and Fabio Marton. Layered point clouds. *Computers and Graphics archive Volume 28 Issue 6,*, 2004.

[5] Howard Butler and Uday Verma. Plasio point cloud viewer. `http://plas.io/`. Accessed: 2015-02-13.

[6] glob3mobile inc. glob3mobile framework. `http://www.glob3mobile.com/`. Accessed: 2015-02-13.

[7] Sharelidar laser scanning sharing platform. `http://www.sharelidar.com/`. Accessed: 2015-02-13.

[8] Opentopography. `http://www.opentopography.org/`. Accessed: 2015-02-14.

[9] sigeom sa. `http://www.sigeom.ch/`. Accessed: 2015-02-14.

[10] Mario Botsch, Alexander Hornung, Matthias Zwicker, and Leif Kobbelt. High-quality surface splatting on todays gpus. *Eurographics Symposium on Point-Based Graphics*, 2005.

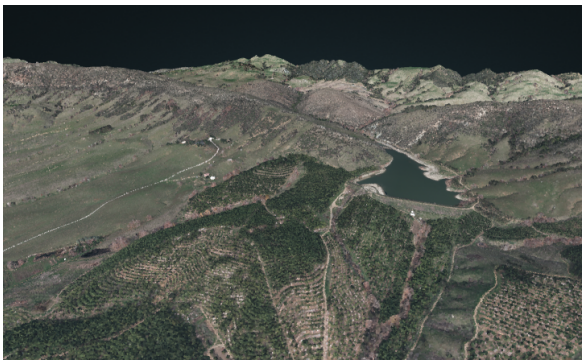
[11] Anan survey. `http://anan.skr.jp/`. Accessed: 2015-02-14.

[12] Spatial reference. `http://spatialreference.org/`. Accessed: 2015-02-13.

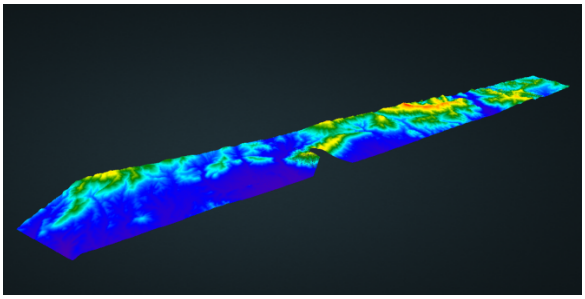
[13] Openstreetmap. `http://www.openstreetmap.org/`. Accessed: 2015-02-14.

[14] Bing maps. `http://www.bing.com/maps`. Accessed: 2015-02-14.

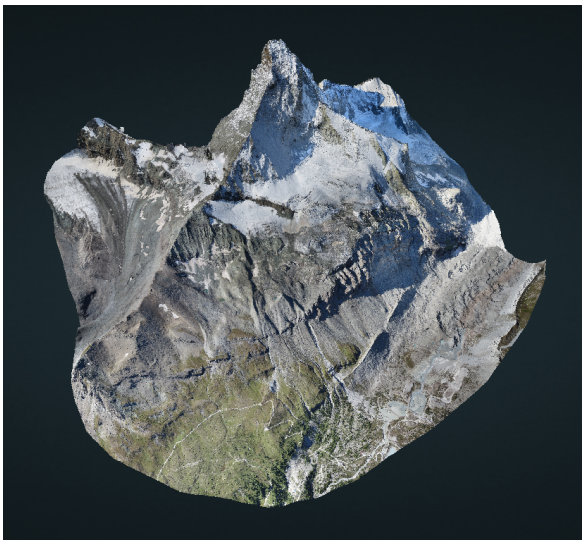
[15] Pix4d. `https://pix4d.com/`. Accessed: 2015-02-14.



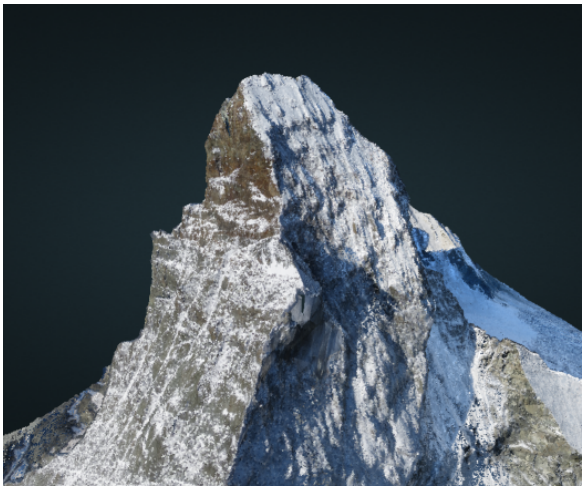
(a) CA13 point cloud 1.4M points rendered



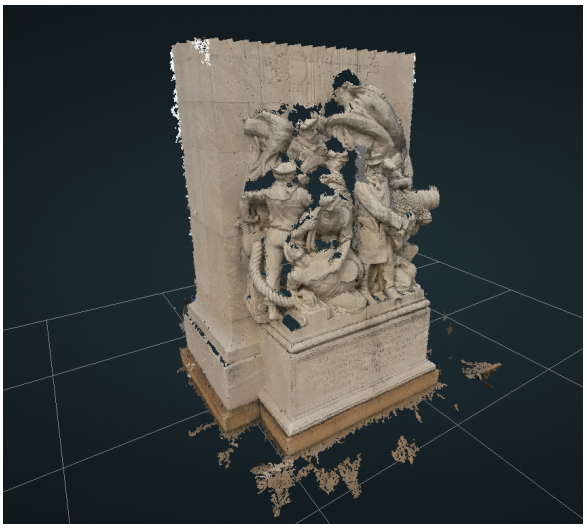
(b) CA13 point cloud 1.1M points rendered



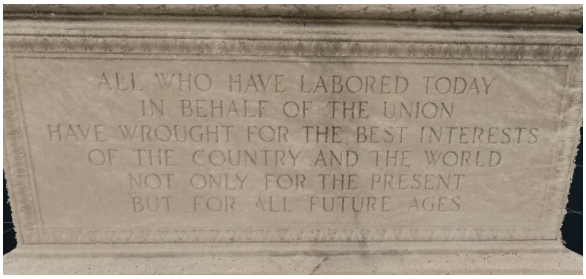
(c) Matterhorn, 1M points rendered



(d) Matterhorn summit, 1M points rendered



(e) Statue in Philadelphia, PA, 1M points rendered



(f) Closeup of statue inscription, 1M points rendered with interpolation shader

Figure 7: Screenshots of different point clouds, rendered either in Firefox or Chrome. CA13 point cloud courtesy of [8]. Matterhorn point cloud courtesy of [15]

.